Welcome

Modern Fortran

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

Adrian Jackson
a.jackson@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 revision: co-arrays and sub modules
  - 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 \times
    fred = 10042 \checkmark
\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 ‘‘ and ‘’ 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
  • \texttt{intent(in)}: Variable data will be used inside the routine but not modified
  • \texttt{intent(out)}: Variable will be modified in the routine but the initial value will not be used
  • \texttt{intent(inout)}: Variable initial data required and will be modified in the routine
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)

- `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, $0.000001$, can be achieved

- `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)
  X = X + log(P)
  Y = Y - 1.0/P
END WHERE
nonnegP = COUNT(P > 0.0)
sumP = SUM(P)
P = MOD(P,2)
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

```plaintext
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 TriangleUser
  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 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:
  \[
  \text{type(person) :: fred, me}
  \]
- Initialisation (construction) possible as well:
  \[
  \text{fred = person("Fred Jones", 21)}
  \]
- \text{fred} is a variable containing 2 elements: \text{name}, \text{officeNumber}
- Elements (individual components) of derived type can be accessed by component selector: \%
  \[
  \text{fred\%name} \quad \text{! contains the name of you}
  \text{fred\%officeNumber} \quad \text{! contains the age of you}
  \]
Supertypes

• Derived type can be used in other derived types:

```fortran
  type corridor
      type(person), dimension(:), allocatable :: rooms(:)
      integer :: numberOfRooms
  end type corridor

  type(corridor) :: a1
  ...
  a1%rooms(1)%name
  a1%numberOfRooms = 10
```
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:

```plaintext
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)
Psuedo OO programming with F90

- Modules and interfaces allow semi-OO programming
  - 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
- Does not provide full OO functionality but can be very powerful
  - Often enough functionality with this without using the F2003 additions
Advice for moving to F90 from F77

• Text changes required
  • Comments `c` -> `!`
  • 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
  character(MAXLEN) :: getName
  getName = self%name
end subroutine
```

- Allowed unlimited polymorphic type
  - class(*)
- Can define abstract classes, extend classes, overload procedures, etc…
Class variables

- If allocatable
  - Either type needs specified:
    ```
    class(*), allocatable :: fred
    allocate(person::fred)
    ```
  - Or source type needs specified:
    ```
    person :: bob
    class(*), allocatable :: fred
    allocate(fred, source=bob)
    ```
    - In this case the allocation is made and the values copies into the new object
**Type guarding**

- Type inquiry/type guarding is possible
- type is
  - Type of object is the specified type
- class is
  - Class of the object is the same as the specified class or an extension of that class

```plaintext
select type (bob)
type is (manager)
  print *, 'This is a manager'
class is (person)
  print *, 'This could be a manager or person'
class default
  print *, 'Unknown type used'
end select
```
Type comparison functions

• Two new intrinsic functions to inquire about types:

\textsc{extends\_type\_of} (X, Y)

• Returns true if the type of X is the same as, or extends the type of Y
• Some subtleties if Y is unallocated unlimited polymorphic type

\textsc{same\_type\_as} (X, Y)

• Returns true if the type of Y is the same as the type of X
Overloading in F2003

- The `generic` keyword specifies polymorphism for type-bound procedures.
- Polymorphism without interface block.
- Without this, type-bound procedures only resolve to a single method.

```f90
GENERIC [, access-spec ] :: generic-spec => binding-name1 [, binding-name2]...

type maths_functions
contains
  procedure real_sum
  procedure int_sum
  generic :: my_sum => real_sum, int_sum
end type
```
Overloading pre F2003

- Generic interfaces can enable procedure overloading, but are not bound to a specific type:

```fortran
module maths_functions
  interface my_sum
    module procedure real_sum
    module procedure int_sum
  end interface
contains
function real_sum (a, b)
  implicit none
  real, intent(in) :: a,b
  real_sum = a + b
end function real_sum

function int_sum (a, b)
  implicit none
  integer, intent(in) :: a,b
  int_sum = a + b
end function int_sum
end module
```
Class constructor

• Can specify a constructor
  • Using interface with same name as the derived type

... public :: person
type person
  character(MAXLEN) :: name
  integer :: officeNumber
contains
  procedure, public :: getName
  procedure, public :: setName
  procedure, public :: getOfficeNumber
  procedure, public :: setOfficeNumber
end type person
interface person
  module procedure initialise_person
end interface

• Can be overloaded
• Not mandatory
Class destructor

- **final** keyword can be used to define procedure(s) to be called on object destruction

```fortran
public :: person

  type person
    character(MAXLEN) :: name
    integer :: officeNumber
  contains
    procedure, public :: getName
    procedure, public :: setName
    procedure, public :: getOfficeNumber
    procedure, public :: setOfficeNumber
    final :: cleanUp
  end type person

interface person
  module procedure initialise_person
end interface
```
Abstract classes

- Can define **abstract classes and deferred procedures**
  - Define data
  - Define procedures and interfaces
  - Define implement procedures
  - Define procedures to be implement by further classes
- Abstract class cannot be instantiated or allocated
  - Can be used for class declaration in methods
    - Important for type hierarchies
Newer features

• Pointers
  • Alias to variables

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

• Recursive procedure support

• select…case… functionality
Summary

• Moving from F77 to F90+ is beneficial both in terms of functionality and easy of correctness checking/programmability etc…
• Well engineered programs can be written in F90/95 that satisfy most requirements for scientific programming
• F2003 and beyond provides a wider range of functionality that may be useful, particularly for re-use
• Performance does not have to be significantly impacted by such functionality, depending on the compiler
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/