Fortran 2008

Program Units: Functions, Subroutines and Modules

Fiona Reid
f.reid@epcc.ed.ac.uk

Lecture 2 - overview

- Lecture 1
  - Language evolution
  - Basic syntax, iteration, conditionals etc.
- This session: program units and subprograms
- Procedures
  - User-defined functions
  - Subroutines
- Modules
  - Using module data and procedures
  - Public and private scope
- Other topics
  - External procedures and explicit interface blocks
  - Internal procedures
  - Recursive procedures

Program units

- Could write complete program as a single unit
- Preferable to break the program into smaller more manageable units
- In Fortran there are three types of program unit
  - Main program
  - External subprogram (e.g. library routines)
  - Module
- Program units
  - Perform a simple manageable task(s)
  - Can be written, compiled and tested in isolation
  - Built up to form the complete program

Program units

- A complete program is constructed from several program units
- It *must* contain exactly one main program
  - The main program may contain any number of modules and external subprograms
- Normally this main program will contain calls to subsidiary programs known as subprograms
- Subprograms are constructed from functions and subroutines
- Subroutines and functions are known as procedures

Program units

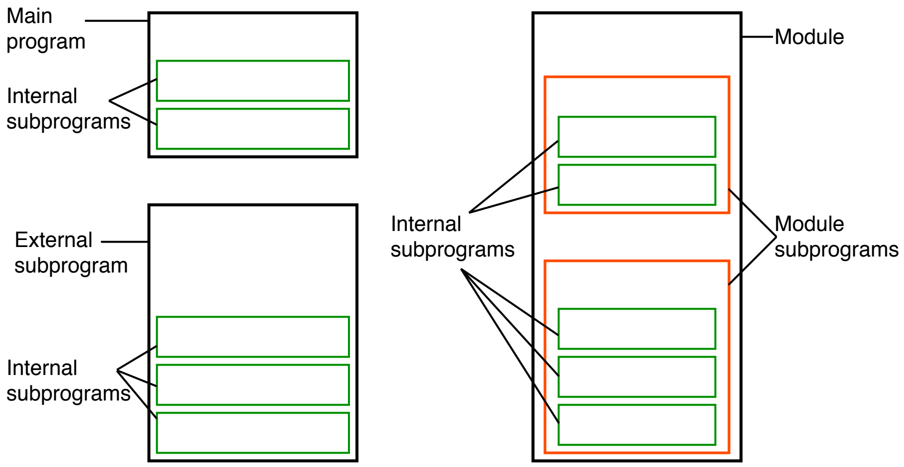

Figure reproduced from Metcalf and Reid (1999)

Procedures

- What are they?
  - Abstracted block of code which performs a specific task
  - E.g. Intrinsic functions

    `sin(x), cos(x), log(x), min(x1, x2)`
  - User-defined functions
  - Subroutines
- When to use them?
  - If a task has to be performed two or more times then you should probably use a procedure
  - Cuts down on code duplication and potential sources of error
  - Should first check whether the routine already exists

Procedures

- Guidelines for writing procedures
  - Check to see whether it already exists
  - Should be as flexible as possible to allow for re-use
  - Do not hardwire dimensions, e.g. array sizes
  - Try to pass the variables referenced in the procedure as actual arguments rather than using global variables
  - Considerably easier to update a parameter once within the main program than updating each procedure separately
  - Think about how easy it would be for another programmer to utilise your procedure
    - Use appropriate variable names, comments etc

User-defined functions

- Often used for short computations

```
distance = get_dist (a, b, c)
...
real function get_dist (x, y, z)
  real, intent(in)    :: x, y, z    ! protects arguments
  get_dist = sqrt(x*x + y*y + z*z)! assigned to distance
end function get_dist
```

- Final assigned value is that returned
  - No return statement (c.f. C/C++/Java etc)
  - Can be declared recursive
  - Values of `a`, `b` and `c` are passed by **reference**
  - Generally should not contain I/O or have side-effects
  - Values of the arguments should not be changed = side-effect
  - Use `intent(in)` to ensure arguments are protected

Subroutines

- Subroutines are "called"

```
call sort(nmax, a, ipass)
...
subroutine sort(nmax, a, ipass)
  integer, intent(in) :: nmax
  real, dimension(nmax), intent(inout) :: a
  integer, intent(out) :: ipass
  ! ... computation here ...
  return
end subroutine sort
```

- Never a return value
  - If you want to return before the end of the subroutine (e.g. if some condition is met) can use `return` to exit subroutine

Arguments and intent

- *All* arguments in functions and subroutines are passed by reference!
  - Changes to the dummy arguments change in the caller
- To avoid unwanted side effects
  - Tell compiler what is allowed with `intent` attribute
- Choices are

```
– intent(in)    ! argument cannot be changed
– intent(inout) ! set on entry; can be updated
– intent(out)   ! not set on entry; to be set
```

- Violations will cause an error at compile-time
  - Vital part of safe programming practice
- Use the `intent` attribute as:
  - Allows compilers to check for coding errors
  - Can facilitate efficient compilation and optimisation

Variable scope/ local objects

- Consider

```
subroutine sub1(m,n,rdata)
  integer, intent(in)      :: m, n
  real, intent(inout)      :: rdata
  real                     :: a
  real, dimension(m,n)     :: x
  ...
end subroutine sub1
```

- Variables **a** and **x** are known as local objects; they
  - Are created each time procedure sub1 is invoked
  - Destroyed when procedure sub1 completes
    - i.e. do not retain their values between calls
    - and may not even exist in the computer memory, between calls
- Can protect the values of local variables between calls with the **save** attribute

Save attribute

- If local objects are required outside the procedure they can be protected with the **save** attribute

```
subroutine sub1()
  integer, save :: i  ! saves value of i between calls
end subroutine sub1
```

- Beware - any variable given an initial value in its declaration statement has the save attribute by default

```
subroutine sub2()
  integer :: i = 0    ! saves value of i between calls
end subroutine sub2
```

- If you *do* want to reset value of **i** between calls use

```
subroutine sub3()
  integer :: i
  i = 0     ! value of i reset to 0 for each call of sub3
end subroutine sub3
```

Modules

- A module is a collection of variables and procedures

```
module sort
  implicit none
  ! variable specifications
  ...
contains
  ! procedure specifications
  subroutine sort_sub1()
  ...
  end subroutine sort_sub1
  ...
end module sort
```

- Variables declared above **`contains`** are in scope
  - Everywhere in the module itself
  - Can also be made available by *using* the module

Using a module

- Contents of a module are made available with **`use`** :

```
program main
  use sort
  implicit none
  ...
  call sort_sub1()
end program main
```

  - The **`use`** statement(s) should go directly after the program statement
  - **`implicit none`** should go directly *after* any use statements
- There are important benefits
  - Procedures contained within modules have explicit interfaces
  - Number and type of the arguments is checked at compile time
  - Not the case for external procedures (discussed later)
- Can implement data hiding or encapsulation
  - Via **`public`** and **`private`** statements and attributes

Using a module - example

```
program scope_main
  use scope_mod
  implicit none

  real  :: a=10.0, b=20.0
  write(*,*)"Before calling sub1 a = ",a," b = ",b
  call sub1(a)
  write(*,*)"After calling sub1 a = ",a," b = ",b

end program scope_main
```


Using a module - example

```
module scope_mod
  implicit none   ! Applies to everything after
   contains

contains

  subroutine sub1(x)
    real, intent(in) :: x
    real :: b = 5.0
    b = b + 1
    write(*,*)"Inside sub1 x = ",x," b = ",b
    return
  end subroutine sub1

end module scope_mod
```

Using a module - example

- Program output:

```
Before calling sub1   a =  10.0   b =  20.0
Inside sub1           x =  10.0   b =  6.0
After calling sub1    a =  10.0   b =  20.0
```

- Variable **`b`** is local to subroutine **`sub1`**
- Number and type of arguments checked at compile time
- Potential for errors greatly reduced

Compiling code with modules

- Consider the program main (main.f90) which uses module sort (sort.f90)

```
program main
  use sort
  implicit none
  ...
  call sort_sub1()
end program main
```

- **`main.f90`** and **`sort.f90`** are separate files
- To compile this program use

```
gfortran sort.f90 main.f90 -o progsort      (GNU)
pgf90 sort.f90 main.f90 -o progsort         (PGI)
```

- As the program main *uses* module sort, sort should be compiled *before* main
- Many compilers will insist on this

Compiling code with modules

- If you execute the command

```
gfortran sort.f90 main.f90 –o progsort   (GNU)
pgf90 sort.f90 main.f90 –o progsort      (PGI)
```

- You will notice that a file with a .mod extension is created for each module file
  – For this example a file **`sort.mod`** will be created
  – These .mod files contain information about global files and interfaces
  – The compiler needs the .mod files
  – Do not delete them

Public and private

- May wish to restrict accessibility of module entities
- Consider the module

```
module another_module
  implicit none
  private                               ! set default
  public :: nlimit                      ! public
  integer, parameter :: nlimit = 20     ! public
  integer            :: nlocal = 0      ! private
  real, public       :: tmp             ! public attribute
contains
  ...
end module another_module
```

- All variables are available within the module
  – But can only “use” public objects
  – The default case is **`public`**

Some dos and don'ts

- Can have:

```
module a
end module a
module b
  use a
end module b
program c
  use b
end program c
```

- But not:

```
module a
  use b
end module a
module b
  use a
end module b
```


External procedures

- Consider a program calling a subroutine:

```
program main
  ...
  call sub1()
end program main
```

```
subroutine sub1()
  ...
end subroutine sub1
```

- These program units are separate (different files)
  - Do not share scope
  - Any common information must be passed via argument list
  - No argument checking between main program and external procedure
- These are referred to as "external" procedures

External procedures

```
program scope_external
 implicit none
 real   :: a = 10.0, b = 20.0
 write(*,*)"Before calling sub1 a = ",a," b = ",b
 call sub1(a)
 write(*,*)"After calling sub1 a = ",a," b = ",b
end program scope_external
```

```
subroutine sub1(x)
 implicit none
 real, intent(in)      :: x
 real   :: b = 5.0
 b = b + 1
 write(*, *)"Inside sub1 x = ",x," b = ",b
 return
end subroutine sub1
```


External procedures

- Program output:

```
Before calling sub1     a =   10.0       b =   20.0
Inside sub1             x =   10.0       b =   6.0
After calling sub1      a =   10.0       b =   20.0
```

- Variable **`b`** is local to subroutine **`sub1`**
- No problems with variables being overwritten
- But, *no* checking of argument types between main program and **`sub1`**
- Fortran can provide argument checking by defining an explicit interface (see Metcalf and Reid sections 5.11 and 5.18 for details)

Explicit interfaces

- Argument checking is achieved via an explicit interface

```
program main
  real :: a, b, f
  interface
    real function fun(x)
      real, intent(in) :: x
    end function fun
  end interface
  ...
end program main
real function fun(x)
  . . .
end function fun(x)
```

- May be beneficial when using legacy codes or library routines/functions

Explicit interfaces

- Allows the compiler to know the expected shape, type and number of subroutine (or function) arguments
- This can provide
  - Aid to debugging code
  - Potential increase in efficiency
- What should go inside an interface block?
  - Function/subroutine header
  - Dummy argument declarations
  - Intent declarations
  - Compiler directives (if any)
  - Local declarations
- If you require external procedures you should use interface blocks if at all possible

Internal procedures

- Can be contained within the main program

```
program main
  ...
  call sub1()
contains
  subroutine sub1()
    ...
  end subroutine sub1
end program main
```

- Declares an “internal” subroutine
  - Variables declared in **main** are in scope in **sub1**
  - Not vice-versa
- However, ideally should use modules instead

Internal procedures

```
program scope_internal
  implicit none
  real  :: a = 10.0, b=20.0
  write(*, *)"Before calling sub1 a = ",a," b = ",b
  call sub1(a)
  write(*,*)"After calling sub1 a = ",a," b = ",b

contains
  subroutine sub1(x)
    implicit none
    real, intent(in)    :: x
    b = b + 1
    write(*,*)"Inside sub1 x = ",x," b = ",b
    return
  end subroutine sub1
end program scope_internal
```

Internal procedures

- Program output:

```
Before calling sub1    a =  10.0    b =  20.0
Inside sub1                  x =  10.0    b =  21.0
After calling sub1           a =  10.0    b =  21.0
```

- Variable **b** is altered inside subroutine **sub1**
- As **b** is in the scope of the main program the compiler does not interpret this as an error
- Difficult to trace such an error especially in a complex program
- Should use modules instead

Recursive procedures

- Recursive procedures call themselves repeatedly
- Must be explicitly declared using **recursive**
- Recursive procedures must contain a **result** keyword
- Subroutines can also be recursive
- Recursion may incur efficiency overheads
  - Space required for local variables, may be proportional to number of nested calls to the same procedure
  - Time incurred when calling the procedure, memory allocation

Recursive function

- Computation of n! using a recursive function
- Function name and **result** variable have same type

```
res = factorial(m)
...
recursive function factorial(n) result(nfact)
  implicit none
  integer, intent(in)    :: n
  integer                :: nfact
  if (n > 0) then
    nfact = n * factorial(n-1)
  else
    nfact = 1
  end if
end function factorial
```


Recursive subroutine

- Computation of n! using a recursive subroutine

```
call factorial(m,res)
...
recursive subroutine factorial(n,result)
  integer, intent(in)     :: n
  integer, intent(inout) :: result
  if (n > 0) then
    call factorial(n-1,result)
    result = result * n   ! Need to update result
  else
    result = 1
  end if
  return
end subroutine factorial
```

Summary

- Various program units
  - Main program
  - Procedures: functions and subroutines
  - Modules: interfaces etc
- Used correctly can contribute to well-structured code
  - Safe, understandable, maintainable code