Chapter 7: Additional Topics

In this chapter we'll briefly cover selected advanced topics in fortran programming. All the topics come in handy to add extra functionality to programs, but the feature you'll use most often is dynamic arrays, i.e., an array the size of which can be redefined at execution time.

7.1 Overloading Operators

We've seen a few examples of overloading ***intrinsic*** assignment operators, like the `=` operator. However we can also define our own operators using interfaces. For example, we can create a dot product operator:

Dot Product Operator

```
   INTERFACE OPERATOR(.DOT.)
! Operator syntax: u.DOT.v
      MODULE PROCEDURE dotproduct
   END INTERFACE

   REAL(KIND=RP) FUNCTION dotproduct(vec1,vec2) RESULT(dot)
      REAL(KIND=RP),INTENT(IN) :: vec1(:),vec2(:)
      IF (SIZE(vec1).EQ.SIZE(vec2)) THEN
         dot = DOT_PRODUCT(vec1,vec2)
      ELSE
         dot = 0.0_RP
        PRINT*,'ERROR: Vector size mismatch'
      END IF
   END FUNCTION dotproduct
```

7.2 Dynamic Arrays

When we dealt with arrays in Chap. 5 we made the implicit assumption that we already knew how large we wanted to make the array. We can remove such an assumption if we add the ALLOCATABLE attribute to a variable declaration. This attribute will tell the compiler to reserve a chunk of memory for possible use during execution. It also gives us the freedom to resize arrays without recompiling.

For example, if we want a three dimensional ALLOCATABLE array we use a `:` as a placeholder for the dimension size, e.g.,

```
REAL(KIND=RP),ALLOCATABLE,DIMENSION(:,:,:) :: array1
```

To make the reserved memory available for use we use the ALLOCATE command. Always remember that if a PROGRAM, FUNCTION, or SUBROUTINE contains an ALLOCATE command it should have a corresponding DEALLOCATE command to release the memory. This will help prevent ***memory leaks***, which occur when a program mismanages memory. Memory leaks can slow down performance and can even exhaust available system memory (which leads to ***segmentation faults***). Fortran does offer some built-in error checking when allocating memory:

```
ALLOCATE(var_name(lowerBound:upperBound),STAT=ierr)
```

where the STAT option returns an INTEGER type. If `ierr` $\neq 0$, then there was an error allocating the memory for `var_name`.

We also show a quick example of reallocating memory during a program's execution

AllocateExample.f90

```
PROGRAM AllocateExample
   IMPLICIT NONE
   INTEGER,ALLOCATABLE,DIMENSION(:)    :: array1
```

```fortran
   INTEGER,ALLOCATABLE,DIMENSION(:,:) :: array2

   ALLOCATE(array1(-2:8),array2(-1:2,0:10))
   PRINT*,LBOUND(array1),UBOUND(array1)
   PRINT*,SHAPE(array2)
   DEALLOCATE(array1,array2)
!
   ALLOCATE(array1(0:100),array2(1:5,-5:5))
   PRINT*,LBOUND(array1),UBOUND(array1)
   PRINT*,SHAPE(array2)
   DEALLOCATE(array1,array2)
END PROGRAM AllocateExample
```

7.3 Optional Arguments

When we write a FUNCTION or SUBROUTINE sometimes an argument may not need to be present at all times. For example, in a function that prints a matrix we can write to a file, but default to the terminal if no file name is provided. In this way we can make the file name an OPTIONAL argument. The procedure can then check if the argument is PRESENT (returns a LOGICAL) and operate accordingly.

OptionalPrint.f90

```fortran
   SUBROUTINE PrintMatrix(mat,fileName,N)
      INTEGER                     ,INTENT(IN) :: N
      REAL(KIND=RP),DIMENSION(N,N),INTENT(IN) :: mat
      CHARACTER(LEN=*),OPTIONAL   ,INTENT(IN) :: fileName
! Local Variables
      INTEGER                                 :: i,fileUnit

      IF (PRESENT(fileName)) THEN
         OPEN(UNIT=fileUnit,FILE=fileName)
      ELSE
         fileUnit = 6 ! 6 is the terminal screen
      END IF
!
      DO i = 1,N
         WRITE(fileUint,*)mat(i,:)
      END DO
!
      IF (PRESENT(fileName)) THEN
         CLOSE(fileUnit)
      END IF
   END SUBROUTINE PrintMatrix
```

NOTE: If we put the above subroutine in its own file, it MUST be contained in a module.

7.4 Advanced Input/Output Options

We know how to read in or display data to the terminal or to a file, but there are some useful I/O options we glossed over. Next we'll explore some of the advanced options available to us when inputting and outputting information.

7.4.1 Non-Advancing I/O

First is the ADVANCE option in READ or WRITE statements. Effectively, this will tell the READ/WRITE statement whether or not to advance to the next line. By default ADVANCE is set to 'YES'. For example,

Advance NO

```fortran
READ(fileUnit,ADVANCE='NO')var1,var2,var3,var4
```

will read in four variables from the file pointed to by `fileUnit` where all variables are on the same line. However, if we read in using

Advance YES
```
READ(fileUnit,*)var1,var2,var3,var4
```

reads in four variables from the file pointed to by `fileUnit` where each variables is on its own line, i.e., separated by a carriage return.

7.4.2 Formatted File I/O

Sometimes unformatted file reading and writing is insufficient. For example, fortran can create and read binary files, which require special arguments. To format data for reading/writing fortran uses a notation like `'A4'`, which means a character of length 4. Or in general, a variable type and the length of the data to be written. The most common variable outputs are

- `A` – characters/strings
- `I` – integer
- `F` – real number, decimal form
- `E` – real number, exponential form
- `ES` – real number, scientific notation
- `EN` – real number, engineering notation

For real number outputs we specify the width of the number with a `#` symbol as well as the number of digits to appear after a decimal points, e.g., `F10.6` is a real number with 10 digits, 6 of which appear after the decimal.

We can combine several different formatting parameters by replacing the `*` option with `'(parameters)'` as we do in the next example

FormatPrint.f90
```
  SUBROUTINE PrintMatrix_Formatted(mat,fileName,N)
     INTEGER                     ,INTENT(IN) :: N
     REAL(KIND=RP),DIMENSION(N,N),INTENT(IN) :: mat
     CHARACTER(LEN=*),OPTIONAL   ,INTENT(IN) :: fileName
! Local Variables
     INTEGER                                 :: i,j,fUnit

     IF (PRESENT(fileName)) THEN
        OPEN(UNIT=fUnit,FILE=fileName)
     ELSE
        fUnit = 6 ! 6 is the terminal screen
     END IF
!
     DO i = 1,N
        DO j = 1,N
           WRITE(fUint,'(A2,I3,A2,I3,A2,F10.6,A1)',ADVANCE='NO')'A[',i,'][',j,']='mat(i,j),' '
        END DO
         WRITE(fUint,'(A2,I3,A2,I3,A2,F10.6)')'A[',i,'][',j,']='mat(i,j)
     END DO
!
     IF (PRESENT(fileName)) THEN
        CLOSE(fUnit)
     END IF
  END SUBROUTINE PrintMatrix_Formatted
```

7.5 Recursive Procedures in Fortran

Many mathematical formulae lend themselves to a recursive formulation, like the Fast Fourier Transform (FFT). But, as we mentioned in Chap. 4, normally a FUNCTION or SUBROUTINE cannot reference itself, directly or indirectly. However, if we invoke that the procedure is RECURSIVE self-reference is possible.

7.5.1 Recursive Functions

We start with a canonical example of a recursive function to compute the factorial, $n!$, for some integer n. This example also introduces a SELECT CASE, which is a common alternative to IF statements.

Recursive Function

```
   RECURSIVE FUNCTION factorial(n) RESULT(factorial_n)
      IMPLICIT NONE
      INTEGER,INTENT(IN) :: n
! Determine if recursion is required
      SELECT CASE(n)
      CASE (0)
! Recursion reached the end
         factorial_n = 1.0_RP
      CASE (1:) ! any integer above 0
! Recursion call(s) required
         factorial_n = n*factorial(n-1)
      CASE DEFAULT
! If n is negative, return error
         PRINT*,'ERROR: n is negative'
         factorial_n = 0.0_RP
      END SELECT
   END FUNCTION factorial
```

7.5.2 Recursive Subroutines

We can also create recursive subroutines. Another example arises in the bisection method, where one recursively halves an interval based on the function $f(x)$ to locate the root of a function. We also enable the termination of the subroutine once we reach a certain number of iterations (interval halvings)

halveInterval.f90

```
   RECURSIVE SUBROUTINE halveInterval(f,xL,xR,tol,iter_count,zero,delta,err)
      IMPLICIT NONE
      REAL(KIND=RP),INTENT(IN)    :: tol
      REAL(KIND=RP),INTENT(INOUT) :: xL,xR
      INTEGER      ,INTENT(INOUT) :: iter_count
      REAL(KIND=RP),INTENT(OUT)   :: zero,delta
      INTEGER      ,INTENT(OUT)   :: err
      REAL(KIND=RP),EXTERNAL      :: f
! Local Variables
      REAL(KIND=RP)               :: xM

      delta = 0.5_RP*(xR-xL)
! Check to see if you've reached the tolerance for the root
      IF (delta.LT.tol) THEN
! Yes! - Return result
         err  = 0
         zero = xL + delta
      ELSE
! No root yet - check iterations and halve again
         iter_count = iter_count - 1
         IF (iter_count.LT.0) THEN
! Max iterations w/o solution - return error
```

```
          err  = -2
          zero = xL + delta
        ELSE
! Keep iterating
          xM = xL + delta
          IF (f(xL)*f(xm).LT.0.0_RP) THEN
             CALL halveInterval(f,xL,xM,tol,iter_count,zero,delta,err)
          ELSE
             CALL halveInterval(f,xM,xR,tol,iter_count,zero,delta,err)
          END IF
        END IF
      END IF
   END SUBROUTINE halveInterval
```