Chapter 19
Introduction to Subroutines

A man should keep his brain attic stacked with all the furniture he is likely to use, and the rest he can put away in the lumber room of his library, where he can get at it if he wants.
Sir Arthur Conan Doyle, Five Orange Pips

Aims

The aims of this chapter are:

- To consider some of the reasons for the inclusion of subroutines in a programming language.
- To introduce with a concrete example some of the concepts and ideas involved with the definition and use of subroutines.
  - Arguments or parameters.
  - The `intent` attribute for parameters.
  - The `call` statement.
  - Scope of variables.
  - Local variables and the `save` attribute.
  - The use of parameters to report on the status of the action carried out in the subroutine.
- Module procedures to provide interfaces.

19.1 Introduction

In the earlier chapter on functions we introduced two types of function

- Intrinsic functions - which are part of the language.
- User defined functions - by which we extend the language.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_19

We now introduce subroutines which collectively with functions are given the name procedures. Procedures provide a very powerful extension to the language by:

- Providing us with the ability to break problems down into simpler more easily solvable subproblems.
- Allowing us to concentrate on one aspect of a problem at a time.
- Avoiding duplication of code.
- Hiding away messy code so that a main program is a sequence of calls to procedures.
- Providing us with the ability to put together collections of procedures that solve commonly occurring subproblems, often given the name libraries, and generally compiled.
- Allowing us to call procedures from libraries written, tested and documented by experts in a particular field. There is no point in reinventing the wheel!

There are a number of concepts required for the successful use of subroutines and we met some of them in Chap. 12 when we looked at user defined functions. We will extend the ideas introduced there of parameters and introduce the additional concept of an interface via the use of modules. The ideas are best explained with a concrete example.

Note that we use the terms parameters and arguments interchangeably.

19.2 Example 1: Roots of a Quadratic Equation

This example is one we met earlier that solves a quadratic equation, i.e., solves

$$ax^2 + bx + c = 0$$

The program to do this originally was just one program. In the example below we break that problem down into smaller parts and make each part a subroutine. The components are:

- Main program or driving routine.
- Interaction with user to get the coefficients of the equation.
- Solution of the quadratic.

Let us look now at how we do this with the use of subroutines:

```
module interact_module
contains
  subroutine interact(a, b, c, ok)
    implicit none
    real, intent (out) :: a
    real, intent (out) :: b
```

```
    real, intent (out) :: c
    logical, intent (out) :: ok
    integer :: io_status = 0

    print *, &
      ' type in the coefficients a, b and c'
    read (unit=*, fmt=*, iostat=io_status) a, b, &
      c
    if (io_status==0) then
      ok = .true.
    else
      ok = .false.
    end if
  end subroutine interact
end module interact_module

module solve_module
contains
  subroutine solve(e, f, g, root1, root2, ifail)
    implicit none
    real, intent (in) :: e
    real, intent (in) :: f
    real, intent (in) :: g
    real, intent (out) :: root1
    real, intent (out) :: root2
    integer, intent (inout) :: ifail
!   local variables
    real :: term
    real :: a2

    term = f*f - 4.*e*g
    a2 = e*2.0
!   if term < 0, roots are complex
    if (term<0.0) then
      ifail = 1
    else
      term = sqrt(term)
      root1 = (-f+term)/a2
      root2 = (-f-term)/a2
    end if
  end subroutine solve
end module solve_module

program ch1901
  use interact_module
```

```
  use solve_module
  implicit none
! simple example of the use of a main program
! and two subroutines.
! one interacts with the user and the
! second solves a quadratic equation,
! based on the user input.
  real :: p, q, r, root1, root2
  integer :: ifail = 0
  logical :: ok = .true.

  call interact(p, q, r, ok)
  if (ok) then
    call solve(p, q, r, root1, root2, ifail)
    if (ifail==1) then
      print *, ' complex roots'
      print *, ' calculation abandoned'
    else
      print *, ' roots are ', root1, ' ', root2
    end if
  else
    print *, ' error in data input program ends'
  end if
end program ch1901
```

19.2.1 Referencing a Subroutine

To reference a subroutine you use the `call` statement:

```
call subroutine_name(optional actual argument list)
```

and from the earlier example the call to subroutine `interact` was of the form:

```
call interact(p,q,r,ok)
```

When a subroutine returns to the calling program unit control is passed to the statement following the call statement.

19.2.2 Dummy Arguments or Parameters and Actual Arguments

Procedures and their calling program units communicate through their arguments. We often use the terms parameter and arguments interchangeably through out this text. The `subroutine` statement normally contains a list of dummy arguments, separated by commas and enclosed in brackets. The dummy arguments have a type associated with them; for example, in subroutine `solve` `x` is of type `real`, but no space is put aside for this in memory. When the subroutine is referenced e.g., `call solve(p,q,r,root1,root2,ifail)`, then the dummy argument points to the actual argument `p`, which is a variable in the calling program unit. The dummy argument and the actual argument must be of the same type - in this case `real`.

19.2.3 The `intent` Attribute

It is recommended that dummy arguments have an `intent` attribute. In the earlier example subroutine `solve` has a dummy argument `e` with `intent(in)`, which means that when the subroutine is referenced or called it is expecting `e` to have a value, but its value cannot be changed inside the subroutine. This acts as an extra security measure besides making the program easier to understand. For each parameter it may have one of three attributes:

- `intent(in)`, where the parameter already has a value and cannot be altered in the called routine.
- `intent(out)`, where the parameter does not have a value, and is given one in the called routine.
- `intent(inout)`, where the parameter already has a value and this is changed in the called routine.

19.2.4 Local Variables

We saw with functions that variables could be essentially local to the function and unavailable elsewhere. The concept of local variables also applies to subroutines. In the example above `term` and `a2` are both local variables to the subroutine `solve`.

19.2.5 Local Variables and the `save` Attribute

Local variables are usually created when a procedure is called and their value lost when execution returns to the calling program unit. To make sure that a local variable

retains its values between calls to a subprogram the `save` attribute can be used on a type statement: e.g.,

```
integer , save :: i
```

means that when this statement appears in a subprogram the value of the local variable `i` is saved between calls.

19.2.6 Scope of Variables

In most cases variables are only available within the program unit that defines them. The introduction of argument lists to procedures immediately opens up the possibility of data within one program unit becoming available in one or more other program units.

In the main program we declare the variables `p`, `q`, `r`, `root1`, `root2`, `ifail` and `ok`.

Subroutine `interact` has no variables locally declared. It works on the arguments `a`, `b`, `c` and `ok`; which map onto `p`, `q`, `r` and `ok` from the main program, i.e., it works with those variables.

Subroutine `solve` has two locally defined variables, `term` and `a2`. It works with the variables `e`, `f`, `g`, `root1`, `root2` and `ifail`, which map onto `p`, `q`, `r`, `root1`, `root2` and `ifail` from the main program.

19.2.7 Status of the Action Carried Out in the Subroutine

It is also useful to use parameters that carry information regarding the status of the action carried out by the subroutine. With the subroutine `interact` we use a logical variable `ok` to report on the status of the interaction with the user. In the subroutine `solve` we use the status of the integer variable `ifail` to report on the status of the solution of the equation.

19.2.8 Modules 'Containing' Procedures

At the same time as introducing procedures we have 'contained' them in a module and then the main program 'uses' the module in order to make the procedure available. Procedures 'contained' in modules are called module procedures.

With the `use` statement the interface to the procedure is available to the compiler so that the types and positions of the actual and dummy arguments can be checked. This was a major source of errors with Fortran 77.

The `use` statement must be the first statement in the main program or calling unit, also the modules must be compiled before the program or calling unit.

We will cover modules in more depth in later chapters.

There are times when an interface is mandatory in Fortran so it's good practice to use module procedures from the start. There are other ways of providing explicit interfaces and we will cover them later.

19.3 Why Bother with Subroutines?

Given the increase in the complexity of the overall program to solve a relatively straightforward problem, one must ask why bother. The answer lies in our ability to manage the solution of larger and larger problems. We need all the help we can get if we are to succeed in our task of developing large-scale reliable programs.

We need to be able to break our problems down into manageable subcomponents and solve each in turn. We are now in a very good position to be able to do this. Given a problem that requires a main program, one or more functions and one or more subroutines we can work on each subcomponent in relative isolation, and know that by using features like module procedures we will be able to glue all of the components together into a stable structure at the end. We can independently compile the main program and the modules containing the functions and subroutines and use the linker to generate the overall executable, and then test that. Providing we keep our interfaces the same we can alter the actual implementations of the functions and subroutines and just recompile the changed procedures.

19.4 Summary

We now have the following concepts for the use of subroutines:

- Module procedures providing interfaces.
- `Intent` attribute for parameters.
- Dummy parameters.
- The use of the call statement to invoke a subroutine.
- The concepts of variables that are local to the called routines and are unavailable elsewhere in the over all program.
- Communication between program units via the argument list.
- The concept of parameters on the call that enable us to report back on the status of the called routine.

19.5 Problems

19.1 Type the program and module procedures for Example 1 into one file. Compile, link and run providing data for complex roots to test this part of the code.

19.2 Split the main program and modules up into three separate files. Compile the modules and then compile the main program and link the object files to create one executable. Look at the file size of the executable and the individual object files. What do you notice?

The development of large programs is eased considerably by the ability to compile small program units and eradicate the compilation errors from one unit at a time. The linker obviously also has an important role to play in the development process.

19.3 Write a subroutine to calculate new coordinates (x', y') from (x, y) when the axes are rotated counter clockwise through an angle of a radians using:

$x' = xcosa + ysina$

$y' = -xsina + ycosa$

Hint:

The subroutine would look some thing like

```
subroutine ChangeCoordinate(x, y, a, xd, yd)
```

Write a main program to read in values of `x`, `y` and `a` and then call the subroutine and print out the new coordinates. Use a module procedure.