UNIVERSITY of HOUSTON

CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

Introduction to Fortran 90 programming, part II

Martín Huarte-Espinosa

mhuartee@central.uh.edu

UNIVERSITY of HOUSTON

CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

VSCSE Summer Training Courses

CACDS to host two summer school classses on Harness the Power of GPU's: Introduction of GPGPU Programming on June 16th - 20th, 2014 and Data Intensive Summer Scool on June 30th - July 2nd 2014. Registration is open!

Continue Reading »

Summer Mini-Series in HPC

CACDS and the HPCTools group will host a block of presentations by the Supercomputing Program Commitee between June 13-20, 2014, at the University of Houston Philip Guthrie Hoffman Hall, Building 547, Room 232, to attract interest from researchers across campus who are engaged in HPC.

Continue Reading »

Argonne Training Program

This two-week program provides intensive hands-on training and offers renowned scientists, HPS experts, and leaders who serve as lecturers and guide hands-on laboratory sessions.

Continue Reading »

cacds.uh.edu/?q=training

UNIVERSITY of HOUSTON
CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

ABOUT | RESEARCH | EDUCATION | RESOURCES | NEWS

TRAINING COURSES
EVENTS
LECTURES
WORKSHOPS
TECHNICAL SEMINARS

Training Courses

EDUCATION

Training Courses>>
- Events
- Lectures
- Workshops
- Technical Seminars

CACDS has launched several HPC training courses for members of the UH research community.
Registration is required to take the courses. Please click the date you would like to attend under each course to register.

CURRENT TRAINING

Upcoming training events at CACDS

UNIVERSITY of HOUSTON

CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

Course Name	Dates
Introduction to C++ Programming I	February 27, 2015 10:00 AM - 12:30 PM
Introduction to C++ Programming II	March 3, 2015 10:00 AM - 12:30 PM
Intel Xeon Phi Programming I	February 24, 2015 2:00 - 4:00 PM
Intel Xeon Phi Programming II	February 27, 2015 2:00 - 4:00 PM

Introduction to **MPI programming with Fortran 90**, 3/13.

Getting started

1. Login: hpc_user{1-47}, follow your seats' order

   Password=cacds2014

2. Go to *Applications > Systems tools > Terminal*

   Type the following 3 commands, and press ENTER after each one:

   include period

```
cp  /share/apps/tutorials/intro2f90.zip  .
unzip  intro2f90.zip
cd  intro2f90
```

3. See the slides, type:

```
firefox  intro2f90_2.pdf
```

   and press ENTER.

Inside intro2f90

- Files with extension .f90 are exercise programs.

- Files with extension .solution.f90 are solutions to the program with the same name prefix.

- Files with extension .dat are data files to be used along with some of the .f90 programs.

- intro2f90_2.pdf are these slides. intro2f90_1.pdf are the slides from the previous workshop.

- Some programs were studied in the previous workshop.

Creating the executable (binary) from the source code

1. Code is saved into a .f90 file.

2. Compile your code, type:

```
f95  prog.f90  -o  prog.exe
```

3. Run the program, type:

```
./prog.exe
```

Arrays

```
REAL, DIMENSION (5) :: A, B   !arrays of 5 elements, with
                                 !indices 1,2…, 5.
REAL, DIMENSION (16) :: B !arrays of 16 elements, with
                               !indices 1,2..., 16

A = (/ 2, 4, 6, 8, 10 /)   !note the (/ … /)

B = (/    1, 2, 3, 4, 5, 6, 7, 8, &
       2, 4, 6, 8, 10, 12, 14, 16 /)
```

Very important for STEM applications.

Arrays

```
REAL, DIMENSION (5) :: A, B      !2 arrays of 5 elements, with
                                 !indices 1,2...5.
REAL, DIMENSION (0:4) :: c       !1 array of 5 elements, index
                                 !0,1...4

A = (/ 2, 4, 6, 8, 10/) or          !note the (/ … /)
A = (/  (2*I, I = 1,5) /) or        !I must be declared as INTEGER
A = (/ 2, (2*I, I = 2, 4), 10 /)

!Operations may be applied to arrays of equal dimensions:
B=A            !Array syntax, powerful feature of Fortran 90
C=5*C+A
WHERE (A > 6)     !This gives B = 0,0,0,1,1
      B = 1.
ELSEWHERE
      B = 0.
END WHERE
```

Arrays -- multi dimensional

```
REAL, DIMENSION(3,4) :: y                 !Rank 2, 3x4 elements
REAL, DIMENSION(0:2,0:3) :: x             !Same
INTEGER, DIMENSION(12,22,4) :: z          !Rank 3
                                       !Max rank per array = 7
z=0 !constant.

!First index is the row. Second one is the column
x(0,0)=1. ; x(0,1)=1. ; x(0,2)=1. ;  x(0,3)=1.
x(1,0)=2. ; x(1,1)=2. ; x(1,2)=2. ;  x(1,3)=2.
x(2,0)=3. ; x(2,1)=3. ; x(2,2)=3. ;  x(3,3)=3.
                         !; used for many commands in one line
DO i = 0, 3             !Transpose a matrix
   DO j = i+1, 3
      d = x(i,j)        !d defined REAL earlier
      x(i,j) = x(j,i)
      x(j,i) = d
END DO ; END DO
```

Arrays -- multi dimensional

The construct

```
x(0,0)=1. ; x(0,1)=1. ; x(0,2)=1. ;  x(0,3)=1.
x(1,0)=2. ; x(1,1)=2. ; x(1,2)=2. ;  x(1,3)=2.
x(2,0)=3. ; x(2,1)=3. ; x(2,2)=3. ;  x(3,3)=3.
```

can also be written as: `x(0,:)=1. ; x(1,:)=2. ; x(2,:)=3.`,
where the colon, ***:***, means “all elements of that dimension”.

Exercise -- Arrays 1

Just do the algorithm (no need to compile, etc.).

Declare an integer array `iarray`, which contains 3 rows and 4 columns. Initialize the first row with integer values 1 to 4 (from left to right), the second row with integers from 5 to 8, and fill the last row with -2. Then print the `iarray`, row by row so that each output line contains the elements of one row at a time.

From http://napsu.karmitsa.fi/courses/supercomputing/lssc/fortran_ex.txt

Exercise -- Arrays 1, solution

Just do the algorithm.

Declare an integer array `iarray`, which contains 3 rows and 4 columns. Initialize the first row with integer values 1 to 4 (from left to right), the second row with integers from 5 to 8, and fill the last row with -2. Then print the `iarray`, row by row so that each output line contains the elements of one row at a time.

```
...
INTEGER, DIMENSION(3,4) :: iarray
INTEGER i,j

iarray(1,1:4) = (/ (j, j=1,4) /)
iarray(2,1:4) = (/ (j, j=5,8) /)
iarray(3,:) = -2
DO i=1,3
    WRITE(*,*) iarray(i,:)
END DO
```

From http://napsu.karmitsa.fi/courses/supercomputing/lssc/fortran_ex.txt

Exercise -- Arrays 2

Continue. Just the algorithm.

Build a new array; 3-by-8 integer array `bigarray`, where the 4 first columns are identical to the `iarray`, and the 4 last columns are obtained from the columns of the array `iarray` by multiplying them with the number 3 and adding 5. **Use array syntax**.

From http://napsu.karmitsa.fi/courses/supercomputing/lssc/fortran_ex.txt

Exercise -- Arrays 2, solution

Continue. Just the algorithm.

Build a new array; 3-by-8 integer array `bigarray`, where the 4 first columns are identical to the `iarray`, and the 4 last columns are obtained from the columns of the array `iarray` by multiplying them with the number 3 and adding 5. **Use array syntax**.

```
...
INTEGER, DIMENSION(3,8) :: bigarray
INTEGER, DIMENSION(3,4) :: iarray
INTEGER i,j
iarray(1,1:4) = (/ (j, j=1,4) /)
iarray(2,1:4) = (/ (j, j=5,8) /)
iarray(3, : ) = -2
bigarray(:,1:4) = iarray(:,:)
bigarray(:,5:8) = iarray(:,1:4) * 3 + 5
```

From http://napsu.karmitsa.fi/courses/supercomputing/lssc/fortran_ex.txt

Arrays -- intrinsic functions

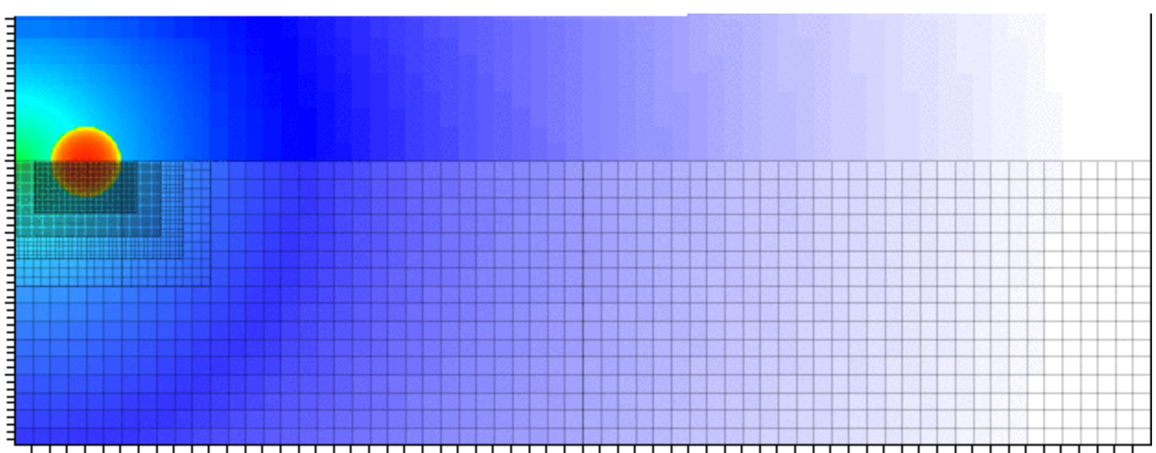
DOT_PRODUCT(A, B)	
MAXVAL(A)	maximum value in A
MAXLOC(A)	one-element 1D array whose value is the location of the first occurrence of the maximum value in A
PRODUCT(A)	product of the elements of A
SUM(A)	sum of the elements of A
MAXVAL(A, D)	array of one less dimension than A; the A maximum values along dimension D
MAXLOC(A)	one-element 1D array whose value is the location of the first occurrence of the maximum value in A
SUM(A, D)	array of one less dimension than A; sums of A elements along dimension D (if D omitted, returns entire array elements sum)
MATMUL(A, B)	matrix product of A and B
TRANSPOSE(A)	

Arrays -- dynamic allocation

Assign memory for arrays during execution.
Scientific application: adaptive-mesh simulations.

E.g.

```
INTEGER :: i,j,AllocateStatus
REAL, DIMENSION(:,:), ALLOCATABLE :: A
…
READ (*,*)  i,j  !read array dimension from keyboard
ALLOCATE ( A(i,j), STAT = AllocateStatus)
   [IF (AllocateStatus /= 0) STOP "Insufficient A memory"]
```



Arrays -- dynamic allocation

Assign memory for arrays during execution.
Scientific application: adaptive-mesh simulations.

E.g.

```
INTEGER :: i,j,AllocateStatus
REAL, DIMENSION(:,:), ALLOCATABLE :: A
…
READ (*,*)  i,j  !read array dimension from keyboard
ALLOCATE ( A(i,j), STAT = AllocateStatus)
   [IF (AllocateStatus /= 0) STOP "Insufficient A memory"]
```

The allocated memory has to be released once it it's not used:

```
DEALLOCATE (A, STAT = DeAllocateStatus)
   [IF (AllocateStatus /= 0) STOP "Memory deallocation
error"]
```

Arrays dynamic allocation -- examples

readArrayDynamically.f90 is a little program illustrating how to read an array of numbers, of unknown size, from a file. It makes use of array dynamic allocation.

1. Open the file, type: `nano readArrayDynamically.f90`
2. Compile and run, type:

   `f95 readArrayDynamically.f90 -o readArrayDynamically.exe`

   then

   `./readArrayDynamically.exe`

Do you understand what's happening?

Arrays dynamic allocation -- examples

readArrayDynamically.f90 is a little program illustrating how to read an array of numbers, of unknown size, from a file. It makes use of array dynamic allocation.

1. Open the file, type: `nano readArrayDynamically.f90`
2. Compile and run, type:

   `f95 readArrayDynamically.f90 -o readArrayDynamically.exe`

   then

   `./readArrayDynamically.exe`

Exercise

Modify it to read a 10x10 array.

Solution: read2dArrayDynamically.f90, **DON'T CHEAT!**

How can you modify it so that the write command is more reader friendly? How can you improve these programs?

Program units

All Fortran 90 programs have a main unit called PROGRAM.
Equivalent to C++' main.

```
!This is a Fortran 90 program
PROGRAM example

   … all the program's statements

END PROGRAM example
```

Other optional program units are:

- MODULES,
- SUBROUTINES,
- FUNCTIONS.

Program units

- **FUNCTIONS**, return ONE computed result via the function name, and are usually shorter than subroutines.

- **SUBROUTINES**, use them when more than ONE value is wanted as a result

- **MODULES**: collections of declarations and subprograms which can be imported into other program units,
  - may be in different .f90 file(s) than PROGRAM,
  - may contain SUBROUTINE(S) and/or FUNCTION(S).

Functions()

- Can be internal or external to the program
- Always take a prefix type (REAL, INTEGER, LOGICAL)
- Always take () at the end
- () may contain 1 to several arguments.

E.g.

Factorial

```
INTEGER FUNCTION Factorial(n)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: n
   INTEGER :: i, Ans

   Ans = 1
   DO i = 1, n
      Ans = Ans * i
   END DO
   Factorial = Ans
END FUNCTION Factorial
```

Read and return a positive

```
REAL FUNCTION GetNumber()
   IMPLICIT NONE
   REAL :: Input_Value
   DO
      WRITE(*,*) 'A positive number: '
      READ(*,*) Input_Value
      IF (Input_Value > 0.0) EXIT
      WRITE(*,*) 'ERROR. try again.'
   END DO
   GetNumber = Input_Value
END FUNCTION GetNumber
```

From http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/F90-Subprograms.pdf

INTENT() Attribute

- Control the direction of data transfer from/to functions, subroutines, modules, and the program.

- INTENT(IN) only receives a value; one we don’t want to change in the receiving program unit.

- INTENT(OUT) only returns a value; not used in internal computations.

- INTENT(INOUT) receives a value from, and returns a value to, its corresponding actual argument.

Common problems

forget function type

```
FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
   DoSomthing = SQRT(a*a + b*b)
END FUNCTION DoSomething
```

forget `INTENT(IN)` – not an error

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER :: a, b
   DoSomthing = SQRT(a*a + b*b)
END FUNCTION DoSomething
```

From http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/F90-Subprograms.pdf

Common problems

forget function type

```
FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
   DoSomthing = SQRT(a*a + b*b)
END FUNCTION DoSomething
```

forget INTENT(IN) – not an error

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER :: a, b
   DoSomthing = SQRT(a*a + b*b)
END FUNCTION DoSomething
```

change INTENT(IN) argument

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
      IF (a > b) THEN
         a = a - b
      ELSE
         a = a + b
   END IF
   DoSomthing = SQRT(a*a+b*b)
END FUNCTION DoSomething
```

forget to return a value

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
   INTEGER :: c
   c = SQRT(a*a + b*b)
END FUNCTION DoSomething
```

From http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/F90-Subprograms.pdf

Internal Functions

- Located inside the program, but always after the **CONTAINS** statement

```
PROGRAM program-name

    IMPLICIT NONE
    [specification part]
    [execution part]

CONTAINS
    [functions]

END PROGRAM program-name
```

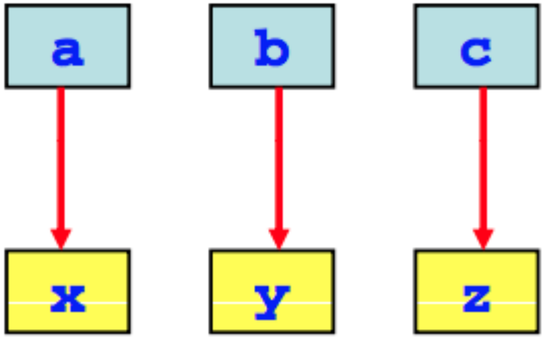
- Cannot have internal (nested) functions.

Functions()

- Arguments are variables: can be constants or expressions

```
PROGRAM EXAMPLE
    …
    WRITE(*,*) Sum(a,b,c)
    …
END PROGRAM EXAMPLE

!External function
INTEGER FUNCTION Sum(x,y,z)
    IMPLICIT NONE
    INTEGER INTENT(IN)::x y z
    …
END FUNCTION Sum
```



From http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/F90-Subprograms.pdf

Example External Function

```
program prog
    implicit none
    real :: b,c
    real, external :: f    !Informs
       ¡compiler it's a function defined
       ¡elsewhere and not a variable.
    b=2.
    c=f(b)
    write(*,*) 'c=', c
end program prog

real function f(a)  !Returns 1 value. Since
       !it's named f, the value f must be
       !set in the function.
     implicit none
     real :: a
     f = a**4          !it calculates a^4
end function f
```

Application problem

The program **taylor.f90*** uses a Taylor expansion to estimate the value of the exponential function evaluated at 1. It also compares the intrinsic Fortran exponential function vs. the Taylor estimation. This is a good exercise on mathematical error tracking and how computers and calculators actually calculate well-known math functions such as the exponential, log(x) and the trigonometric ones.

Read the source code, type: `nano taylor.f90`

Do not modify it.

Compile, type: `f95 taylor.f90 -o taylor.exe`

Then, run, type: `./taylor.exe`

Understand what's happening. Note the KIND specification.

Homework: do a similar program but for the SIN(x) function.

*From http://faculty.washington.edu/rjl/uwamath583s11/sphinx/notes/html/fortran_taylor.html#fortran-taylor

Exercise -- functions

Let’s compare 2 versions of this program.

Open taylor.f90, type:

```
nano taylor.f90
```

Open taylor.function.f90, type:

```
nano taylor.function.f90
```

1. Compare the codes. Let’s discuss.

2. Solve the 4 questions marked with a “?” in taylor.
   function.f90.

3. Compile & run both programs. Is the result the same?

Subroutines

```
PROGRAM Example
   …
   CALL name
   …

   CONTAINS
   SUBROUTINE name (x1,...,xn)
      IMPLICIT NONE
      [specification part]
      [execution part]
      [subprogram part]
   END SUBROUTINE name

END PROGRAM Example
```

Subroutines

```
PROGRAM Example
  …
  CALL name
  …

  CONTAINS
  SUBROUTINE name (x1,...,xn)
    IMPLICIT NONE
    [specification part]
    [execution part]
    [subprogram part]
  END SUBROUTINE name

END PROGRAM Example
```

- Similar to functions:
  - always take ()
  - Optional arguments

    `SUBROUTINE(x1,...,xn)`

- But:
  - SUBROUTINES do not return any value with the subroutines' name
  - SUBROUTINES DO NOT NEED A PREFIX TYPE
  - SUBROUTINES are usually larger

Two simple examples:

Am, Gm and Hm are used to return the results

```
SUBROUTINE Means(a, b, c, Am, Gm, Hm)
   IMPLICIT NONE
   REAL, INTENT(IN)  :: a, b, c
   REAL, INTENT(OUT) :: Am, Gm, Hm
   Am = (a+b+c)/3.0
   Gm = (a*b*c)**(1.0/3.0)
   Hm = 3.0/(1.0/a + 1.0/b + 1.0/c)
END SUBROUTINE Means
```

values of a and b are swapped

```
SUBROUTINE Swap(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(INOUT) :: a, b
   INTEGER :: c
   c = a
   a = b
   b = c
END SUBROUTINE Swap
```

From http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/F90-Subprograms.pdf

Subroutines -- arguments

Since a formal argument with the **INTENT(OUT)** or **INTENT(INOUT)** attribute will pass a value back to the corresponding actual argument, the *actual argument must be a variable*.

```
PROGRAM Errors
  IMPLICIT NONE
  INTEGER :: a, b, c
    ..........
  CALL Sub(1,a,b+c,(c),1+a)
    ..........
END PROGRAM Errors
```

```
SUBROUTINE Sub(u,v,w,p,q)
  IMPLICIT NONE
  INTEGER, INTENT(OUT) :: u
  INTEGER, INTENT(INOUT) :: v
  INTEGER, INTENT(IN) :: w
  INTEGER, INTENT(OUT) :: p
  INTEGER, INTENT(IN) :: q
    ..........
END SUBROUTINE Sub
```

Are these two correct or incorrect?

From http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/F90-Subprograms.pdf

Subroutines -- argunets

The number of arguments and their types must match properly.

There is no type-conversion between arguments!

```
PROGRAM Error                 SUBROUTINE ABC(p, q)
  IMPLICIT NONE                  IMPLICIT NONE
  INTEGER :: a, b                INTEGER, INTENT(IN)  :: p
  CALL ABC(a, b)                 REAL, INTENT(OUT)    :: q
  CALL ABC(a)                       ..........
CONTAINS                      END SUBROUTINE ABC
  ..........
END PROGRAM Error
```

What are the 2 problems here?

From http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/F90-Subprograms.pdf

Subroutines -- may be nested

```
SUBROUTINE outer
    REAL :: x, y, z       !global variables
    …
    CONTAINS
    SUBROUTINE inner1
        REAL :: y
        y=x+1
        …
    END SUBROUTINE inner1
    SUBROUTINE inner2
        REAL :: z
        z=x+2
        …
    END SUBROUTINE inner2
END SUBROUTINE outer
```

Example

```
PROGRAM SUBDEM
IMPLICIT NONE
REAL A,B,C,SUM,SUMSQ
  CALL INPUT( + A,B,C)
  CALL CALC(A,B,C,SUM,SUMSQ)
  CALL OUTPUT(SUM,SUMSQ)
END PROGRAM SUBDEM

SUBROUTINE INPUT(X, Y, Z)
REAL X,Y,Z
  WRITE(*,*),'ENTER THREE NUMBERS => '
  READ(*,*) X,Y,Z
END SUBROUTINE INPUT

SUBROUTINE CALC(A,B,C, SUM,SUMSQ)
REAL A,B,C,SUM,SUMSQ
  SUM = A + B + C
  SUMSQ = SUM **2
END SUBROUTINE CALC

SUBROUTINE OUTPUT(SUM,SUMSQ)
REAL SUM, SUMSQ
  WRITE(*,*) 'The sum of the numbers you entered are: ',SUM
  WRITE(*,*) 'And the square of the sum is:',SUMSQ
END SUBROUTINE OUTPUT
```

http://www.chem.ox.ac.uk/fortran/subprograms.html

Exercise -- Subroutines

Write a small program. Call it squareArray.f90. It should:

- create an array B, of 1 dimension and 4 components
- call a subroutine which:
  - receives the array B
  - creates another array F which contains the squared components of B
  - returns the F
- finally *write(*,*)* the components of the squared array onto the screen.

Q:

- what should the intent() be for the arrays B & F?
- how to make the dimensions of the received array B correct in the subroutine?

Exercise -- Solution

```
program squareArray
    implicit none
    real, dimension(4) :: b,c
    b = (/2., 3., 4., 5./)
    call f_sub(b,c)
    write(*,*) "c = ",c

contains
  subroutine f_sub(a,f)
     implicit none
     real, dimension(:), intent(in) :: a
     real, dimension(size(a)), intent(out) :: f
     f = a**2
  end subroutine f_sub
end program squareArray
```

Modules

General form:

```
module <MODULE-NAME>
…                             ! Declare variables
    [contains]
    [! Define subroutines or functions]
end module <MODULE-NAME>
```

A program can use this module in this way:

```
program <NAME>
use <MODULE-NAME>
    …                         ! Declare variables
    ! Executable statements
end program <NAME>
```

Modules -- justification

- Clean way to separate large programs into different files. E. g. using modules for: IO, computations, solving, plotting, etc.
- Can define global variables in modules to be used in several different routines
- Can define new data types to be used in several routines

file1.f90

```
PROGRAM prog
    USE modu
    USE modu2
    IMPLICIT NONE
    …
END PROGRAM prog
MODULE modu
    IMPLICIT NONE
    …
    [RETURN]
END MODULE modu
```

file2.f90

```
MODULE modu2
    IMPLICIT NONE
    …
    CONTAINS
    [SUBROUTINE1]
    [SUBROUTINE2...]
    [FUNCTIONS1]
    [RETURN]
END MODULE modu2
```

Compile:

```
f95 file2.f90 file1.f90 -o prog.exe
```

First list modules that do not use any other modules, followed by those modules that only use previously listed modules, followed by your main program. When each module is compiled, a .o file is created.

Modules -- example

```fortran
program main !main.f90
use circle_mod
implicit none

    real(kind=8) :: a

    !print pi defined in module:
    write(*,*) 'pi = ', pi

    !test area fun. from module:
    a = area(2.d0)
    write(*,*) 'area for circle &
    of radius 2: ', a

end program main
```

```fortran
module circle_mod !circle_mod.f90
implicit none
    real(kind=8) :: &
      pi = 3.141592653589793d0

contains
    real(kind=8) function area(r)
    real(kind=8), intent(in) :: r
      area = pi * r**2
    end function area

!below, may be used by other programs
    real(kind=8) &
      function circumference(r)
    real(kind=8), intent(in) :: r
      circumference = 2.d0 * pi * r
    end function circumference
end module circle_mod
```

Result:

pi = 3.14159265358979

area for a circle of radius 2: 12.5663706143592

From: http://faculty.washington.edu/rjl/uwamath583s11/sphinx/notes/html/fortran_modules.html#fortran-modules

Using Modules

```
PROGRAM Main
   USE SomeConstants
   IMPLICIT NONE
     ..........
END PROGRAM Main
```

```
MODULE SomeConstants
   IMPLICIT NONE
   REAL, PARAMETER :: PI = 3.1415926
   REAL, PARAMETER :: g = 980
   INTEGER         :: Counter
END MODULE SomeConstants
```

```
MODULE DoSomething
   USE SomeConstants, ONLY : g, Counter
   IMPLICIT NONE
CONTAINS
   SUBROUTINE Something(…)
     ……
   END SUBROUTINE Something
END MODULE DoSomething
```

PI is not available

From http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/F90-Subprograms.pdf

Modules -- privacy

- Fortran 90 allows a module to have *private* and *public* items. However, *all global entities of a module, by default, are public* (*i.e.*, visible in all other programs and modules).
- To specify public and private, do the following:

```
PUBLIC  :: name-1, name-2, …, name-n
PRIVATE :: name-1, name-2, …, name-n
```

- The **PRIVATE** statement without a name makes all entities in a module *private*. To make some entities visible, use **PUBLIC**.
- **PUBLIC** and **PRIVATE** may also be used in type specification:

```
INTEGER, PRIVATE :: Sum, Phone_Number
```

From http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/F90-Subprograms.pdf

Modules -- privacy example

- Any global entity (*e.g.*, **PARAMETER**, variable, function, subroutine, etc) can be in **PUBLIC** or **PRIVATE** statements.

```
MODULE TheForce
   IMPLICIT NONE
   INTEGER :: SkyWalker, Princess
   REAL, PRIVATE :: BlackKnight
   LOGICAL :: DeathStar
   REAL, PARAMETER :: SecretConstant = 0.123456
   PUBLIC   :: SkyWalker, Princess
   PRIVATE :: VolumeOfDeathStar
   PRIVATE :: SecretConstant
CONTAINS
   INTEGER FUNCTION VolumeOfDeathStar()
     ..........
   END FUNCTION WolumeOfDeathStar
   REAL FUNCTION WeaponPower(SomeWeapon)
     ..........
   END FUNCTION ..........
END MODULE TheForce
```

Is this public?

By default, this **PUBLIC** statement does not make much sense

From http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/F90-Subprograms.pdf

Application problem

Program efficiency & modules. It is important to understand the execution speed of a program and identify its slowest parts. **timealloc.f90*** explores how dynamic memory allocation of arrays affects performance. It also uses: the `CONTAINS` statement; a subroutine; it uses `system_clock()`, which measures computing time; it uses `random_number()`, which returns a pseudorandom number(s) from a uniform distribution within 0 <= x < 1.

1. Read the source code. Do not modify it. Compile & run. Make sure you understand what's happening.
2. Move the module into a new file, module.f90. Debug. Compile:

   f95 timealloc.f90 module.f90 -o modules.exe. Then, run. How does time compare? Which version is faster? Does it make sense?

Solution: timeallocMain.f90 & timeallocMod.f90, **DON'T CHEAT!**

*Program taken from http://flibs.sourceforge.net/timealloc.f90

References

- Many books and online sites available
  - E.g. "Modern Fortran in Practice", published by Cambridge University Press.
- Specify “...fortran **90**…” in your searches; you may get info on older distributions (e.g. 77 is popular for legacy code).
- Some useful links:
  - *http://fortranwiki.org/fortran/show/Fortran+Wiki*
  - Reference card, **PRINT IT**!,

http://www.pa.msu.edu/~duxbury/courses/phy480/fortran90_refcard.pdf

  - The NAG libraries have hundreds of routines (may be commercial?) for science and engineering applications

    http://www.nag.com/numeric/fl/FLdescription.asp
  - *http://www.lahey.com/other.htm*, many fortran links
  - *http://flibs.sourceforge.net/**examples_modern_fortran**.html*

Thank you

- Please fill the course assessment forms, *just click the home icon in your browser*

- Email yourselves the **slides** from
  /share/apps/tutorials/intro2f90_2.pdf

- Upper menu > System > **Log Out** hpc_user...

Upcoming training events at CACDS

Introduction to C++ Programming I	February 27, 2015 10:00 AM - 12:30 PM
Introduction to C++ Programming II	March 3, 2015 10:00 AM - 12:30 PM
Intel Xeon Phi Programming I	February 24, 2015 2:00 - 4:00 PM
Intel Xeon Phi Programming II	February 27, 2015 2:00 - 4:00 PM

Introduction to **MPI programming with Fortran 90**, this March.

Advanced topics
(not covered here)

Much more to say about Fortran 90:

- Variable attributes: parameter, intent, save, public, private, etc.,
- Variable kind,
- Namelists,
- Pointers & the NULL() function,
- Linked list operations,
- Interfacing with C, MPI, openMP, CUDA, etc.,

Beyond the scope of this introductory course.

Example on OOP applications

Example of how classes and objects can be used in Fortran 90.

http://fortranwiki.org/fortran/show/Object-oriented+programming2

Note the use of:

- the construct `TYPE`*, to define classes,*
- the character `%`, ”equivalent” to C++’ periods.

Getting started

1. Login: hpc_user{1-47}, follow your seats' order

   Password=cacds2014

2. Go to *Applications > Systems tools > Terminal*

   Type the following 3 commands, and press ENTER after each one:

   include period

```
cp  /share/apps/tutorials/intro2f90.zip  .
unzip  intro2f90.zip
cd  intro2f90
```

3. See the slides, type:

```
firefox  intro2f90_2.pdf
```

   and press ENTER.

The Nano Text Editor

^X means ``hold down the CTRL key and press the x key''. Commands listed at the bottom of your screen.

To edit a file called *filename*, type `nano` *filename*.

^O	save contents without exiting (you will be prompted for a file to save to)
^X	exit nano (you will be prompted to save your file if you haven't)
^T	when saving a file, opens a browser that allows you to select a file name from a list of files and directories

^A	move to beginning of line
^E	move to end of line
^Y	move down a page
^V	move up a page
^_	move to a specific line (**^_^V** moves to the top of the file, **^_^Y** to the bottom)
^C	find out what line the cursor is currently on
^W	search for some text.

When searching, you will be prompted for the text to search for. It searches from the current cursor position, wrapping back up to the top if necessary.

^D	delete character currently under the cursor
BackSpace	delete character currently in front of the cursor
^K	delete entire line
^	search for (and replace) a string of characters