Introduction to Fortran: Part 2

Reuben D. Budiardja

National Institute for Computational Sciences

Seminar Series on High Performance Computing

Modern Fortran Features

Program organization and style:

- free format
- modules, sub-module
- **implict none**
- **portable** precision model

Fortran 90/95 OO features:

- derived type
- array operations, dynamic allocation
- procedure and **operator** overloading (*static polymorphism*)
- Fortran pointer

Modern Fortran Features

Program organization and style:

- free format
- modules, sub-module
- **implict none**
- **portable** precision model

Fortran 90/95 OO features:

- derived type
- array operations, dynamic allocation
- procedure and **operator** overloading (*static polymorphism*)
- Fortran pointer

Fortran 2003 adds more:

- type-bound procedures with `pass` attribute
- procedure pointers
- Type-bound procedure by name
- final procedure, automatic deallocation (*garbage collection*)
- type extension, **abstract** type (*templating*)
- polymorphic entities
- procedure **overriding**, deferred binding (*dynamic polymorphism*)

Arrays

- **Array**: a collection of values (of certain data types, e.g. integer, real, etc) indexed by *array index* (*key*)
- Static-ly (compile time) and Dynamic-ly(runtime) allocated arrays are possible
- The majority of **intrinsic** functions supports array argument
- Fortran supports true multi-dimensional arrays

Arrays

- **Array**: a collection of values (of certain data types, e.g. integer, real, etc) indexed by *array index* (*key*)
- Static-ly (compile time) and Dynamic-ly(runtime) allocated arrays are possible
- The majority of **intrinsic** functions supports array argument
- Fortran supports true multi-dimensional arrays
  - as opposed to C & C++, for example, where multi-D array is implemented as a vector of pointers (array of array)
  - true multi-D array in contiguous memory block is more efficient for many array operations

Arrays Terminologies

- The **rank** of the array is the number of dimensions
  - maximum rank is 31 (F2008)
  - previous max is 7 (F90)
- The number of elements in a dimension is the **extent**
- The **shape** of an array is a vector where each element of the vector is the extent in the corresponding dimension

Array Syntax

```
1  real                              :: x
2  real, dimension(10)               :: a, b
3  real, dimension(10,10)            :: c, d
4  real, dimension(0:4,3:8,7,9) &
5    :: e
6
7  a          = b
8  c          = d
9  a(1:10)    = b(1:10)
10 a(2:3)     = b(4:5)
11 a(1:10)    = c(1:10,2)
12 a          = x
13 c          = x
14 a(1:5)     = b(1:5) * cos(c(6:10,1))
15 a(1:3)     = b(1:5:2)
16
17 print*, shape(e)  !- prints 5,6,7,9
```

- Array assignment must be *conformable*: size and dimension (i.e. shape) have to agree
- Scalars are conformable
- Strides can be used too
- Default lower bound is 1, but can be explicitly specified

Array Constructor

Example: Array Constructor

```
1 real   dimension(3) :: x = [0.0, 1.0, 2.0]
2 real, dimension(5) :: a, b
3
4 a = [x, 3.0, 4.0]
5 b = [ (sqrt(real(i)), i=1.5) ]  !- constructor with implicit do-loop
```

Dynamically Allocated Arrays

- Array can be dynamically allocated at runtime $\Longrightarrow$ flexible size

Dynamically Allocated Arrays

- Array can be dynamically allocated at runtime $\Longrightarrow$ flexible size
- Dynamic array is allocated on the *heap* memory
  - vs. static array on the stack
  - the size of stack is often limited (default: 2 GB)
  - remedies are somewhat problematic (e.g. compiler dependent, not portable)

Dynamically Allocated Arrays

- Array can be dynamically allocated at runtime $\Longrightarrow$ flexible size
- Dynamic array is allocated on the *heap* memory
  - vs. static array on the stack
  - the size of stack is often limited (default: 2 GB)
  - remedies are somewhat problematic (e.g. compiler dependent, not portable)
- Allocation/deallocation is done with **allocate/deallocate** statement
  - allocation and deallocation take time
  - e.g. do not allocate/deallocate in a loop
- Use allocatable for large arrays

Example: Allocatable Array

```
1  program allocatable_array
2    implicit none
3    integer :: iSize, jSize, kSize
4    real, dimension(:,:,:), allocatable :: a_3d
5
6    print*, "Input the size of the array"
7    read*, iSize, jSize, kSize
8
9    allocate(a_3d(iSize, jSize, kSize))
10   print*, shape(a_3d)
11
12   !- Use array
13
14   deallocate(a_3d)
15 end program allocatable_array
```

Array Construct: where

The **where** construct is used when we want to make selective assignment to array.

Example: where

```
1  real , dimension(10,10) :: &
2    a, recip_a
3
4  !- fill in a ...
5
6  where( a/= 0.0 )
7    recip_a = 1.0/a
8  elsewhere
9    recip_a = 1.0
10 endwhere
```

- Arrays must be conformable
- code block executes when condition is true

Array Construct: forall

The **forall** construct provide a mechanism to specify an indexed parallel assignment to array.

Example: forall

```
1  !- With do-loops:
2  do j = 1, m
3    do i = 1, n
4      a(i, j) = i + j
5    end do
6  end do
7
8  !- with forall
9  forall(i = 1:n, j = 1:m)
10   a(i, j) = i + j
11 end forall
```

- The **forall** and **where** constructs may aid in vectorization and parallelization on certain architecture
- These constructs provide more natural mechanism to express often found mathematical formulas

any Function

The function **any** returns true if any array elements are true.

Example: any

```
1 integer, dimension(4) :: a
2
3 a = [0, -1, 1, 2]
4
5 if (any(a < 0)) then
6   print*, 'found negative number'
7 end if
```

all Function

The function **all** returns true if all array elements are true.

Example: all

```
1  integer, dimension(4) :: a, b
2
3  a = [0, -1, 1, 2]
4  b = [5, 3, 6, 4]
5
6  if (all(a > 0)) then
7    print*, 'All numbers are positive'
8  end if
9
10 if (all(a /= b))
11   print*, 'a and b are mutually exclusive'
12 end if
```

Passing Arrays to Subroutines

Example: Array Arguments

```
1 subroutine ex2 (a, b, x)
2  real, dimension(:), intent(in)  :: a
3  real, dimension(0:), intent(in) :: b
4  real, dimension(2:, :), intent(inout) :: x
5  ...
6 end subroutine ex2
7
8 call ex2(u, v, w(4:9, 2:6))
```

- An assumed-shape array is a *dummy* argument that takes the shape of the *actual* argument
- The default lower bound is 1, but can be redefined
- The *extent* of the array is determined from the actual argument

Automatic Array

- An *automatic array* can be created upon entry to a procedure
- The size can vary and determined at runtime
- The array is allocated on the stack
- The object disappear when the procedure exit
- Requires explicit interface of the procedure (see module section)

Example: Automatic Array

```
1 subroutine swap(a, b)
2  real, dimension(:), intent(in)   :: a, b
3  real, dimension(size(a))         :: work
4
5  work = a
6  a    = b
7  b    = work
8 end subroutine swap
```

Elemental Procedures

- A procedure (function or subroutine) can be declared as **elemental**
- An **elemental** procedure can be passed an array of any dimension
- Each element of the array is acted upon one element at a time
- The argument and result must be scalar
- The procedure must have an explicit interface visible to the caller (more on this later in *module*)
- Elemental procedure must be **pure**:
  - A pure procedure has no side effect: it does not change the state of the program except,
  - For function: it returns a value
  - For subroutines: it modifies **intent(out)** and **intent(inout)** arguments
  - All intrinsic functions are pure

Example: Elemental Procedures

```
1  module geometry
2    implicit none
3
4  contains
5
6    elemental function circumference(r) result (c)
7      real, intent(in) :: r   !- intent(in) required
8      real             :: c
9      c = 2.0 * acos(-1.0) * r
10   end function circumference
11
12   elemental subroutine area(r, a)
13     real, intent(in)  :: r   !- intent(in) required
14     real, intent(out) :: a
15     a = acos(-1.0) * r**2
16   end subroutine area
17
18 end module geometry
```

Example: Invoking Elemental Procedures

```
1  program circles
2    use geometry
3
4    real, dimension(5) :: r, a
5
6    r = [2.0, 4.0, 4.3, 5.0, 7.0]
7
8    print*, circumference(r)
9
10   call area(r,a)
11   print*, a
12
13 end program circle
```

Derived (Structures) Data Types

- A collection of intrinsic data types (integer, real, logical, etc)
- Can hold static and allocatable arrays
- Can hold other derived types
- A derived type must first be **declared** before it can be used

Example: Derived Type

```
1  program DerivedTypeExample
2
3    implicit none
4
5    type :: color
6      integer                    :: shade
7      real, dimension(3)         :: rgb
8      character(len=30)          :: name
9    end type color
10
11   type(color) :: background
12
13   background % shade    = 10
14   background % rgb      = [128.0, 255.0, 132.0]
15   background % name     = "greyish"
16
17   print*, background
18 end program DerivedTypeExample
```

Procedures and Derived Type

- Procedures (functions and subroutines) can have derived-types as arguments and results (for functions)
- Use derived-types just like intrinsic type

Program Organization

Why should we use Subprogram (Function / Subroutines)?

- Decompose complex tasks into simpler, manageable code blocks
  - no "super main" program
- Enable reuse of code, reduce code duplication
- Hide implementation details, limit scope of variables

Module

Modules are another, more flexible tool to organize code base.

A **module** contains definition that can be made accessible to other program units.

Modules may contain:

- Derived Type definitions
- Variables
- Subroutines, Functions

A module collects related type definition, subroutines, and functions into a *package* that can be re-used (by other program, modules, etc).

Example: Module

```
1  module geometry
2    implicit none
3    private
4
5    type, public :: circleType
6      real                  :: radius
7      real, dimension(2) :: center
8    end type circleType
9
10   real, parameter :: PI = acos(-1.0)
11
12 contains
13
14   elemental function circumference(g) result (c)
15     type(circleType), intent(in) :: g
16     real                         :: c
17     c = 2.0 * PI * g % radius
18   end function circumference
19
20 end module geometry
```

How To / Why Use Module ?

- The **public** entities (default) of a module is accessible to other program unit via **use** statement

How To / Why Use Module ?

- The **public** entities (default) of a module is accessible to other program unit via **use** statement
- Modules provide realible mechanism for specifying global data, including variables, type definitions, and procedure interface

How To / Why Use Module ?

- The **public** entities (default) of a module is accessible to other program unit via **use** statement
- Modules provide realible mechanism for specifying global data, including variables, type definitions, and procedure interface
- Modules facilitate object-oriented concepts and modular programming

How To / Why Use Module ?

- The **public** entities (default) of a module is accessible to other program unit via **use** statement
- Modules provide realible mechanism for specifying global data, including variables, type definitions, and procedure interface
- Modules facilitate object-oriented concepts and modular programming
- Modules provide information hiding:
  - **private** entities not visible outside the module,
  - **protected** entities are visible as *read-only* outside the module
  - Internal design can be altered without the need to alter the program
  - Prevent accidental alteration of internal data
- Modules provide explicit interfaces (see next slide)

module is arguably the most important feature for program organization in modern Fortran. Use it!

Implicit and Explicit Interface

Implicit interface:

- Position of arguments to procedures are the only available information
- The call to procedure is matched and resolved during program linking
- This is all that is available in Fortran 77
- Drawbacks:
  - No compile-time type checking
  - Programmer is responsible for checking the interfaces
  - If there is a mismatch of type, precision, etc, may result in strange program behavior, segmentation fault, and other runtime error

Implicit and Explicit Interface

Explicit interface:

- More information is available to the compilers: types, precision, argument names, etc.
- Provided automatically when procedures are in a module
- Module has to be compiled first, then use-d by the other program unit (other module or program). (Usually this is in the form of .mod file.)
- Benefits:
  - At compile time, compiler does consistency checking of argument passed by the caller to procedures.
  - Required for many features of modern Fortran: optional parameters, elemental procedures, assume-shape arrays, procedure overloading, etc.

Always use explicit interface. The easiest way to do this is to put all the procedures inside module(s).

Putting It All Together: Module Example

Generic Interface: Procedures Overloading

- Function/suboutine overloading allows several procedures to be called by the same generic name

Generic Interface: Procedures Overloading

- Function/suboutine overloading allows several procedures to be called by the same generic name
- Useful for implementing different variations of the same basic concepts, while hiding implementation details
- In Object-oriented principle, this is known as *polymorphism*.

Generic Interface: Procedures Overloading

- Function/suboutine overloading allows several procedures to be called by the same generic name
- Useful for implementing different variations of the same basic concepts, while hiding implementation details
- In Object-oriented principle, this is known as *polymorphism*.
- Examples:
  - The area of any object has the same basic concept, but computed differently for circle, rectangle, triangle, etc.

Generic Interface: Procedures Overloading

- Function/suboutine overloading allows several procedures to be called by the same generic name
- Useful for implementing different variations of the same basic concepts, while hiding implementation details
- In Object-oriented principle, this is known as *polymorphism*.
- Examples:
  - The area of any object has the same basic concept, but computed differently for circle, rectangle, triangle, etc.
- A procedure can be overloaded if at least one of its arguments is distinguishable
  - Anything distinguishable works: different precision, array shape, types

Example: Generic Interface

Generic Interface (2)

- Generic interface works across modules in different files
- A better way for the previous example:
  - Create one module for Circle, another for Rectangle
  - Scalable programming: different people can work on different object
- Intrinsic subroutines / functions can also be overloaded

Operator Overloading

- Intrinsic operator (+, -, /, =) can be overloaded for user-defined and derived-type entities
- Using interface block, new meaning can be given to intrinsic operator

Example: Operator Overloading

```
1  module geometry
2    implicit none
3    type, public :: circleType
4      real                   :: radius
5      real, dimension(2) :: center
6    end type circleType
7
8    interface operator (+)
9       module procedure circle_sum
10   end interface
11
12 contains
13
14   function circle_sum(c1,c2) result (cs)
15     type(circleType), intent(in) :: c1, c2
16     type(circleType)                  :: cs
17     cs%radius = c1%radius + c2%radius
18   end function circle_sum
19
20 end module geometry
```

Example: Operator Overloading

```
1  program circle_sum
2    use geometry
3    implicit none
4
5    type(circleType) :: c1, c2, c3
6
7    c1%radius = 3.0
8    c2%radius = 5.0
9
10   c3 = c1 + c2
11
12   print%, c3%radius  !- prints 8.0
13
14 end module geometry
```

Best Practices in Fortran Programming

- Take advantage of free format
  - Use spaces, indentations, line breaks as necessary for better readability
  - Use comments as necessary

Best Practices in Fortran Programming

- Take advantage of free format
  - Use spaces, indentations, line breaks as necessary for better readability
  - Use comments as necessary
- Use implicit none to avoid implicit declaration

Best Practices in Fortran Programming

- Take advantage of free format
  - Use spaces, indentations, line breaks as necessary for better readability
  - Use comments as necessary
- Use implicit none to avoid implicit declaration
- Put all subroutines in modules to get explicit interface checking

Best Practices in Fortran Programming

- Take advantage of free format
  - Use spaces, indentations, line breaks as necessary for better readability
  - Use comments as necessary
- Use implicit none to avoid implicit declaration
- Put all subroutines in modules to get explicit interface checking
- Write module-oriented (modular) code:
  - Put related subroutines, data types, into a module
  - Use module for access control, information hiding
  - Create one file per module
  - Break up modules as needed, to manage complexity and to get scalable programming
- Use module variables and common blocks judiciously

Best Practices in Fortran Programming

- Take advantage of free format
  - Use spaces, indentations, line breaks as necessary for better readability
  - Use comments as necessary
- Use implicit none to avoid implicit declaration
- Put all subroutines in modules to get explicit interface checking
- Write module-oriented (modular) code:
  - Put related subroutines, data types, into a module
  - Use module for access control, information hiding
  - Create one file per module
  - Break up modules as needed, to manage complexity and to get scalable programming
- Use module variables and common blocks judiciously
  - module variables are global
  - not thread-safe by default

Best Practices in Fortran Programming

- Use derived-type to organize your data
- One derived-type (with its type-bound procedures) and related methods per module: **a class** (object and methods)
- Use intent for subroutine arguments

Best Practices in Fortran Programming

- Use derived-type to organize your data
- One derived-type (with its type-bound procedures) and related methods per module: **a class** (object and methods)
- Use intent for subroutine arguments
- Function should not have side-effects: pure function

Best Practices in Fortran Programming

- Use derived-type to organize your data
- One derived-type (with its type-bound procedures) and related methods per module: **a class** (object and methods)
- Use intent for subroutine arguments
- Function should not have side-effects: pure function
- Use portable precision model with kind

Modern Fortran Features (Recap)

Program organization and style:

- free format
- modules, sub-module
- **implict none**
- **portable** precision model

Fortran 90/95 OO features:

- derived type
- array operations, dynamic allocation
- procedure and **operator** overloading (*static polymorphism*)
- Fortran pointer

Fortran 2003 adds more:

- type-bound procedures with `pass` attribute
- procedure pointers
- Type-bound procedure by name
- final procedure, automatic deallocation (*garbage collection*)
- type extension, **abstract** type (*templating*)
- polymorphic entities
- procedure **overriding**, deferred binding (*dynamic polymorphism*)