Introduction to Modern Fortran

Dr. Axel Kohlmeyer

Associate Dean for High-Performance Computing
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu

Master in High Performance Computing

Overview

- Fortran is a general purpose, imperative programming with focus on numeric computing
- Developed at IBM in the 1950s for use with mainframe computers
- Widely used in High-Performance and Scientific Computing due to large number of intrinsics and portability via use of a runtime library
- Popular standards Fortran 77, Fortran 90/95 and Fortran 2003/2008 each with extensive support for backward compatibility to standards

Fortran Coding Styles

- Modern Fortran compilers support most (new) features from the Fortran 77 standard upward
- Fortran 90 introduced many new features and a new formatting style. Not widely adopted until the mid-2000s due to limited compiler support
- A lot of legacy code remains in part Fortran 77
- We will focus on using modern Fortran format and styles, with focus on commonly used features and code constructs, most of which are already present in many Fortran 77 style codes

Historic Fortran Program Format (AKA ‘Fixed Format’)

```
c        1         2         3         4         5         6         7
c234567890123456789012345678901234567890123456789012345678901234567890123456789012
      SUBROUTINE DSCAL(N,DA,DX,INCX)
c
      DOUBLE PRECISION DA
      INTEGER INCX,N
      DOUBLE PRECISION DX(*)
      INTEGER I,M,MP1,NINCX
      INTRINSIC MOD

      IF (N.LE.0 .OR. INCX.LE.0) RETURN
      IF (INCX.EQ.1) THEN
         M = MOD(N,5)
```

Col 1: c,C,d,D,!,*: line is a comment, ignored

Col 1-5: Labels

Col 6: Continuation

Col 7-72: Program text

Col 73-80: Ignored

Modern Fortran Format (AKA ‘Free Format’)

- One instruction per line (lines **truncated** to 132 chars)
- Longer instructions have to use a continuation character ‘&’ as the last character
- Code is case insensitive (Only strings are not)
- Strings can use single or double quotation marks
- A popular convention is to write Fortran keywords upper case and everything else lower case (emacs)
- Symbols (variable/function names) must start with a character and can use numbers and ‘_’
- Comments start with the ‘!’ character

Compilation and Filename Conventions

- Fortran code needs to be compiled with a compiler, e.g. gfortran, f95, ifort, flang, pgf95
- The filename extension usually serves as a hint to the compiler what source file style to expect:
  - .f .F => fixed format Fortran 77 style
  - .f90 .F90 => free format Fortran 90/95/03/08 style
  - Uppercase extension => use C-style preprocessor
  - Example:
    `gfortran -Wall -O -o hello.exe hello.f90`
- For details consult compiler documentation

Basic Data Types

- Data types **may** have implementation specific storage size, but most common is 4-byte.
- INTEGER: signed integer, 4-byte → 32-bit
- REAL: floating point, 4-byte → single precision
- (DOUBLE PRECISION): obsolete (2x REAL)
- COMPLEX: pair of two REAL numbers
- CHARACTER: for storing strings
- LOGICAL: value either `.TRUE.` or `.FALSE.`

Different ‘Kinds’ of Basic Data Types

- Fortran can use different storage sizes for data types; these are referred to a “kind” and represented by the number of bytes it uses
- Intrinsic functions SELECTED_INT_KIND() and SELECTED_REAL_KIND() are used to determine the number of bytes needed
- Default INTEGER is usually INTEGER(KIND=4) Old syntax: INTEGER*4
- Default REAL is typically REAL(KIND=4), DOUBLE PRECISION is then REAL(KIND=8)

Literals

- INTEGER: numbers <u>without</u> a decimal point; define ‘kind’ of integer with appending ‘_<kind>’
Example: 120_1 (this is invalid: 200_1)
- REAL: numbers <u>with</u> a decimal point, optionally with an exponent. Kind is default kind (i.e. 4) unless set or exponent is using ‘d’ instead of ‘e’
Examples: 29. 61495.209 2.5e50_8 1.d100
(these are **invalid**: 2.5e50, 1.0d10_4)
- COMPLEX: pair of real numbers with the same conditions as for REAL (1.0_8,0.0_8)

Program Blocks

- Every Fortran program must have exactly one ‘PROGRAM’ block. This is the entry point of the program (same as main() in C/C++ programs)
- ‘PROGRAM’ blocks must have a name and are terminated with an ‘END’ statement
- Example:

```
! Minimal Fortran program example
PROGRAM hello
   PRINT*,’Hello, World!’
END PROGRAM hello
```

Variable Declaration

- Variables <u>must</u> be declared at the <u>start</u> of a PROGRAM, SUBROUTINE, FUNCTION block
- By default, variables are declared implicitly: any variable with the first character a-h,o-z will be implicitly defined as REAL, i-n as INTEGER (→ “god” is REAL, unless declared INTEGER)
- Implicit variable declaration is a **VERY BAD** idea, thus **always** use IMPLICIT NONE
- Variables may have <u>attributes</u> and <u>initializers</u>

Variable Declaration Examples

```
! test
PROGRAM test
  IMPLICIT NONE
  INTEGER :: i,j,k=0                  ! normal integers (k is static)
  INTEGER, SAVE :: l                  ! static integer (retains value)
  INTEGER(kind=8) :: m                ! long (=64-bit) integer
  INTEGER, PARAMETER :: o=10          ! constant (must initialize)
  REAL :: a,b=1.0                     ! single precision float
  REAL(kind=8) :: c=0.5_8             ! double precision
  CHARACTER(len=2) :: h = 'hi'        ! 2 character string
  LOGICAL :: y=.true.,n=.false.       ! boolean variables
  PRINT*, h,i,j,k,m,o,a,b,c,y,n       ! i,j,m,a are not initialized
END PROGRAM test

$ gfortran -Wall test.f90
$ ./a.out
 hi   486112256           0           0  3619698626323808256
    10   1.09286526E-08   1.00000000      0.50000000000000000
 T F
```

Operators

- Arithmetic operators: addition(+), subtraction(-), multiplication(*), division(/), exponentiation(**) (exponentiation faster if right value is integer)
- String operator: concatenation(//)
- Arithmetic comparisons: equal(==),
  not equal (/=), less(<), greater(>), and so on.
- Logical operators: not(.not.), and(.and.), or (.or.), equal(.eqv.), non-equal(.neqv.)
  Note: use .eqv./.neqv. to compare results of logical operations or comparisons, not == or /=

Structured Programming

- Fortran supports several common structured programming constructs:
  - IF … THEN … ELSE IF … ELSE … END IF
  - DO var=start,end,increment … END DO
  - DO WHILE (condition) … END DO
- Note that conditions are always evaluated <u>in full</u> (i.e. must **not** expect C-style **short circuiting**)
- Use EXIT to *break* out of construct and CYCLE to *continue* with next iteration immediately

Intrinsic Functions

- Fortran has a large collection of intrinsic functions: mod(x,y), min(x,y), max(x,y), floor(x), abs(x), sqrt(x), exp(x), log(x), sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), sinh(x), cosh(x), and many more. Many operate on integer, real and complex data
- Intrinsic functions usually return the data type of the argument unless promotion to floating point is required.
- Typecasts with int(x), dble(x), real(x), cmplx(x)

Bitwise Manipulations

- Many Fortran compilers supported extensions for bitwise operations since Fortran 77, but they were standardized only in Fortran 2008
- SHIFTL(x,n) returns the bits of integer variable ‘x’ shifted ‘n’ times to the left; SHIFTR(x,n) similarly shifts to the right;
- IAND(i,j), IOR(i,j), IEOR(i,j) compute and return the bitwise “and”, “or”, or “exclusive or” value of the arguments “i” and “j”; NOT(i) negates all bits
- IBITS(x,i,n) extracts “n” bits starting at the “i”-th

Arrays in Fortran

- Arrays in Fortran are declared either with the "dimension" attribute or by giving a dimension
```
REAL, DIMENSION(20) :: X,Y
REAL :: Z(20)
```
- Arrays are indexed starting from 1 (**not 0**)
- Multidimensional arrays are stored in "flat" memory, i.e. 1-d array plus information about the size of the dimensions (→ reshape).
- Leftmost array index follows elements that are consecutive in memory (opposite of C/C++)

Array Intrinsic Operation/Functions

- Fortran supports operations on entire arrays:

```
REAL:: A(10),B(10),C(10),Z,D(10,10)
A = B+C ! or: A(:) = B(:)+C(:)
C = cos(B)
Z = dot_product(A,B)
```

- It is also possible to operate on ranges:

```
A(6:10) = sqrt(B(1:5))
```

- Length of array can be determined with size()

```
PRINT*,size(D),size(D,1),size(D,2)
         100            10          10
```

Dynamic Memory Management

- Variables and arrays are defined on the stack unless flagged for dynamic allocation:

```
REAL,ALLOCATABLE :: a,x(:),y(:,:)
ALLOCATE(a,x(10),y(10,20))
a = 10.0
x(:) = 1.0
DEALLOCATE(a,x,y)
```

- The Fortran standard requires that dynamically allocated data is freed when the variable goes out of scope. Deallocate may not be required.

Subroutines

- Subroutines are code blocks executed with the CALL command that can have arguments
- Subroutines may not be called recursively
- Subroutines must have unique names across the entire program and – by default – correct number and type of arguments across different files are not checked, thus a mismatch can lead to crashes or undefined behavior
- Subroutine arguments can have the “INTENT” attribute signaling intended use: IN,OUT,INOUT

Subroutine Arguments

- Fortran has call by reference conventions, so changing an argument changes it in the calling program (unlike C/C++ with call by value)
- Arrays are passed as reference to the first element, thus using CALL func(a) and CALL func(a(1)) are the same
- Array dimensions can use “wildcards”
  REAL, INTENT(IN) :: a(:), b(:,:)
  REAL, INTENT(IN) :: c(10,10,*), d(*)
  Only in the first case dimensions are passed

More on Subroutine Arguments

- Variable initializer is only applied on first call
- To retain variable value between calls use SAVE attribute (same as “static” in C/C++)
- Arguments may be flagged with the OPTIONAL attribute and then can be left out.
- Multiple optional arguments are assigned on call either by order or with <name>=<value>
- Optional arguments may not be used unless they are present (→IF (PRESENT(<name>))

Subroutines versus Functions

- Functions are similar to subroutines, but have a return value and thus a return type
- The return value is set by assigning a value to the variable that has the name of the function
- Because of the return value, functions **must** be declared or else Fortran will assume it is an implicitly declared array
- An alternate variable may be used as return value via the RESULT keyword:
  FUNCTION add(x,y) RESULT z

Master in High Performance Computing

Interfaces and Overloading

- To have argument checking, you need to declare subroutines and functions from other files in an “INTERFACE” block.
- An INTERFACE block repeats declaration, list of arguments, type and attributes of arguments, IMPLICIT and USE statements
- To overload create an INTERFACE with “name” and declare multiple interfaces inside. When “name” is called, the compiler will substitute a call to the matching interface from that block

Modules

- Modules are probably the most important feature of modern Fortran versions, somewhat similar to C++ namespaces or Python modules
- A module can contain data and code
- Both can be either public or private
- When compiling interfaces and types are stored in a (compiler specific) <name>.mod file
- Import variables and code with USE <name>
- Import can be selective (via ONLY) or aliased

Benefits to using Modules

- Select visibility of code, avoid naming conflicts
- Automatic generation of interfaces
- Simpler syntax for overloading
- Allows to organize code “by topic”
- Specific functions/subroutines can be kept local by declaring them PRIVATE (cf. static in C/C++)
- Cleaner alternative to global variables

Derived Types

- Derived types allow to build custom compound data types (similar to “struct” in C/C++)
- Variables with derived types are declared like other variables using TYPE(<name>) and can have attributes and dimensions as well
- Members of a derived type are accessed with the “%” operator: mytype%member
- Derived types can be extended:
  TYPE, extends(one) :: two
  END TYPE two

More on Derived Types

- Derived types may contain other derived types
- Derived types may contain subroutines or functions (type bound procedures) and thus function similar to classes in C++
- Derived types can be simply output with PRINT* unless they have allocatable members.
- Similar for reading: the members are just filled as if given as individual variables

Simple Fortran I/O

- The built in I/O library of Fortran supports writing in binary, text and direct access mode and allows writing to files and strings
- Different I/O streams are identified by integers
- Its formatted I/O is unusual in that it will print a set of stars '****' if output would overflow the given format (motivated by output to a printer)
- We will avoid this by using the default channels (represented by a '*') and default formats (also represented by a '*') here and avoid complexity:

```
WRITE(*,*) var,a(1) ! or PRINT*,var,a(1)
READ(*,*) var1,var2,a(1),a(2)
```

Fortran READ/WRITE Statements

- Fortran I/O works in “records” where each READ or WRITE command represents a record
- For writing in text-mode a “record” is a line unless the format limits the number of items per line; then additional newlines are inserted
- For reading in text-mode a “record” is a line unless there are more elements to be read; then data from the next line is read
- When there are fewer elements to be read than contained in the line, the remainder is discarded

Fortran I/O: Implicit Loops

- To output or read a whole array (or a subset) implicit loops may be used:
  `WRITE(*,*) x,y,(a(i),i=1,num)`
  and:
  `READ(*,*) x,y,(a(i),i=1,num)`
- Implicit loops can be nested:
  `WRITE(*,*) ((a(i,j),i=1,n1),j=1,n2)`
- When the dimensions are known (not declared with a ‘*’) the array or array range can be used:
  `WRITE(*,*) a,b(:,:),c(1,:),d(2:5)`

Modern Fortran vs C Interoperability

- Fortran 2003 introduces a standardized way to tell Fortran how C functions look like and how to make Fortran functions have a C-style ABI
- Module “iso_c_binding” provides kind definition: e.g. C_INT, C_FLOAT, C_SIGNED_CHAR
- Subroutines can be declared with “BIND(C)”
- Arguments can be given the property “VALUE” to indicate C-style call-by-value conventions
- String passing tricky, needs explicit 0-terminus

Calling C from Fortran 03 Example

```
int sum_abs(int *in, int num) {
  int i,sum;
  for (i=0, sum=0; i<num; ++i) sum += abs(in[i]);
  return sum;
}
/* fortran code:
  USE iso_c_binding, ONLY: c_int
  INTERFACE
    INTEGER(c_int) FUNCTION sum_abs(in, num) BIND(C)
      USE iso_c_binding, ONLY: c_int
      INTEGER(c_int), INTENT(in) :: in(*)
      INTEGER(c_int), VALUE :: num
    END FUNCTION sum_abs
  END INTERFACE
  INTEGER(c_int), PARAMETER :: n=200
  INTEGER(c_int) :: data(n)
  PRINT*, SUM_ABS(data,n)  */
```

Calling Fortran 03 From C Example

```
SUBROUTINE sum_abs(in, num, out) BIND(c)
   USE iso_c_binding, ONLY : c_int
   INTEGER(c_int), INTENT(in)  :: num,in(num)
   INTEGER(c_int), INTENT(out) :: out
   INTEGER(c_int),             :: i, sum
   sum = 0
   DO i=1,num
     sum = sum + ABS(in(i))
   END DO
   out = sum
END SUBROUTINE sum_abs

!! c code:
!   const int n=200;
!   int data[n], s;
!   sum_abs(data, &n, &s);
!   printf("%d\n", s);
```

Master in High Performance Computing

Pointers in Fortran vs. C/C++

- In Fortran “POINTER” is an attribute to be used when declaring a variable (like “PARAMETER” or “ALLOCATABLE”)
- In C/C++ a pointer is kind of variable (stores a memory address) with an associated type
- Unlike in C/C++ when using a pointer variable in Fortran, you access the data it is pointing to, i.e. it is more like a reference
- To make a pointer variable point to a “target”, you need to use the association operator: =>
  Associate with NULL() to “disconnect” a pointer
- ASSOCIATED() returns .true. for pointers associated to a target
- Only variables that have the “TARGET” attribute added or other “POINTER” variables or NULL() can be used for associations

Simple Example using Pointers

```
PROGRAM pointers
  REAL, POINTER :: q => NULL()
  REAL, TARGET  :: c = 0.0, d = -1.0

  PRINT*,ASSOCIATED(q)  ! prints F
  q => c
  PRINT*,ASSOCIATED(q)  ! prints T
  PRINT*,c,q            ! prints 0.0  0.0
  c = 1.0
  PRINT*,c,q            ! prints 1.0  1.0
  q = 2.0
  PRINT*,c,q            ! prints 2.0  2.0
  q => d
  PRINT*,c,q            ! prints 2.0 -1.0
END PROGRAM pointers
```

POINTER versus ALLOCATABLE

- Variables with the POINTER attribute can also be used with ALLOCATE() and DEALLOCATE()
- This is considered equivalent to defining an “anonymous” “ALLOCATABLE,TARGET” variable and then associating the pointer with it
- Because of the “anonymous” nature there is no automatic deallocation when the pointer goes out of scope (unlike for ALLOCATABLE), so the POINTER behaves like ALLOCATABLE, SAVE
- => use DEALLOCATE() to avoid memory leaks

Pointers in Derived Types

- The POINTER attribute may also be used for members of a derived type and include references to its own type. Derived types can then be allocated and associated. Example:

```
TYPE llitem
    TYPE(llitem), POINTER :: next => NULL()
    INTEGER :: val
END TYPE llitem
TYPE(llitem), POINTER :: head, item => NULL()

ALLOCATE(item)
item%val = 1
head => item
```

Building a Data Structure

- The code from the previous example can be extended to build a longer linked list:

```
TYPE(llitem), POINTER :: item, head

ALLOCATE(item)
item%val = 1   ! item%next defaults to NULL()
head => item
ALLOCATE(item)
item%val = 2
item%next => head
head => item
ALLOCATE(item)
item%val = 3
item%next => head
head => item
```

Traversing a Data Structure

- After the code from the previous example we have a linked list with 3 items. Now we want to output its values:

```
item => head
DO WHILE (ASSOCIATED(item))
    PRINT*,item%val
    item => item%next
END DO
```

- We can delete the linked list in a similar fashion:

```
DO WHILE (ASSOCIATED(head))
  item => head%next
  DEALLOCATE(head)
  head => item
END DO
```

Type-Bound Procedures (1)

- First step toward classes in Fortran is to include functions and subroutines into derived types. these are called type-bound procedures:

```
MODULE zoo
  PRIVATE
  TYPE animal
     CHARACTER(len=8) :: word=' '
  CONTAINS
     PROCEDURE :: say
  END TYPE ANIMAL
  PUBLIC :: animal
CONTAINS
  SUBROUTINE say(this)
  CLASS(animal) :: this
    PRINT*,'This animal says',this%word
  END SUBROUTINE say
END MODULE zoo
```

Type-Bound Procedures (2)

- Derived type needs to be defined in a module
- Type-bound procedure must have the derived type as the first argument, usually named “self” or “this” (same as in Python)
- Instead of TYPE(name) use CLASS(name) for first argument referring to the instance itself
- Contained procedure can be any subroutine or function inside the module (may be private)
- Access with <name>%<procedure>

Constructors

- The constructor is a function contained in the module (but not type-bound) that has the type of the class as return value
- It is made a constructor by defining an interface that has the same name as the derived type
  → that also allows to overload the constructor

```
MODULE zoo
! [...]
  INTERFACE animal
    MODULE PROCEDURE :: init_animal
  END INTERFACE animal
CONTAINS
  TYPE(animal) init_animal FUNCTION(w)
    init_animal%word = w
  END SUBROUTINE init_animal
END MODULE zoo
```

```
PROGRAM sounds
  USE zoo
  IMPLICIT NONE
  TYPE(animal) :: one, two

  one = animal(‘woof’)
  two = animal(‘meow’)
  CALL one%say
  CALL two%say
END PROGRAM sounds
```

Destructor

- A destructor is a type-bound procedure that is called automatically if an allocated derived type is deallocated
- It must have a TYPE (not CLASS) as first argument

```
MODULE zoo
  TYPE animal
      CHARACTER(len=10) :: word
    CONTAINS
      PROCEDURE :: say
      FINAL :: theend
  END TYPE animal
CONTAINS
  SUBROUTINE theend(self)
    TYPE(animal) :: self
    PRINT*,'the end of me ',self%word
  END SUBROUTINE theend
  SUBROUTINE say(self)
    CLASS(animal) :: self
    PRINT*, 'Say: ‘,self%word)
  END SUBROUTINE say
```

```
PROGRAM sounds
  USE zoo
  IMPLICIT NONE
  TYPE(animal),ALLOCATABLE :: one
  TYPE(animal) :: two

  ALLOCATE(one)
  one = animal(‘woof’)
  two = animal(‘meow’)
  CALL one%say
  CALL two%say
! this will trigger the destructor
  DEALLOCATE(one)
END PROGRAM sounds
```

Introduction to Modern Fortran

Dr. Axel Kohlmeyer

Associate Dean for High-Performance Computing
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu