Fortran 2008

Introduction: Syntax, loops, conditionals & I/O

Fiona Reid
f.reid@epcc.ed.ac.uk

Lecture 1 - overview

- Plan for this lecture
  - Introduction to Fortran 2008
  - Evolution of the language
  - Syntax, loops, conditionals, I/O

Language evolution

- Ancient History
  - Name comes from **FOR**mula **TRAN**slation
  - Fortran 66 was the first language to have a standard (1967)
- Fortran 77
  - New standard to overcome divergence in different implementations (1978)
- Fortran 90
  - Major revision - a new and vastly improved language emerged
  - Added modules, derived data types, dynamic memory allocation, intrinsics
  - But retained backward compatibility!
- Fortran 95
  - Minor revision but added several HPC related features; **`forall`**, **`where`**, **`pure`**, **`elemental`**, pointers
- Fortran 2003
  - Major revision with many new features including; OO capabilities, procedure pointers, IEEE arithmetic, C interoperability,
- Fortran 2008
  - Minor change: added co-arrays & sub modules

Key strengths

- Mathematical / Scientific applications
  - Formula translation
  - Closely related to the need for mathematical operations
  - e.g., intrinsic **`complex`** data type
  - Many numerical resources / libraries available
- Array handling
  - Problems typically exist on grids or lattices
  - Naturally described in terms of multi-dimensional arrays
  - Fortran 2008 has powerful array syntax / operations
- Highly optimisable
  - Compiler should make code as fast as possible
  - e.g. use of pointers is restricted
  - Natural language for HPC applications

Weaknesses

- Things one would not do …
  - Interact with the operating system (use e.g., C)
  - Build GUIs (use e.g., Java)
  - Handle strings (use, e.g., Perl)
- Backward compatibility
  - Huge amount of legacy code
  - Language has grown somewhat large
- However …
  - Used in the right context, it is easy and very powerful

Software engineering

- Fortran 90 introduced new features
  - Structured, sane, safe programming!
- Modules
  - Provide excellent possibilities for encapsulation
  - Provide interfaces for subroutines (argument type-checking)
  - Provide structure
- Portability
  - Concept of "type" for data objects
  - Opens the way to obtaining portable behaviour, particularly for floating point arithmetic
- Subsequent incarnations (95, 2003, 2008) have built on this
  - Result is a modern language that is very good for HPC applications

Hello World

- The canonical introductory program

```
program hello
  ! Display a message to standard output    (usually the screen)
  implicit none
  write (unit = *, fmt = *) "Hello World!"
end program hello
```

- Basic syntax is based on lines
  - Statements occupy lines of up to 132 characters
  - **Case insensitive** (c.f. C, C++, Java)
  - Comments are introduced with an exclamation mark `!`
- You will see many variations in style

Main program and syntax

- Formally main program

```
[program program-name]
  [specification-statements]
  [executable-statements]
end [program [program-name]]
```

- Text inside square brackets `[]` is optional
- Long lines can be split using continuation `&`

```
write (unit = *, fmt = *) &
"Long and somewhat convoluted Hello World line!"
```

- Multiple statements on a single line
  - Can be split using a semi-colon `;`
  - Not recommended for readability – use one statement per line

Variables

- Intrinsic data types are declared

```
implicit none          ! Enforce strong typing
integer                :: i        ! 10
real                   :: a        ! 3.14159
character              :: letter   ! a
character (len = 12)   :: month    ! January
logical                :: switch   ! .false.
complex                :: z0, z1   ! (1.0, 1.0)
```

- Variables
  - Must be declared *before* any executable statements
  - Have an acceptable name made up of alphanumeric characters (or underscores _) of which the first character must be a letter
  - Acceptable: **a1, a_letter, a123b**
  - Not acceptable: **1abc, quid$in**

Implicit None

- Undeclared variables always have an implicit type
  - If the first letter begins with an i, j, k, l, n, m type is **integer**
  - If the first letter begins with any other letter type is **real**
- Implicit typing is very dangerous and should always be turned off using **implicit none**
  - Consider the following

```
real :: l1 = 1.2345
write(*,*)"The value of l1 = ", ll
```

  - The variable **ll** is implicitly assumed to be of integer type
  - The compiler will not complain
  - Using **implicit none** would catch this typographical error
  - Can be *very* difficult to debug

Arrays

- Arrays hold a collection of values at the same time
- Elements are accessed by *subscripting* the array
  - A 10 element 1D array can be visualised as:

1	2	3	4	5	6	7	8	9	10

  - A 4x2, 2D array can be visualised as:

Dimension 2

Dimension 1

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

- In Fortran arrays are stored in memory by *columns* – known as column major (C, C++, Java all store by row)

Arrays

- Arrays are declared with dimension attribute

```
implicit none
integer, dimension(4) :: n4
```

- Provides 4 elements
  - Elements: `n4(1), n4(2), n4(3), n4(4)`
  - First element is, by default, 1
- Can set the *lower* and *upper bounds*

```
real, dimension(-5:4) :: r
```

  - Elements: `r(-5), r(-4),` … `r(0),` … `r(4)`
  - Total number of elements in the array is the *size*
  - Here n4 has size = 4 and r has size = 10

Multidimensional arrays

- Arrays can have more than one dimension

```
complex, dimension(1:10, 1:20) :: z
```

- Terminology
  - Number of dimensions is the *rank* (here 2)
  - Number of elements in given dimension is the *extent*
  - Sequence of the extents is the *shape,* here (10, 20)
- Up to 7 dimensions are allowed

```
real, dimension(2, 3, 4, 5, 6, 1) :: vast
```

  - Has six dimensions (i.e., rank 6)
  - Extent in the fourth dimension is 5
  - Shape is (2, 3, 4, 5, 6, 1)
  - Size is 2x3x4x5x6x1 = 720 elements

More on character variables

- Declared in similar way to numeric types
- Character variables can
  - Refer to a single character
  - Refer to a string (achieved by adding a length specifier)
- The following are all valid declarations

```
character             :: sex
character (len = 20)  :: name
character (len = 10), dimension(10,10) :: carray
```

- Assigned using either double ”” or single quotes ’’

```
sex = ’f’
name = ”Joe Bloggs”
```

Parameter attribute

- Named constants may be defined and used

```
integer, parameter   :: n = 100
real, dimension(2*n) :: r
real, parameter      :: pi = 3.14
```

- Values set at compile time must not change
  – Constant expressions involving parameters are evaluated at compile time
  – Attempt to assign a new value will give a compiler error
  – Any intrinsic type may have the parameter attribute, including arrays
- The general declaration is

```
type [, attributes] :: variable
```


Variable initialisation

- Variables can be initialised either at point of declaration

```
program initial_declare
  implicit none
  integer :: i = 10
  real    :: pi = 3.14159
  character (len = 12) :: month = "January"
end program initial_declare
```

- Or within the main program

```
complex    :: ci
logical    :: iostatus
ci = (0.0, 1.0)
iostatus = .true.
```

- Beware: initialising arrays at declaration can result in very large executable sizes (intialised at compile time)

Concept of type

- Floating point variables
  - Variables declared real are of default precision
  - Standard does not specify what this is (but usually 4 bytes)
- Mechanism for ensuring get desired type
  - E.g., by specifying the range or decimal precision required
  - Uses the kind type parameter (processor dependent)

```
integer, parameter :: sp = kind(1.0)
real (kind = sp), dimension(10) :: variable
```

- Extended precision (double)

```
integer, parameter :: dp = kind(1.0d0)
real (kind = dp)    :: variable
```


Numerical expressions

- Arithmetic operators are

```
**      ! exponentiation
*       ! multiplication
/       ! division
+       ! addition
-       ! subtraction
```

  - decreasing order of precedence
- Otherwise expressions evaluated left-to-right
  - e.g., `a*b*c` evaluated as `(a*b)*c`
  - Except `a**b**c` evaluated as `a**(b**c)`
- Care! Integer division rounded toward zero
  - e.g., `(2*4)/5` gives 1 but `2*(4/5)` gives 0
- Type promotion during arithmetic
  - Promotes to higher type, e.g. `integer * real = real`

Mixed assignments

- Promotion during arithmetic **(+ - * /)**
  - Expression **a** ***operator*** ***b*** is evaluated as

type of a	type of b	type of result
integer	integer	integer
integer	real	real
integer	complex	complex
real	real	real
real	complex	complex
complex	complex	complex

- Explicit conversions are also possible
  - Intrinsic functions **int(), real(), cmplx()**
  - e.g., **z = cmplx(r1,r2)**, where **r1** and **r2** are variables of type real containing the real and imaginary parts of the complex number respectively

Intrinsic functions

- Over 100 intrinsic functions in Fortran 2008
  - array operations, bit manipulations, character strings
  - check whether there's an intrinsic available (List of intrinsic functions in Metcalf and Reid or the Standard)
- Conversion

  int() real() cmplx() abs() nint() aint() aimag() ceiling() floor()
- Mathematical

  sqrt(x) exp(x) log(x) log10(x)
  sin(x) cos(x) tan(x) asin(x) acos(x) atan(x) sinh(x) cosh(x) tanh(x)
- Others

  min(x1, x2, ...) max(x1, x2, ...) mod(a, p)
  conjg() tiny(x) huge(x)

Relational operators

- These are

```
<       ! less than
<=      ! less than or equal
>       ! greater than
>=      ! greater than or equal
==      ! equal
/=      ! not equal
```

- Logical expressions are then, e.g.,

```
a < b
char1 == "a"
a+b >= c+d
```

- For integer and real numeric types
  - Not complex

Logical operators

- Logical variables take on one of two values

```
.true.
.false.
```

- Relational operators are

```
.not.          ! unary not
.and.          ! logical and
.or.           ! logical or
.eqv.          ! equivalent
.neqv.         ! not equivalent
```

- Decreasing order of precedence
  - e.g., `i .or. j .and. .not. k` evaluated as `i .or. (j .and. (.not. k))`

Conditionals

- Very similar to other languages

```
if (logical-expression) then
  block
[else if (logical-expression) then
  block]...
[else
  block]
end if
```

- May be nested
  - but not interleaved
- Also a select case statement (cf switch in Java)

Select case

- Select case provides an alternative to a series of repeated **`if...then...else if`** statements
- The general form of the case construct is

```
[name:] select case (expression)
     [case selector [name]
       block]...
     [case default
       block]
end select [name]
```

- Where **`expression`** can be any of
  - A single integer, character, or logical depending on type
  - min: any value from a minimum value upwards
  - :max any value from a maximum value downwards
  - min : : max any value between the two limits

Iteration

- Bounded iteration

```
do n = 1, 100
  ! do something
end do
```

- Formally

```
do [variable = expr1, expr2[, expr3]]
  block
end do
```

  – where ***expr1***, ***expr2***, and ***expr3*** are integers
  – number of iterations will be `max(0, (expr2-expr1+expr3)/expr3)`

- Arbitrary stride is allowed (including negative stride)

```
do n = 10, 1, -2
  !  do something
end do
```


Controlling iteration

- Consider the unbounded loop

```
do
  ! go around for ever
end do
```

- This can be terminated

```
do
  ! do some computation
  if (condition) exit  ! exits from current loop
  ! do something else
end do
```

- Can also go to next iteration using **`cycle`**

Simple I/O

- The **print** statement is the simplest form of directing unformatted data to the standard output

```
print*,"The temperature is ",temperature," degrees"
```

  - Each print statement begins on a new line
  - Print statement can transfer any object of intrinsic type to standard output
  - Strings are delimited by either double " " or single ' ' quotes
  - Two occurrences of string delimiter produce one occurrence in the output, e.g. `print*,"Fred says ""Hello!"""`
- `print` only allows access to standard output – screen
- `write()` is much more useful as it can also handle files

Simple I/O – write statement

- Use `write()` statement

```
write ([unit =] unit, &
       [fmt =] format_string ...) [list]
```

  - can take default `write (*,*)`
  - i.e., standard output and free format
- To write to an external file

```
open (unit = 20, file = "file.dat", &
      form = "formatted", action = "write")
write (unit = 20, fmt  = *) [list]
close (unit = 20, status = "keep")
```

- Input is via `read()`
  - e.g. `read(*,*)temperature` to read the value of `temperature` from the keyboard

Summary

- Fortran is an evolving language
  - Now has many powerful features
  - Natural language for scientific / engineering problems
  - Hence commonly found in HPC applications
  - Vast amount of legacy code
- This lecture we have looked at
  - Basic syntax, loops, conditionals & I/O
- The next two lectures will look at how to structure code with Fortran 2008 and array syntax

References

- References
  - Metcalf and Reid "Fortran 90/95 Explained" (2nd Ed.), ISBN 0198505582
  - Metcalf, Reid and Cohen "Fortran 95/2003 Explained", ISBN 0198526938
  - T.M.R. Ellis et-al "Fortran 90 Programming", ISBN 0201544466
- Standards can be found at: http://www.nag.co.uk/sc22wg5/
- Wiki page: http://fortranwiki.org/fortran/show/HomePage

Fortran 2008

Extra slides: Examples of conditionals, loop control, select case, file I/O

Fiona Reid
f.reid@epcc.ed.ac.uk

Conditionals (example)

- For example

```
if (t < 0) then
  ! It’s cold
  ice = .true.
else if (t > 100) then
  ! It’s hot
  steam = .true.
else
  water = .true.
  wet = .true.
  washout = .true.
end if
```

Select case - example

- General form of **selector** is a list of non-overlapping values/ ranges of the same type as **expression**
- Values of **expression** not included in **selector** can be caught by **case default**, e.g.

```
seasons: select case (month)          ! month is of type integer
  case (1:2,12)                       ! Winter, Dec, Jan, Feb
    write(*,*)"It is winter"
  case(3:5)                           ! Spring, Mar, Apr, May
    write(*,*)"It is spring"
  case(6:8)                           ! Summer, Jun, Jul, Aug
    write(*,*)"It is summer"
  case(9:11)                          ! Autumn, Sep, Oct, Nov
    write(*,*)"It is autumn"
 case default                         ! if month outside 1-12
    write(*,*)"Must enter 1-12"
 end select seasons
```


Controlling iteration - example

```
mainloop: do
    write(*,*)"Input student id"
    read(*,*)stid
    if (stid == 0) exit mainloop
    average = 0
    innerloop: do i = 1, 5
        write(*,*)"Please enter mark"
        read(*,*)mark
        if (mark < 0) then
            write(*,*)"Mark < 0, start again"
            cycle mainloop
        end if
        average = average + mark
    end do innerloop
    average = average/5.0
    write(*,*)"Average of student",stid," is = ",average
end do mainloop
```

Simple I/O – write statement

- Can use write and read statements to access standard input (i.e. screen and keyboard)

```
write(*,*)"This text will appear on the screen"
write(*,*)"Input temperature (C)"
read(*,*)temperature  ! Reads value input via
                            ! the keyboard and assigns
                            ! to variable temperature
```

- Multiple values can be read in from a single line

```
write(*,*)"Input 3 results"
read(*,*)result1,result2,result3
```


Simple I/O – unknown file length

- To continue reading values from an external file until the end of the file is reached

```
integer :: i, icount = 0
integer, parameter :: maxln=500
real, dimension(maxln) :: a
open(unit=10, file="temps.dat", status="old", action="read")
do i = 1, maxln
  read(10,*,end=100)a(i)
  icount = icount + 1
end do
100 continue                ! 100 is a label
close(10)
write(*,*)"No. of lines read in from file =",icount
. . .
```