Organising the program code

Arjen Markus

Deltares

June 6, 2018

Program units

Fortran distinguishes a number of organisational units:

- The main program – starting point of the execution
- Subroutines and functions – do a particular job, where functions are very much like mathematical functions.
- Modules and submodules: ways of "packaging" routines and variables

Some guidelines:

- Use `IMPLICIT NONE`
- Use modules (and submodules)
- Limit the accessibility of items in modules

The main program

The main program is the starting point:

```
program solve_equation
    use solve_pde

    implicit none

    type(solution_data) :: solution
    integer             :: i

    call initialise( solution )
    do i = 1,100
        call next_step( solution, rhs )
        call write_solution
    enddo
contains
subroutine rhs( solution )  !<== Fortran 2008
    ...                     !<== Interface defined in module
end subroutine rhs
...
end program solve_equation
```

The main program - explanation

The code as shown contains:

- The main program – `program ... end program`
- It *uses* the module `solve_pde`, which contains the definition of the derived type `solution_data`.
- The `contains` statement indicates the start of the *internal* routines.

Internal routines have access to the variables defined in the containing program unit:

```
contains

subroutine write_solution
    write(*,'(a,i0)')    'Time: ', i    ! <== Defined in main program
    write(*,'(10f10.4)') solution%array ! <== Also in main program
    ...
end subroutine write_solution
```

Modules: advantages

In FORTRAN 77 no "packaging" mechanism was available: every routine was accessible everywhere. Name conflicts possible!
Modules have several advantages:

- Limit the access to routines
- Provide an explicit interface to routines – with newer features necessary!
  But also important: the compiler can check the interface in detail
- One location for definitions: derived types, interfaces, variables
- Resolve naming conflicts

Modules: syntax

Modules are defined as follows:

```
module solve_pde
    implicit none

    type solution_data
        real, dimension(10,10) :: array
    end type solution_data

contains
subroutine next_step( solution, eval_rhs )
    type(solution_data), intent(inout) :: solution
    interface
        subroutine eval_rhs( solution )
            import :: solution_data
            type(solution_data), intent(inout) :: solution
        end subroutine eval_rhs
    end interface
    ...
end subroutine next_step
...
end module solve_pde
```

Modules: using them

Items in modules are not automatically available. You need to *use* the module:

```
program solve_equation
    use solve_pde
    ...
end program solve_equation
```

Or, only a selection:

```
program solve_ode_equation
    use solve_ode, only: euler => method_euler
    ...
    call euler( array, rhs )  !<== Instead of "call method_euler"
    ...
end program solve_ode_equation
```

Limiting access

You can use several types of access in a module:

- By default, items (variables, parameters, derived types, subroutines, functions) are *public*, that is useable whenever you *use* the module.
- You can make items *private*:

```
module solve_pde
    implicit none

    private   !< Default access is private
    public :: solution_data, next_step
    ...
end module solve_pde
```

- Variables can be *protected*: they are read-only outside the module defining them.

Modules and submodules

Disadvantages of modules (pre-Fortran 2003):

- All code defined in a single file
- A change in a routine's body – no change in the interface – results in the "need" to recompile the module and everything that uses it. Known as the *compilation cascade*

Solution: submodules
The module defines the interface, the submodule defines the implementation.

Submodules (1)

Submodules depend on a module or on another submodule:

```
module calculus
    implicit none

    interface
        module function getgradient( f, x )
            interface
                real function f(x)
                    real, intent(in) :: x
                end function f
            end interface
            real, intent(in) :: x
        end function
    end interface
end module calculus
```

The module contains the *interface*

Submodules (2)

Definition of the submodule:

```
submodule(calculus) gradients
    implicit none
    real, private :: dx = 1.0e-3 ! Very naive implementation

contains
module function getgradient  ! <== No argument list - comes from module

    getgradient = (f(x+dx) - f(x-dx)) / 2.0 / dx
end function getgradient
end submodule gradients
```

Note that it could repeat the interface:

```
module function getgradient( f, x )
    interface
        real function f(x)
        ...
    end interface
    ...
end function getgradient
```

Generic interfaces (1)

Modules are also very useful for defining *generic interfaces*:

```
module reporting
    implicit none

    private
    public :: report_results

    interface report_results
        module procedure report_integer
        module procedure report_integer_array
        ...
    end interface
contains
subroutine report_integer( text, ivalue )
    ...
end subroutine report_integer

subroutine report_integer_array( text, iarray )
    ...
end subroutine report_integer_array
```

Generic interfaces (2)

Such interfaces can then be used instead of the specific names:

```
program solve_problem
    use report_results
    implicit none

    integer                  :: x
    integer, dimension(10) :: y

    do x = 1, 100
        call actually_solve( x, y )

        call report_results( 'X parameter',     x )
        call report_results( 'Solutions found', y )
    enddo
end program solve_problem
```

Generic interfaces (3)

Interfaces with the same name need not be defined in one module:

```
module find_root_single
    use precision, only: wp => single
    include 'find_root.f90'
end module find_root_single

module find_root_double
    use precision, only: wp => double
    include 'find_root.f90'
end module find_root_double

module find_root
    use find_root_single
    use find_root_double
end module find_root
```

Generic interfaces (4)

The *include file* contains the actual code – but independent of the precise meaning of `wp`:

```
    interface find_root
        module procedure find_root_impl
    end interface
contains
subroutine find_root_impl( f, root )
    interface
        real(kind=wp) function f(x)
            import :: wp
            real(kind=wp), intent(in) :: x
        end function
    end interface
    real(kind=wp) :: root
    ...
end subroutine find_root_impl
```