Welcome
Modern Fortran (F77 to F90 and beyond)
Virtual tutorial starts at 15.00 BST
Modern Fortran: F77 to F90 and beyond

Adrian Jackson
adrianj@epcc.ed.ac.uk
@adrianjhpc
Fortran

- Ancient History (1967)
  - Name comes from FORmula TRANslation
  - Fortran 66 was the first language to have a standard
- Fortran 77 (1978)
  - New standard to overcome divergence in different implementations
- Fortran 90 (1991)
  - Major revision – much improved programmability
  - Added modules, derived data types, dynamic memory allocation, intrinsics
  - But retained backward compatibility!
- Fortran 95 (1997)
  - Minor revision but added several HPC related features; `forall, where, pure, elemental, pointers`
  - Major revision: OO capabilities, procedure pointers, IEEE arithmetic, C interoperability,
- Fortran 2008 (2010)
  - Minor change: co-arrays and sub modules
- Fortran 2015 (2018?)
  - Minor revision: planned improvements in interoperability between Fortran and C, parallel features, etc..
F90 text/character changes

• Names (variables, program units, labels) maximum size increased:
  • Up to 31 characters, only 6 character in F77
• Comments start with !
  • Also allows inline comments: i.e. \texttt{a = b + c ! My sum}
  • F77: \texttt{c} or \texttt{C} in column 1
• Free-format
  • Up to 132 character lines
  • No specification about where on a line characters are
  • Spaces not allowed inside constants or variable names
  
    \begin{verbatim}
    fred = 1 00 42
    fred = 10042
    \end{verbatim}
  • Continuation of lines done using \& at end of line
    
    \begin{verbatim}
    a = b + \&
    c ! My sum
    \end{verbatim}
  • Breaking character strings requires \& at end of line and the beginning of the next line
    
    \begin{verbatim}
    mystring = ‘hello\&
    & and welcome’
    \end{verbatim}
  • Can use \texttt{“} and \texttt{’} for character strings (allows “you’re an idiot” type strings)
Typing

- **IMPLICIT NONE**
  - Instruct the compiler to disable implicit typing for a program unit
  - Implemented in most F77 compilers prior to F90
  - Required for main program, subroutines/functions (unless in contains), and modules

- **New variable definition format**
  - :: used to separate attributes from variable names
    ```
    integer, parameter :: bob = 6
    ```
  - rather than
    ```
    integer bob
    parameter (bob=6)
    ```
Typing

- intents to provide compiler checking and optimisation options
  - `intent(in)`: Variable data will be used inside the routine but not modified
  - `intent(out)`: Variable will be modified in the routine but the initial value will not be used
  - `intent(inout)`: Variable initial data required and will be modified in the routine
Modules

- Constants, variables, and procedures can be encapsulated in modules for use in one or more programs.
- A module is a collection of variables and procedures
  ```
  module sort
    implicit none
    ! variable specifications
    ...
  contains
    ! procedure specifications
    subroutine sort_sub1()
    ...
    end subroutine sort_sub1
    ...
  end module sort
  ```
- Variables declared above `contains` are in scope
  - Everywhere in the module itself
  - Can also be made available by `using` the module
Points about modules

- Within a module, functions and subroutines are known as module procedures
- Module procedures can contain internal procedures
- Module objects can be given the `SAVE` attribute
- Modules can be `USEd` by procedures and modules
- Modules must be compiled before the program unit which uses them
  - This can complicate your build process
  - Some use scripts or small applications to work out the correct compile order
Using modules

- Contents of a module are made available with `use`:

```fortran
PROGRAM TriangUser
  USE Triangle_Operations
  IMPLICIT NONE
  REAL :: a, b, c
  The `use` statement(s) should go directly after the program statement
  `implicit none` should go directly after `after` any use statements

- There are important benefits
  - Procedures contained within modules have explicit interfaces
  - Number and type of the arguments is checked at compile time
  - Not the case for external procedures
  - Can implement data hiding or encapsulation
    - via `public` and `private` statements and attributes
Derived data types

- F90 allows the use of derived data types
  - Groups of data structures
  - Enables building of more sophisticated types than the intrinsic ones, i.e. linked data structures, lists, trees etc…

- Imagine we wish to specify objects representing persons
  - Each person is uniquely distinguished by a name and room number
  - We can define a corresponding “person” data type as follows:

```fortran
  type person
    character (len=10) :: name
    integer :: officeNumber
  end type person
```


Derived data types

• To create a derived type variable you use the syntax:
  type(person) :: fred, me
• Initialisation (construction) possible as well:
  fred = person(“Fred Jones”, 21)
• fred is a variable containing 2 elements: name, officeNumber
• Elements (individual components) of derived type can be accessed by component selector: %
  fred$name ! contains the name of you
  fred$officeNumber ! contains the age of you
Operators

- Comparison operators:
  - New characters for operators, either can be used, can be mixed

  - `.lt. => <`  ! less than
  - `.le. => <=`  ! less than or equal
  - `.gt. => >`  ! greater than
  - `.ge. => >=`  ! greater than or equal
  - `.eq. => ==`  ! equal
  - `.ne. => /=`  ! not equal

- Logical variables should be compared with

  - `.eqv.`
  - `.neqv.`
Operator overloading

• Using interfaces it is possible to overload operators (or define your own operators) as well:

```fortran
implicit none

interface operator(+)
    module procedure real_sum, int_sum
end interface
```

• Only really makes sense if you define your own operators or datatypes
  • Can’t override existing definitions (the above example isn’t actually allowed)
Loops

• do loop terminated by end do
  do i=1,10
    x = x + y
  end do
• rather than
  do 10 i=1,10
10  x = x + y
• cycle keyword will skip a loop iteration
  do i=1,10
    if(i .eq. 5) cycle
    x = x + y
  end do
• exit keyword will finish the loop
  do i=1,10
    x = x + y
    if(x>100) exit
  end do
Dynamic memory

- Dynamic memory supported by allocatable attribute, allocate, deallocate and allocated routines
  - Automatically deallocated when out of scope unless saved

```fortran
real, allocatable :: charles(:,:)
integer :: myerror
...
allocate(charles(1000,10))
...
if(.not. allocated(charles)) then
  allocate(charles(1000,10), stat=myerror)
  if(myerror /= 0) stop
end if
...
deallocate(charles)
```
Portable precision

• F77 defined variable precision by specify the number of bytes data stored in:
  integer*4, real*8
• F90 introduces more control, can specify required variable range
  • SELECTED_INT_KIND: define the minimum number of decimal digits required
  • SELECTED_REAL_KIND: define minimum number of decimal digits and exponent range

INTEGER, PARAMETER :: large_int = SELECTED_INT_KIND(9)
INTEGER(KIND=large_int) :: i
• large_int is non-negative if the desired range of integer values, $-10^9 < n < 10^9$ can be achieved
Portable precision

INTEGER, PARAMETER :: small_real = SELECTED_REAL_KIND(6,37)

• \texttt{small\_real} is non-negative if the desired exponent range of real values, \(-10^{37} < n < 10^{37}\) can be achieved, and the desired number of decimal digits, .000001, can be achieved

• \texttt{selected\_real\_kind} returns:
  • -1 if the precision cannot be achieved
  • -2 if the range cannot be achieved

REAL(KIND=small_real) :: x
real(small_real), allocatable :: my_data(:,:)

• Constants can be specified with a kind type (like 7.d0)
INTEGER(KIND=large_int) :: I = 7\_large\_int
REAL(KIND=small_real) :: x = 5.0\_small\_real
Array operations

- Fortran can operate on whole arrays
  - whole or subsections
  
a = 0.0 ! scalar conforms to any shape
  
b = c + d ! b, c, d must be conformable
  
e = \sin(f) + \cos(g) ! and so must e, f, g

- Subsection selection:
  - REAL, DIMENSION(1:15) :: A
  - A(:, ) whole array
  - A(m:, ) elements m to 15 inclusive
  - A(:, n) elements 1 to n inclusive
  - A(m:n) elements m to n inclusive
  - A(:, :2) elements 1 to 15 in steps of 2
  - A(m:m) 1 element section of rank 1
  - WHERE (P > 0.0) P = log(P)
Array operations

• Range of array intrinsics

WHERE (P > 0.0) P = \log(P)
WHERE (P > 0.0)
  \begin{align*}
  X &= X + \log(P) \\
  Y &= Y - 1.0/P
  \end{align*}
END WHERE
nonnegP = \text{COUNT}(P > 0.0)
sumP = \text{SUM}(P)
P = \text{MOD}(P,2)
Advice for moving to F90 from F77

• Text changes required
  • Comments \( \rightarrow \) !
  • Continuation lines & at column 6 becomes & at end of the line

• Implicit none
  • Make sure typing is explicit
  • If code uses implicit typing this require lots of variable declarations
  • Rename variables if you are declaring them for the first time
  • Use kind parameters if you are declaring new variables

• Use modules
  • Split code into sensible groupings
  • Convert groupings into modules
  • Use those modules where required
  • Common blocks to modules
  • Files to modules
Advice for moving to F90 from F77

• Using modules
  • Make module private by default
  • Only use the components you require
• Convert do loops
• Rename variables
• Declare variables using module defined kind parameters
  • Enables easy change of precision if required
• Move to dynamic allocation from static
• Consider array syntax for new code development
Interoperable code F77 and F90

• Occasionally it’s necessary to have code that works in both F77 and F90
  • i.e. include file for library
• Can be done by including continuation characters in correct place
  • & at the end of the line but after the 72 character (F90)
  • & at the beginning of the line in column 6 (F77)
• No inline comments
• Comment character ! in 1st column
  • Not strictly F77 compliant but compilers will generally accept this
Newer features

• C interoperability
  • New module ISO_C_BINDING
    • Has the kind types for C intrinsic variables
    • Defined types and structures can be inter-operable:
      TYPE, BIND(C) :: matrix
      ....
      END TYPE matrix
    • Some restrictions on what can be in the types or structures
    • Same can be done for procedures with defined interfaces
Newer features

- Object oriented programming
  - Modules and derived types can be used to make “semi-classes”
    - Encapsulation of data and functions with modules
    - Controlled access to data or functions with private and public keywords
    - Polymorphism with interfaces
    - Operator overloading with interfaces
  - F2003 introduces further OO functionality
  - Type bound procedures

```fortran
module building
  implicit none
  integer, parameter :: MAXLEN = 100
  type person
    character(MAXLEN) :: name
    integer :: officeNumber
  contains
    procedure, nopass :: getName
    procedure :: setName
    procedure :: getOfficeNumber
    procedure :: setOfficeNumber
  end type person
end module building
```
Newer features

• Class variable passed to type bound procedures
  • Allows polymorphic procedures
• Type bound procedures must take a class variable
  • Variable name is not prescribed (self is not a keyword)
  • Automatically passed
  • Allows for data polymorphism

```fortran
function getName(self)
class(person), intent(inout):: self
class_variable character(MAXLEN) :: getName
  getName = self%name
end subroutine
```

• Allowed unlimited polymorphic type
  class(*)
• Can define abstract classes, extend classes, overload procedures, etc…
Newer features

• Pointers
  • Alias to variables

• Co-arrays
  • Parallel programming using partitioned global address space (PGAS) approach

• Recursive procedure support

• select…case… functionality
Goodbye

Virtual tutorial has finished
Please check here for future tutorials and training

http://www.archer.ac.uk/training
http://www.archer.ac.uk/training/virtual/