6 Subroutines and Functions

6.1 Aims

By the end of this worksheet you will be able to:

- Understand the use of subroutines and functions to make your code more efficient and easier to read.

6.2 Re-using code – the subroutine

Examine the following program

```
 program output
 implicit none
 real,dimension(3) :: a,b,c
 character :: answer*1
!initialise arrays
 a = 1.5
 b = 2.5
 c = 3.5
 write(*,1) 'a',a
 print *, 'type y to continue or any other key to finish'
 read *, answer
 if (answer /= 'y') stop
 write(*,1) 'b',b
 print *, 'type y to continue or any other key to finish'
 read *, answer
 if (answer /= 'y') stop
 write(*,1) 'c',c
 print *, 'type y to continue or any other key to finish'
 read *, answer
 if (answer /= 'y') stop
 write(*,1) 'a*b*c',a * b * c
1 format(a,3f8.3)
 end program output
```

The program sets up some arrays and then outputs them. At three stages in the program (bolded), it asks whether it should continue; it stops if the answer is not 'y'. Notice that the three bolded parts of the code are *identical.*

Simple enough – but look at the amount of code! Most of it is the same – wouldn't it be nice to re-use the code and cut down on the typing? The answer is to use **subroutines**.

```
        program output1
        implicit none
        real,dimension(3) :: a,b,c
!initialise arrays
        a = 1.5
        b = 2.5
        c = 3.5
        write(*,1) 'a',a
        call prompt()
        write(*,1) 'b',b
        call prompt()
        write(*,1) 'c',c
        call prompt()
        write(*,1) 'a*b*c',a * b * c
1       format(a,3f8.3)
        end program output1
!+++++++++++++++++++++++++++++++++++++++++++++
        subroutine prompt()
!prompts for a keypress
        implicit none
        character answer*1
        print *, 'type y to continue or any other key to finish'
        read *, answer
        if (answer /= 'y') stop
        end subroutine prompt
```

Examine the code, each time we use type

call prompt()

the program jumps to the line

subroutine prompt()

then executes each line of the code it finds in the subroutine until it reaches the line

end subroutine prompt

and then returns to the main program and carries on where it left off.
The program is much easier to understand now. All the code for prompting is in one place. If we ever need to change the code which prompts the user to continue, we will only ever need to change it once. This makes the program more **maintainable.**

6.3 Arguments to subroutines

We have seen that subroutines are very useful where we need to execute the same bit of code repeatedly.

The subroutine can be thought of as a separate program which we can call on whenever we wish to do a specific task. It is independent of the main program – it knows nothing about the variables used in the main program. Also, the main program knows nothing about the variables used in the subroutine. This can be useful – we can write a subroutine using any variable names we wish and we know that they will not interfere with anything we have already set up in the main program.

This immediately poses a problem – what if we **want** the subroutine to do calculations for us that we can use in the main program? The following program uses **arguments** to do just that.

Example: a program that calculates the difference in volume between 2 spheres.

```
        program vols
!Calculates difference in volume of 2 spheres
        implicit none
        real :: rad1,rad2,vol1,vol2
     character :: response
     do
        print *, 'Please enter the two radii'
        read *, rad1,rad2
        call volume(rad1,vol1)
        call volume(rad2,vol2)
        write(*,10) 'The difference in volumes is, ',abs(vol1-vol2)
10      format(a,2f10.3)
        print *, 'Any more? - hit Y for yes, otherwise hit any key'
        read *, response
        if (response /= 'Y' .and. response /= 'y') stop
     end do
     end program vols
!_____________________________________________________
     subroutine volume(rad,vol)
     implicit none
     real :: rad,vol,pi
!calculates the volume of a sphere
     pi=4.0*atan(1.0)
     vol=4./3.*pi*rad*rad*rad
!It's a little quicker in processing to do r*r*r than r**3!
     end subroutine volume
```

When the program reaches the lines

```
        call volume(rad1,vol1)
```

It jumps to the line

```
        subroutine volume(rad,vol)
```

The values, **rad1** and **vol1** are passed to the subroutine. The subroutine calculates a value for the volume and when the line :

```
        end subroutine volume
```

is reached, the value of the volume is returned to the main program

Points to notice – these are very important – please read carefully

- You may have several subroutines in your program. Ideally, a subroutine should do a specific task – reflected by its name.
- All the variables in subroutines, apart from the ones passed as arguments, are 'hidden' from the main program. That means that you can use the same names in your subroutine as in the main program and the values stored in each will be unaffected – unless the variable is passed as an argument to the subroutine.
- It is very easy to forget to declare variables in subroutines. Always use **implicit none** in your subroutines to guard against this.

- All the variables in the subroutine **must** be declared.
- The positioning of the arguments (in this case, rad and vol) is **important**. The subroutine has no knowledge of what the variables are called in the main program. It is **vital** that the arguments agree both in **position** and **type**. So, if an argument to the subroutine is **real** in the main program, it must also be **real** in the subroutine.
- If an argument to the subroutine is an array, it must also be declared as an array in the subroutine.

Exercise 6.1

Write a program that calculates the difference in area between two triangles. Your program should prompt the user for the information it needs to do the calculation. Use a subroutine to calculate the actual area. Pass information to the subroutine using arguments.

6.4 User Defined Functions

We have already met FORTRAN **intrinsic functions** like **abs, cos, sqrt**. We can also define our own functions – they work in a similar way to **subroutines**.

As an example, let's write a program (**func.f95**) that does some trigonometry. As you know, the trig routines in FORTRAN use radians, not degrees - so it would be nice to write a function that does all the conversion for us.

```
print *,'Enter a number'
read *, a
pi=4.0*atan(1.0)
print *,'the sine of ',a,' is ',sin(a*pi/180)
```

In this snippet, we are having to code the conversion from degrees to radians directly into the main part of the program. That's OK for a 'one-off', but what if we needed to do the conversion several times. Now look at this:

```
      program func
!demonstrates use of user defined functions
      implicit none
      integer, parameter :: ikind=selected_real_kind(p=15)
      real (kind=ikind):: deg,rads
      print *, 'Enter an angle in degrees'
      read  *, deg
      write(*,10) 'sin = ',sin(rads(deg))
      write(*,10) 'tan = ',tan(rads(deg))
      write(*,10) 'cos = ',cos(rads(deg))
10    format(a,f10.8)
      end program func
!_____________________________________________
      function rads(degrees)
      implicit none
      integer, parameter :: ikind=selected_real_kind(p=15)
!     returns radians
      real (kind=ikind) :: pi,degrees,rads
      pi=4.0_ikind*atan(1.0_ikind)
      rads=(degrees*pi/180.0_ikind)
      end function rads
```

What we have done, in effect, is to create our own function **rads**, which is used in an identical way to the intrinsic ones you have used already like **sqrt**, **cos**, and **abs**.

When the line

```
write(*,10) 'sin = ',sin(rads(deg))
```

is reached, the program jumps to

```
function rads(degrees)
```

the value, **degrees**, is passed to the function. The function does some computation, then finally returns the calculated value to the main program with the line

```
rads=(degrees*pi/180.0_ikind)
```

Note carefully that it doesn't return the value in the argument list (as does a subroutine) but actually **assigns** the value to its own name **rads**.

- The function **rads** converts the value of the argument, degrees, to radians.
- Notice that we must declare the data type of the function both in the main program, and in the function itself **as if it were a variable**.
- Functions return **one** value. This value, when calculated, is assigned to the name of the function as if it were a variable –

  rads=(degrees*pi/180.0_ikind)

Exercise 6.2

Write a program that includes a function called

real function average(n,list)

where **n** is integer and is the number of items in the **list**, and **list** is a **real array**.

Write suitable code for reading the numbers from a file (or keyboard), and output the average of the numbers.

Exercise 6.3

Write a program that allows a user to enter the size of a square matrix. In the program write a subroutine to compute a **finite difference matrix**. Ensure your output is neatly formatted in rows and columns.

So, for a 10 by 10 matrix, we expect output to look like this

```
 2 -1  0  0  0  0  0  0  0  0
-1  2 -1  0  0  0  0  0  0  0
 0 -1  2 -1  0  0  0  0  0  0
 0  0 -1  2 -1  0  0  0  0  0
 0  0  0 -1  2 -1  0  0  0  0
 0  0  0  0 -1  2 -1  0  0  0
 0  0  0  0  0 -1  2 -1  0  0
 0  0  0  0  0  0 -1  2 -1  0
 0  0  0  0  0  0  0 -1  2 -1
 0  0  0  0  0  0  0  0 -1  2
```

Check your attempt with finite.diffs.f95 on the website.