Transitioning to Fortran 90 for Scientific and Engineering Calculations

Robert J. Ribando and Mark J. Fisher

Department of Mechanical, Aerospace and Nuclear Engineering
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903-2442

ABSTRACT

This article gives a brief introduction to the use of Fortran 90 for use in scientific and engineering computation. It is based on a single, actual working program containing a wider variety of Fortran 90 syntax than probably any one programmer would ever need. The original program comment statements have been expanded in this document to give a fairly descriptive picture of allowable programming syntax. This program has been run successfully under three different Fortran 90 compilers and thus is quite portable.

Keywords:

Fortran 90
Structured Programming
Functions
Subroutines
Compilers

Introduction

The program that forms the basis of this paper was originally developed to help distance education students who were taking a graduate-level computational fluid dynamics (CFD) course through Virginia's Commonwealth Graduate Engineering Program (CGEP). Many of them were older students who had not taken a programming course lately or even programmed in several years. The coding was originally in Watfor-77, the compiler that was being used in our first-year programming course at the time and which we were able to distribute to our non-resident students. As Fortran 90 began to be adopted, that Fortran 77 program was upgraded to ease the transition into Fortran 90. Eventually a separate Fortran 90 program was developed, and that also has evolved over the past few years.

One of the reputed advantages of Fortran 90 is backward compatibility with Fortran 77 and the first edition of this program did retain many Fortran 77 features. When our largest public microcomputer facility was upgraded recently to the Windows NT operating system and because the particular Fortran 90 product previously being used was not NT-compatible, we found ourselves evaluating several other Fortran 90 implementations. One of them, Elf90, is not backwardly compatible with Fortran 77; instead all Fortran 77 (and earlier) syntax that is considered obsolescent is simply not included. Thus, in order to run our "Everything-you-want-to-know-about-Fortran 90" program using that compiler, many more, mostly minor, changes were required. These included removing x descriptors (for skipping spaces) in all format statements, adding double colons in all declaration statements, providing explicit interfaces for the external subprograms and removing all unused variables, all instances of print*, data commons, data statements, etc. The some 700+ line, now-totally-Fortran 90-compliant program is discussed in this paper.

Comments statements in the actual program were converted to prose in this document and expanded and are interspersed here among the source code. Meanwhile all executable statements were retained in their original form. An electronic copy of the program ref90a.f can be downloaded at:

http://www.itc.virginia.edu/itcweb/help/hints/unix/programming/fortran90/

The earlier ref90.f program, which retains some Fortran 77 features, can also be found at the same site.

Those using the xlf90 compiler on an IBM RS/6000 must cut off the two modules at the top of this program and compile them separately, using, for example:

```
xlf90 -c one.f
```

where one.f is the name given to the file containing the first module. Then, at the time that the program needing them is compiled, one would link in the two object files produced:

```
xlf90 -o run_me ref90a.f one.o two.o
```

Here, of course, ref90a.f is the name given to the file containing the rest of the program. An executable file named run_me will be produced.

Everything included below also compiles and runs under Salford/NAG FTN90 Compiler Version 2.1 in both fixed and free format. The ".for" extension is used for fixed format and the ".f " is used to indicate free format. Under the NAG (and Elf90) compilers the modules do not have to be clipped off and should appear first in the single source file. For Salford/NAG free format one would use:

```
ftn90 ref90a.f/lgo
```

while for fixed format

```
ftn90 ref90.for/lgo
```

is used. Separate instructions apply to Elf90.

Preliminaries

Ampersands (&) have been positioned in column 73 to indicate *continuation* (although in this word-processed document they may not appear to line up). These are not read in NAG Fortran 90 with fixed format, but are necessary when this program is run on the RS/6000 under xlf90.

Modules are used to share global data with a *subroutine*. These appear first in a NAG and Elf90 program, but are separate files under IBM xlf90. Modules can be used in place of *commons* to share data among program segments. The latter are considered obsolescent and are prohibited in Elf90. In the following modules some variables are both declared and initialized.

```
Module one
Implicit None
Real :: Alpha = 3.0, Beta = 3.0, Gamma = 2.0, Delta
End Module one

Module two
Implicit None
Real :: Epsilon=4.0, Zeta=3.0, Eta=2.0, Theta=1.0
End Module Two
```

This following program statement marks the actual beginning.

```
Program ref90a
```

The above two modules are included here:

```
Use One
Use Two
```

Next are *declarations* for all the variables used in the program; they have been grouped by type for clarity. The double colon syntax is required with the Elf90 compiler. Variables may have up to 31 alphanumeric characters and underlines.

```
Implicit None
```

```
Integer:: I, j, K=3, Caseno, Badnum, n, L=4, M=5
Integer:: mg1, ng1
Integer, Parameter::Imax = 10, Jmax =10

Real :: Array(Imax, Jmax)
Real :: Barray(3, 3), Carray(3, 3), Darray(3, 3)
Real, Target :: Earray(2, 2), Farray(3, 3)
Real :: Ashft(4,4), Bshft(4,4)
Real :: Vec(10), Vect1(-4:5)
Real :: vector1(5), vector2(5)
```

The G1 array is set up on the next line for *dynamic redimensioning*, which is implemented further down, while on the following line several of the variables are declared and initialized at the same time. Data statements are prohibited in Elf90.

```
Real, Dimension (:,:), Allocatable :: G1
Real :: A= .7132, B = .532, Numgrad, Magnit, Diam = 4.2, Circum,  &
   &    Area
Real :: Dotprod, Top_of_the_line
Real :: Angle, Xval = -1.0, Yval = -1.0
```

A *double precision* real variable is declared in the following line; note that some compilers use kind = 2 to specify double precision. The line after that declares a *pointer* variable.

```
Real (kind = 8) :: Pidoub
Real, Pointer :: Apoint(:,:)

Character (len=20)  :: Name
Character (len=50)  :: Warning
Character (len=10)  :: Strng
Character (len=1)   :: Ltrgrad
Character (len=8)   :: date
Character (len=10)  :: time
Character (len=80)  :: Linetext
Character (len=1)   :: Aselect

Complex :: Z

Logical :: Mask(Imax, Jmax)
Logical :: Check = .true.
```

The next few lines set up a *derived data type* and then declare the Flintstone vector to be a variable of that type:

```
Type :: Person
            Character (len =20)  :: Full_Name
            Character (len = 7)  :: Sex
            Integer :: Age, Weight
            Character (len = 20) :: Occupation
End Type Person
Type (Person) :: Flintstone(4)
```

Statement functions would ordinarily appear here, but are prohibited in Elf90.

Explicit interfaces are needed for all *external subprograms* (in Elf90); two of them are seen here, while the subprograms themselves appear after the end of the main program:

```
Interface
        Function Hypot(L,M)
        Implicit None
        Integer, Intent(IN) :: L, M
        Real :: Hypot
        End Function Hypot

        Subroutine Circle(Diam, Circum, Area)
        Implicit None
        Real, Intent (IN) :: Diam
        Real, Intent (OUT) :: Circum, Area
        End Subroutine Circle

End Interface
```

The following group includes *assignment* statements. Multiple statements are allowed on one line in some Fortran 90's, although not Elf90. One of the statements is continued on the following line; the ampersand in column 73 is used for compatibility with xlf90.

```
Epsilon = 3.0
Zeta    = 0.5
Eta = Alpha + Beta + Gamma                              &
&    + Epsilon + Zeta
Top_of_the_line = 17.0

Write(6,100,advance='no')                   ! Non-advancing IO
  100 format(' Please input your name:')
Read(5,fmt=1020) Name
Write(6,fmt=1030) Name
Write(6,*) 'Top of the line = ',Top_of_the_line
```

Repetition Structures

This section focuses on *repetition* structures, the first of them a simple *Do Loop*. Note that line numbered loops are not allowed in Elf90. The Read(5,*) statements are used to stop the program temporarily pending user action; the earlier "Pause" statement being considered obsolescent. Within the Do Loop both an *intrinsic* (library) function (Sin) and a *user-defined* function (Square) are invoked. The latter is defined below. The values are printed using *list-directed* output.

```
Write(6,*) 'Next is output from simple DO LOOP:'
Read(5,*)
Do J = 1,10
        Vec(J) =  Sin(Real(J)) + Square(J)
        Vect1(J-5) = Real(J)
        Write(6,*) Vec(J),Vect1(J-5)
End Do
Write(6,*)
```

This nested Do Loop uses names to indicate the beginning and termination of each loop. An *Implied Do Loop* is also used in the last Write statement.

```
Write(6,*) 'Next is output from Nested DO LOOP:'
Read(5,*)
Fred : Do J = Jmax,1,-2
        Barney : Do I = 1,Imax,2
                Array(I,J) = Hypot(I,J)
                Write(6,fmt=1000) I,J,Array(I,J)
        End Do  Barney
        Write(6,fmt=1010) (Array(I,J), I = 1,Imax,2)
End Do  Fred
Write(6,*)
```

The following demonstrates a pretest loop, the *Do While*. Note that in Watfor 77 this was implemented as the non-standard While ( ) do … End While.

```
Write(6,*) 'Next is output from DO WHILE:'
Read(5,*)
Do While (k .lt. 10)
        K = K + 1
        Write (6,*) K
End Do
Write(6,*)
```

The next *Do - End Do* with an EXIT allows the stopping criterion to be placed anywhere in the loop and is preferred over previous repetition structures:

```
Write(6,*) 'Next is output from DO - END DO with an EXIT:'
Do
        If (k > 20) Exit
        k = k + 1
        Write(6,*) k
End Do
Write(6,*)
```

Selection

A full complement of syntax is available for *selection*. These include the *Logical If, Block If, If - Else If* and the *Case* structure. The following is a simple *Logical If*:

```
Read(5,*)
Write(6,*) 'Next is output from LOGICAL IF statement:'
IF (K .GE. 0) Write(6,1040) Name
Write(6,*)
```

The next is a *Block If* statement with compound logical operator. Note use of symbolic representation (rather than, e.g., .GE., .LT.) for the relational operators.

```
Read(5,*)
Write(6,*) 'Next is output from BLOCK IF statement:'
If (Jmax <= 100 .and. Imax >= 0 .and. k > -20) Then
        Do J = 1,3
                Vec(J) = 0.0
```

```
                    Do I = 2,4
                        Vec(J) = Vec(J) + Real(I)**3
                    End Do
            End Do
            Write(6,fmt=1050) J, Vec(J), Vec(J)
End If
Write(6,*)
```

The following *If, Else IF, End If* structure allows multiple alternatives. The first few lines seek input of the numerical grade, which is then converted to a letter grade. The user is remonstrated for doing poorly using an illustration of *concatenation*. Iostat is used to trap input errors. The decision structure itself is given the name "Grade_Assign".

```
Write(6,*)  'Next is output from IF - ELSE IF:'
Write(6,*)  'Input your numerical grade, please. (0. - 100.)'
Read(5,*,Iostat = Badnum) Numgrad
Do While (Badnum .GT. 0)
            Badnum = 0
            Write(6,*) 'Bad Data, Try again.'
            Write(6,*) 'Input your numerical grade, please. (0. -100.)'
            Read(5,*,Iostat = Badnum) Numgrad
End Do

Grade_Assign: IF (Numgrad .LT. 60.) Then
            Ltrgrad = 'F'
Else IF (Numgrad .LT. 70.) Then
            Ltrgrad = 'D'
Else IF (Numgrad .LT. 80.) Then
            Ltrgrad = 'C'
Else IF (Numgrad .LT. 90.) Then
            Ltrgrad = 'B'
Else
            Ltrgrad = 'A'
End If Grade_Assign
Write(6,fmt=1060) Numgrad, Ltrgrad
If (Numgrad < 65 ) Then
            Warning = 'Shape up, '
            Write(6,*) Trim(Warning)//' '//Trim(Name)//'!'
End If
Write(6,*)
```

The *Select Case - End Select* construct will take care of multiple alternatives also. Caseno is the decision variable for which the user is first prompted.

```
Write(6,*) 'Next is output from SELECT - END SELECT:'
Write(6,*) 'Input an integer, please:'
Read(5,*,Iostat = Badnum) Caseno
Do While (badnum .gt. 0)
            Badnum = 0
            Write(6,*) 'Bad Data, Try again.'
            Write(6,*) 'Input an integer, please:'
            Read(5,*,Iostat = Badnum) Caseno
End Do

Select Case(Caseno)
Case (-1000 : 0)
```

```
            Write(6,*) 'This is Case 1'
Case (1 : 4)
            Write(6,*) 'This is Case 2'
Case (5, 7, 9)
            Write(6,*) 'This is Case 3'
Case (6, 8, 10:1000)
            Write(6,*) 'This is Case 4'
Case Default
            Write(6,*) 'This is everything else'
End Select
Write (6,*)
```

Miscellaneous

In the following call of an external subroutine the variables Delta and Theta are shared via modules (one and two above), while other variables are passed as arguments.

```
Write(6,*) 'Next is output from SUBROUTINE:'
Read (5,*)
Call Circle(Diam, Circum, Area)
Write(6,fmt=1100) Diam, Circum, Area
Write(6,*) 'Delta =  ', Delta
Write(6,*) 'Theta =  ', Theta
Write(6,*)
```

Recursive functions are permitted in Fortran 90; here the factorial is computed using an *Internal Function* which is defined just *before* the end of the program below.

```
Write(6,*) 'RECURSIVE FUNCTIONS.'
Write(6,*) '(Handled as an internal function)'
Write(6,*) 'Input the value of an integer less than or equal '
Write(6,*) 'to 12 that you want to compute the factorial of:'

Read(5,*) n
Write(6,fmt=1130) n, F(n)
If (n > 12) Then
            Write(6,*) 'WARNING!'
            Write(6,*) 'The computed value is WRONG!  The correct value is'
            Write(6,*) 'greater than the biggest integer allowed!'
End if
```

This one uses an *External Function*, which is also at the bottom of the file just *after* the program end:

```
Write(6,*)
Write(6,*) 'Hypotenuse = ', Hypot(L,M)
```

Here are some operations involving *complex numbers*, including use of several intrinsic functions, while the brief section after this uses a logical variable:

```
Write(6,*) 'Next is output from COMPLEX ARITHMETIC:'
Read(5,*)
Z = Cmplx(A, B)
Magnit = Abs(Z)
Write(6,*) Z
```

```
Write(6,fmt=1080) Real(Z)
Write(6,fmt=1090) Aimag(Z), Magnit
Write(6,*)

IF (Check) Write (6,*) 'Check was True'
Write(6,*)
```

The following use of *integer arithmetic* allows one to count by 5's or 10's, etc. It would be helpful, for example, if one just wanted to print out, say only every fifth item in a list.

```
Write(6,*) 'Next is output based on INTEGER ARITHMETIC:'
Read(5,*)
Do K = 1,100
        IF (K/10*10 == K) Write(6,*) K
End Do
Write(6,*)
```

Besides demonstrating the use of *"Double Precision"* (Real*8), this example shows how to find pi from the inverse cosine.

```
Write(6,*) 'Next is "DOUBLE PRECISION" (Real*8 - 15-16 significant&
  & figures.)'
Write(6,*) '(and shows how to find pi from inverse cosine.)'
Read(5,*)
Pidoub = acos(-1.0)
Write(6,*) Pidoub
Write(6,*)
```

One often needs to distinguish between the Atan2 and the Atan intrinsic functions. Atan takes a single argument and runs only between -pi/2 and pi/2; Atan2 takes two arguments and runs between -pi and +pi. The latter will return the appropriate quadrant for the angle, while the former may not.

```
Angle = Atan(Yval/Xval)
Write(6,*) 'Angle using Atan function = ', Angle
Angle = Atan2(Yval,Xval)
Write(6,*) 'Angle using Atan2 function = ', Angle
```

Sometimes the Date and Time functions are handy:

```
Write(6,*)
Write(6,*) 'Next gives the DATE and TIME:'
Read(5,*)
Call date_and_time(date,time)
Write(6,*)  'date = ',date
Write(6,*)  'time = ',time
Write(6,*)
```

Internal Write statements may be used to change an integer or real to a *character variable* so that it can be used in a legend or label.

```
Write(6,*) 'Next is output from INTERNAL WRITE section:'
Read(5,*)
Write(Strng,'(I10)') I
Write(6,fmt=1021) 'I =', Strng
```

```
Write(Strng,'(F5.2)') Alpha
Write(6,fmt=1021) 'Alpha =', Strng
Write(Strng,'(1PE9.2)') Alpha
Write(6,fmt=1021) 'Alpha =', Strng
```

The next section assigns values to the *Data Structures* declared above and manipulates them using functions such as *Concatenation* and *Trim.* In the first part data is assigned to each of the four vectors, while in the second section (the Do Loop), the data is formatted and printed out.

```
Write(6,*)
Write(6,*) 'Use of DATA STRUCTURE:'
Write(6,*)
Flintstone(1) = Person("Fred_Flintstone","male",38,190,"quarryman"&
   &)
Flintstone(2) = Person("Wilma_Flintstone","female",36,120,"systems&
   & analyst")
Flintstone(3) = Person("Pebbles_Flintstone","female",8,75,"student&
   &")
Flintstone(4) = Person("Dino_Flintstone","male",3,120,"pet")

Do k = 1, 4
        Write(Strng, '(I2)') Flintstone(k)%Age
        If (Strng(1:1) == '8' .or. Strng(1:2) == ' 8') Then
                Write(6,*) Trim(Flintstone(k)%full_name)//' is an '      &
           &     //Trim(Strng)//'-year-old, '//Trim(Flintstone(k)%Sex)// &
           &     ' '//Trim(Flintstone(k)%Occupation)//'.'
        Else
                Write(6,*) Trim(Flintstone(k)%full_name)//' is a '       &
           &    //Trim(Strng)//'-year1-old, '//Trim(Flintstone(k)%Sex)// &
           &     ' '//Trim(Flintstone(k)%Occupation)//'.'
        End If
End Do

Write(6,*)
Linetext = Flintstone(1)%Full_name
If (Flintstone(1)%Weight > Flintstone(2)%Weight) then
        Write(6,*) Trim(Linetext)//' weighs more than '//          &
   &    Trim(Flintstone(2)%Full_Name)//'.'
Else if (Flintstone(1)%Weight < Flintstone(2)%weight) then
   Write(6,*) Trim(Linetext)//' weighs less than '//               &
   &    Trim(Flintstone(2)%full_name)//'.'
Else
Write(6,*) Trim(Linetext)//' and '//                               &
   &  Trim(Flintstone(2)%full_name)//'weigh the same.'
End If
Write(6,*)
```

Simplified Array Operations

If one is involved in finite-difference, finite-volume or finite-element calculations (as in a CFD course), it is the simplified array operations that make Fortran 90 so attractive. Fortran 90 allows one to assign values to a whole array at a time, add whole arrays element-by-element without having to have nested Do Loops, operate on sections of arrays, etc. The line count for typical scientific computing operations is reduced considerably. Here two 2-dimensional arrays

are each initialized (to a constant value) in only one line apiece and then are added - without needing Do Loops for either operation.

```
Barray = 3.
Carray = 4.
Write(6,*) 'Next is an ARRAY ADDITION:'
Read(5,*)
Darray = Barray + Carray
Do i = 1,3
        Write(6, fmt=1010) (darray(i,j), j =1,3)
End Do
Write(6,*)
```

One can also address just a particular section of an array:

```
Carray(1:2,3) = 0.0
```

The programmer can set up *masks* to work with the *Where* construct in order to treat different sections of an array differently. Here they are used to handle the left, right, bottom and top rows/columns of an array separately from the interior, as one might do in setting boundary conditions.

```
Mask = .true.                   ! All entries
Mask(1,   1:jmax) = .false.     ! Left side.
Mask(imax,1:jmax) = .false.     ! Right side.
Mask(1:imax,1   ) = .false.     ! Bottom
Mask(1:imax,jmax) = .false.     ! Top

Write(6,*) 'Next is use of the Where construct:'
Read (5,*)
Array = 9.9                     ! Initialize whole array here, then reset part
Where (.not. mask)              ! of it (the boundaries) in this statement.
        Array = 0.0
End where
Do j = 10,1,-1
        Write(6, fmt=1010) (array(i,j), i=1,10)
End Do
```

The *Cshift* (Circular Shift) and *Eoshift* (End off shift) allow one to access neighbors (as one frequently does while solving partial differential equations numerically) without having to use i,j subscripts in conjunction with Do Loops. The Eoshift operator loses data shifted over the edge of the array; while the circular shift brings it back in on the opposite edge.

```
      Read(5,*)
      Write(6,*) 'Next is use of CSHIFT/EOSHIFT operators - Original arr&
         &ay:'
      Write(6,fmt=119) 'Label1', 'Label2', 'Label3', 'Label4'
  119 Format(t12, A6, t32, A6, t52, A6, t72, A6)    ! Note use of T (Tab) descriptor
      Do J = 1,4
              Do I = 1,4
                      Ashft(i,j) = 10*i + j             ! This is just to initialize the array.
              End Do
      End do
      Do J = 4, 1, -1
            Write(6,fmt=120) (Ashft(i,j), i = 1,4)
```

```
        End Do
120   Format(t10, f8.3, t30, f8.3, t50, f8.3, t70, f8.3)          ! Note use of Tab (t) descriptor

        Bshft = Cshift(Ashft, 1, Dim = 2)
        Write(6,*)
        Write(6,*) 'Here the array is shifted down 1 using CSHIFT (circular shift)'
        Do J = 4, 1, -1
                Write(6,120) (Bshft(i,j), i = 1,4)
        End do

        Bshft = Eoshift(Ashft, -1, Dim = 1)
        Write(6,*)
        Write(6,*) 'Here the array is shifted to right using EOSHIFT (end-off shift)'
        Do J = 4, 1, -1
                Write(6,fmt=120) (Bshft(i,j), i = 1,4)
        End do
```

Dynamic array allocation, a major new feature of Fortran 90, allows one to allocate only the needed array space at run time, rather than having to lock it in at compilation time as in Fortran 77. This means that the programmer does not need to know how big the user needs the arrays to be at the time the program is developed. This way, for example, the maximum grid size in a typical multi-dimensional field solution can be set up to be limited only by the available memory of the user's computer.

```
        Read(5,*)
        Write(6,*) 'Next is use of DYNAMIC REDIMENSIONING.'
        Write(6,*) 'Input dimensions of a 2-D array separated by comma.'
        Write(6,*) '(It would be preferable to keep the integers less'
        Write(6,*) 'than 10 or so to minimize the printing later.)'
        Read(5,*)  mg1, ng1
        Allocate (G1(mg1,ng1))                    !  Allocate the desired memory
        Do J = 1, NG1
                Do I = 1, mg1
                        G1(i,j) = i-j             !  Assign some values.
                End do
                Write(6,1150) (G1(i,j), i = 1, mg1)
        End do
        Deallocate (G1)         !  Return the memory.
```

There are also plenty of new intrinsic functions specifically for vectors. Here a couple are used:

```
        Write(6,*)
        Write(6,*) 'Next is use of an INTRINSIC FUNCTION for vectors:'
        Read(5,*)
        Call random_seed()                        ! Use the supplied random number
        Call random_number(vector1)               ! generator to get some values.
        Call random_number(vector2)
        Dotprod = dot_product(vector1,vector2)
        Write(6,fmt=1120) dotprod
        Write(6,*)
```

The following example shows a use of *Pointers* and *Targets*.

```
        Write(6,*)
```

```
Write(6,*) 'Next is use of POINTERS and TARGETS for arrays:'
Read(5,*)
Earray = 5.                    ! Initialize Earray
Farray = 7.                    ! Initialize Farray
Write(6,*) 'Chose the array you would like to display, Earray or F&
   &array'
Write(6,*) 'Enter e or f (it must be lower case)'
Read(5,*) Aselect

If (Aselect == 'e') Apoint => Earray          ! Assign Apoint to Earray
If (Aselect == 'f') Apoint => Farray          ! Assign Apoint to Farray
Write(6,*) Apoint
Nullify (Apoint)
Write(6,*)
```

File processing

This section demonstrates a number of ways to create and read from data files. All data is written here only to "Scratch" files, so no files will be left when this job is done. The first example is of *List directed* I/O, while the second uses *formatted* I/O.

```
Write(6,*) 'Next is output from file processing section (LIST-DIRE&
   &CTED):'
Read(5,*)
Open (Unit = 12, Status = 'Scratch')
Write(12,*) Vec
Rewind(12)
Read(12,*) Vec
Write(6,*) Vec
Close(12)

Write(6,*)
Write(6,*) 'Next is output from the file processing section (FORMA&
   &TTED): '
Read(5,*)
Open(unit = 15, status = 'scratch')
Write(15, fmt=1140) vec
Read(15,fmt = 1140) vec
Rewind(15)
Write(6, fmt=1140) vec
Close(15)                      ! Files should be closed at end.
```

The next section demonstrates *Unformatted* I/O. The default record length might have to be increased to handle longer vectors. The file is written, then read unformatted, then printed list-directed. If the data is to be written and then read back in again in the same environment, then unformatted I/O is much faster than formatted.

```
Write(6,*) 'Next is output from file processing section, written a&
   &nd read unformatted, then printed list-directed.'
Read(5,*)
Open (Unit = 13, Form = 'Unformatted', Status = 'Scratch')
Write(13) Vec
Rewind(13)
Read(13) Vec
Write(6,*) Vec
```

```
      Close(13)
```

The "End = " construct demonstrated here may be useful when the exact number of entries in a file are not known.

```
      Open(Unit = 14, Status = 'Scratch')
      Write(6,*)
      Write(6,*) 'Next is when you do not know the exact number of entri&
         &es in a file'
      Do I = 1,7
               Write(14,*) Vec(I)
      End do
      Rewind(14)
      Do I = 1, 10
               Read(14,*,End = 420) Vec(I)
               Write(6,*) I, Vec(I)
      End Do
420   I = I-1
      Write(6,fmt=1110) I
      Close(14)
```

Format statements

All the format statements used above are grouped in the next section. Note that the use of _x, i.e., 3x, to give blanks is forbidden in Elf90; one uses ' ' instead. Hollerith (_h) notation is a thing of the past.

```
1000 Format('  ','I = ',I3,'  J = ',I3,'  array(I,J) = ',F9.2)
1010 Format(5('  ',EN10.3))   ! Note multiple!
1020 Format(A20)
1021 Format(2A10)
1030 Format(//,' You are in for the thrill of your life, ',A20,/)
1040 Format(' We''re doing well, ',A20)
1050 Format('  ',I5,'  ',E15.8,'  ',EN11.4)
1060 Format('  ','Number Grade =', F8.1,'     ', 'Letter Grade =',A2)
1080 Format('  ','Magnitude of a complex number whose real part is',F6.&
   &2)
1090 Format('  ','and imaginary part is', F6.2, '  is:',F6.2)
1100 Format('  ','Diameter = ',F6.2,'    Area = ',F6.2,'    Circumferen&
   &ce = ',F9.2)
1110 Format('  ','There were ',I2,' entries in the file.',/)
1120 Format('  ','Dotproduct of the two vectors = ', f10.4)
1130 Format('  ','The factorial of ',I3, ' = ', I20)
1140 Format(4('  ',e15.8))
1150 Format(5g15.8)
```

Here is the end of the main program (the text following the Stop will be printed on-screen for the user) and the internal subprograms follow:

```
      Stop '**** O.K.'
      Contains
```

Subprograms

Internal subprograms, here one simple and one *recursive* function, are grouped after the "contains" and before the program end:

```
Function Square(J)
Implicit None
Real:: Square
Integer, Intent(In) :: J
Square = Real(J**2)
Return
End Function Square

Recursive Function F(n)  Result(Fac)
!    This calculation of the factorial is set up as an internal function here.  Note that it
!    calls itself.
!    WARNING:  This gives a wrong answer for n > 12 !!

Implicit None
Integer, Intent(In) :: N
Integer :: Fac
If (n == 1) then
            Fac = 1
Else
            Fac = N*f(n-1)
End If
Return
End Function F

End Program ref90a
```

After the end of the program are the *external subprograms*. The first is a function, while the other is a subroutine. The *Intent(IN)* causes the compiler to balk if the subprogram attempts to change a variable that is supposed to be only an input. *Intent(OUT)* means the variable is an output, while *Inout* can be either. This feature is highly useful when several programmers work on the same job. It is used in the internal functions above as well.

```
Function Hypot(L,M)
Implicit None
Integer, Intent(IN) :: L, M
Real :: Hypot
Real :: Square2
Square2 = Real(L)**2 + Real(M)**2
Hypot =  Sqrt(Square2)
Return
End Function Hypot

Subroutine Circle(Diam, Circum, Area)
Use one              ! Use of a module to share data
Use two
Implicit None
Real, Intent(In)  :: Diam
Real, Intent(Out) :: Circum, Area
Real :: PI, H

Pi = Acos(-1.0)
Circum = Pi * Diam
Area = Pi * Diam**2 / 4.0
```

```
Delta = Alpha + Beta + Gamma
h = epsilon + zeta + theta
Area = h*area
Diam = 3.0  ! The compiler would choke if this weren't commented out because it has
!  been declared above as an input.
Return
End Subroutine Circle
```

Conclusion

Within the rambling program above are enough examples of syntax to get the Fortran 90 neophyte started and to keep the intermediate programmer engaged. Indeed, there are many more structures included than would ever appear in a typical scientific program. Most of the books listed below as references delve into at least the topics presented here, as well as many other Fortran 90 features not even mentioned here.

References

1. W.S. Brainerd, C.H.Goldberg, and J.C. Adams, *Programmer's Guide to Fortran 90*, 2nd Ed., Unicomp, Albuquerque (1994).

2. G. Coschi and J.B.Schueler, *WATFOR-77 Language Reference Manual*, WATCOM Publications Limited, Waterloo, Canada (1985).

3. *Elf90 Essential Lahey Fortran*, Revision B, Lahey Computer Systems (1996).

4. J.F. Kerrigan, *Migrating to Fortran 90*, O'Reilly and Associates, Sebastopol, CA, (1993).

5. W.E. Mayo and M. Cwiakala, *Introduction to Computing for Engineers*, McGraw-Hill, NY (1991).

6. *Nagware FTN90 User's Guide*, Salford Software (1995).

7. L. Nyhoff and S. Leestma, *Introduction to FORTRAN 90 for Engineers and Scientists*, Prentice-Hall (1997).

8. J.M.Ortega, *An Introduction to Fortran 90 for Scientific Computing*, Saunders (1994).

9. W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery, M.Metcalf, *Numerical Recipes in Fortran 90, The Art of Scientific Computing, Vol. 2 of Fortran Numerical Recipes*, Cambridge University Press, New York (1996).

Appendix - Some Sources of Fortran 90 Compilers

Numerical Algorithms Group (NAG), Inc.
(630) 971-2337

http://nag.com

Lahey Computer Systems, Inc.
865 Tahoe Blvd., P.O.,Box 6091
Incline Village, NV 89450-6091
(800) 548-4778
sales@lahey.com
htp://www.lahey.com

Digital (the compiler formerly known as Microsoft Fortran)
http://www.digital.com/fortran/dvf-spd.html

Absoft
http://www.absoft.com

IBM xlf90
http://www.software.hosting.ibm.com/ad/fortran/xlfortran/function.html

Copyright and Disclaimer