Chapter 4: Program Organization

Now that we know the basics of how to code in fortran we will learn how to break a program into manageable parts. This makes code easier to read, easier to test and easier to reuse. We'll hit the high notes of how to use the FUNCTION, SUBROUTINE, and MODULE constructs to break-up the workflow of a program. A quick way to remember the difference between each piece is that a

- FUNCTION takes in multiple arguments and returns a single argument.
- SUBROUTINE takes in and returns multiple arguments.
- MODULE is a construct that contains variable declarations, functions, and subroutines that can be made available to a program or another module.

When we discuss the FUNCTION and the SUBROUTINE we include special attributes that tell the compiler how to handle arguments passed in by the user. There are three options:

- INTENT(IN) – Use with functions and subroutines. It informs the compiler that an argument may not be changed by the function/subroutine
- INTENT(OUT) – Use with subroutines. It informs the compiler that an argument will return information from the subroutine to the calling procedure. The argument's value is undefined on entry to the procedure and must be given a value by some means.
- INTENT(INOUT) – Use with subroutines. It informs the compiler that an argument may transmit information into a subroutine, be operated upon, and then returned to the calling procedure.

4.1 Functions

A function in fortran is a procedure that accepts multiple arguments and returns a single result. They come in two flavors intrinsic and external.

4.1.1 Intrinsic Functions

Intrinsic functions are built-in functions that do not need declared. There are ***lots*** of intrinsic functions available (a quick Google search reveals as much), some of them are

- EXP – exponential function
- SIN – sine, argument in radians (other trig functions available as well)
- ASIN – arcsine, argument in radians (other inverse trig functions available as well)
- LOG – natural logarithm
- LOG10 – common logarithm (base 10)
- ABS – absolute value

4.1.2 External Functions

These are procedures written by a user that can be called by another PROGRAM (or FUNCTION or SUBROUTINE). We'll write the external function and the program that call it in separate files. This example we'll write a function to evaluate the function

$$f(x) = x^3 - x + \cos(x).$$

First, we write the function

function.f90

```
FUNCTION f(x)
   IMPLICIT NONE
   INTEGER,PARAMETER         :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP),INTENT(IN) :: x
   REAL(KIND=RP)             :: f

   f = x**3 - x + COS(x)
END FUNCTION f
```

In `function.f90` we have a variable `f` with the same name as our function. This is a special variable, known as the RESULT variable, and it is the means by which a FUNCTION returns information to a calling procedure. Always remember that every FUNCTION ***must*** contain a variable that has the same name as the FUNCTION, or we could rewrite the function declaration above as FUNCTION f(x) RESULT(v); in either case the variable f(x) or v ***must*** be assigned. Next, we write the main program that will call the function $f(x)$.

functionMain.f90

```
PROGRAM functionExample
   IMPLICIT NONE
   INTEGER,PARAMETER :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP)     :: x,y
   REAL(KIND=RP),EXTERNAL   :: f

   x = 3.0_RP
   y = f(x)
   PRINT*,y
END PROGRAM functionExample
```

Now that we have these two fortran source files, how do we compile the program? We have several options, we can

1. Compile the files all at once on the command line.

   Compile all files at once

   ```
   gfortran function.f90 functionMain.f90 -o functionExample
   ```

2. Compile the files one at a time at the command line.

   Compile files on at a time

   ```
   ! the -c flag means 'compile' but don't create an executable
   gfortran -c function.f90
   gfortran -c functionMain.f90
   gfortran function.o functionMain.o -o functionExample
   ```

3. Compile using a make file.

For this example, one of the first two options are straightforward. However, once we have a lot of source files it becomes unwieldy due to the nature of dependencies. Dependencies, essentially, boils down to checkinig

- If a program CALLs a subroutine or function from another file, this other file must be compiled either at the same time or before the program is compiled.
- If a program USEs a module, or a module USEs another module, then the USEd module needs to be compiled **before** the program or USEs module.

A make file makes the compilation of many source files simple because it automatically detects inheritance and which files need compiled. Later, in Sec. 4.4.1, we cover make file construction.

In general, a function in fortran has the form

General Function

```
FUNCTION function_name( passing arguments )
   IMPLICIT NONE
   INTEGER,PARAMETER    :: RP = SELECTED_REAL_KIND(15)
   data_type,INTENT(IN) :: arguments that shouldn't be changed
   data_type            :: any arguments that can be changed
   data_type            :: function_name
! Local variables
   data_type            :: available locally, calling procedure doesn't know of
                           these variables
   ...
   code
END FUNCTION function_name
```

4.2 Subroutines

Subroutines are more general than functions as they allow multiple input arguments and output results. However, you do have to be diligent when you assign the INTENT of each argument. Let's start with a simple example that mimics the FUNCTION we just wrote.

subroutine.f90

```
SUBROUTINE f(x,y)
   IMPLICIT NONE
   INTEGER,PARAMETER         :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP),INTENT(IN)  :: x
   REAL(KIND=RP),INTENT(OUT) :: y

   y = x**3 - x + COS(x)
END SUBROUTINE f
```

In `subroutine.f90` we have an argument that transmits the value of x into the subroutine and the argument y which returns information to the calling program. Note that the INTENT of each input argument matches accordingly.

Next, we have the main program that calls the subroutine.

subroutineMain.f90

```
PROGRAM subroutineExample
   IMPLICIT NONE
   INTEGER,PARAMETER :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP)     :: x,y

   x = 3.0_RP
   CALL f(x,y)
   PRINT*,y
END PROGRAM subroutineExample
```

The two source files are compiled identically to compiling the function source files (modulo file names).

In general, a subroutine in fortran has the form

General Subroutine

```
SUBROUTINE subroutine_name( passing arguments )
   IMPLICIT NONE
   INTEGER,PARAMETER       :: RP = SELECTED_REAL_KIND(15)
   data_type,INTENT(IN)    :: arguments that shouldn't be changed
   data_type,INTENT(INOUT) :: arguments that may be changed
   data_type,INTENT(OUT)   :: arguments for output only
! Local variables
```

```
   data_type               :: available locally, calling procedure doesn't know of
                              these variables
   ...
   code
END SUBROUTINE subroutine_name
```

4.3 Passing Functions

Let's return to the example of using the left endpoint quadrature rule to approximate an integral. Recall we had to hard code the function we wanted to integrate. Now, however, with our knowledge of functions we can write a general quadrature routine and pass in different functions to integrate.

We'll write the functions to integrate, the left hand rule and then a driver to collect the quadrature results. Let's consider two functions

$$f(x) = e^x - x^2 \quad \text{and} \quad g(x) = \sin(x) - \cos(x),$$

with corresponding source code

functions.f90

```
FUNCTION f(x)
   IMPLICIT NONE
   INTEGER,PARAMETER         :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP),INTENT(IN) :: x
   REAL(KIND=RP)             :: f

   f = EXP(x) - x**2
END FUNCTION f
!
FUNCTION g(x)
   IMPLICIT NONE
   INTEGER,PARAMETER         :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP),INTENT(IN) :: x
   REAL(KIND=RP)             :: g

   g = SIN(x) - COS(x)
END FUNCTION g
```

Next we write a function that implements the left point rule quadrature method.

quadratureRules.f90

```
FUNCTION leftHandRule(a,b,N,func)
   IMPLICIT NONE
   INTEGER       ,PARAMETER  :: RP = SELECTED_REAL_KIND(15)
   INTEGER       ,INTENT(IN) :: N
   REAL(KIND=RP),INTENT(IN) :: a,b
   REAL(KIND=RP),EXTERNAL   :: func
   REAL(KIND=RP)             :: leftHandRule
! Local Variables
   INTEGER       :: i
   REAL(KIND=RP) :: dx,x
!
   dx  = (b-a)/N
   leftHandRule = 0.0_RP
   DO i = 0,N-1
      x   = a + i*dx
      leftHandRule = leftHandRule + func(x)*dx
```

```
   END DO
END FUNCTION leftHandRule
```

We applied the EXTERNAL attribute to the `func` input argument, which informs the compiler that the argument is a REAL FUNCTION and not a REAL variable. This, in effect, allows us to pass a function to another function.

Finally, we write a driver for the quadrature program.

QuadExample.f90

```
PROGRAM Quadrature
   IMPLICIT NONE
   INTEGER       ,PARAMETER :: RP = SELECTED_REAL_KIND(15)
   INTEGER       ,PARAMETER :: N = 150
   REAL(KIND=RP),PARAMETER :: a = -1.0_RP, b = 2.0_RP
   REAL(KIND=RP)           :: Integral
   REAL(KIND=RP),EXTERNAL  :: leftHandRule,f,g

   Integral = lefthandRule(a,b,N,f)
   WRITE(*,*)'Approximation for f(x) integral: ',Integral
   WRITE(*,*)
   Integral = leftHandRule(a,b,N,g)
   WRITE(*,*)'Approximation for g(x) integral: ',Integral
END PROGRAM Quadrature
```

We finish the example by providing the output of the quadrature program for the two functions $f(x)$, $g(x)$.

QuadExample - Commands and Output

```
gfortran functions.f90 quadratureRules.f90 QuadExample.f90 -o quad
./quad
 Approximation for f(x) integral:    3.9809989288432961

 Approximation for g(x) integral:  -0.82136496727329411
```

4.4 Modules

Modules are similar to header files in C++. With modules we can easily organize our code, allow for code reuse and obtain additional benefits we will discuss later (in Chap. 6). Modules are extremely useful and powerful. We will only scratch the surface of what modules are capable. Since source files can contain multiple modules, ***follow the philosophy of one file = one module***.

Modules have the general structure

General Module Structure

```
MODULE module_name
!  variable definitions
CONTAINS
!  functions and subroutines
END MODULE module_name
```

Let's create a simple module that will make our lives simpler. In every function, subroutine and program we've had to declare

```
INTEGER,PARAMETER :: RP = SELECTED_REAL_KIND(15)
```

It would be nice if we could always have the precision definition available to all procedures. We can use a module to achieve this goal. For good measure we'll also include the constant π, just in case we need it later.

Constants.f90

```
MODULE Constants
   IMPLICIT NONE
   INTEGER      ,PARAMETER :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP),PARAMETER :: pi = 4.0_RP*ATAN(1.0_RP)
END MODULE Constants
```

Note that we still use IMPLICIT NONE in the module. Also, since the module `Constants.f90` contains no functions or subroutines we omit the CONTAINS command.

Now in a main program (or other module) we USE the module

main.f90

```
PROGRAM main
   USE Constants
   IMPLICIT NONE
...
END PROGRAM main
```

Notice we use the module ***before*** we invoke IMPLICIT NONE in `main.f90`. Also, we still use IMPLICIT NONE everywhere.

To compile with modules is similar to compiling with multiple `.f90` files. We have the option of compiling all at once or individually. However, since we usually have more than a couple module (or source) files, there is an easier way to compile. We use makefiles.

4.4.1 Compiling Source Code with Makefiles

Makefiles are special format files that, together with the `make` utility, help us automatically compile source code, get the correct dependencies and linking, and manage programming projects. A makefile has the basic components of defining the compiler to be used, any compiler flags, the name of the compiled executable, and then any objects to be built and their dependencies.

Writing the syntax for a general makefile would be very unwieldy and confusing. So instead, let's learn through an example. We'll create a makefile for the quadrature program written earlier in this chapter. We name the compiler, flags, and executable name at the top of the file, name the objects to built, and then compile. The pound symbol `#` is the comment character for makefiles. We'll colorize the makefile to facilitate its readability. Also, we'll update the files a bit to include the `Constants` module.

quad.mak

```
COMPILER = gfortran
FLAGS = -O3 # this is an optimizer which makes loops more efficient
EXECUTABLE = quad
# Object Files for build
OBJS = \
Constants.o \
functions.o \
quadratureRules.o

$(EXECUTABLE) :  QuadExample.f90 $(OBJS)
        $(COMPILER) $(FLAGS) -o $(EXECUTABLE) QuadExample.f90 $(OBJS)

# Object dependencies and compilation
Constants.o : Constants.f90
        $(COMPILER) $(FLAGS) -c Constants.f90 # The commands must be exactly one ‘‘tab’’ over

functions.o : functions.f90 Constants.o
        $(COMPILER) $(FLAGS) -c functions.f90
```

```
quadratureRules.o : quadratureRules.f90 Constants.o
	$(COMPILER) $(FLAGS) -c quadratureRules.f90
```

Now we show how to use a makefile to compile the program. To make a makefile with a specific name we use the `-f` flag.

quad.mak - Commands and Output

```
make -f quad.mak
   gfortran  -c ./Constants.f90
   gfortran  -c ./functions.f90 Constants.o
   gfortran  -c ./quadratureRules.f90 Constants.o
   gfortran  -o quad ./QuadExample.f90 functions.o quadratureRules.o
./quad
   Approximation for f(x) integral:    3.9809989288432961

   Approximation for g(x) integral:  -0.82136496727329411
```

For this simple program a makefile is somewhat overkill. However, once we start object oriented programming makefiles can make our lives much simpler when building projects.