Introduction to Fortran

Peter Hill

Section 1: A Brief History of Fortran

Overview

- A brief history of Fortran
- Some programming fundamentals
- Hello World

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

Further reading

- “Modern Fortran Explained: Incorportating Fortran 2018”, Metcalf, Reid, Cohen (2018) OUP
- “Fortran for Scientists and Engineers”, Chapman (2018) McGraw-Hill Education
- “Guide to Fortran 2008 Programming”, Brainerd (2015), Springer
- Fortran wiki: http://fortranwiki.org
    - Lots of resources linked from there!
- `gfortran` documentation: https://gcc.gnu.org/onlinedocs/gcc-10.2.0/gfortran/
- Intel Fortran language reference: https://software.intel.com/content/www/us/en/develop/documentation/fortran-compiler-developer-guide-and-reference/top/language-reference.html

What is Fortran?

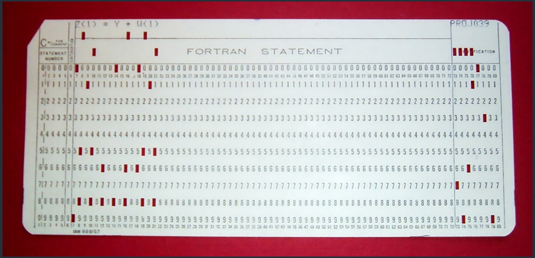
- Old language! 1956 – 62 years old
  - Oldest “high-level” language
  - Many versions over the years, latest is 2018! And next one is being worked on as we speak
- Some parts feel outdated
  - Backwards compatibility is important
  - But original reasons for some features no longer hold
- But still 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



By Arnold Reinhold, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=775153

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
  - Added whole array operations, modules, derived data types
- Further revisions:
  - Fortran 95 - minor revision to F90
  - Fortran 2003 - major revision, support for object-oriented programming
  - Fortran 2008 - native support for parallelisation via coarrays
  - Fortran 2018 - minor revision to F2008, more support for parallelisation and interoperability with C

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
- Slow support from compiler vendors

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:

```
1111001100001111000100000000011111110011000011110101100111000000111000011
```

- 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!
- Words like `this` are either keywords or tiny code snippets
- Words like *this* are (usually) technical words, either common usage or from the Fortran standard
- Words like **this** are just emphasis
- I use Fortran 2018, but everything should compile with gfortran 7

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

Section 2: Types, Grammar and Control

Overview

- Types and variables
- Basic grammar and operations
- Control flow
- Programming style

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!

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

Variable assignment

- Variables starts out *uninitialised*
- It's possible to read an uninitialised variable, but its value will be junk!
    - This is a frequent cause of bugs
- We can *assign* it a value with `=`
- Can assign from *literals* or *expressions*

```
real :: mass, velocity, energy
mass = 2.0
velocity = 3.5
energy = 0.5 * mass * (velocity**2)
```

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`”

Types

There are 5 fundamental types in Fortran:

- `integer`
- `real` – floating point numbers
- `complex` – floating point complex numbers
- `logical` – booleans, two values: `.true.`/`.false.`
- `character(len=<n>)` – fixed length text, also called strings

(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?

- Type tells computer how to interpret set of bits
- Type carries *semantic* information
- Also what operations can be done

```fortran
integer :: a = 1, b = 2
real :: c = 2.0
character(len=1) :: d = "2"
a + b   ! Ok
a + c   ! Ok
a + d   ! Error
```

- Compiler checks semantics based on types
  - Won't check other semantics like `mass + velocity`
  - Some programming languages go further like `Haskell`

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`!

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
```

- Lines 3-4: all variable declarations need to come after `implicit none` and before *executable statements*
- 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

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 / 2.
print*, real(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
- *Call* the function with `()` brackets/parentheses
- *Arguments* or *parameters* go in `()` brackets
  - Can be literals or expressions
- Functions *return results* that can be used in expressions

```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*

```fortran
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:

```fortran
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> {, <stride>}
   ! 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
- Last form does `<upper-bound> - <lower-bound> + <stride>` 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!

```fortran
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

```fortran
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:

```fortran
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:

```fortran
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

Some points on writing Fortran: 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!

Some points on writing Fortran: comments

- Comments are very useful for documenting code
- Not just for other people, you will forget how it works in six months!
- Do explain reasons for doing something

```fortran
! FFT isn't normalised
frequency = rfft(signal)/size(signal)
```

- Don't just repeat what the code does

```fortran
! Divide inverse fourier transform by length of signal
frequency = rfft(signal)/size(signal)
```

- Too many useless comments make code harder to read

```fortran
x = 5 ! assign 5 to x
```

Some points on writing Fortran: whitespace

- Whitespace mostly doesn't matter:

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

- But very important for readability!

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

Some points on writing Fortran: 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 writing Fortran: lines

- Statements must be on a single line unless you use a &, *line continuation*

  ```
  if (some_very_long_and_complicated_condition > some_other_very_long_anc
  ```

  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 writing Fortran: capitalisation

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

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

  is *identical* to

  ```
  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

Some points on Fortran grammar: file names

- The standard doesn't mention source files **at all**
- Linux also doesn't care about file extensions
- Early, fixed-form sources files used `.f` file extension
- With Fortran 90, people started using `.f90` for free-form source files
- Some people thought that `.f95`, `.f03`, etc. should be used for later standards
- Not the case!
- Just use `.f90` and you'll be fine

Section 3: Array, Parameters and Kinds

Overview

- Arrays
- Parameters
- Kinds

Arrays

- Arrays are one of the “killer features” of Fortran
  - Big reason why it’s lasted so long!
- Use `dimension` *attribute* to make anything an array
- 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:

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

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

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

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

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

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

Array indexing

- Just like with `do` loops, we can also specify a *stride*:

```fortran
print*, "vector1(1:3:2):", vector1(1:3:2)
```

- And similarly, the stride can even be negative:

```fortran
print*, "vector1(3:1:-1):", vector1(3:1:-1)
```

- Note that we must also reverse the upper/lower bounds to actually reverse the array

Arrays

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

```fortran
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 `,`:

```fortran
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>.../)`
  - Called *array constructor*
- Can use this to initialise or assign to arrays:

```fortran
integer, dimension(3) :: array1 = [1, 2, 3]
```

- Or even pass to functions:

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

- Unfortunately, only works for 1D arrays! Multidimensional arrays need to use `reshape`
- Note: we can even specify type/kind for all elements:

```fortran
array2 = [real(real64)::1, 2, 3]
```

Literal multidimensional arrays

```
  integer :: matrix2(3, 3)

  matrix2 = reshape([ &
       9, 8, 7, &
       6, 5, 4, &
       3, 2, 1], shape(matrix2))
```

- Second argument to `reshape` is shape of result
- Array of integers with size in each dimension, for example `[3, 3]` above
- For assigning to arrays, we can just use `shape` intrinsic to get the right answer

Operations with arrays

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

```fortran
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?
- These whole-array operations make Fortran very powerful:

```fortran
pressure = density * temperature
```

- Assuming all three variables are the same shape, this just does the right thing

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 :: vector1(3)
integer :: matrix1(3, 3)
integer :: vector2(4)
integer :: matrix2(3, 4)
```

Conformability

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

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

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

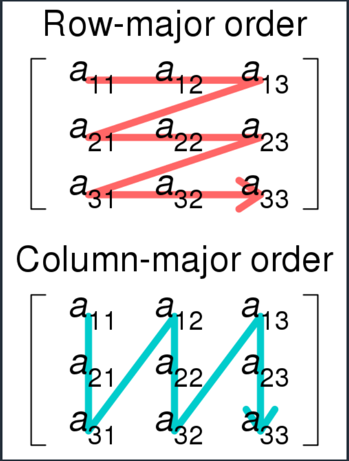
Memory layout

- Maths matrix:

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

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*
- "Which index increases fastest?"



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:

```fortran
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

```fortran
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

```

Useful intrinsics

- The basic mathematical intrinsic functions also work on arrays:
- `sin`, `cos`, `tan`, `sqrt`, `exp`, `log`, etc.

```
real, dimension(N) :: x = [(2.*pi * real(i)/N, i=1, N)]

print*, x
print*, sin(x)
print*, cos(x)
print*, abs(sin(x))
print*, exp(-(x - pi)**2 / (pi / 4.))
```

Useful intrinsics

There are many useful functions for working with arrays built in to Fortran

Many of these intrinsics take an optional `dim` argument to take the function just along that dimension

Working with vectors/matrices

- `dot_product(vector_a, vector_b)`: Returns the dot product of two 1D arrays of the same size
- `matmul(matrix_a, matrix_b)`: Returns the matrix multiplication of two matrices
- `product(array)`: Return the product of all the elements
- `sum(array)`: Return the sum of all the elements
- `norm2(array)`: Return the L_2 norm, essentially `sqrt(dot_product(x, x))`

Useful intrinsics

Logical inquiries

- `all(mask)`: Returns true if all the elements in the logical array `mask` are true, otherwise returns false
  - `mask` can be an expression like: `all(array > 0)`
- `any(mask)`: True if any of the elements in `mask` are true
- `count(mask)`: Returns how many elements in `mask` are true (an integer)

Useful intrinsics

Finding elements and values

- `maxval(array)`: Returns the maximum value in `array`
- `minval(array)`: Returns the minimum value in `array`
- `maxloc(array [, mask])`: Returns the *location* of the maximum value
- `minloc(array [, mask])`: Returns the *location* of the minimum value
  - Get the element closest to `value`: `minloc(abs(array - value))`
- `findloc(array, value)`: Return the location of `value` in `array`
  - Note that this is not so useful for `real` arrays!

Useful intrinsics

Array size and shape inquiries

- `lbound(array)`: Return a 1D array of the lower bounds of `array`
- `ubound(array)`: Return a 1D array of the upper bounds of `array`
- `shape(array)`: Return a 1D array of the size of each dimension
- `size(array)`: Return a scalar of the total size of `array` (i.e. `product(shape(array))`)

Useful intrinsics

Making new arrays

- `merge(tsource, fsource, mask)`: Return `tsource` where `mask` is true and `fsource` where `mask` is false
- `spread(source, dim, ncopies)`:
- `transpose(array)`: Returns the transpose of a 2D array
- `reshape(source, shape [, pad] [, order])`: Returns a copy of `source` with `shape`. `pad` can be used to fill in values if the result is larger than `source`. `order` can be used to do a general n-D transpose

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, dimension(:), allocatable :: array1
real, allocatable :: array2(:)
! 3D array:
real, 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, dimension(:, :), allocatable :: array
allocate(array(10, 5))
! array is now 10*5
```

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:

```fortran
deallocate(array)
```

- Less important than C-like languages due to *automatic* variables and scope – will cover this later
- It is an error to `deallocate` an unallocated array, so:

```fortran
if (allocated(array)) deallocate(array)
```

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

if (status /= 0) then
  print*, errmsg
  print*, status
  ! Note non-constant stop code is technically F2018
  error stop status
end if
```

Guarding `allocate`

- Note `errmsg` may not always be accurate. . .
- Bare `allocate` will terminate on error, possibly with more useful error message
- If you use `stat=` keyword, **always** check it!
    - otherwise program will continue and be wrong

Aside: heap and stack

- Two (main) areas of memory in the computer: the heap and the stack
  - Only a model, not necessarily physically separate!
- The stack is first-in, first-out, like a stack of plates
- Program uses this for passing arguments to functions, local variables, where to return to, etc.
- Stack is only a limited size, so very large arrays as local variables may *overflow* the stack
- The heap is just a big pile of memory
- Slower to find what you want, but there's more of it
- `allocate` puts variables on the heap instead of the stack
- So even if you know the size, can be useful to `allocate`

parameter

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

```fortran
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!
- Super useful for things like `pi`, `speed_of_light`, `electron_mass`, etc.

parameter examples

```fortran
program parameters
  implicit none
  real, parameter :: pi = 4.*atan(1.)
  integer, parameter :: grid_size = 4
  real, dimension(grid_size), parameter :: x_grid = [0., .25, .5, .75]
  real, dimension(grid_size), parameter :: x = sin(2 * pi * x_grid)

  print*, x
end program parameters
```

Kinds of types

- Most important for `real`s (and `complex`)
  - Sometimes important for `integer`s
- Floating point representation
- Doing lots of maths with floating point numbers can lose precision $=>$ need more precision in our `real`s
- Default `real` kind is (normally) 32-bit (4 bytes) (`float` in C)
  - Can represent numbers $\pm 3.4 \times 10^{38}$ to about 7 decimal places
- Using 64-bits (8 bytes) we can represent numbers $\pm 1.7 \times 10^{308}$ to about 15 decimal places

Kinds of types

- Different ways to specify the kind
- 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
- Also possible to change default kind via compiler flags

Kinds of types

- Don't use those! Use either this:

```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

- F2008 added standardised names for common kinds
- Prefer to 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
```

- We will cover what the `use` line means later

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_real64
```

- Can also use `D` instead of `E`: `3.142d0`
- 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`

Kinds of types

- Similar story for `integer`s
- Default kind is usually 32-bit again
- Can represent the numbers -2^{31} to $2^{31} - 1$
- A 64-bit `integer` can represent -2^{63} to $2^{63} - 1$
- Can choose this kind with either:

```fortran
! Get the kind that can represent an integer with 18 digits
integer, parameter :: ip = selected_int_kind(18)
integer(ip) :: x

! or, better:
use, intrinsic :: iso_fortran_env, only : int64
integer(int64) :: x
```

Section 4: Functions and Subroutines

Overview

- Functions and subroutines

Breaking up programs

- Large `program`s become difficult to understand and *maintain*
- Quickly come across chunks of code we want to reuse
- Very good idea to break programs up
- Two ways of breaking up `program`s in Fortran:
    - Functions/subroutines
    - Modules

Functions/subroutines

- `functions`/`subroutines`: reusable chunks of code
- Take *arguments* or *parameters* and (may) *return results*
- Generically called *procedures* or *subprograms*
- May also refer to them both as *functions* – will make it clear when I mean `function`s in particular
- Procedures should have a single responsibility
- Prefer short procedures

Benefits of procedures

- Testing
  - Can test individual parts of the code in isolation
- Reuse
  - Same function can be used in different contexts
  - Can even build up a *library* of functions
- Abstraction
  - Can now think of chunk of code as a thing itself
  - Good names help here!
- Maintainability
  - Code can be easier to understand
  - Can fix implementation without touching rest of code
- Encapsulation
  - Hide internal details from other parts of the `program`
  - Program against the *interface*

Functions

- Takes arguments and returns a single result (may be array)
- **Always** returns a value
- Set result by assigning to function name
- Two ways to write the same thing:
  - Left-hand version required when `<type>` has attributes (for example, `dimension`)

```
function <name>({<argument> {, ...}})
  implicit none
  <type> :: <name>
  {<type> :: <argument>}
  ! body
  <name> = ! result
end function <name>
```

```
<type> function <name>({<argument> {,
  implicit none
  {<type> :: <argument>}
  ! body
  <name> = ! result
end function <name>
```

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 :: 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:

```fortran
program basic_function
  implicit none

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

contains
  function kronecker_delta(i, j) result(delta)
    integer :: i, j
```

Functions

- Use function like `y = kronecker_delta(x, x)`
  - Note you must do **something** with the result!
- Use `()` even if a function requires no arguments: `t = current_time()`
- As long as `implicit none` is in your `program` (or `module`, see later), not necessary in procedures
  - Some people advise as good practice though!
  - I will be skipping this from examples for space reasons

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>({<argument> {, ...}})
  implicit none
  {<type> :: <argument>}
  ! body
end subroutine <name>
```

- Subroutines are used via the `call` statement:

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

Subroutine example

```fortran
program basic_subroutine
  implicit none
  integer :: a = 2

  print*, a
  call add_x_to_y(3, a)
  print*, a

contains
  subroutine add_x_to_y(x, y)
    integer :: x, y
    y = x + y
  end subroutine add_x_to_y
end program basic_subroutine
```

`intent`

- Compiler can optimise better when it has more information
- Useful to know whether arguments are inputs or outputs
- 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 (“read-only”)
- `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! (“write-only”)
- `intent(inout)`: for arguments are to be modified by the procedure (“read-write”)
  - If you don’t explicitly provide an `intent`, this is the default

intent

```fortran
program intents
  implicit none
  integer :: a = 2

  print*, a
  call add_x_to_y(3, a)
  print*, a

contains
  subroutine add_x_to_y(x, y)
    integer, intent(in) :: x
    integer, intent(inout) :: y
    y = x + y
  end subroutine add_x_to_y
end program intents
```

`intent`

- **Always**, **always** include `intent`!
    - Removes class of bugs
    - Adds documentation
    - Can improve performance
- Prefer `function`s over `subroutine`s with one `intent(out)` argument

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!
  - Can use intrinsics `int`, `real`, `cmplx` to convert types and kinds

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`

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)
```

Dummy arguments and arrays

Three choices for passing arrays:

- `dimension(n, m, p)`: explicit size
  - Need to pass `n`, `m`, `p` as well (or get them from elsewhere)
  - Actual argument has to be exactly this size
  - Compiler can only check size is correct 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
  - `n`, `m`, `p` need to be passed in (or got from elsewhere)

Dummy arguments and arrays

```fortran
    call print_array_explicit(array)
    call print_array_explicit_passed_sizes(array, 2, 2, 2)
    call print_array_assumed_size(array, 2, 2)
    call print_array_assumed_shape(array)
    call print_array_assumed_shape_lowerbound(array)
```

Local variables

- Variables declared inside procedures are *local* to that routine
  - Also called *automatic* variables
- Their *scope* or *lifetime* 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 or globals

Local variables

```fortran
program local_variables
  implicit none
  integer :: x = 4
  print*, add_square(x), x
contains
  integer function add_square(number)
    integer, intent(in) :: number
    integer :: x
    x = number * number
    add_square = number + x
  end function add_square
end program local_variables
```

- `x` in the main program and `x` within `add_square` are different variables
  - The inner `x` *shadows* the outer one

Global variables

- Possible for procedures to access variables in the containing scope
    - Technically called *host association*

```fortran
  integer :: x = 4
  print*, x
  print*, add_square(x)
  print*, x
contains
  integer function add_square(number)
    integer, intent(in) :: number
    x = number * number
    add_square = number + x
```

- This is surprising, especially given `intent(in)`!
- Hard to see where `x` comes from
- There are uses for this, but prefer to explicitly pass in variables via arguments

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:

```fortran
subroutine count_to_10_wrong()
  integer :: count = 0
  integer :: i
  do i = 1, 10
    count = count + 1
  end do
  print*, "Initialisation in declaration =", count
end subroutine count_to_10_wrong
```

Initialising local variables

- This behaviour can be useful for *caching* results of expensive calculations
- Nicer to have the explicit `save` attribute

```fortran
subroutine count_to_10_cache()
  logical, save :: first_call = .true.
  integer, save :: count = 0
  integer :: i
  if (first_call) then
    do i = 1, 10
      count = count + 1
    end do
    first_call = .false.
  end if
  print*, "Explicit 'save' and cached flag =", count
end subroutine count_to_10_cache
```

Returning early from procedures

- Sometimes we want to finish a procedure early
    - For example, to avoid deeply nested `if` statements
    - Or to check *preconditions*
- Use the `return` keyword
- Finishes the procedure there and then

```fortran
real function my_abs(x)
  real, intent(in) :: x
  if (x > 0) then
    my_abs = x
    return
  end if
  my_abs = -x
end function my_abs
```

Recursion

- Due to historical reasons, procedures are not recursive by default: they cannot call themselves directly or indirectly
  - Luckily, always possible to write recursive algorithms as iterative instead!
- Need to use `result` keyword to change name of function result
- Use `recursive` keyword:

```fortran
recursive function fibonacci(n) result(res)
  integer, intent(in) :: n
  integer :: res
  if ((n == 1) .or. (n == 2)) then
    res = 1
    return
  end if
  res = fibonacci(n - 1) + fibonacci(n - 2)
end function fibonacci
```

Pure procedures

- It turns out to be very useful for the compiler to know if a function has *side-effects*
  - Does it modify any arguments except those marked `intent(out|inout)`?
  - Does it modify any variables from a different scope?
  - Does it have a local variable with `save`?
  - Does it attempt to `read` or `write` anything?
- A *pure* procedure has no side-effects
  - `function`s must only return values
  - `subroutine`s can have `intent(out)` arguments
- Given the same input, you will always get the same output
  - e.g. `abs(-0.5)` always returns `0.5`
- Useful documentation!
- Compiler can catch more bugs/mistakes
- Compiler may be able to apply some more aggressive optimisations

Elemental procedures

- We saw some intrinsics can handle scalars or arrays
- It is also possible to write our own functions that can operate on both scalars and arrays
- These functions apply the function to each element, so they are called *elemental*
- Elemental functions must be `pure` or marked `impure`

```fortran
  integer, dimension(3) :: x = [1, 2, 3]
  print*, double(x)

contains
  elemental integer function double(x)
    integer, intent(in) :: x
    double = 2 * x
  end function double
```

Returning allocatable variables from functions

- Both functions and subroutines can allocate variables
- Returning an allocatable from a function can be quite convenient, because it will automatically reallocate the assignee

```fortran
function make_grid(grid_size)
  integer, intent(in) :: grid_size
  real, dimension(:), allocatable :: make_grid
  integer :: i
  allocate(make_grid(grid_size))
  do i = 1, grid_size
    make_grid = real(i) / grid_size
  end do
end function make_grid
```

Use like:

```fortran
grid = make_grid(grid_size)
```

Section 5: Input, Output and Text

Overview

- More control flow
- Input and output
- Characters

Choosing an option: `select case`

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

```
select case (<expression>)
case (<case-1>)
  ! do something 1
case (<case-2>)
  ! do something 2
{case default}
  ! do something 3
end select
```

- `case default` matches when nothing else does
  - Not necessary, but absence might hide bugs!

`select case`

- The expression in the `select case` must be an `integer`, `logical` or `character` scalar variable
- Can specify value or range:
  - `value`
  - `:upper`
  - `lower:`
  - `lower:upper`
- Ranges must be of same type
- Case must be known at compile time
  - i.e. literals, `parameter`s, and expressions using them only
- Cases may not overlap!
  - Compare to multiple `if` statements

select case

```fortran
  select case(number)
  case (42)
    print*, "You picked exactly forty-two"
  case (:10)
    print*, "You picked a number less than or equal to ten"
  case (50:)
    print*, "You picked a number more than or equal to fifty"
  case (11:20)
    print*, "You picked a number between eleven and twenty"
  case default
    print*, "You picked a number between twenty and fifty,&
            & excluding forty-two"
  end select
```

select case with characters

- Neat thing about select case in Fortran: works with strings!

```fortran
select case (animal)
case ("cat")
  print*, "meow!"
case ("dog")
  print*, "woof!"
case ("pig")
  print*, "oink!"
case default
  print*, "<generic animal noise>"
end select
```

- Can be useful for parsing user arguments
  - careful not to overdo it though, this is expensive!

Selective array expressions: `where`

- It can be very useful to perform operations on an array only on certain elements
  - e.g. taking `log` of non-zero elements
- The `where` construct does exactly this:

```
where (<array-logical-expression>)
  ! do something on "true" elements
elsewhere
  ! do something on "false" elements
endwhere
```

- Equivalent of `do` loop and `if/else` construct acting on every element
- The `where` mask must conform to the array(s) used in the construct body
- Not very common, but useful to have when you need it

where example

```
where (pressure > 0)
  log_pressure = log(pressure)
elsewhere
  log_pressure = -1.e20
end where
```

is the same as:

```
do i = 1, 6
  if (pressure(i) > 0) then
    log_pressure(i) = log(pressure(i))
  else
    log_pressure(i) = -1.e20
  end if
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:

```fortran
outer: do i = 1, 4
  inner: do j = 1, 4
    if ((i + j) == 6) exit outer
    print row_format, i, j, i + j
  end do inner
end do outer
```

Skip an iteration: `cycle`

- Sometimes need to skip a loop iteration, but keep looping
- Other languages use `continue`: unfortunately, this is an old keyword for `end do`!
- Fortran uses `cycle` instead

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

Skip an iteration: `cycle`

- Like `exit`, can also give `cycle` a loop label:

```
outer: do i = 1, 5
  inner: do j = 1, 5
    if ((i + j) == 7) cycle outer
    print*, i, "+", j, "=", i + j
  end do inner
end do outer
```

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

- Instead of star, can give a *format string*
    - Technically *data edit descriptor*
- Basic form is `'(<something>)'`, where `<something>` is a comma-separated list of format codes

```
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
```

- Lots of format codes: use your favourite search engine!

Formatted I/O

Basic format codes

- `a`: `character`
- `i`: `integer`
- `f`: `real`
- `e`: `real`, but using 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(*, '("        |1234567890123456 78|")')
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:

```
write(*, '(2(i0, 2(f8.3)))') ...
```

- expands to:

```
write(*, '(i0, f8.3, f8.3, i0, f8.3, f8.3)') ...
```

- More complicated example:

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

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

Formatted I/O

- 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
- This will print the whole of `array`, two elements per line:

```fortran
write(*, '("|", i2.1, " ", i2.1, "|")') array
```

Formatted I/O

Other notes:

- Compiler will check types, but unfortunately only at runtime
  - This is because format string can be built dynamically!
- Nice tip: can put format string into a `character`
  - Good if reusing the same format a lot
- If the format string is too small for the data, program will print stars (`****`) taking up the full space instead

Unformatted I/O

- Formatted I/O is about writing variables as text
- Fine for small amounts of data
- But can rapidly become bottleneck at large amounts
- For example, printing a `real` as text: sign, leading digit, decimal point, 8 digits, exponent symbol, exponent sign, 2 exponent digits: total 15 characters
- Assuming ASCII, then 1 byte per character: 15 bytes vs 4 for internal binary representation
- Almost 4x larger!
- Plus additional cost of converting
- Possible rounding errors as well

Unformatted I/O

- When we `open` a file (next section), can choose to open it `form='unformatted'`
- Avoids problems from before
- Now can `read`/`write` directly the internal representation
- 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:

```fortran
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>` is an `integer` (that you've already declared)
- `newunit` will make sure it's unique (and negative)
- `newunit` is F2008. On older versions you need to manage the unit numbers yourself:

```fortran
integer, parameter :: rectangle_unit = 11
open(unit=rectangle_unit, file="rectangle.shape")
```

- There are some pre-declared units, so use values > 10 to be safe

`open` arguments

Lots of possible arguments, but two useful ones are `status` and `action`:

`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!

`open` arguments

`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>)
```

- If <unit> is not an opened unit, close does nothing

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

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

Error checking I/O: `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 :: iostat
character(len=200) :: error_msg
open(newunit=unit_num, file="filename", iostat=iostat &
     iomsg=error_msg)
if (iostat /= 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 iostat
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
```

Namelists

- Another nifty feature of Fortran
- Great for simple things, not so great for more complicated things
- Similar to list/record-based I/O above: give a list of variables to read from/write to file
- But now with reading, have a bit more flexibility

```
<type> :: variable1 {, <variable2> ...}
namelist /<name>/ <variable1> {, <variable2> ...}
read(<unit>, nml=<name>)
```

and the input file looks like:

```
&<name>
  <variable1> = <value1>   ! Comment
  <variable2> = <value2>
/
```

Namelists

- Can also write namelists
  - useful for recording what inputs were actually used
- Variables can be in any order
- Can even miss variables! useful for default values
- Extra ones not in the `namelist` statement is an error though
- Also, `logical`s can be `T`/`F`, `.true.`/`.false.`
- Variables need to be declared first
- Comments are allowed (and are ignored), case insensitive (as usual), and whitespace is mostly ignored (except within names, as usual)
- Unfortunately, `namelist`s are not quite *first-class entities*
  - Meaning you cannot put them in a variable of any kind, pass them to a function, etc.
  - This becomes a pain if you want to do more complicated things, e.g. have multiple particle species, each with identical namelists

Namelist example

```fortran
  real :: height = 2.0, width = 3.0

  namelist /rectangle/ height, width

  open(newunit=file_unit, file="rectangle.nml")
  read(unit=file_unit, nml=rectangle)
  close(file_unit)

  write(output_unit, output_format) height, width, height * width
  print*, ""

  write(output_unit, nml=rectangle)
```

Working with `characters`

- `character` declarations must include the `len`gth:
- `characters` of a fixed length are terminated by *blanks*, i.e. spaces

```
character(len=10) :: cat = "Ziggy     "
```

- `characters` can be indexed similarly to arrays, with one restriction: the colon (`:`) is **required**:

```
cat(:3) == "Zig"
cat(4:4) == "g"
cat(5:) == "y     "
```

Working with `characters`

- The `character len` must match how long it actually is
- For `parameters` though, we can use `len=*`

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

- Only works for `parameters` and dummy `intent(in)` arguments though!

Working with `characters`

- If we don't know how long the `character` needs to be, we can make it `allocatable`
- Also allows us to change the size later
- Use `len=:` in declaration:

```fortran
character(len=:), allocatable :: filename
```

- Normal `a = b` assignment will take care of allocation
- If allocating manually, need to specify the type and `len`:

```fortran
allocate(character(len=10)::filename)
```

character examples

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

print*, len(fixed_string), fixed_string

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

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

Working with `character`s

- We can stick two `character`s together with the *concatenation operator*, `//`:

```fortran
character(len=*), parameter :: cat1 = "Ziggy", cat2 = "Lana"
character(len=*), parameter :: both_cats = cat1 // " and " // cat2
```

- This is the easiest way to build up a string dynamically
- But we can also `write` to a `character`!
- Just use the variable in place of the `unit` argument

Writing to a `character`

```fortran
character(len=7) :: run_name
integer :: run_number

run_number = 456
write(run_name, '(A, I4.4)') "run", run_number
print*, run_name

! Too long for the character!
run_number = 89988
write(run_name, '(A, I4.4)') "run", run_number
print*, run_name
```

Useful intrinsics for `character`s

- `len(string)`: How long is `string`, including the trailing spaces
- `len_trim(string)`: How long is `string`, **excluding** the trailing spaces
- `trim(string)`: Remove all trailing spaces
- `adjustl(string)`, `adjustr(string)`: “Adjust” the string left/right, moving leading/trailing spaces to the end/beginning
- `index(string, substring)`: Return the starting position of `substring` within `string`, or zero if it’s not found.
- `new_line(a)`: Get the new line `character` of the same kind as `a`
  - This is the equivalent of `\n` in C, Python, etc.

Section 6: Modules and Derived Types

Overview

- Modules
- Derived types
- Interfaces
- Miscellaneous

Modules

- Very big programs become difficult to develop and maintain
- Becomes useful to split up into separate files
    - Can group related functions together
- May want to reuse functions between projects
- Early versions of Fortran just stuck subprograms into separate files and linked 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

Modules

- Modules are their own scope
  - Names in one `module` do not clash with names in another
  - Similar to *namespaces* in other languages
- Compiler gets access to same information about procedures as if they were inside the `program`
- Always, **always** use `module`s when using multiple files
- Not limited to one `module` per file
  - But usually a good idea!
- `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`:

```fortran
module <name>
  ! use other modules
  implicit none
  ! variables, types, parameters
contains
  ! functions, subroutines
end module <name>
```

- Just like `program`s, `implicit none` applies to the whole `module`

Modules

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

```
module badbad
  implicit none
  print*, "This won't compile!"
end module badbad
```

Using modules

- Using a `module` is simple:

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

- This makes everything in `particle_properties` available to the whole `program`
  - Equivalent to `from particle_properties import *` in Python
- Only one module per `use`:

```
program track_particles
  use file_utilities
  use particle_properties
  use physical_constants
  implicit none
```

Using modules

- Usually better practice to `use module`s only in the particular functions that need them
- We can also just `use` certain things from a `module`:

```fortran
subroutine push_particle
  use particle_properties, only : kinetic_energy, coulomb_force
  use physical_parameters, only : electron_mass, electron_charge
```

- These can be either variables or procedures
- This is great!
  - More obvious where things come from
  - Doesn't bring in unneeded names
  - Can even rename things if they clash locally

```fortran
subroutine push_particle
  use physical_parameters, only : c => speed_of_light
```

- Note: `use` statements must come before `implicit none`

Module visibility

- Plain `use <module>` brings in **everything** from `<module>` to the local scope
  - This is called *use association*
- Sometimes we want to have some variable or procedure that is used "internally" to a module, and don't want to be able to access it from outside that module
- We can use `public` and `private` statements and attributes to control which names are available to be `use`d
- `private` entities (i.e. variables or functions) won't be visible outside the module
- `private`/`public` statement by itself marks the default visibility
- By default, `public` is assumed
- Then can add either as an attribute to individual entities
- Entities `use`d from other modules can also have visibility attributes applies to them

Module visibility syntax

- As an attribute on a variable:

```
<type>, <visibility-spec> {, <other-attributes>} :: <variable>
! For example:
integer, dimension(2, 2), parameter, private :: internal_array = ...
```

- Or a separate visibility specification statement:

```
<visibility-spec> :: <entity>
! For example:
public :: some_public_thing
```

Module visibility

```fortran
module particle_properties
  use, intrinsic :: iso_fortran_env, only : real64
  use physical_constants, only : proton_mass, speed_of_light
  implicit none
  private ! Marks all entities as private by default
  ! Public attribute on a variable
  real(real64), parameter, public :: deuterium_mass = 1.99955249 * proton_
  ! Separate attribute for procedure
  public :: kinetic_energy
  ! Separate attribute for entity used from another module
  public :: proton_mass
contains
  function kinetic_energy(mass, velocity)
    use, intrinsic :: iso_fortran_env, only : real64
```

Compiling modules

- Compiling a `module` results in a `.mod` file as well as the built object file (`.o`)
  - (Actual file extensions may vary)
- This file describes the names in the `module` as well interfaces to the functions, 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 essentially the *Application Binary Interface* (ABI) and is a Hard Problem to maintain backwards compatibility while still adding new features

Compiling modules

- Getting an executable that you can run is (essentially) two step process:
  - *Compile* source code to *object files*
  - *Link* object files to *executable*
- Compiler normally takes care of both compiling and linking for us
- If we tell the compiler about all the files we want to compile and link together in one go, we don't need to do anything special
- Modules are not executable, so if we don't want to compile everything at once, need to tell compiler to stop at the object file stage
- To link the final executable, we then need to tell the linker about all the object (`.o`) and module (`.mod`) files

Compiling modules

- For `gfortran`, use the `-c` flag to just compile and not link:

```
# Creates my_module.o and my_module.mod
$ gfortran -c my_module.f90
# Just creates my_program.o
$ gfortran -c my_program.f90
```

- To link, we can use `gfortran` again:

```
# Link together the object files
$ gfortran my_module.o my_program.o -o my_program
```

- Note that `gfortran` looks in the current directory for the `.mod` files
  - Don't need to explicitly list the `.mod` files
- Can use `-I<directory>` to tell `gfortran` to look somewhere else

Modules in practice

- Cannot have circular dependencies
  - `module_A` `use`s `module_B` which `use`s `module_A`
  - This won't work!
  - Ways round it using `submodule`s: not covered here!
- Some trickiness: there is now an order in which you have to compile files:
  - If `module_A` `use`s `module_B` which `use`s `module_C`, then:
  - need to compile `module_C` then `module_B` then `module_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

- Not uncommon to need to call a set of functions with the same few arguments
  - For example, may always need `mass` and `velocity` together
- Or we may have some variables that we need to keep in sync
  - For example, the `kinetic_energy` of a particle with its `velocity`

```
ke = kinetic_energy(mass1, velocity1, position1, charge1, E_field)
force = coulomb_force(charge1, charge2, position1, position2)
update_position(position1, mass1, velocity1, force)
```

- We keep passing around the same bundle of information!

Derived types

- One solution to these problems is a *derived type*
  - Other languages call these *structs* or *classes*
- A derived type contains *components* (or *members*) which can be intrinsic types (`real`, `integer`, etc.) or other derived types
- Fortran does have the ability to do some *object-oriented programming* (OOP)
  - Will only cover a little bit here

```fortran
ke = particle1%kinetic_energy(E_field)
call particle1%set_coulomb_force(particle2)
call particle1%push()
```

Derived types – basic syntax

- Declaring a derived type looks like so:

```
type :: <name>
  <type> :: <component name>
  ...
end type <name>
```

- Declaring a variable of that type is done as follows:

```
type(<name>) :: <variable name>
```

- And refer to a component with %:

```
<variable>%<component>
```

Derived types – example

```fortran
type :: velocity_type
  real(real64) :: x, y, z
end type velocity_type

type :: particle_type
  real(real64) :: mass
  type(velocity_type) :: velocity
end type particle_type

type(particle_type) :: proton
```

- Due to Fortran being case insensitive, type names can accidentally clash with variable names
  - Hence my personal preference for `_type` suffix
- Note: `velocity` is probably better off as a 1D array in real code!

Derived types – example continued

- Now only have to pass one parameter to `kinetic_energy`:

```fortran
function kinetic_energy(particle)
  type(particle_type), intent(in) :: particle
  real(real64) :: kinetic_energy

  kinetic_energy = 0.5 * particle%mass &
       * (particle%velocity%x**2 &
          + particle%velocity%y**2 &
          + particle%velocity%z**2)
end function kinetic_energy
```

- Note: we can access components of components
  - But might be a sign the design is wrong
  - Can also use `associate` block to make this easier (not covered here)

Derived type initialisation

- Fortran makes a default *structure constructor* for us
- This initialises all the members in order:

```fortran
type :: particle_type
  real :: mass
  real :: velocity
end type particle_type

type(particle_type) :: proton = particle_type(1., 0.)
! Equivalent to:
proton%mass = 1.
proton%velocity = 0.
```

- Keyword arguments also work here
- Later we will see a way of customising *constructors*

Derived type components default values

- Often useful to give default values to (some) components
- Then when declaring a variable, those components will already be initialised

```fortran
type :: particle_type
  real(real64) :: position = 0._real64
  real(real64) :: velocity = 1._real64
end type particle_type
```

- Now if we make declare a new `particle_type`:

```fortran
type(particle_type) :: particle
```

- We have `particle%position == 0.0` and `particle%velocity == 1.0`

Arrays of derived types

- Fortran's approach to arrays extends to derived types
- Accessing a component on an array gives an array of that component

```fortran
type(particle_type), dimension(3) :: particles

call random_number(particles%position)
call random_number(particles%velocity)

print*, particles%position

particles%position = particles%position &
     + (timestep * particles%velocity)

print*, particles%position
```

- Note: in performance-sensitive parts of code, there may be faster methods!
  - Search: Array of Structures vs Structure of Arrays

Derived type inheritance

- One of the big benefits of derived types and OOP is *inheritance*
- We can *extend* a type and add new components
- The new type *inherits* all the properties of the old type

```fortran
type :: particle_type
  real(real64) :: mass
  real(real64) :: velocity
end type particle_type

type, extends(particle_type) :: charged_particle_type
  real(real64) :: charge
end type charged_particle_type

type(particle_type) :: neutron = particle_type(1._real64, 3._real64)
type(charged_particle_type) :: proton &
     = charged_particle_type(1._real64, 3._real64, 1._real64)

```

Derived type inheritance

- A proton is both a charged_particle_type **and** a particle_type
- Now we can write code that works for all particle_types, plus special code that just deals with charged_particle_type
    - For example, gravitational_force applies to both protons and neutrons, but coulomb_force only applies to protons
- Only need to change one thing:
- Functions that can take a derived type as well as types that extend it, need to use class(<base-type>):

```fortran
    class(particle_type), intent(in) :: particle
    real(real64) :: kinetic_energy

    kinetic_energy = 0.5 * particle%mass &
         * particle%velocity**2
  end function kinetic_energy
end program type_inheritance
```

Derived type polymorphism

- This ability to use the same function to act on different types is called *polymorphism*
  - Polymorphism == "many shapes"
- Polymorphism is one of the four pillars of OOP
  - Along with: abstraction, encapsulation and inheritance
- Possible to use `class(<name>), allocatable` to make a variable whose exact type is determined at runtime
- Won't be covering this in-depth here!

Type-bound procedures

- As well as data members, we can associate procedures with types
    - Procedures can also remain as a *free function*
- These are *type-bound procedures*
    - Called *methods* in other languages
- Three slight annoyances:
    1. The `type` needs to be in a `module` not a `program`
        - Ways round this, but more annoying!
    2. The function definition still needs to be in the `contains` section of the `module`
    3. The list of methods has to be a `contains` section in the `type`

Derived type methods

```fortran
module particle_mod
  use, intrinsic :: iso_fortran_env, only : real64
  implicit none
  type :: particle_type
    real(real64) :: position = 0._real64
    real(real64) :: velocity = 1._real64
  contains
    procedure :: update_position
  end type particle_type
contains
  elemental subroutine update_position(particle, timestep)
    class(particle_type), intrinsic(inout) :: particle
    real(real64), intent(in) :: timestep
    particle%position = particle%position &
         + (timestep * particle%velocity)
  end subroutine update_position
```

Derived type methods

- Can now call the method on our objects:

```fortran
call particles%update_position(timestep)
! Or:
call update_position(particles, timestep)
```

- Because we're calling the method with the `%` syntax, Fortran passes the object as the first argument
  - Very similar to Python's `self`
  - Can use `nopass` attribute to disable this
- We don't know if the type will be `extend`ed, so must use `class` not `type`
- Identical to calling the method and passing the argument ourselves!
- Note `update_position` is `elemental` so can act on the whole array
- We can hide the free function version with `private` – still have access to the method though!

Derived type methods

- Why are methods useful?
- We can rename them!

```
type :: particle_type
  ...
contains
  procedure :: push => update_position
end type particle_type
```

- Now `particle_type%push` doesn't conflict with other procedures called `push`
- If a method is in the base type, we can *override* it in child types

Overriding derived type methods

```fortran
  type :: animal
  contains
    procedure, nopass :: make_noise
  end type animal

  type, extends(animal) :: cat
  contains
    procedure, nopass :: make_noise => meow
  end type cat

  type, extends(animal) :: dog
  contains
    procedure, nopass :: make_noise => bark
  end type dog
```

Overriding derived type methods

- Now if we have something that just takes a `class(animal)`, we can use `make_noise` on it:

```fortran
subroutine speak(creature)
  class(animal), intent(in) :: creature
  call creature%make_noise
end subroutine speak
```

- We can pass in a `cat` or `dog` and get the right noise
- At runtime, Fortran works out exactly which `make_noise` it should call

Derived type member visibility

- Just like `module`s can have declare the visibility of their entities, `type`s can specify visibility of their members
- This is very useful if there's some member that needs to be kept in sync with the state of the type
  - e.g. a particle might store its energy, instead of recalculating it every time
  - Or a container might store how many items it holds
- These are called *invariants*
- Can make the invariant `private` and then provide `public` methods to get or set the value
- Visibility is at the `module` scope

Invariant example

```fortran
type :: particle_type
  private
  real(real64) :: velocity = 0._real64
  real(real64) :: mass = 0._real64
  real(real64) :: kinetic_energy = 0._real64
contains
  procedure, public :: set_velocity
  procedure, public :: set_mass
  procedure, public :: get_kinetic_energy
  procedure :: set_kinetic_energy
end type particle_type
```

- Hiding all the members means we can't use the default constructor
  - Could provide our own, see later
- We provide *setters* and *getters* for the private members
  - Skipped getters for `mass`, `velocity` here for simplicity!

Invariant example

```fortran
subroutine set_velocity(particle, velocity)
  class(particle_type), intent(inout) :: particle
  real(real64), intent(in) :: velocity

  particle%velocity = velocity

  call particle%set_kinetic_energy()
end subroutine set_velocity
```

- Setting either the `mass` or `velocity` instantly updates the `kinetic_energy`
- Setter for `kinetic_energy` is `private`: user of our type can't accidentally change it
- Now all the properties of the particle are always in sync
- Very useful if `kinetic_energy` is used more often than we change `velocity` or `mass` – calculation only done on updates
  - This is called *caching*, useful performance technique

Interfaces

- Early Fortran standards didn't have `contains` or `module`s
- Functions could be external to the program, e.g. in a library
- Return type for functions was `implicit`, i.e. based on first letter of name
- Type and number of arguments was completely unknown to the compiler
    - Programmer had better get them right!
- Very bug prone!

Missing interface example

External library:

```fortran
integer function idouble(x)
  integer, intent(in) :: x
  idouble = 2 * x
end function idouble

integer function iadd(a, b, c)
  integer, intent(in) :: a, b, c
  iadd = a + b + c
end function iadd
```

Program:

```fortran
program use_external_function
  print*, idouble(2.0)
  print*, iadd(2, 4)
end program use_external_function
```

Interfaces

- Fortran 90 introduced `contains`, `module`s and `interface`s
- Interface is essentially the procedure minus the executable statements
- Procedures in `module`s and `program`s have *interfaces* generated by the compiler
    - Also known as *signatures* or *prototypes*
- The interface allows the compiler to give warnings/errors if arguments are wrong
- Programmers could also write an explicit `interface` block for external procedures themselves
    - Useful for legacy libraries or interoperability with other languages

Interfaces

```fortran
  interface
    integer function idouble(x)
      integer, intent(in) :: x
    end function idouble

    integer function iadd(a, b, c)
      integer, intent(in) :: a, b, c
    end function iadd
  end interface

  print*, idouble(2)
  print*, iadd(2, 3, 4)
```

Now compiler knows that `idouble(2.0)` and `iadd(2, 4)` are mistakes!

Passing procedures

- With an interface, we can now pass procedures to other procedures
- Useful for writing e.g. integration routines

Let's take two functions we want to apply:

```
integer function double(x)
  integer, intent(in) :: x
  double = 2 * x
end function double

integer function add_two(x)
  integer, intent(in) :: x
  add_two = x + 2
end function add_two
```

Passing procedures

Using an explicit interface in the function:

```fortran
  integer function apply(f, x)
    interface
      integer function f(y)
        integer, intent(in) :: y
      end function f
    end interface
    integer, intent(in) :: x

    apply = f(x)
  end function apply
```

And calling the function:

```fortran
  print*, apply(double, 42)
  print*, apply(add_two, 6)
```

Passing procedures

- This gets very verbose if we use the same function interface a lot
- Another option is to use an *abstract interface* to name a signature

```fortran
abstract interface
  integer function univariate(x)
    integer, intent(in) :: x
  end function univariate
end interface
```

- We can then declare our dummy argument to be a `procedure` of this type:

```fortran
integer function apply2(f, x)
  integer, intent(in) :: x
  procedure(univariate) :: f

  apply2 = f(x)
end function apply2
```

Generic interfaces

- Very common to want to have the same function for multiple types, or different sets of arguments
  - Called *overloads* in other languages, but not quite the same in Fortran
- Before Fortran 90, needed to have a different name for each function
  - e.g. `idouble` for `double` function on `integer`s, `cdouble` for `complex`
- Only intrinsics could have generic names
- `interface` allows us to "group" functions or subroutines together under a common name
  - Cannot group **both** `function`s and `subroutine`s together though!
- Somewhat annoyingly, can only do this for external procedures or those in `module`s, **not** those in `program`s

Generic interfaces

- For external procedures, need to write out explicit interfaces
- Don't need to repeat interface for modules, can just say

```
interface <generic-name>
  module procedure <specific-name-1>
  module procedure <specific-name-2>
end interface <generic-name>
```

- Can use this to define new *constructors* for derived types, or even operators:

```
interface operator(+)
  module procedure add_my_type
end interface
```

Generic interfaces

In a module:

```fortran
module generic_interfaces
  implicit none

  interface double
    module procedure idouble
    module procedure rdouble
  end interface double
```

Using it in a program:

```fortran
program generic_interfaces_prog
  use generic_interfaces, only : double
  implicit none
  print*, double(2)
  print*, double(2.0)
end program generic_interfaces_prog
```

Generic interfaces

- There are some rules about what can be put in a generic interface
- Mostly, the dummy arguments must be *distinguishable*:
    - Different number of arguments
    - Different types
    - Different kinds
    - Different ranks
    - `allocatable`/`pointer`
    - Procedure vs variable
- Gets a little tricky if names **and** types clash

Optional arguments

- Common to have a function that only sometimes needs a particular argument
- Can make these arguments `optional`
- We've met a few intrinsics with optional arguments
- Need to check that an `optional` argument is `present` before we can use it
    - Except to pass it to other functions
- Require a bit of special handling if we want to give `optional` arguments default values

Optional arguments

```fortran
  subroutine greet(name, politely)
    character(len=*), intent(in) :: name
    logical, intent(in), optional :: politely

    character(len=:), allocatable :: greeting

    greeting = "Hey, "

    if (present(politely)) then
      if (politely) then
        greeting = "Good morning, "
      end if
    end if

    print*, greeting, name
  end subroutine greet
```

Optional arguments

```fortran
  call greet("Peter")
  call greet("Peter", politely=.true.)
```

Also notice that we need to check if `politely` is `present` before trying to use it at all:

```fortran
    if (present(politely)) then
      if (politely) then
```

The following is dangerous, because standard doesn't mandate short-circuiting:

```fortran
if (present(politely) .and. politely) then
  ...
```

Might try to read `politely` first, but it doesn't exist!

Introducing new scope: `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

```fortran
real :: y = 1
print*, y

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

print*, y
```

`stop` and `error stop`

- Sometimes want to finish a program "early"
  - Maybe some quantity has converged
  - Maybe something's gone wrong and we can't continue
- Two statements: `stop` and `error stop`
  - Essentially equivalent, except `error stop` always indicates an error somehow
- Can take an `integer` or `character` constant (i.e. value has to be known exactly at compile time)
  - Relaxed in Fortran 2018!

```
character(len=*), parameter :: converged = "The simulation has converged!"
...
stop converged
```

- Unfortunately, the standard is vague on what happens with an integer
- **Usually** is the exit code, if between 0–127
  - Can't 100% rely on this, but mostly fine

Other useful intrinsics

There are a few intrinsics for dealing with command line arguments

- `command_argument_count()`: Returns the number of command arguments
- `call get_command_argument(number [, value] [, length] [, status])`: Does one of the following, depending on the argument passed:
  - `value`: (`character(len=*)`) get the value of the argument
  - `length`: (`integer`) get the length of the argument
  - `status`: (`integer`) positive if bad `number`, -1 if `value` argument was too short, zero otherwise
- `call get_environment_variable(name [, value] [, length] [, status]    [,trim_name])`: Similar to `get_command_argument` but for environment variables
  - `trim_name`: (`logical`) if false, trailing whitespace in `name` is significant

Documentation

- Documentation is a vital but dull part of programming
- Everybody hates trying to use undocumented code
  - Even worse trying to modify someone else's!
- Comments can be used to document the code inline
- Good practice to put a comment above procedures to explain what it does and **how**
- Comments on tricky bits of code to explain **why**
- Various tools, e.g. Ford: https://github.com/Fortran-FOSS-Programmers/ford
- Usually special comment syntax for bits you want Ford to recognise, e.g. `!!`

Ford documentation example

```fortran
!! Feeds your cats and dogs, if enough food is available. If not enough
!! food is available, some of your pets will get angry.
subroutine feed_pets(cats, dogs, food, angry)
    !! The number of cats to keep track of.
    integer, intent(in)   :: cats
    !! The number of dogs to keep track of.
    integer, intent(in)   :: dogs
    !! The ammount of pet food (in kilograms) which you have on hand.
    real, intent(inout)   :: food
    !! The number of pets angry because they weren't fed.
    integer, intent(out) :: angry
    ...
end subroutine feed_pets
```

Internal procedures

- `function`s and `subroutine`s can `contain` other procedures
- Sometimes useful for longer functions with repeated chunks
- However: these internal procedures can use variables from their parent function
  - See above on scope!
- This makes it easier to make mistakes
- Usually better to make a separate procedure
  - Can make it `private` in a `module`

```
function integrate(y, dy)
  ...
contains
  function runge_kutta_step(y, dy)
    ...
  end function runge_kutta_step
end function integrate
```

Another way to construct 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!

Further reading

- “Modern Fortran Explained: Incorportating Fortran 2018”, Metcalf, Reid, Cohen (2018) OUP
- “Fortran for Scientists and Engineers”, Chapman (2018) McGraw-Hill Education
- “Guide to Fortran 2008 Programming”, Brainerd (2015), Springer
- Fortran wiki: http://fortranwiki.org
    - Lots of resources linked from there!
- `gfortran` documentation: https://gcc.gnu.org/onlinedocs/gcc-10.2.0/gfortran/
- Intel Fortran language reference: https://software.intel.com/content/www/us/en/develop/documentation/fortran-compiler-developer-guide-and-reference/top/language-reference.html