Modules

Reusing this material

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 simple manageable task(s)
  - Can be written, compiled and tested in isolation
  - Built up to form the complete program

Modules

- Constants, variables, and procedures can be encapsulated in modules for use in one or more programs.
- 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

Points about modules

- Within a module, functions and subroutines are known as module procedures
- Module procedures can contain internal procedures
- Module objects can be given the `SAVE` attribute
- Modules can be `USE`d by procedures and modules
- Modules must be compiled before the program unit which uses them.

Module syntax

```
MODULE module-name
[ <declarations and specification statements> ]
[ CONTAINS
<module-procedures> ]
END [ MODULE [ module-name ]]
```

Module example

```
MODULE Triangle_Operations
  IMPLICIT NONE
  REAL, PARAMETER :: pi=3.14159
CONTAINS
  FUNCTION theta(x,y,z)
   ...
  END FUNCTION theta
  FUNCTION Area(x,y,z)
   ...
  END FUNCTION Area
END MODULE Triangle_operations
```

Using modules

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

```
PROGRAM TriangUser
  USE Triangle_Operations
  IMPLICIT NONE
  REAL :: a, b, c
```

  - 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
  - Can implement data hiding or encapsulation
  - Via **`public`** and **`private`** statements and attributes

Restricting visibility

- The visibility of an object declared in a module can be restricted to that module by giving it the attribute `PRIVATE`

```
REAL :: Area, theta
PUBLIC                         !confirm default
PRIVATE :: theta               !restrict
REAL, PRIVATE :: height!restrict
```

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

USE rename syntax

- Can rename module variables and procedures when using them:

```
USE <module-name> &
   [,<new-name> => <use-name>]
```

i.e.

```
USE Triangle_Operations, &
   Space => Area
```

USE ONLY syntax

- Also possible to restrict what parts of a module to use:

```
USE <module-name> [, ONLY : <only-list>]
```

i.e.

```
USE Triangle_operations, ONLY: &
  pi, Space => Area
```

Module interfaces

- Fortran allows the definition of interfaces
  - Informs compiler of expected shape, type, and number of arguments for routine or function (also optional nature, intent)
  - Can provide
    - Compile time checking and aid to debugging code
    - Potential increase in efficiency
- Can have explicit interfaces, i.e.:

```
interface
  real function fun(x)
    real, intent(in) :: x
   end function fun
 end interface
```

- Not necessary to specify explicit interfaces for module procedures

Module interfaces

- Possible to implement polymorphism with module interfaces, i.e.:

```
module maths_functions
 implicit none
 private

 public :: my_sum

 interface my_sum
    module procedure real_sum
    module procedure int_sum
 end interface

 contains

 function real_sum (a, b)
   implicit none
   real, intent(in) :: a,b
   real_sum = a + b
 end function real_sum

 function int_sum (a, b)
   implicit none
   integer, intent(in) :: a,b
   int_sum = a + b
 end function int_sum
end module
```

Operator overloading

- Using interfaces it is possible to overload operators (or define your own operators) as well:

```
implicit none
private

interface operator(+)
   module procedure real_sum, int_sum
end interface

contains

…
```

- Only really makes sense if you define your own operators or datatypes
    - Can't override existing definitions (**the above example isn't actually allowed**)

Psuedo OO programming with F90

- Modules and interfaces allow semi-OO programming
  - Encapsulation of data and functions with modules
  - Controlled access to data or functions with private and public keywords
  - Polymorphism with interfaces
  - Operator overloading with interfaces
- Does not provide full OO functionality but can be very powerful
  - Often enough functionality with this without using the F2003 additions

Exercise

- Look at the basic module creation practicals
- Move on to covert percolate source code from single file to multiple modules

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
```

- As the program main *uses* module sort, sort should be compiled *before* main

Compiling code with modules

- If you execute the command

```
gfortran sort.f90 main.f90 -o progsort
```

- 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

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
```