Version 2014-1
as of 1 February 2014

Introduction to Scientific Programming

Control Constructs and Arrays

Fortran Part 2

- Control constructs
  - `if, else if, else, endif`
  - `select case, case, end select`
  - `do, enddo, cycle, exit`
- Arrays
  - static arrays, dimension
- Subprograms
  - Functions and Subroutines

- Full story
  - Control constructs
  - Arrays
  - Structures
  - Subprograms: Functions and Subroutines

IF

```
[label0:] if (logical expression) then
                {if-block}
             [else if (logical expression) then
                {else-if-block} ]
             [else
                {else block} ]
             end if [label0]
```

- Use labels in complicated (and/or long) if or do constructs
  In case that an **endif** or **enddo** is missing, the compiler will be able to tell which one is missing
- Always indent code in **if** and **do** constructs

IF

optional label

```
real    :: x
complex :: root

imag: if (x < 0.0) then
  root=cmplx(0.0,sqrt(-x))
else
  root=cmplx(sqrt(x),0.0)
end if imag
```

```
integer :: n, factorial

if (n < 0) then
  print "n should be
positive"
  exit
else if (n == 0) then
  factorial=1
else if (n >= 1) then
  ...
end if
```

SELECT CASE

```
[label:] select case (expression)
           [case selector
              block]
           [case default
              block]
           end select [label]
```

- **`expression`** may be integer or character (or logical)
- **`selector`** is list of non-overlapping values

SELECT CASE

```
select case (n)
case (:-1)      ! Range from smallest integer to -1
   print*, "n should be positive"
   exit
case (0:)       ! Range from 0 to largest integer
   factorial=1
   <factorial code>
end select
```

DO

```
[label:] do variable=expr1, expr2[, expr3]
              block
         end do [label]
```

`variable` is a scalar integer variable
`expr1`, **`expr2`** & **`expr3`** are integer expressions

DO

```
factorial=1
do i=2,n
  factorial=factorial*i
end do
```

- when **n**<=1, loop is not executed
- lower and upper limits could be any valid expression that evaluates to an integer

stride of 3.
stride=1, if not specified

```
dotp = 0.0
do i=1,n,3
   dotp = dotp+a(i)   * b(i)
   dotp = dotp+a(i+1)* b(i+1)
   dotp = dotp+a(i+2)* b(i+2)
end do
```

- this is an example of loop unrolling
- assumes **n** is divisible by 3

Arrays

```
program average

real :: a, b, c, sum, average

a = 3.4
b = 4.5
c = 5.6

sum     = a + b + c
average = sum / 3.

print *, sum, average

end program
```

Output:

```
 13.5000, 4.500000
```

This works to calculate the average of 3 variables (**a, b, c**), but …

How would we deal with an example that has more than 3 variables, say 100?

Arrays

```
program average

integer, parameter :: n = 3
real, dimension(n) :: a

a(1) = 3.4      ! Data from somewhere
a(2) = 4.5
a(3) = 5.6

sum = 0.        ! Initialize sum = 0

do i=1, n
  sum = sum + a(i)          ! Add up
enddo

average = sum / real(n)   ! Divide

print *, sum, average

end program
```

An array is a group of variables (or constants), all of the same type, that are referred to by single name. An individual value within the array is called an array element; it is identified by the name of the array together with a subscript.

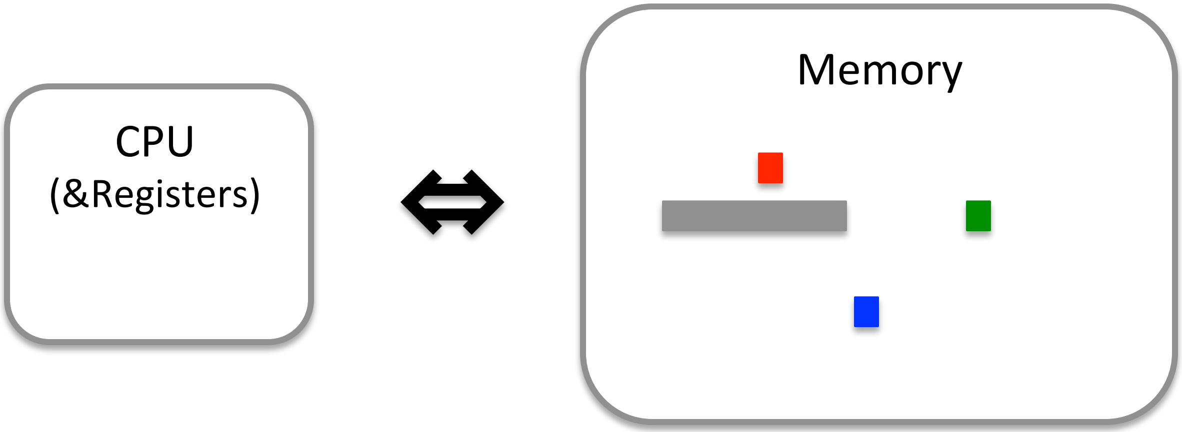
```
type, dimension(<constant>) :: var
```

a is an array of type real
a(1) is the first array element
a(3) is the third array element

Output:

```
 13.5000, 4.500000
```

Computer Architecture: Memory



Variables and arrays are stored somewhere:
a, b, c scalar variables
d array variable, contiguous in memory

Arrays

What is the most important segment in this code?

```
program average_square

integer, parameter :: m = 30
real, dimension(m) :: squares

! Read from keyboard
print *, 'Enter the maximum number'
read  *, n_squares
print *, 'Input is', n_squares

if (n_squares > m) then
  stop 'Error'
endif

! Calculate and store squares
do i=1, n_squares
  squares(i) = real(i)**2
enddo

! Calculate average
sum = 0.
do i=1, n_squares
  sum = sum + squares(i)
enddo
aver = sum / real(n_squares)
print *, 'average = ', aver
end program
```

Multi-Dimensional Arrays

In this example, **`a`** is a 2D array with 3x7 elements, and **`b`** is a 3D array with 2x4x8 elements

```
program average

integer, parameter        :: n = 3
integer, parameter        :: m = 7

real, dimension(n,m)      :: a
real, dimension(2,4,8)    :: b

…

end program
```

Arrays

```
integer, dimension(0:7,2) :: indArray
...
indArray(myTaskID,1) = myStart
indArray(myTaskID,2) = myEnd
```

indArray's size	16
indArray's rank	2
extent along 1st dimension	8
indArray's shape	(8,2)

- Ordered collection of elements
- Each element has an index
- Index may start at any integer number, not only 1
- Array element may be of intrinsic or derived type
- Array ***size*** refers to the number of elements
- The number of dimensions is the ***rank***
- The size along a dimension is called an ***extent***
- Array ***shape*** is the sequence of extents

Derived Data Types and Structures

```
type person       ! Declaration of a
       ! derived type
real                 :: age
integer              :: year_of_birth
character(len=8)     :: name
end type person

! Declaration of a structure of
! the derived type
type(person)                  :: you

! A structure can be an array
type(person), dimension(10) :: we

! Elements are references by %
you%age              = 29.2
you%year_of_birth = 1990
you%name             = 'John Doe'

print *, 'age = ', you%age
print *, 'you = ', you
```

A Derived Type is similar to an array. Like an array, a single derived type can have multiple components. Unlike an array, the components of a derived type may have different types. One component may be an integer (array), while the next component is a real (array), the next a character (array), and so forth.

Components are accessed with percent (%)

`structure%component`

```
Output:
 age =     29.20000
 you =     29.20000     1990 John Doe
```

Derived Data Types and Structures

```
…
! more examples of how to use structures

we(1)%age               = 'John Doe'
we(1)%age               = you%age
we(1)%year_of_birth = you%year_of_birth
we(1)%name              = you%name

we(2)  = you    ! copies all components

we(3)%age = 1.5 * we(3)%age
```

Derived Data Types - Structures

- Composed of one or more components
- Components may also be arrays or of a derived type

```
! Declaration of the type
type particle
   integer                 :: partID
   real, dimension(3) :: pos, vel, force
end type particle

! Declaration of variables of a type
type(particle) :: p1, p2

! Usage
p1%partID = 1
p1%pos     = 0.0; p1%vel = 0.0; p1%force = 0.0
p1%pos(1) = 1.5
p2 = p1; p2%partID = 2
```

Derived Data Types and Structures

```
! structures can contain arrays
! and other structures

type data
  integer :: nx, ny, nz                 ! Number of points in X, Y and Z
  real, dimension(10,10,10) :: value    ! 3D array
end type data

type(data) :: set1, &
              set2

type more_data
  real, dimension(20,20) :: y
  type(data)             :: d
end type more_data

type(more_data) :: more

set1%nx = 5; set2%value(1,1,1) = 7.3

more%y(1,1) = 5.
more%d%nx = 7
more%d%value(2,4,5) = 7.3
```

We gratefully acknowledge the sponsorship of Chevron Corporation, whose generous support of TACC has made possible this Scientific Computing Curriculum and other student-focused initiatives.

License