Quick Guide to FORTRAN 90

Sources:

a) Programmer's Guide to Fortran 90, Brainerd, Goldberg, Adams, Springer 1996, 445p

b) Fortran 90: A Reference Guide. Chamberland; Prentice Hall 1995

General Remarks

• Short sample program illustrating some features:

```
program quote
  implicit none
  integer  :: n,m
  n = 0;  m = n + 1 ! multiple commands in same line
  print *, n, m     ! default format for print
  read  *, v
  call listing
  if (m > n) m = n
  m = n + m**2 &    ! continuation of line
      m + n**2

  contains
  subroutine listing
    ...
  end subroutine listing
end program quote   ! Always write the name
```

• Use of lowercase is allowed. In fact it is good style.

• Free form source: No specified columns. Comments begin with `!`. To continue line use `&`. This is the default if your file has extension `.f90`

• Command `implicit none` means that compiler does not assume type from the name of the variable like before (i,j,k integer, x,y,z reals). Should put this in every program. It's enough to have this declaration just once in a file.

• Multiple statements in the same line, separated by `;`

• The command `stop ["message"]` aborts the execution of the program and displays the `"message"`. Should be used only in error routines.

• The command `include "name"` includes a file in the program.

Structure of the Code

Here is a form of the program. The order of commands that appear below is obligatory.

```
program/function/subroutine/module  name_
[use name_]

implicit none

(parameter statments)
(type declaration/variables/interface blocks)
(code)

[contains]

(internal procedures)

end program/function/subroutine/module  name_
```

Control Commands

• At the `end` command always repeat the name of the structure being finished. Ex: `end program quote`

```
if (a==b) then
   ....
else if ( .... ) then
   ....
end if

if (a>b) a = b ! No then needed here

label:if (..) then
  ...
end if label   ! if with label

[label:] select case (dice)
  case (2:3,12)
    print *, " ..."
  case (7,11)
  case default
end select [label]
```

• Do is the only looping construct in Fortran. Three forms:

a) No loop control. Do forever, leaves only with `exit` command. There are two ways of jumping inside loop:

• `exit` leaves the innermost loop. With a label it's possible to exit all loops nested in.

• `cycle` terminates one iteration. Return to beginning of loop. With label can return to the beginning of [label] loop.

```
[label:] do
  if (..) cycle [label]  ! go to do
  if (..) exit [label]   ! go to end do
end do [label]
```

b) With a do variable. It's not recommended to use a real variable in the loop

```
do n=1,20      or   do n=1,20,2
end do              end do
n=1,2,3 ... 20      n=1,3,5 .. 19
```

c) Do while loop

```
do while (ok)
  ...
end do
```

Procedures

There as two kinds of procedures: `function` and `subroutine`. They're classified as internal (contained in a program or in another procedure); module (contained in a module) and external (old standard, don't use it!). Internal procedures cannot be nested. Always repeat name of procedure at the `end` command.

• The statement `return` finishes the execution of a procedure. Useful, sometimes, in error conditions but should be avoided in general.

```
program bla
  implicit none
  real :: x,y

  call read_all
```

```
  call print_all

  contains   ! now follows procedures

  subroutine read_all
    x = x + y              ! notice scope of x,y here
  end subroutine read_all

  subroutine swap (a,b)  ! with arguments
    real, intent(inout) :: a,b
    real :: temp
    temp = a; a=b; b=temp
  end subroutine swap
end program bla
```

• Functions can return one value or array/structure. There are two forms, one with, the other without the keyword `result`. You need to use it when reference to the current value of the function is made on the right hand side but it is not a recursive call. You can achieve the same result defining another variable to hold the intermediate result. But in recursive call it is obligatory.

```
real function f(x)
  real :: x
  f = x - 2*x*x
end function f

! Here we need result keyword
function factorial(n) result(f_res)
  integer :: n, f_res, i
  f_res = 1
  do i = 1, n
    f_res = n*f_res
  enddo
```

• Do not change the values of dummy variables in a function so as to have side effects. Only subroutines should have side effects. This can confound the compiler.

• To write a recursive procedure you need to declare it `recursive`. For recursive function it is obligatory to use `result`.

```
recursive function fac(n) result(fac_r)
fac_r = n*fac(n-1)
end function fac
```

• When procedures are internal to a program, another procedure or they're in a module, they're preceded by `contains` statement.

```
module f_mod
implicit none

contains

real function f(x)
  real :: x
  ...
  contains

    function g(x) ! this is internal to f
     ...
    end function g
end functions f

end module f_mod
```

Argument Passing

More complicated with new features of F90: Optional argument, keyword arg., generic procedure and assumed shape arrays

• Keyword argument. Here it does not matter the order of the arguments, but you have to give the name of each argument. For example:

```
series(d=0.1, m=400, n=700)
```

• Attribute `optional`: Useful when using keyword argument only. You can omit some arguments and assume default.

```
integer function f(m,n)
integer, optional :: m
integer :: n, temp_m

if present(m) then ! test if m was given
  temp_m = m
else
  temp_m = 0  ! default value for m
end if
...
end function f
```

• Attribute `intent`. Can be `in` (by value), `out`, `inout` (by reference). All arguments should have an intent attribute to facilitate optimization and error checking.

```
subroutine compute(m,n)
  integer, intent(in) :: m
  integer, intent(out) :: n
```

• Procedures can be passed as arguments also, but only intrinsic and module procedures. We need to declare intrinsic procedures with the keyword `intrinsic` (see below), but for modules/external ones we don't need it.

```
program integrate
  implicit none
  intrinsic sin

  print *, int(sin, a=0.0, b=3.14, n=100)

  contains

  real functions int(f,a,b,n)
    real :: f
    real, intent(in) :: a,b
    integer, intent(in) :: n
    ......
    sum = sum + f(a);
  ...
```

• Attribute `save`. Save the value of a variable between calls. Not necessary when variable is initialized.

```
  subroutine count
    integer [,save] :: n = 0
    n=n+1
  end subroutine
```

Variables (General)

• New types of variables with initialization. Besides the ones below we have `double precision` but since it is outdated is should be replaced by `real` with `kind` parameter.

```
integer :: x,y
real    :: w1 = 0  ! initialization
complex :: z
logical :: ok = .true.
character :: c="s"
```

• The attributes for variables are:

```
allocatable, dimension, intent,
intrinsic, optional, parameter, pointer,
private, public, save, target, external
```

They can also be used as a command, but this is not recommended.

```
real a
dimension a(10)
intent(in) a
```

• Definition of constants is done with modifier `parameter`. The value cannot be changed. It is good a practice to put whenever possible because can improve performance.

```
real, parameter :: pi = 3.1415
```

Real/Integer

• Kind parameters for `real` and `integer`

```
 real (kind=2) :: x,y
```

The codes of kind (0,1 ...) are machine dependent and they mean the precision (single/double for reals or max number for integer). To determine the kind independent of machine use

```
  selected_real_kind(8,70)
  selected_int_kind(r)
```

The first one returns kind of real with 8 digits and between -10^{70}, $+10^{70}$. The other one returns kind type of an integer between -10^r, $+10^r$.

• Sometimes need to use cast to convert types explicitly. Keywords are: `int`, `real`, `cmplx`. Whenever needed use it.

```
i= int(r); mean = real(sum)/n;
```

Logical

• Logical variables and operators: `.true.` `.false.` `.not.` `.or.` `logical :: ok`

• Alternate symbols for relational operators. Instead of `.ge.` etc. can use `<`, `>`, `<=` etc. Notice that `.ne.` `.eq.` are, respectively, `/=`, `==`.

Character

Can do assignments, comparisons, concatenations and select substring using same notation as for matrix

```
  character (len=7) :: s
  character (len = *), parameter :: &
     message = "Hello World"   ! length given by string
  print *, message
  len("xxxx");  s = "hello" ! assignment
  word="...:
  plural = trim(word) // "s" ! concatenation
  s(2:3) = "xx"   ! select substring
```

Arrays

• Declaration

```
real, dimension (1:9) :: a            ! 1-d array
real, dimension (0:10,0:10,1:30) :: a ! 3-d array
real, dimension (10) :: a   ! 1:10 as subscript
```

• Assumed shape arrays. The size is assumed from argument given to procedure.

```
subroutine s(a)
  real, dimension (:),intent(in) :: a

  do i=1,size(a)
end subroutine
```

• Allocatable (dynamic) arrays. Array is not given dimensions and is declared with `allocatable` attribute. The dimensions are established later, with `allocate` statement.

```
  integer, dimension(:), allocatable :: a
  integer :: status
  allocate (a(20),stat = status)
  if (status > 0) then ! Error during allocation
  ...
  deallocate(a)  ! free memory
```

• Automatic arrays. Declaration of array may use values from other dummy arguments. Can be used only in procedures. In the main program arrays are either declared with constant bounds or are declared `allocatable`.

```
subroutine s(d_list,n, a)
  real,dimension(:) :: d_list
  real,dimension(n,n) :: a
  real,dimension(size(d_list)) :: local_list
  real,dimension(2*n+1) :: longer_list

end subroutine
```

• Array Constructor. There are three forms

```
x(1:4) = (/ 1.2, 3.5, 1.1, 1.5 /) ! scalar expression
x(1:4) = (/ a(i,1:2), a(i+1,2:3) /) ! array exp.
x(1:4) = (/ (sqrt(real(i)),i=1,4) /)! implied do list
```

The rank is always one, but we can use `reshape]` command.

```
\reshape( (/ 1, 2, 3 /), (/ 2, 3 /) ) ! 2 X 3 array
```

• Implied do list. List of expressions followed by iterative control.

```
(sqrt(real(i)),i=1,4)
((a(i,j), i=1,4), j=1,4) ! note the double loop
print *, (a(i,i), i=1,n)
```

• Arrays sections. Use a subscript triplet [l]:[u] [:s], where l is lower bound, u is upper bound and s is stride. If omited default is assumed (1).

```
  a(2:5) = 1.0 ! a(2)=1.0 ... a(5) = 1.0
  v(0:4)  ! v(0) .. v(4)
  v(0:4:2) ! v(0), v(2), v(4)
  v(:)   ! all elements
  v(::2) ! all elements with stride 2
  b(1:4:3, 6:8:2, 3) =
! b(1,6,3)  b(1,8,3)
! b(4,6,3)  b(4,8,3)
```

• Vector Subscript can be used to select elements.

```
  iv = (/ 3, 7, 2 /)
  v(iv)  ! v(3), v(7), v(2)
  b(8:9, 5, (/4, 5, 2/))
! b(8,5,4) b(8,5,5) b(8,5,2)
! b(9,5,4) b(9,5,5) b(9,5,2)
```

• Array Assignment. Permitted when shape left = shape right or right is a scalar.

```
a = 0.0  ! array a = 0
a(2:4,5:8) = b(3:5,1:4)
```

• Where statement. Assign values only to elements of array where condition is true. Within `where` only assignments are permitted. Cannot be nested.

```
  where (a < 0) b = 0
  where (c /= 0)
    a = b/c
  elsewhere
    a = 0
    c = 1
  end where
```

• Intrinsic Operations. All intrinsic operations can be applied to arrays. Ex: `abs(a(k:n,l))` result in one dimensional array with $n - k + 1$ nonnegative values.

```
  a = a*b  ! multiply element by element
           ! to multiply matrices use matmul
  a = a**2  ! square each element
  a(k,k:n+1) = a(k,k:n+1)/pivot
  a = a+b; a = a-b
```

• Element Renumbering, After operation arrays no longer have same subscripts. The subscript start at 1.

```
integer, dimension (0:6), &
   parameter :: v = (/ 3, 7, 0 /)
   maxloc(v) ! result is (/ 2 /) and not (/ 1 /)
```

• Array functions. See appendices for all list but there is matrix multiplication, transpose, maximal element, sum of elements, etc. Following we have some examples:

```
norm = maxval(abs(a)) ! norm is a scalar


subroutine search (a)
  integer, dimension(:),intent(in) :: a
  integer :: i
  do i=1, size(a) ! dimension of a
  ...
  end do
end subroutine search
```

One can search using `any` (just as in matlab).

```
  found = any(a(1:n) == 3.0)
```

Any is like `.or.` to arrays.

```
! linear systems
real, dimension (size(b), size(b) + 1) :: m
m = size(b)
m(1:n,1:n) = a  ! matrix
m(1:n, n+1) = b ! lhs
```

Suppose we have `m = (/12 13 14 /)` then

```
spread(m,1,2) = 12 13 14  ! 2 X 3 matrix
                12 13 14
```

Pointer

Once pointing to an existing variable it acts as an alias. The other way to use is to allocate memory and point there. To make a variable a target of a pointer need to use `target` attribute.

```
  real, pointer :: p
  real, target :: r ! to be pointed to
  p => r   ! pointer assignment

  real, dimension(:), pointer :: v
  real, dimension(:,:), target :: a
  v => a(4,:) ! point to row 4 of a

  print *, v  ! once pointing is like an alias
              ! same as print *, a(4,:)
  v = 0       ! set fourth row to 0
  allocate(p1 [,stat=alloc_stat]) ! create space
  deallocate(p1)    ! free space
  nullify(p1)       ! p1 = null_pointer
  associated(p1,p2) ! test equality of pointers
```

Structures and Derived Types

To define the type use:

```
  type phone_type
    integer :: area_code, number
  end type
```

To declare a variable of the new type:
   `type (phone_type) :: phone`
To make reference to components: `phone%number = 10`
Can initialize using structure constructor:
   `phone = phone_type(812,8571)`
Can assign the whole structure: `phone1 = phone2`

Modules

Nonexecutable program unit that contains data objects, declarations, derived-type definitions and interface blocks. All this that can be used by other program units with `use` command. Replaces the external procedures of FORTRAN 77. Makes more transparent use of routines, types and data shared by different units, including the main program. Also makes possible for compiler to know prototypes of procedures, to check parameters and make use of optional, keyword argument, argument intent etc.

```
module spherical
  integer,parameters :: N =100, M=50
(type declaration/variables/interface blocks)
[contains]
(internal procedures)
end module

program mani
  use spherical
  ...
end program mani

program p
contains
  subroutine s
    use spherical ! visible only from here
  end subroutine s
end program p
```

• `private, public` statements. Hides or turns public names from module. Default is public. One can use private statement to turn all names, by default, to private. Then need public attribute for each name to be made public.

```
module vector
  private ! now is the default
  public :: mult_mat, vect_prod
  ...
end module vector
```

• `use` statement. Makes visible all data and procedures of a module, unless you have used `private` statement to hide some data/procedures. Depending on where you put the declaration the scope is going to be the whole program or just of one procedure. Use must be the first statement of a program unit. The general form is `use name_of_module`

You can change one name when using a module with the declaration:

```
use spherical, nr_of_unknows => M ! use nr_of_unknows
                                  ! instead of M
```

Also you can specify the only name needed with `only`

```
use spherical,only :: N  ! N  only is accessible
```

External Procedures

Before F90 this was the only kind of procedure. Now they're not necessary. Use modules instead. External procedures can be compiled separately from other parts or kept in one file and compiled together. These same options are available for module procedures.

In some circumstances it is necessary to declare that a procedure is external with `external`. For example if an external procedure has the same name as a intrinsic one (in Fortran 90 there a lot of new intrinsic procedures).

It is possible to declare prototype of external procedures with `interface` command in the calling program unit. Then compiler can check parameters and make use of keyword arguments and other facilities. See the nongeneric form of the `interface` command. Another way is to declare the procedures using `external`

Interface Command

The general form is:

```
interface [name]
  .....
end interface
```

There are two basic types of declaration, one with `[name]` and one without it.

a) Without `name` Specify explicit interface for an external or dummy procedure (used as an argument to be passed on). Also called nongeneric interface block. This will be in the calling unit and are to be used with external procedures. Other alternative is to use `external` command. Here you cannot use `module procedure` command.

```
interface
  real function v(r,s)
    real :: r, s
  end function v
  real function area(h)
    real :: h
  end function area
end interface
```

b) With `name`. This is typically used in modules. There are different uses of this form.

b1) Generic procedures. Specify single name for all procedures in interface block. At most one specific procedures is invoked each time. Compiler determine which one by the type of the variables. When you use `sqrt` function, for example, the compiler determine by the type (real, double, complex, etc.) which routine to call. Observe the use of the `module procedure` command.

```
module swap_module
interface swap
  module procedure swap_reals, swap_integers
end interface
contains
subroutine swap_reals(a,b)
...
subroutine swap_integers(a,b)
...
end module swap_module
```

b2) User Defined Operator. We can define a new unary/binary (depending on the number of arguments) operator. After this we use just like one. For example `print *, .det. M`. Only functions can be in the interface block here.

```
interface operator (.det.)
  module procedure  determinant
end interface
```

b3) Extending Operator. Can extend meaning of usual operators `+, -, *, /` etc. In the example below one can use `+` instead of `.or` for logical variables. Only functions can be in the interface block here.

```
interface operator (+)
   module procedure logical_plus_logical
end interface
```

b4) Extending Intrinsic Functions. We can extend the meaning of intrinsic functions.

```
interface sqrt  ! extending for integer
  module procedure sqrt_int
end interface
```

b5) Extending Assignments. Here we need to associate with a subroutine.

```
interface assignment(=)
  module procedure integer_gets_logical
end interface
subroutine integer_gets_logical(i,l)
  integer,intent(out) :: i
  integer,intent(in) :: l
  ...
```

Format Specifiers

• Here we discuss formatting for `read`, `write` and `print` commands. We don't need `format` command anymore. Now the formatting is done directly in the input/output commands. There are two forms

a) List-directed formatting. This is also called the default formatting or free formatting. It is indicated with an *. This should be used whenever possible because is easy and usually gives good results.

```
print *, a,b  ! a and b are variables
```

b) Format-directed formatting. Here you specify in detail the formatting. The general form is "format-item", where:
format-item = [r]data-edit-desc or control-edit-desc or [r](format-item-list)

The "r" is a repeat specification integer.

data-edit-desc

Descriptor	Data Type
`i` w[.m]	decimal integer
`b` w[.m]	binary integer
`o` w[.m]	octal integer
`z` w[.m]	hexadecimal integer
`f` w.d	real, without exponent
`e` w.d [`e` h]	real, exponential form
`es` w.d [`e` h]	real, scientific form
`en` w.d [`e` h]	real, engineering form
`l` w	logical
`a` [w]	character

The meaning of the letters are: w = width of field, m = digits, d = digits to the right of decimal point and h = digits in exponent

control-edit-desc

Descriptor	Function
`t` n	tab to position n
`tl` n	tab left n positions
`tr` n	tab right n positions
`[r]/`	newline
`:`	stop formatting if list exhausted
`s`	same as `ss`
`sp`	print plus sign
`ss`	suppress plus sign
`bn`	blanks = null (ignore) in numeric input fields
`bz`	blanks = 0 in numeric input fields

• For `complex` type it is required two reals edit-descriptors.
Some examples:

```
print "(e8.3)", x
print "(f5.1, a, i4, /, 3(i4))", x, " and ", n, v
! Here we assume v is a integer vector with 3 elements.
print "(3i2)", 2, 3, 4
print "(a,e10.3)", "The answer is ", x
print "(2f7.1)", z ! z is complex
```

• One can put format in a string:

```
fmt = "(f5.1,a,i4)"
print fmt, x, " and ", n
```

• When inputing values for `read` command separate them by comma or blanks. Ex: `read *, a, b, c`. The user should type `1 2 3` or `1,2,3`. It is good practice to use the default formatting because it is usually tolerant of variations in alignment and user-friendly.

Obsolete Features

• Numeric labels

```
DO 10 i =1. 100
10 a(i,j) = i+j
```

• Arithmetic `IF`: `IF (x-y) 100, 200, 300`
• Alternate return specifier. Recommended: evaluate return code in a case statement: `CALL SUB(A,B,C,*10,*20,*30)`
• Real `DO` variable
• Shared loop terminate (typically with a `RETURN` statement). Recommended: Use `end do` for each loop.
• `GOTO [label]`
• Computed goto. `GOTO (100, 200, 300) I` Replaced by case
• `ASSIGN 200 TO I   ...   GOTO  I`
• `PAUSE` Use dummy `read`.
• `DATA` Statement. Replaced by initialized type statements.
• Double precision data type. Use kind statement
• `COMMON` Blocks. Replaced by modules
• Some format specifiers: `g` (replaced by `*`), `d` (similar to `e`), `x` (same as `tr`) and `p` (tricky to use, give bad results).
• Statements: `EQUIVALENCE, ENTRY, FORMAT, CONTINUE DOUBLE PRECISION, IMPLICIT REAL (a-z)`