

Software design using Fortran 95

The convenience of MATLAB, the speed of a Porche

Jonas Jusélius

27.04.2005



Introduction

F95 should be considered a completely new language, not as a more advanced F77 with memory allocation!

Fortran 90/95 is tailored for writing mathematical codes. For the F77 programmer it offers much more flexibility and structure. For the C programmer it offers a decent and compact programming language.

Why bother? Programming languages should help the programmer to express the problem at hand in a clear and structured way. F95 is a huge leap in the right direction compared to F77. For C programmers F95 offers a more mathematical syntax and better OOP support through overloading.

Major features of F95

Module-Oriented Programming Modules are the single most important addition to the Fortran standard. Using modules one can practice module-oriented programming; a primitive relative of OO programming.

- Interfaces
- User defined types
- Data hiding
- Procedure and operator overloading

Syntax F95 introduces many convenient syntactic elements familiar from languages like MATLAB. A clear syntax helps readability, speeds up development, and reduces the number of bugs.

Dynamic Memory Allocation While the rest of the world got dynamic memory allocation in the seventies, Fortran got in in 1990. DMA also enables the use of range checking to ensure that no array boundaries are violated.

Strict type checking Avoids passing the wrong types in the wrong order to subroutines.

Syntax

Arrays in F95 have operators and plenty of intrinsic functions.

```
real(8), dimension(100,100) :: a, b
real(8), dimension(10,10) :: c
integer(4), dimension(3) :: r, v
integer(4) :: r2

a=0.d0; b=42.d0      ! initialize arrays or vectors

c = a(1:10, 90:100) + b(42:52, 67:77) ! slicing
a = matmul(transpose(a),b)
a = a*b      ! element by element multiply
a = 2.d0*a+b ! daxpy

b(1,1) = dot_product(a(1,:), b(:,5))
r2 = sum(r**2)
r2 = sum(r**v)
r = r-v
```

Syntax

The **case** statement provides a special if-else construct which can be both faster and clearer than a traditional if-else construct.

```
character (3) :: foo
select case (foo)
  case ( 'x ' )
    ...
  case ( 'bar ' )
    ...
  case default
    ...
end select
```

Syntax

The **where** statement performs actions on all array elements satisfying some condition.

```
real(8), dimension(100, 1000) :: a, b
where ( a < 0.d0 )
    a = abs(a)
else where
    b = -a/42.d0
end where
```

Syntax

The **forall** statement is a compact way of looping over arrays

```
real(8), dimension(100, 100) :: a
integer(4) :: i, j

forall ( i=1:100, j=1:100 )
    a(i,j) = i*j+i+j
end forall
```

Syntax

F95 adds a number of general syntax enhancements

Exponentiation **

Comparison < > >= <= == /=

Logical .and. .or. .not. .eqv. .neqv.

Boolean .true. .false.

Comments !

Line continuations &

Free-form syntax, no more 6 spaces before code, lines can be 132 characters long (formerly 72). Tabs are "allowed".

Long variable names (formerly 6)

Data types complex and logical

Precision selected precision using the KIND specifier

Pointers

Pointers provide aliases to types. Pointers can be also be used for limited data hiding in conjunction with modules and user defined types.

```
real(8), dimension(42), target :: foo
real(8), dimension(:), pointer :: bar
bar=>foo
if ( associated(bar) ) bar=42.d0 ! check if pointer is connected
nullify(bar) ! disconnect pointer
```

Dynamic memory allocation

```
real(8), dimension(:, :), allocatable :: aa
real(8), dimension(:), pointer :: ap !pointers are also allocatable

integer(4) :: n
integer(4), dimension(2) :: nn
allocate( aa(1000, 1000) )
allocate( ap(1000) )

! arrays know their shape and size
n=size(aa)          ! 1000000
nn=shape(aa)       ! (/ 1000, 1000 /)

if ( allocated(aa) ) deallocate(aa)
if ( associated(aa) ) deallocate(ap)
```

Subroutines and stating your intent

The **intent** keyword marks and enforces the intended use of procedure arguments.

```
subroutine foobar(a, b, c, d)
  real(8), dimension(:, :), intent(in) :: a    ! no explicit
  real(8), dimension(:), intent(inout) :: b    ! dimensions
  real(8), intent(out) :: c
  integer(4), optional :: d

  do i=1, size(a(1, :)) ! query the array for its size
    b = a(1, :)*b
  end do
  c = sum(a)
  if present(d) then ! check if 'd' was given as an argument
    d = c**2
  end if
end subroutine
```

Functions

Functions always return a type, or a pointer to a type. Function return values cannot be discarded.

```
function foo(a,n) result(r)
  real(8), intent(in) :: a
  integer(4), intent(in) :: n
  integer(4), dimension(n) :: r

  r = 2.d0*a
end function

function foo2(a) result(r)
  real(8), intent(in) :: a
  integer(4), dimension(:), pointer :: r

  allocate(r(10))
  r = int(2.d0*a)    ! cast to int, also dble()
end function
integer(4), dimension(:), pointer :: q
q=>foo2()
```

Recursive functions

```
recursive function bar(a) result(r)
  integer(4), intent(in) :: a
  integer(4) :: r
  a=a+1
  if (a > 100) then
    r=a
  else
    r=bar(a)
  end if
end function
```

Modules

Modules are organizational units which allow for information hiding, function overloading and operator definition.

```
module foo_m ! note the _m convention
  implicit none ! only once per module
  public get_counter, addto_counter ! visible members
  private ! everything else is hidden
  integer(4) :: counter
contains
  function get_counter() result(r)
    integer(4) :: r
    call bar(1)
    r=counter
  end function

  subroutine addto_counter(i)
    integer(4), intent(in) :: i
    counter=counter+i
  end subroutine
end module foo_m
```

User defined types

Types wrap related data into a logical units:

```
type atom_t      ! note the _t convention
  character(3) :: symbol ! Unq, Unp...
  real(8), dimension (:), pointer :: coord
  real(8) :: charge
  type(aobasis_t), pointer :: basis
end type
```

```
type(atom_t) :: a
type(aobasis_t), target :: a_basis
```

```
a%symbol='Au'
a%charge=79.d0
allocate (a%coord(3))
a%coord=(/0.0,0.0,0.0/)
a%basis=>a_basis
```

Interfaces

Interfaces is the Swiss-Army-Knife feature of F95. Interfaces enables safe linking between F95 and C and F77. They also provide the means to do procedure and operator overloading, and they can be used to achieve inheritance and polymorphism.

Interface for linking with external methods (C or F77)

```
interface  
  subroutine foo(a,b)  
    real(8) :: a  
    integer(4) :: b  
  end subroutine  
  
  subroutine bar(a,b)  
    real(8) :: a  
    integer(4) :: b  
  end subroutine  
end interface
```

Procedure overloading

Procedure overloading allows us to use one name for a procedure

```
interface foobar
  subroutine foo(a)
    real(8), dimension(:) :: a
  end subroutine

  subroutine bar(a)
    real(8), dimension(:, :) :: a
  end subroutine
end interface
```

Procedure arguments

Interfaces can also be used to pass functions or subroutines as arguments to procedures

```
subroutine foobar(a, bar)
  integer(4) :: a
  interface
    function bar(q) result(r)
      real(8) :: q, r
    end function
  end interface

  a=bar(4.d0*a)
end subroutine
```

Operators

F95 also lets the programmer (re)define operators

```
module foo_m
  interface operator (.foo.)
    module procedure raboof ! implementation in this module
  end interface
contains
  function raboof(a,b) result(r)
    ...
  end function
end module

q=w.foo.z
```

Module implementation

Small is beautiful: Subroutines should do one thing, and do it well

No public data: Use access functions instead

Quasi-objects using types: Use types, and pass typed variables to module methods in order avoid module globals

Constructors and destructors for every module or "class". New instances are created by calling **call** `new(mytype,...)`, and deleted with **call** `delete(mytype)`

Inheritance can be achieved by extending user defined types and using interfaces for overloading.

Polymorphism is possible using overloading and dispatch functions

What should one keep in mind when designing modules?

Small modules: Try to write modules which handles one logical unit of the problem

Small interfaces: A module should have only a small set of public methods

- Easier to change things without breaking something else
- Others need only to understand the interfaces
- Document the interface

Few interfaces: Modules should rely on few other modules; try to keep modules simply linked

Coding style

A few remarks on coding style: F90 is (unfortunately) case insensitive, but for better readability I suggest the following guide-lines.

- Strict indentation using TAB characters
- Code in small-caps, there is really no need to shout
- Type names end with `_t`
- Module names end with `_m` or `_class`
- Parameters in all-caps

Gotchas

Using F95 in all its glory seldom causes any big performance penalties. There are unfortunately some compiler dependent things to watch out for:

matmul The intrinsic `matmul()` function is very handy. Unfortunately some compilers have really, really bad implementations: Multiplying two 1000x1000 matrices on my laptop using a simplistic handwritten `matmul` takes 2.80 s., using Intel BLAS 2.08 s. and using `ifort8` `matmul` 13.76 s. Using `ifc7.1` `matmul` takes 2.10 s.

Dynamic arrays inside user defined types can cause certain (many) compilers to produce very slow loops over the arrays. This is easily circumvented by using an temporary pointer to the array.

```
type foo_t
    real(8), dimension(:,:), pointer :: a
end type

subroutine run_forrest_run(bar)
    type(foo_t) :: bar

    integer(4) :: i, j
    real(8), dimension(:,:), pointer :: apa
    apa=>bar%a

    forall( i=1:size(apa(1,:), j=1:size(apa(:,1) )
            apa(i,j) = kaboom(i,j)
    end forall
end subroutine
```

BLAS and LAPACK

Nice interface to BLAS and LAPACK via BLAS95 and LAPACK95 modules

```
use blas_dense
real(8), dimension(1000,1000) :: a, b
call gemm(a, b)
call gemm(a, b, blas_trans, blas_notrans, 2.d0, 0.5d0)
```

```
use la_precision, only: wp => sp ! partial use with rename
use f95_lapack, only: la_geev
real(8), dimension(1000,1000) :: a
real(8), dimension(1000) :: wr, wi, vl
call la_geev(a, wr, wi, VL=vl) ! note keyword argument
```

Odds and ends

Libraries Writing module libraries in F95 is no problem. F95 libraries are however compiler dependent, as are the compiler generated include files.

Fortran 2003 F2003 greatly enhances the OO features of F95

- Procedure pointers in user defined types
- True inheritance and polymorphism
- Constructors and destructors
- Better control over type member visibility (e.g. read-only members)