Introduction to F

Fortran (FORmula TRANslation) was introduced in 1957 and remains the language of choice for most scientific programming. The latest standard, Fortran 90, includes extensions that are familiar to users of C. Some of the most important features of Fortran 90 include recursive subroutines, dynamic storage allocation and pointers, user defined data structures, modules, and the ability to manipulate entire arrays.

Fortran 90 is compatible with Fortran 77 and includes syntax that is no longer considered desirable. F is a subset of Fortran 90 that includes only its modern features, is compact, and is easy to learn.

Instructions for running F on Unix workstations and on the Power Macintosh are available.

1. Introduction

In the following, we will assume that the reader is familiar with a programming language such as True BASIC. We first give an example of an F program.

```
program product_example
   real :: m, a, force
   m = 2.0                      ! mass in kilograms
   a = 4.0                      ! acceleration in mks units
   force = m*a                  ! force in newtons
print *, force
end program product_example
```

This program is similar to **Program product** in Chapter 2 of Gould & Tobochnik. The features of F included in the above program include:

- The first statement must be a **program** statement; the last statement must have a corresponding **end** program statement.

- Integer numerical variables and floating point numerical variables are distinguished. The names of all variables must be between 1 and 31 alphanumeric characters of which the first must be a letter and the last must not be an underscore.

- The types of all variables must be declared.

- Real numbers are written as 2.0 rather than 2.

- The case is significant, but two names that differ only in the case of one or more letters cannot be used together. All **keywords** (words that are part of the language and cannot be redefined.) are written in lower case. Some names such as **product** are reserved and cannot be used as names.

- Comments begin with a **!** and can be included anywhere in the program.

‣ Statements are written on lines which may contain up to 132 characters.

‣ The asterisk (*) following **print** is the default format.

We next introduce syntax that allows us to enter the desired values of **m** and **a** from the keyboard. Note the use of the (unformatted) read statement and how character strings are printed.

```
program product_example2
   real :: m, a, force
   ! SI units
   print *, "mass m = ?"
   read *, m
   print *, "acceleration a = ?"
   read *, a
   force = m*a
   print *, "force (in newtons) =", force
end program product_example2
```

2. Do construct

F uses a **do** construct to have the computer execute the same statements more than once. An example of a **do** construct follows:

```
program series
   integer :: n
   real :: sum_series       ! sum is a keyword
   sum_series = 0.0
!  add the first 100 terms of a simple series
   do n = 1, 100
      sum_series = sum_series + 1.0/real(n)**2
      print *, n,sum_series
   end do
end program series
```

Note that **n** is an **integer** variable. In this case the do statement specifies the first and last values of **n**; n increases by unity (default). The block of statements inside a loop is indented for clarity.

Because the product **n*n** is done using integer arithmetic, it is better to convert **n** to a real variable before the multiplication is done. Also note that exponentiation is done using the operator ****.

3. If construct

In the next program example, the do loop is exited by satisfying a test.

```
program series_test
!  illustrate use of do construct
   integer :: n
!  choose large value for relative change
   real :: sum_series, newterm, relative_change
   n = 0
   sum_series = 0.0
   do
      n = n + 1
      newterm = 1.0/(n*n)
      sum_series = sum_series + newterm
      relative_change = newterm/sum_series
      if (relative_change < 0.0001) then
         exit
      end if
      print *, n,relative_change,sum_series
   end do
end program series_test
```

The new features of F included in the above program include:

‣ A **do** construct can be exited by using the **exit** statement.

‣ The **if** construct allows the execution of a sequence of statements (a block) to depend on a condition. The **if** construct is a compound statement and begins with **if** ... **then** and ends with **end if**. The block inside the **if** construct is indented for clarity. Examples of more general **if** constructs using **else** and **else if** statements are given in Program test_factorial.

Table 1. Summary of relational operators.

relation	**operator**
less than	<
less than or equal	<=
equal	==
not equal	/=
greater than	>
greater than or equal	>=

The following program illustrates the use of a **kind parameter** and a named **do** construct:

```
   program series_example
   !  illustrate use of kind parameter and named do loop
      integer, parameter :: double = 8
      integer :: n
      real (kind = double) :: sum_series, newterm, relative_change
      n = 0
      sum_series = 0.0
      print_change: do
         n = n + 1
         newterm = 1.0/real(n, kind = double)**2
         sum_series = sum_series + newterm
         relative_change = newterm/sum_series
         if (relative_change < 0.0001) then
            exit print_change
         end if
         print *, n,relative_change,sum_series
      end do print_change
   end program series_example
```

‣ Variables may have a particular hardware representation such as double precision by using the **kind = double** in parenthesis after the keyword **real** representing the data type. A more general use of the **parameter statement** is given in Program drag.

‣ Double precision in Fortran 90 on the SGI and in F on the Power Macintosh is done by letting

double = 8

However double precision in F on the SGI is done by letting

double = 2

‣ The **do** and **end do** statements must either have the same name or both be unnamed. In general, a **do** construct is named to make explicit which **do** construct is exited in the case of nested do constructs. The use of a named **do** construct in the above example is unnecessary and is for illustrative purposes only.

Subprograms are called from the main program or other subprograms. As an example, the following program adds and multiplies two numbers that are inputed from the keyboard. Note that the variables **x** and **y** are **public** and are available to the main program.

```
module common
   public :: initial,add,multiply
   integer, parameter, public :: double = 8
   real (kind = double), public :: x,y

contains

subroutine initial()
   print *, "x = ?"
   read *,x
   print *, "y = ?"
   read *,y
end subroutine initial

subroutine add(sum2)
   real (kind = double), intent (in out) :: sum2
   sum2 = x + y
end subroutine add

subroutine multiply(product2)
   real (kind = double), intent (in out) :: product2
   product2 = x*y
end subroutine multiply

end module common

program tasks            ! illustrate use of module and subroutines
! note how variables are passed
   use common

   real (kind = double) :: sum2, product2
   call initial()                 ! initialize variables
   call add(sum2)                 ! add two variables
   call multiply(product2)
   print *, "sum =", sum2, "product =", product2
end program tasks
```

‣ Subprograms (subroutines and functions) are contained in modules. The form of a module, subroutine, and a function is similar to that of a main program.

‣ A module is accessed in the main program by the **use** statement.

‣ Subroutines are invoked in the main program by using the **call** statement.

‣ A subprogram always has access to other entities in the module.

‣ The subprograms in a module are preceded by a **contains** statement.

‣ Variables and subprograms may be declared **public** in a module and be available to the main program (and other modules).

‣ Information can also be passed as an argument to each subprogram as are the variables sum2 and product2. A open parenthesis () is needed even if there are no arguments. The **intent** of each dummy argument of a program must be indicated.

- intent **in** means that the dummy argument cannot be changed within the subprogram.
- intent **out** means that the dummy argument cannot be used within the subprogram until it is given a value with the intent of passing a value back to the calling program.
- intent **in out** means that the dummy argument has an initial value which is changed and passed back to the calling program. (It also is correct to write **inout**.)

‣ The module(s) can be a separate file.

4. Formatted output

The structure of Program cool is similar to Program tasks. Note the use of the **modulo** function and the use of format specifications. In place of the asterisk * denoting the default format, we have used a **format specification** which is a list of **edit descriptors**. An example from Program cool is

```
print "(t7,a,t16,a,t28,a)", "time","T_coffee","T_coffee - T_room"
```

The **t** (tab) edit descriptor is used to skip to a specified position on an output line. The edit descriptor **a** (alphanumeric) is for character strings. An example of the **f** (floating point) descriptor is given by

```
print "(f10.2,2f13.4)",t,T_coffee,T_coffee - T_room
```

The edit descriptor **f13.4** means that a total of thirteen positions are reserved for printing a real value rounded to 4 places after the decimal point. (The decimal point and a minus sign occupy two out of the thirteen positions.) The edit descriptor **2f13.4** means that the edit descriptor **f13.4** is done twice. The other common edit descriptor is **i** (integer).

Comment on Program drag
The only new syntax in Program drag is the use of the parameter statement:

```
real (kind = double), public, parameter :: g = 9.8
```

A **parameter** is a **named constant**. The value of a parameter is fixed by its declaration and cannot be changed during the execution of a program.

5. Files

Program save_data illustrates how to open a new file, write data in a file, close a file, and read data from an existing file.

```
program save_data
!  illustrate writing and reading file
   integer :: i,j,x
   character(len = 32) :: file_name
   print *, "name of file?"
   read *, file_name
   open (unit=5,file=file_name,action="write",status="new")
   do i = 1,4
```

```
         x = i*i
         write (unit=5,fmt=*) i,x
      end do
      close(unit=5)
      open (unit=1,file=file_name,action="read",status="old")
      do i = 1,4
         read (unit=1,fmt = *) j,x
         print *, j,x
      end do
      close(unit=5)
   end program save_data
```

Input/output statements refer to a particular file by specifying its unit. The read and write statements do not refer to a file directly, but refer to a file number which must be connected to a file. There are many variations on the open statement, but the above example is typical. The values of the action specifier are read, write, and readwrite (default). Values for status are old, new, replace, or scratch.

If we plan to reuse data on the same system with the same compiler, we can use unformatted input/output to save the overhead, extra space, and the roundoff error associated with the conversion of the internal representation of a value to its external representation. Of course, the latter is machine and compiler dependent. Unformatted access is very useful when data is generated by one program and then analyzed by a separate program on the same computer. To generate unformatted files, omit the format specification. Examples of programs which use direct access and records are available.

6. Arrays

The definition and use of arrays is illustrated in Program vector.

```
   module common
      public :: initial,cross

      contains

      subroutine initial(a,b)
         real, dimension (:), intent(out) :: a,b
         a(1:3) = (/ 2.0, -3.0, -4.0 /)
         b(1:3) = (/ 6.0, 5.0, 1.0 /)
      end subroutine initial

      subroutine cross(r,s)
         real, dimension (:), intent(in) :: r,s
         real, dimension (3) :: cross_product
         ! note use of dummy variables
         integer :: component,i,j
         do component = 1,3
            i = modulo(component,3) + 1
            j = modulo(i,3) + 1
            cross_product(component) = r(i)*s(j) - s(i)*r(j)
         end do
         print *, ""
         print *, "three components of the vector product:"
         print "(a,t10,a,t16,a)", "x","y","z"
         print *, cross_product
      end subroutine cross

      end module common

      program vector             ! illustrate use of arrays
         use common
         real, dimension (3) :: a,b
         real :: dot
         call initial(a,b)
         dot = dot_product(a,b)
         print *, "dot product = ", dot
         call cross(a,b)
      end program vector
```

The main features of arrays include:

‣ An array is declared in the declaration section of a program, module, or procedure using the dimension attribute. Examples include

real, dimension (10) :: x,y
real, dimension (1:10) :: x,y
integer, dimension (-10:10) :: prob
integer, dimension (10,10) :: spin

‣ The default value of the lower bound of an array is 1. For this reason the first two statements are equivalent to the first.

‣ The lower bound of an array can be negative.

‣ The last statement is an example of two-dimensional array.

‣ Rather than assigning each array element explicitly, we can use an array constructor to give an array a set of values. An array constructor is a one-dimensional a list of values, separated by commas, and delimited by "(/" and "/)". An example is

```
a(1:3) = (/ 2.0, -3.0, -4.0 /)
```

is equivalent to the separate assignments

```
a(1) = 2.0
a(2) = -3.0
a(3) = -4.0
```

‣ Note that the array **cross_product** can be referenced by one statement:

```
print *, cross_product
```

F has many vector and matrix multiplication functions. For example, the function **dot_function** operates on two vectors and returns their scalar product. Some useful array reduction functions are **maxval**, **minval**, **product**, and **sum**.

7. Allocate statement

One of the better features of Fortran 90 is dynamic storage allocation. That is, the size of an array can be changed during the execution of the program. The use of the **allocate** and **deallocate** statements are illustrated in the following. Note the use of the implied do loop.

```
program dynamic_array
!  simple example of dynamic arrays
   real, dimension (:), allocatable :: x
   integer :: i,N
   N = 2
   allocate(x(N:2*N))
!  implied do loop
   x(N:2*N) = (/ (i*i, i = N, 2*N) /)
   print *, x
   deallocate(x)
   allocate(x(N:3*N))
   x = (/ (i*i, i = N, 3*N) /)
   print *, x
end program dynamic_array
```

8. Random number sequences

Fortran 90 includes several useful built-in procedures. One of the most useful ones is **subroutine random_number**. Although it is a good idea to write your own random number generator using an algorithm that you have tested on the particular problem of interest, it is convenient to use subroutine random_number when you are debugging your program or if accuracy is not important. The following program illustrates several uses of subroutine **random_number** and **random_seed**. Note that the argument rnd of random_number must be real, has intent out, and can be either a scalar or an array.

```
program random_example
   real :: rnd
   real, dimension (:), allocatable :: x
   integer, dimension(2) :: seed, seed_old
   integer :: L,i,n_min,n_max,ran_int,sizer
!  generate random integers between n_min and n_max
!  dimension of seed is one in F and two in Fortran 90.
   call random_seed(sizer)
   print *, sizer
!  illustrate use of put and get
   seed(1) = 1239
   seed(2) = 9863        ! need for Fortran 90
   call random_seed(put=seed)
   call random_seed(get=seed_old)
!  confirm value of seed
   print *, "seed = ", seed_old
   L = 100         ! length of sequence
   n_min = 1
   n_max = 10
   do i = 1,L
      call random_number(rnd)
      ran_int = (n_max - n_min + 1)*rnd + n_min
      print *,ran_int
   end do
!  assign random numbers to array x as another example
   allocate(x(L))
   call random_number(x)
   print "(4f13.6)", x
   call random_seed(get=seed_old)
   print *, "seed = ", seed_old
end program random_example
```

Note how subroutine random_seed is used to specify the seed. This specification is useful when the same random number sequence is used to test a program.

9. Recursion

A simple example of a recursive definition is the factorial function

$$\text{factorial}(n) = n! = n(n-1)(n-2) \ldots 1$$

A recursive definition of the factorial is

factorial(1) = 1 factorial(n) = n factorial(n-1)

A program that closely parallels the above definition follows. Note how the word **recursive** is used.

```
module fact

public :: f
contains

recursive function f(n) result (factorial_result)
   integer, intent (in) :: n
   integer :: factorial_result

   if (n <= 0) then
      factorial_result = 1
   else
      factorial_result = n*f(n-1)
   end if
end function f

end module fact

program test_factorial
   use fact

   integer :: n
   print *, "integer n?"
   read *, n
   print "(i4, a, i10)", n, "! = ", f(n)
end program test_factorial
```

A less simple example (taken from pp. 98-99 in *The Fun of Computing,*John G. Kemeny, True BASIC (1990)) is given two integers, n and m, what is their greatest common divisor, that is, the largest integer that divides both? For example, if n = 1000 and m = 32, than the greatest common divisor (gcd) is gcd = 8.

One method for finding the gcd is to integer divide n by m. We write n = q m + r, where q is the quotient and r is the remainder. If r = 0, then m divides n and m is the gcd. Otherwise, any divisor of m and r also divides n, and hence gcd(n,m) = gcd(m,r). Because r < m, we have made progress. As an example, take n = 1024 and m = 24. Then q = 42 and r = 16. So we want gcd(24,16). Now q = 1 and r = 8 and we calculate gcd(16,8). Finally q = 2, and r = 0 so gcd = 8. The following program implements this idea.

```
module gcd_def

public :: gcd
contains

recursive function gcd(n,m) result (gcd_result)
   integer, intent (in) :: n,m
   integer :: gcd_result
   integer :: remainder

   remainder = modulo(n,m)
   if (remainder == 0) then
      gcd_result = m
   else
      gcd_result = gcd(m,remainder)
   end if
end function gcd

end module gcd_def

program greatest
   use gcd_def

   integer :: n,m
   print *, "enter two integers n, m"
   read *, n,m
   print "(a,i6,a,i6,a ,i6)", "gcd of",n," and",m,"=",gcd(n,m)
end program greatest
```

The example of recursion given in almost all introductory textbooks is the towers of Hanoi. To save space, a discussion is given elsewhere together with the program. (not finished)

The volume of a d-dimensional hypersphere of unit radius can be related to the area of a (d - 1)-dimensional hypersphere. The following program uses a recursive subroutine to integrate numerically a d-dimensional hypersphere:

```
module common
   public :: initialize,integrate

   integer, parameter, public :: double = 8
   real (kind = double), parameter, public :: zero = 0.0
   real (kind = double), public :: h, volume
   integer, public :: d

contains

subroutine initialize()
   print *, "dimension d?"
   read *, d                         ! spatial dimension
   print *, "integration interval h?"
   read *, h
   volume = 0.0
end subroutine initialize

recursive subroutine integrate(lower_r2, remaining_d)
!  lower_r2 is contribution to r^2 from lower dimensions
   real(kind = double),intent (in) :: lower_r2
   integer, intent (in) :: remaining_d  ! # dimensions to integrate
   real (kind = double) :: x

   x = 0.5*h   ! mid-point approximation
   if (remaining_d > 1) then
      lower_d: do
         call integrate(lower_r2 + x**2, remaining_d - 1)
         x = x + h
         if (x > 1) then
            exit lower_d
         end if
      end do lower_d
   else
      last_d: do
         if (x**2 + lower_r2 <= 1) then
           volume = volume + h**(d - 1)*(1 - lower_r2 - x**2)**0.5
         end if
         x = x + h
         if (x > 1) then
            exit last_d
         end if
      end do last_d
   end if
end subroutine integrate

end module common

program hypersphere
!  original program by Jon Goldstein
   use common

   call initialize()
   call integrate(zero, d - 1)
   volume = (2**d)*volume         ! only considered positive octant
   print *, volume
end program hypersphere
```

10. Character variables

The only instrinsic operator for character expressions is the concatenation operator //. For example, the concatenation of the character constants *string* and *beans* is written as

"string"//"beans"

The result, stringbeans, may be assigned to a character variable. A useful example of concatenation is given in the following program.

```
   program write_files
   !  test program to open n files and write data
      integer :: i,n
      character(len = 15) :: file_name
      n = 11
      do i = 1,n
   !     assign number.dat to file_name using write statement
         write(unit=file_name,fmt="(i2.2,a)") i,".dat"
   !     // is concatenation operator
         file_name = "config"//file_name
         open (unit=1,file=file_name,action="write",status="replace")
         write (unit=1, fmt=*) i*i,file_name
         close(unit=1)
      end do
   end program write_files
```

Note the use of the **write** statement to build a character string for numeric and character components.

11. Bit manipulation

12. Complex variables

Fortran 90 is uniquely suited to handle complex variables. The following program illustrates the way complex variables are defined and used.

```
   program complex_example
      integer, parameter :: double = 2
      real (kind = double), parameter :: pi = 3.141592654
      complex (kind = double) :: b,bstar,f,arg
      real (kind = double) :: c
      complex :: a
      integer :: d
      ! A complex constant is written as two real numbers, separated by
      ! a comma and enclosed in parentheses.
      a = (2,-3)
      ! If one of part has a kind, the other part must have same kind
      b = (0.5_double,0.8_double)
      print *, "a =", a      ! note that a has less precision than b
      print *, "a*a =", a*a
      print *, "b =", b
      print *, "a*b =", a*b
      c = real(b)     ! real part of b
      print *, "real part of b =", c
      c = aimag(b)      ! imaginary part of b
      print *, "imaginary part of b =", c
      d = int(a)
      print *, "real part of a (converted to integer) =", d
      arg = cmplx(0.0,pi)
      b = exp(arg)       ! done in two lines for ease of reading only
      bstar = conjg(b)    ! complex conjugate of b
      f = abs(b)       ! absolute value of b
      print *, "properties of b =", b,bstar,b*bstar,f
   end program complex_example
```

13. Pointers

14. References

Walter S. Brainerd, Charles H. Goldberg, and Jeanne C. Adams, *Programmer’s Guide to F*, Unicomp (1996).

Michael Metcalf and John Reid, *The F Programming Language,* Oxford University Press (1996).

15. Links

- related programs
- emacs tutorial
- unix tutorial
- Fortran sites

Please send comments and corrections to Harvey Gould, hgould@clarku.edu.

Updated 9 October 1999.