Source file: `fdemo1.f`

```
c===========================================================
c     fdemo1:  Program which demonstrates many of the
c     essential features of Fortran 77. Some 'safe' language
c     extensions are used; all extensions are valid
c     Fortran 90.
c===========================================================

c===========================================================
c     Source code formatting rules:
c
c     Columns    Use
c
c     1-5        numeric statement label
c     6          continuation character: '&' recommended
c     7-72       statement
c
c     BE EXTREMELY CAREFUL NOT TO TYPE BEYOND COLUMN 72!
c===========================================================
C     COMMENT LINES: Use 'c' 'C' or '*' IN FIRST COLUMN
*===========================================================

c-----------------------------------------------------------
c     The 'program' statement names a Fortran main routine.
c     Optional, but recommended and note that there can
c     only be one 'program' (main routine) per executable.
c-----------------------------------------------------------
      program         fdemo1

c===========================================================
c     BEGINNING OF DECLARATION STATEMENTS
c
c     Declarations (or specification statements) must
c     ALWAYS appear before ANY executable statements.
c===========================================================

c-----------------------------------------------------------
c     The 'implicit none' statement is an extension which
c     forces us to explicitly declare all variables and
c     functions (apart from Fortran built in functions).
c     HIGHLY RECOMMENDED.
c-----------------------------------------------------------
      implicit        none

c-----------------------------------------------------------
c     PARAMETERS
c-----------------------------------------------------------
c     The parameter declaration effectively assigns a
c     CONSTANT value to a name.  Note that each
c     parameter statement must be accompanied by an
c     appropriate declaration of the type of the
c     parameter.  Also note that, except in strings,
c     blanks (spaces) are ignored in Fortran---you can
c     use this fact to make code more readable.
c-----------------------------------------------------------
      integer         zero
      parameter     ( zero = 0 )

c-----------------------------------------------------------
c     Always specify floating point constants using
c     scientific notation.  Use 'd' (instead of 'e') for
c     real*8 constants.
c-----------------------------------------------------------
      real*8          pi
      parameter     ( pi   = 3.141 5926 5358 9793 d0 )

      real*8          tiny
      parameter     ( tiny = 1.0 d-50 )

c-----------------------------------------------------------
c     VARIABLES
c-----------------------------------------------------------
c     The main data types we will be using are
c
c     integer,  real*8,  logical,
c     character*1,  character*2, ... etc., character*(*)
c
c     but note that Fortran has support for complex
c     arithmetic.  Note that complex*16 means real*8
c     values are used for both the real and imaginary
c     parts of the variable.
c-----------------------------------------------------------

c-----------------------------------------------------------
c     (a) SCALARS
c-----------------------------------------------------------
      real*8         a,        b,        c
      real*8         res1,     res2,     res3,     res4
      integer        i,        j,        k,        n
      integer        ires1,    ires2,    ires3,    ires4
      logical        switch
      logical        lres1,    lres2,    lres3
      complex*16     ca,       cb
c-----------------------------------------------------------
c     (b) ARRAYS
c-----------------------------------------------------------
      integer        n1,       n2,       n3
      parameter    ( n1 = 4,   n2 = 3,   n3 = 2)

c-----------------------------------------------------------
c     (b.1) 1-D ARRAYS:  Note, in a main program, all
c     dimension bounds must be integer parameters or
c     integer constants.
c-----------------------------------------------------------
      real*8         r1a(n1),   r1b(n2)
      integer        i1i(n1)
c-----------------------------------------------------------
c     (b.2) 2-D ARRAYS:
c-----------------------------------------------------------
      real*8         r2a(n1,n2)
c-----------------------------------------------------------
c     (b.3) 3-D ARRAYS:
c-----------------------------------------------------------
      real*8         r3a(n1,n2,n3)

c===========================================================
c     END OF DECLARATION STATEMENTS
c===========================================================

c===========================================================
c     BEGINNING OF EXECUTABLE STATEMENTS
c===========================================================

c***********************************************************
c     Assignment statements and simple arithmetic
c     expressions
c***********************************************************

c-----------------------------------------------------------
c     Assignment to scalar variables ... again, note
c     the use of scientific notation (d0) to specify
c     a real*8 constant.
c
c     The only valid logical constants are .true. and
c     .false.  (don't forget to include the .'s)
c-----------------------------------------------------------
      a =  0.025d0
      b = -1.234d-16
      c =  1.0d0
      i =  3000
      switch = .true.

c-----------------------------------------------------------
c     Note the use of the continuation character in
c     column 6 to continue a statement on a second line.
c-----------------------------------------------------------
      write(*,*) 'a = ', a,' b = ', b
      write(*,*) ' c = ', c, ' i = ', i,
     &           ' switch = ', switch
      call prompt('Through scalar assignment')

c-----------------------------------------------------------
c     Arithmetic expressions.  Fortran has standard
c     operator precedences except that the exponentiation
c     operator '**' associates RIGHT to LEFT:  e.g.
c
c     i ** j ** k  is equivalent to  i ** (j ** k)
c
c     Parentheses force evaluation of subexpressions.
c-----------------------------------------------------------
      a = 2.0d0
```

```
      b = 3.0d0
      c = 3.0d0

      res1  = a + b
      res2  = a**2 + b**2
      res3  = (a**2 + b**2)**(0.5d0)
      write(*,*) 'res1 = ', res1,' res2 = ', res2
      write(*,*) ' res3 = ', res3
      call prompt('Through real*8 arithmetic expressions')

c-----------------------------------------------------------
c     Notice the integer truncation which occurs when
c     dividing the integer 2 by the integer 3.
c-----------------------------------------------------------
      i = 2
      j = 3
      k = 2

      ires1 = 2 + 3
      ires2 = 2 / 3
      ires3 = i ** j ** k
      ires4 = (i ** j) ** k
      write(*,*) 'ires1 = ', ires1, ' ires2 = ', ires2
      write(*,*) 'ires3 = ', ires3, ' ires4 = ', ires4
      call prompt('Through integer arithmetic expressions')

c-----------------------------------------------------------
c     "Mixed-mode" computations
c-----------------------------------------------------------

c-----------------------------------------------------------
c     i + j  is  computed using integer arithmetic and
c     the result is converted to a real*8 value before being
c     assigned to res2.
c-----------------------------------------------------------
      res1 = i + j

c-----------------------------------------------------------
c     3 / 4 is evaluated using integer arithmetic (yielding
c     0) and then the value is converted to real*8.
c-----------------------------------------------------------
      res2 = 3 / 4

c-----------------------------------------------------------
c     The appearance of a double precision constant
c     forces the division to be computed using real*8
c     arithmetic
c-----------------------------------------------------------
      res3 = 3 / 4.0d0
      write(*,*) 'res1 = ', res1, ' res2 = ', res2
      write(*,*) ' res3 = ', res3
      call prompt('Through mixed-mode arithmetic')

c***********************************************************
c     CONTROL STATEMENTS
c***********************************************************

c***********************************************************
c     DO LOOPS
c
c     Note that 'end do' is not Fortran 77, but a safe
c     extension (it is legal Fortran 90).
c***********************************************************

      do i = 1 , 3
         write(*,*) 'Loop 1: i = ', i
      end do
      call prompt('Through loop 1')

c-----------------------------------------------------------
c     The same do loop with the optional loop increment
c     specified explicitly
c-----------------------------------------------------------
      do i = 1 , 3 , 1
         write(*,*) 'Loop 2: i = ', i
      end do
      call prompt('Through loop 2')

c-----------------------------------------------------------
c     Another do-loop with a non-default loop increment ...
c-----------------------------------------------------------
      do i = 1 , 7 , 2
         write(*,*) 'Loop 3: i = ', i
      end do
      call prompt('Through loop 3')

c-----------------------------------------------------------
c     ... and one with a negative increment
c-----------------------------------------------------------
      do i = 3 , 1 , -1
         write(*,*) 'Loop 4: i = ', i
      end do
      call prompt('Through loop 4')

c-----------------------------------------------------------
c     Nested do-loops.
c-----------------------------------------------------------
      do i = 1 , 3
         do j = 1 , 2
            write(*,*) 'Loop 5: i, j = ', i, j
         end do
      end do
      call prompt('Through loop 5')

c-----------------------------------------------------------
c     Any of the do-loop parameters can be variables,
c     expressions or parameters: safest to ALWAYS use
c     integer values.
c-----------------------------------------------------------
      n = 6
      do i = 2 , n , n / 3
         write(*,*) 'Loop 6: i = ', i
      end do
      call prompt('Through loop 6')


c***********************************************************
c     LOGICAL EXPRESSIONS
c
c     Note that the Fortran comparison and logical
c     operators all have the form: .operator.
c
c     Comparison:  .eq.    .ne.    .gt.    .lt.
c                  .ge.    .le.
c     Logical:     .not. (unary)
c                  .and.   .or.
c***********************************************************

      a = 25.0d0
      b = 12.0d0

      lres1 = a .gt. b
      lres2 = (a .lt. b)  .or.  (b .ge. 0.0d0)
      lres3 = a .eq. b
      write(*,*) 'lres1 = ', lres1, ' lres2 = ', lres2,
     &           ' lres3 = ', lres3
      call prompt('Through basic conditionals')

c***********************************************************
c     IF-THEN-ELSE STATEMENTS.
c***********************************************************

      if( a .gt. b ) then
         write(*,*) a, ' > ', b
      end if
      call prompt('Through if 1')

      if( b .gt. a ) then
         write(*,*) b, ' > ', a
      else
         write(*,*) a, ' > ', b
      end if
      call prompt('Through if 2')

c-----------------------------------------------------------
c     Nested IF statement.
c-----------------------------------------------------------
      if( a .gt. b ) then
         if( a .gt. 2 * b ) then
            write(*,*) a, ' > ', 2 * b
         else
            write(*,*) a, ' <= ', 2 * b
```

```
         end if
      else
         write(*,*) a, ' <= ', b
      end if
      call prompt('Through nested if')

c-----------------------------------------------------------
c     IF ... ELSE IF .. IF construct can be used in lieu
c     of 'CASE' statement.
c-----------------------------------------------------------
      do i = 1 , 4
         if(      i .eq. 1 ) then
            write(*,*) 'Case 1'
         else if( i .eq. 2 ) then
            write(*,*) 'Case 2'
         else if( i .eq. 3 ) then
            write(*,*) 'Case 3'
         else
            write(*,*) 'Default case'
         end if
      end do
      call prompt('Through case via if')

c***********************************************************
c     WHILE LOOPS
c
c     The do while( ... ) ... end do construct is valid
c     Fortran 90, and a safe Fortran 77 extension.
c***********************************************************
      a = 0.1d0
      b = 0.0d0
      do while ( b .le. 1.0d0 )
         write(*,*) 'Do while loop: b = ', b
         b = b + a
      end do
      call prompt('Through while loop')

c***********************************************************
c     USING BUILT-IN (INTRINSIC) FUNCTIONS
c***********************************************************
      res1 = sin(0.3d0 * Pi)
      res2 = cos(0.3d0 * Pi)
      res3 = res1**2 + res2**2
      res4 = sqrt(res3)
      write(*,*) 'res1 = ', res1, ' res2 = ', res2
      write(*,*) 'res3 = ', res3, ' res4 = ', res4
      call prompt('Through built-in fcn 1')

c-----------------------------------------------------------
c     atan, acos, asin, etc. return arctangent, arcosine,
c     arcsine etc. in RADIANS
c-----------------------------------------------------------
      res1 = atan(1.0d0)
      write(*,*) 'res1 = ', res1
      call prompt('Through built-in fcn 2')

c-----------------------------------------------------------
c     min and max will return the minimum and maximum
c     respectively of an arbitrary number of arguments
c     of any UNIQUE data type.  Do NOT mix types in
c     a single statement as in
c
c     write(*,*) min(1,2.0d0)
c-----------------------------------------------------------
      write(*,*) 'min(3.0d0,2.0d0) = ', min(3.0d0,2.0d0)
      write(*,*) 'min(1,-3,5,0) = ', min(1,-3,5,0)
      call prompt('Through built-in fcn 3')

c-----------------------------------------------------------
c     mod is particularly useful for calculating when one
c     integer divides another evenly
c-----------------------------------------------------------
      do i = 0 , 1000
         if( mod(i,100) .eq. 0 ) then
            write(*,*) 'i = ', i
         end if
      end do
      call prompt('Through built-in fcn 4')

c-----------------------------------------------------------
c     Stop program execution
c-----------------------------------------------------------
      call prompt('Through fdemo1')
      stop

c===========================================================
c     END OF EXECUTABLE STATEMENTS
c===========================================================

c-----------------------------------------------------------
c     End of program unit (fdemo1)
c-----------------------------------------------------------
      end

c===========================================================
c     Prints a message on stdout and then waits for input
c     from stdin.
c===========================================================
c-----------------------------------------------------------
c     A new program unit (prompt).
c-----------------------------------------------------------
      subroutine prompt(pstring)

         implicit      none

         character*(*) pstring
         integer       rc
         character*1   resp

         write(*,*) pstring
         write(*,*) 'Enter any non-blank character & '//
     &              'enter to continue'

         read(*,*,iostat=rc,end=900)  resp
c-----------------------------------------------------------
c        Return to calling program.
c-----------------------------------------------------------
         return

 900     continue
c-----------------------------------------------------------
c        Stop program execution.  This section of code is
c        the "end-of-file" handler for standard input
c        (via the end=900 clause of the read statement).
c        In this case, it is perfectly good style simply
c        to quit.
c-----------------------------------------------------------
         stop
c-----------------------------------------------------------
c     End of program unit (prompt).
c-----------------------------------------------------------
      end
```

Source file: `Makefile`

```
############################################################
#  Note that this 'Makefile' assumes that the following
#  environment variables are set:
#
#     F77         ->  name of f77 compiler
#     F77FLAGS    ->  generic f77 flags
#     F77CFLAGS   ->  f77 flags for compilation phase
#     F77LFLAGS   ->  f77 flags for load phase
############################################################
.IGNORE:

F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS)
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)

.f.o:
   $(F77_COMPILE) $*.f

EXECUTABLES = fdemo1

all: $(EXECUTABLES)

fdemo1: fdemo1.o
   $(F77_LOAD) fdemo1.o -o fdemo1

clean:
   rm *.o
   rm $(EXECUTABLES)
```

Source file: `fdemo1-output`

```
############################################################^
Wed Oct  6 15:17:09 PDT 2004
############################################################

lnx1 1> cat Makefile
############################################################
#  Note that this 'Makefile' assumes that the following
#  environment variables are set:
#
#     F77         ->  name of f77 compiler
#     F77FLAGS    ->  generic f77 flags
#     F77CFLAGS   ->  f77 flags for compilation phase
#     F77LFLAGS   ->  f77 flags for load phase
############################################################
.IGNORE:

F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS)
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)

.f.o:
   $(F77_COMPILE) $*.f

EXECUTABLES = fdemo1

all: $(EXECUTABLES)

fdemo1: fdemo1.o
   $(F77_LOAD) fdemo1.o -o fdemo1

clean:
   rm *.o
   rm $(EXECUTABLES)

############################################################
lnx1 3> env | grep '^F77'
F77=pgf77
F77FLAGS=-g
F77CFLAGS=-c
F77LFLAGS=-L/usr/local/PGI/lib

############################################################
lnx1 4> make

pgf77 -g -c fdemo1.f
pgf77 -g -L/usr/local/PGI/lib fdemo1.o -o fdemo1

############################################################
# I encourage you to download 'fdemo1.f', compile it,
# and run it INTERACTIVELY yourself.  You should see
# output essentially identical to that shown below.
# Note, however, that both because I'm lazy, as well
# as to illustrate the use of I/O re-direction, I have
# previously prepared a file called 'INPUT', which
# contains many lines consisting of a single character
# These lines will be read by the 'prompt' subroutine
# which, when run interactively, writes a prompt to
# stdout and then waits for input from stdin.
############################################################

lnx1 5> head -10 < INPUT
q
q
q
q
q
q
q
q
q
q

############################################################
lnx1 6> fdemo1 < INPUT
 a =   2.5000000000000000E-002 b =  -1.2339999999999998E-016
  c =    1.000000000000000      i =         3000 switch =  T
 Through scalar assignment
 Enter any non-blank character & enter to continue

############################################################
```

```
# Note: For readability, all other instances of the
# following output from the 'prompting' routine have been
# converted to blank lines with a text editor command.
############################################################


 res1 =    5.000000000000000      res2 =    13.00000000000000
  res3 =    3.605551275463989
 Through real*8 arithmetic expressions

 ires1 =           5 ires2 =           0
 ires3 =         512 ires4 =          64
 Through integer arithmetic expressions

 res1 =    5.000000000000000      res2 =   0.0000000000000000E+000
  res3 =   0.7500000000000000
 Through mixed-mode arithmetic

 Loop 1: i =           1
 Loop 1: i =           2
 Loop 1: i =           3
 Through loop 1

 Loop 2: i =           1
 Loop 2: i =           2
 Loop 2: i =           3
 Through loop 2

 Loop 3: i =           1
 Loop 3: i =           3
 Loop 3: i =           5
 Loop 3: i =           7
 Through loop 3

 Loop 4: i =           3
 Loop 4: i =           2
 Loop 4: i =           1
 Through loop 4

 Loop 5: i, j =           1           1
 Loop 5: i, j =           1           2
 Loop 5: i, j =           2           1
 Loop 5: i, j =           2           2
 Loop 5: i, j =           3           1
 Loop 5: i, j =           3           2
 Through loop 5

 Loop 6: i =           2
 Loop 6: i =           4
 Loop 6: i =           6
 Through loop 6

 lres1 =  T lres2 =  T lres3 =  F
 Through basic conditionals

    25.00000000000000      >    12.00000000000000
 Through if 1

    25.00000000000000      >    12.00000000000000
 Through if 2

    25.00000000000000      >    24.00000000000000
 Through nested if

 Case 1
 Case 2
 Case 3
 Default case
 Through case via if

 Do while loop: b =   0.0000000000000000E+000
 Do while loop: b =   0.1000000000000000
 Do while loop: b =   0.2000000000000000
 Do while loop: b =   0.3000000000000000
 Do while loop: b =   0.4000000000000000
 Do while loop: b =   0.5000000000000000
 Do while loop: b =   0.6000000000000000
 Do while loop: b =   0.7000000000000000
 Do while loop: b =   0.7999999999999999
 Do while loop: b =   0.9000000000000000
 Do while loop: b =   0.9999999999999998
 Through while loop

 res1 =   0.8090169943749475      res2 =   0.5877852522924732
 res3 =    1.000000000000000      res4 =    1.000000000000000
 Through built-in fcn 1

 res1 =   0.7853981633974483
 Through built-in fcn 2

 min(3.0d0,2.0d0) =    2.000000000000000
 min(1,-3,5,0) =          -3
 Through built-in fcn 3

 i =           0
 i =         100
 i =         200
 i =         300
 i =         400
 i =         500
 i =         600
 i =         700
 i =         800
 i =         900
 i =        1000
 Through built-in fcn 4

 Through fdemo1

lnx1 7> exit
exit
```

Source file: `fdemo2.f`

```
c===========================================================
c     fdemo2:  Program which demonstrates basic usage
c     of character variables in Fortran 77.
c===========================================================
      program        fdemo2

      implicit       none

c-----------------------------------------------------------
c     See below for definition of integer function
c     'indlnb'.   Note that this and other useful routines
c     are available in the 'p410f' library.
c-----------------------------------------------------------
      integer        indlnb

c-----------------------------------------------------------
c     Define some character variables of various lengths
c
c     Note that
c
c        character*1     foo
c
c     and
c
c        character       foo
c
c     are synonymous, i.e. if an explicit length
c     specification is not given, the variable will
c     be a single character long.
c-----------------------------------------------------------
      character*1    c1
      character*2    c2
      character*4    c4
      character*26   lcalph
      character      cc1*1,    cc2*2,   cc4*4
      character*60   buffer

c-----------------------------------------------------------
c     Assignment of constant strings to char. variables.
c     If length of character expression being assigned
c     is less than length of character variable, variable
c     is 'right-padded' with blanks.
c-----------------------------------------------------------
      c1     = 'a'
      c2     = 'bc'
      c4     = 'defg'
      lcalph = 'abcdefghijklmnopqrstuvwxyz'

      write(*,*) 'c1 = ', c1
      write(*,*) 'c2 = ', c2
      write(*,*) 'c4 = ', c4
      write(*,*) 'lcalph = ', lcalph
      call prompt('Through constant assignment')

c-----------------------------------------------------------
c     // is the string concatentation operator
c-----------------------------------------------------------
      write(*,*) 'c1 // c2 // c4 = ', c1 // c2 // c4
      call prompt('Through concatenation')

c-----------------------------------------------------------
c     The integer intrinsic (built-in) function 'len'
c     returns the length of its string argument
c-----------------------------------------------------------
      write(*,*) 'len(c1) = ', len(c1)
      write(*,*) 'len(buffer) = ', len(buffer)
      call prompt('Through string length')

c-----------------------------------------------------------
c     Substring extraction
c-----------------------------------------------------------
      write(*,*) 'lcalph(1:13) = ',  lcalph(1:13)
      write(*,*) 'lcalph(18:18) = ', lcalph(18:18)
      call prompt('Through substring extraction')

c-----------------------------------------------------------
c     Substring assignment
c-----------------------------------------------------------
      c4(4:4) = 'Z'
      write(*,*) 'c4 = ', c4
      call prompt('Through substring assignment')

c-----------------------------------------------------------
c     Use of 'indlnb'
c-----------------------------------------------------------
      buffer = 'somefilename'
      write(*,*) '<' // buffer // '>'
      write(*,*) '<' // buffer(1:indlnb(buffer)) // '>'
      buffer = 'Some multi-word message'
      write(*,*) '<' // buffer // '>'
      write(*,*) '<' // buffer(1:indlnb(buffer)) // '>'
      buffer = ' '
      write(*,*) 'indlnb(buffer) = ', indlnb(buffer)
      call prompt('Through indlnb usage')

      call prompt('Through fdemo2')

      stop
      end

c-----------------------------------------------------------
c     Prints a message on stdout and then waits for input
c     from stdin.
c-----------------------------------------------------------
      subroutine prompt(pstring)

         implicit      none

         character*(*) pstring
         integer       rc
         character*1   resp

         write(*,*) pstring
         write(*,*) 'Enter any non-blank character & '//
     &              'enter to continue'
         read(*,*,iostat=rc,end=900)  resp
         return

 900     continue
         stop
      end

c-----------------------------------------------------------
c     Returns index of last non-blank character in 's',
c     or 0 if the string is completely blank.
c-----------------------------------------------------------
      integer function indlnb(s)

         character*(*)    s

         do indlnb = len(s) , 1 , -1
            if( s(indlnb:indlnb) .ne. ' ' ) return
         end do
         indlnb = 0

         return

      end
```

Source file: `fdemo2-output`

```
############################################################
# Blank lines added for readability.
############################################################
lnx1 1> fdemo2
 c1 = a
 c2 = bc
 c4 = defg
 lcalph = abcdefghijklmnopqrstuvwxyz
 Through constant assignment
 Enter any non-blank character & enter to continue
a

 c1 // c2 // c4 = abcdefg
 Through concatenation
 Enter any non-blank character & enter to continue
a

 len(c1) =            1
```

```
 len(buffer) =           60
 Through string length
 Enter any non-blank character & enter to continue
a

 lcalph(1:13) = abcdefghijklm
 lcalph(18:18) = r
 Through substring extraction
 Enter any non-blank character & enter to continue
a

 c4 = defZ
 Through substring assignment
 Enter any non-blank character & enter to continue
a

 <somefilename
 <somefilename>
 <Some multi-word message
 <Some multi-word message>
 indlnb(buffer) =            0
 Through indlnb usage
 Enter any non-blank character & enter to continue
a

 Through fdemo2
 Enter any non-blank character & enter to continue
a
```

Source file: `first100-generate`

```
############################################################
# 'iota' is an APL-inspired script I wrote to generate
# the integers from 1 to n, one per line.  It comes in
# useful in many instances.  In Linux, there is also
# a command 'seq', which can so the same thing.
############################################################
lnx1 1> iota
usage: iota <n> [<origin|1>]

lnx1 2> which iota
/usr/local/bin/iota

############################################################
# 'mw' is another script which attempts to locate
# the source for a script or other executable, and if
# successful, displays the source.
############################################################
lnx1 3> mw iota
</usr/local/bin/iota>
#!/bin/sh

Usage="usage: iota <n> [<origin|1>]"

case $# in
1) n=$1; origin=1;;
2) n=$1; origin=$2;;
*) echo "$Usage"; exit 1;;
esac

if printf "%d" $n > /dev/null 2>&1 && \
   printf "%d" $n > /dev/null 2>&1 $origin; then
   awk 'BEGIN{for(i=0; i<'$n'; i++) printf "%d\n", i+'$origin'}' < /dev/null
else
   echo "$Usage"; exit 1;
fi

############################################################
# Sample 'iota' invocation.
############################################################
lnx1 4> iota 10
1
2
3
4
5
6
7
8
9
10

############################################################
# Create 'first100' file.
############################################################
lnx1 5> iota 100 > first100

############################################################
# Display first 10 lines of 'first100' using Unix 'head'
# command.  Note use of '!$' (last argument to previous
# command).
###########################################################
lnx1 6> head -10 !$
head -10 first100
1
> 2
3
> 4
5
6
7
8
9
10

############################################################
# Display last 10 lines of 'first100' using Unix 'tail'
# command.
############################################################
lnx1 7> tail -10 !$
tail -10 first100
91
92
93
94
95
96
97
98
99
100
```

Source file: `mysum.f`

```
c===========================================================
c     mysum:  reads numbers one per line from stdin
c     and writes sum on stdout.  Ignores invalid inputs
c     but counts number encountered and reports on stderr.
c===========================================================
      program      mysum

      implicit     none

c-----------------------------------------------------------
c     vi:    Current number read from stdin
c     sum:   Current sum of numbers read
c     rc:    For storing return status from READ
c     nbad:  Count of number of bad inputs
c-----------------------------------------------------------
      real*8       vi,         sum
      integer      rc,         nbad

c-----------------------------------------------------------
c     Initialize ...
c-----------------------------------------------------------
      nbad = 0
      sum  = 0.0d0

c-----------------------------------------------------------
c     The following construct is roughly equivalent to
c     a while loop, execution keeps returning to the
c     top of the loop until end of file is detected on
c     stdin.
c-----------------------------------------------------------
 100  continue
         read(*,*,iostat=rc,end=200)  vi
         if( rc .eq. 0 ) then
c-----------------------------------------------------------
c           Read a bona fide real*8 value, update sum.
c-----------------------------------------------------------
            sum = sum + vi
         else
c-----------------------------------------------------------
c           Input was invalid.
c-----------------------------------------------------------
            nbad = nbad + 1
         end if
      go to 100
 200  continue

c-----------------------------------------------------------
c     Write sum on standard output.
c-----------------------------------------------------------
      write(*,*) sum

c-----------------------------------------------------------
c     Report # of invalid inputs only if there were some.
c-----------------------------------------------------------
      if( nbad .gt. 0 ) then
c-----------------------------------------------------------
c        Unit 0 is stderr (standard error) on most Unix
c        systems: if you redirect stdin using '>' and this
c        message is tripped, it will still appear on the
c        terminal.
c-----------------------------------------------------------
         write(0,*) nbad, ' invalid inputs'
      end if

      stop

      end
```

Source file: `mysum-s.f`

```
c===========================================================
c     Less-commented (i.e. more reasonable level of
c     comments) version of mysum.
c===========================================================
c     mysum_s:  reads numbers one per line from stdin
c     and writes sum on stdout.  Ignores invalid inputs
c     but counts number encountered and reports on stderr.
c===========================================================
      program      mysum

      implicit     none

      real*8       vi,         sum
      integer      rc,         nbad

      nbad = 0
      sum  = 0.0d0

 100  continue
         read(*,*,iostat=rc,end=200)  vi
         if( rc .eq. 0 ) then
            sum = sum + vi
         else
            nbad = nbad + 1
         end if
      go to 100
 200  continue

      write(*,*) sum

      if( nbad .gt. 0 ) then
         write(0,*) nbad, ' invalid inputs'
      end if

      stop

      end
```

Source file: `mysum-ng.f`

```
c===========================================================
c     go-to-less version of mysum.  No more or less correct
c     than go-to-full version, of course, but some would assert
c     that this version is "cleaner"/"safer" in some sense,
c     or at the very least, better stylistically.
c===========================================================
      program      mysum

      implicit     none

      real*8       vi,         sum
      integer      rc,         rceof,     nbad

      nbad = 0
      sum  = 0.0d0

c-----------------------------------------------------------
c     'rc' will be set to a NEGATIVE value when end of file
c     is encountered.  Initialize 'rc' to 0 to ensure that
c     we get into the 'do while' loop.
c-----------------------------------------------------------
      rc = 0
      do while ( rc .ge. 0 )
         read(*,*,iostat=rc)  vi
         if( rc .eq. 0 ) then
            sum = sum + vi
c-----------------------------------------------------------
c           New logic to ensure that EOF doesn't get counted
c           as a bad input.
c-----------------------------------------------------------
         else if ( rc .gt. 0 ) then
            nbad = nbad + 1
         end if
      end do

      write(*,*) sum

      if( nbad .gt. 0 ) then
         write(0,*) nbad, ' invalid inputs'
      end if

      stop

      end
```

Source file: `mysum-output`

```
lnx1 1> mysum
1
2
8
10
^D
    21.00000000000000
lnx1 2> mysum < first100
    5050.000000000000

lnx1 3> mysum
12
2
8
a
10
b
^D
    32.00000000000000
            2 invalid inputs

lnx1 4> mysum < first100 > mysum_result

lnx1 5> more !$
more mysum_result
    5050.000000000000
```

Source file: `dvfrom.f`

```
c===========================================================
c     Returns a double precision vector (one-dimensional
c     array) read from file 'fname'.  If 'fname' is the
c     string '-', the vector is read from standard input.
c
c     The file should contain one number per line; invalid
c     input is ignored.
c
c     This routine illustrates  a general technique for
c     reading data from a FORMATTED (ASCII) file.  In
c     Fortran, one associates a "logical unit number"
c     (an integer) with a file via the OPEN statement.
c     The unit number can then be used as the first
c     "argument" of the READ and WRITE statements to
c     perform input and output on the file.
c
c     Fortran reserves the following unit numbers:
c
c     5      terminal input (stdin)
c     6      terminal output (stdout)
c     0      error output on Unix systems (stderr)
c===========================================================

      subroutine dvfrom(fname,v,n,maxn)
c-----------------------------------------------------------
c     Arguments:
c
c        fname:  (I)    File name
c        v:      (O)    Return vector
c        n:      (O)    Length of v (# read)
c        maxn:   (I)    Maximum number to read
c-----------------------------------------------------------
         implicit         none
c-----------------------------------------------------------
c        The integer functions 'indlnb' and 'getu' are
c        defined in the 'p410f' library.
c-----------------------------------------------------------
         integer          indlnb,      getu
c-----------------------------------------------------------
c        Declaration of routine arguments: note
c        "adjustable dimensioning" of v; any array which
c        is declared with adjustable dimensions must be
c        a subroutine argument; any adjustable dimensions
c        must also be subroutine arguments.
c-----------------------------------------------------------
         character*(*)    fname
         integer          n,           maxn
         real*8           v(maxn)

c-----------------------------------------------------------
c        Programming style: Use parameter (ustdin) rather
c        than constant value (5) for stdin logical unit #
c-----------------------------------------------------------
         integer          ustdin
         parameter      ( ustdin = 5 )

c-----------------------------------------------------------
c        Local variables:
c
c        vn:     Current number read from input
c        ufrom:  Logical unit number for READ
c        rc:     For storing return status from READ
c-----------------------------------------------------------
         real*8           vn
         integer          ufrom,       rc

c-----------------------------------------------------------
c        Intialize
c-----------------------------------------------------------
         n = 0

c-----------------------------------------------------------
c        Read from stdin?
c-----------------------------------------------------------
         if( fname .eq. '-' ) then
c-----------------------------------------------------------
c           Set unit number to stdin default
c-----------------------------------------------------------
            ufrom = ustdin
         else
c-----------------------------------------------------------
c           Get an available unit number
c-----------------------------------------------------------
            ufrom = getu()
c-----------------------------------------------------------
c           Open the file for formatted I/O
c-----------------------------------------------------------
            open(ufrom,file=fname(1:indlnb(fname)),
     &           form='formatted',status='old',iostat=rc)
            if( rc .ne. 0 ) then
c-----------------------------------------------------------
c              Couldn't open the file, print error message
c              and return.
c-----------------------------------------------------------
               write(0,*) 'dvfrom: Error opening ',
     &                    fname(1:indlnb(fname))
               return
            end if
         end if

c-----------------------------------------------------------
c        Input numbers into vector (one per line) until
c        EOF or maximum allowable number read
c-----------------------------------------------------------
 100     continue
            read(ufrom,*,iostat=rc,end=200)  vn
            if( rc .eq. 0 ) then
               n = n + 1
               if( n .gt. maxn ) then
                  write(0,*)  'dvfrom: Read maximum of ',
     &                         maxn, ' from ',
     &                         fname(1:indlnb(fname))
                     n = maxn
                     go to 200
               end if
               v(n) = vn
            end if
         go to 100
 200     continue

c-----------------------------------------------------------
c        If we are reading from a file, close the file.
c        This releases the unit number for subsequent use.
c-----------------------------------------------------------
         if( ufrom .ne. ustdin ) then
            close(ufrom)
         end if

         return

      end
```

Source file: `tdvfrom.f`

```
c===========================================================
c     Test program for subroutine 'dvfrom'.
c
c     Program expects one argument which is the filename
c     to be passed to 'dvfrom'
c===========================================================
      program        tdvfrom

      implicit       none

c-----------------------------------------------------------
c     The integer function 'iargc' returns the number of
c     arguments supplied to the program.  It is
c     automatically available to all Fortran programs on
c     most Unix systems, as is 'getarg' (see below).
c-----------------------------------------------------------
      integer        iargc,      indlnb

      integer        maxn
      parameter    ( maxn = 100 000 )
      real*8         v(maxn)
      integer        n

      character*256  fname
```

```
c-----------------------------------------------------------
c     Unless exactly one argument is supplied, print usage
c     message and exit.
c-----------------------------------------------------------
      if( iargc() .ne. 1 ) then
         write(0,*) 'usage: tdvfrom <file name>'
         write(0,*)
         write(0,*) '      Use ''tdvfrom -'' to read ',
     &              'from standard input'
         stop
      end if

c-----------------------------------------------------------
c     The subroutine 'getarg' (Unix) takes 2 arguments.
c     The first is an integer input argument specifying
c     which argument is to be fetched, the second is
c     a character output argument which, on return,
c     contains the fetched argument.
c
c     Get the filename.
c-----------------------------------------------------------
      call getarg(1,fname)
c-----------------------------------------------------------
c     Call the routine ...
c-----------------------------------------------------------
      call dvfrom(fname,v,n,maxn)
c-----------------------------------------------------------
c     ... and report how many numbers were read.
c-----------------------------------------------------------
      write(0,*) 'tdvfrom: ', n, ' read from '//
     &           fname(1:indlnb(fname))

      stop
      end
```

Source file: `tdvfrom-output`

```
lnx1 1> tdvfrom
 usage: tdvfrom <file name>

       Use 'tdvfrom -' to read from standard input

lnx1 2> tdvfrom -
1
2
3
4
5
^D
 tdvfrom:            5 read from -

lnx1 3> tdvfrom first100
 tdvfrom:          100 read from first100
```

Source file: `dvto.f`

```
c===========================================================
c     Writes a double precision vector to file 'fname'.
c     If fname is the string '-' then the vector is written
c     to standard output.
c===========================================================

      subroutine dvto(fname,v,n)
c-----------------------------------------------------------
c     Arguments:
c
c        fname:  (I)    File name
c        v:      (I)    Vector to be written
c        n:      (I)    Length of vector
c-----------------------------------------------------------
         implicit         none

         integer          getu,       indlnb

         character*(*)    fname
         integer          n
         real*8           v(n)

         integer          ustdout
         parameter      ( ustdout = 6 )

         integer          i,      uto,       rc

         if( fname .eq. '-' ) then
            uto = ustdout
         else
            uto = getu()
            open(uto,file=fname(1:indlnb(fname)),
     &           form='formatted',iostat=rc)
            if( rc .ne. 0 ) then
               write(0,*) 'dvto: Error opening ',
     &                    fname(1:indlnb(fname))
               return
            end if
         end if

         do i = 1 , n
            write(uto,*) v(i)
         end do

         if( uto .ne. ustdout ) then
            close(uto)
         end if

         return

      end
```

Source file: `tdvto.f`

```
c===========================================================
c     Test program for subroutine 'dvto'.
c
c     Program expects two arguments, the name of a file
c     for output ('-' for stdout) and the length of the
c     test vector to be written.
c===========================================================
      program        tdvto

      implicit       none

c-----------------------------------------------------------
c     The integer function 'i4arg' is defined in the
c     'p410f' library.  It takes two arguments, the first
c     is an integer specifying which program argument is
c     to be parsed as an integer, and the second is a
c     default value which will be returned if the argument
c     was not supplied or could not be converted to an
c     integer.
c-----------------------------------------------------------
      integer        iargc,      i4arg

      integer        maxn
      parameter    ( maxn = 100 000 )
      real*8         v(maxn)
      integer        n

      integer        i
      character*256  fname

c-----------------------------------------------------------
c     Unless exactly two arguments are supplied, print usage
c     message and exit.
c
c     Note the use of the "logical-if" statement (no then)
c-----------------------------------------------------------
      if( iargc() .ne. 2 ) go to 900

      call getarg(1,fname)
      n = i4arg(2,-1)
      if( n .eq. -1 ) go to 900
c-----------------------------------------------------------
c     Limit the value of n
c-----------------------------------------------------------
      n = min(n,maxn)
c-----------------------------------------------------------
c     Define test vector
c-----------------------------------------------------------
      do i = 1 , n
         v(i) = i
```

```
      end do

c-----------------------------------------------------------
c     Call the routine ..
c-----------------------------------------------------------
      call dvto(fname,v,n)

c-----------------------------------------------------------
c     Normal exit
c-----------------------------------------------------------
      stop

c-----------------------------------------------------------
c     Usage exit
c-----------------------------------------------------------
 900  continue
         write(0,*) 'usage: tdvto <file name> <n>'
         write(0,*)
         write(0,*) '       Use ''tdvto -'' to write ',
     &              'to standard output'

      stop

      end
```

Source file: `tdvto-output`

```
lnx1 1> tdvto
 usage: tdvto <file name> <n>

        Use 'tdvto -' to write to standard output

lnx1 2> tdvto -
 usage: tdvto <file name> <n>

        Use 'tdvto -' to write to standard output

lnx1 3> tdvto - 10
    1.000000000000000
    2.000000000000000
    3.000000000000000
    4.000000000000000
    5.000000000000000
    6.000000000000000
    7.000000000000000
    8.000000000000000
    9.000000000000000
    10.00000000000000

lnx1 4> tdvto foo 5

lnx1 5> cat foo
    1.000000000000000
    2.000000000000000
    3.000000000000000
    4.000000000000000
    5.000000000000000
lnx1 6> tdvfrom foo
 tdvfrom:            5 read from foo

lnx1 7> tdvto - 100 | tdvfrom -
 tdvfrom:          100 read from -
```

Source file: `Makefile`

```
.IGNORE:

F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS)
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)

.f.o:
   $(F77_COMPILE) $*.f

EXECUTABLES = fdemo2 mysum tdvfrom tdvto

all: $(EXECUTABLES)

fdemo2: fdemo2.o
   $(F77_LOAD) fdemo2.o -o fdemo2

mysum: mysum.o
   $(F77_LOAD) mysum.o -o mysum

tdvfrom: tdvfrom.o dvfrom.o
   $(F77_LOAD) tdvfrom.o dvfrom.o -lp410f -o tdvfrom

tdvto: tdvto.o dvto.o
   $(F77_LOAD) tdvto.o dvto.o -lp410f -o tdvto

clean:
   rm *.o
   rm $(EXECUTABLES)
```

Source file: `make-output`

```
############################################################
# Do the default make (all: $(EXECUTABLES))
############################################################
lnx1 1> make
pgf77 -g -c fdemo2.f
pgf77 -g -L/usr/local/PGI/lib fdemo2.o -o fdemo2
pgf77 -g -c mysum.f
pgf77 -g -L/usr/local/PGI/lib mysum.o -o mysum
pgf77 -g -c tdvfrom.f
pgf77 -g -c dvfrom.f
pgf77 -g -L/usr/local/PGI/lib tdvfrom.o dvfrom.o -lp410f -o tdvfrom
pgf77 -g -c tdvto.f
pgf77 -g -c dvto.f
pgf77 -g -L/usr/local/PGI/lib tdvto.o dvto.o -lp410f -o tdvto

############################################################
# Here's an alias which lists all the executables in a
# directory using the fact that the -F flag to ls appends
# a '*' to the name of such files.  I've included it here
# just to keep you thinking about tailoring your Unix
# environment to suit your own needs. 'sed' is the stream-
# editor, which, like 'awk' and 'perl' can be used to
# manipulate and modify text.
############################################################
lnx1 2> alias lsx  '/bin/ls -F | fgrep \* | sed s/\*//g'

lnx1 3> lsx
fdemo2
mysum
tdvfrom
tdvto

############################################################
# For those of you who think that there must be a find
# command that does about the same thing, you're right ...
############################################################
lnx1 4> find . -perm +111
.
./fdemo2
./mysum
./tdvfrom
./tdvto
############################################################
# ... and I'd *still* alias it 'lsx'!
############################################################

############################################################
# Clean up ...
```

```
############################################################
lnx1 5> make clean
rm *.o
rm fdemo2 mysum tdvfrom tdvto

lnx1 6> ls
Makefile  dvto.f    first100   mysum.f    tdvto.f
dvfrom.f  fdemo2.f  mysum-s.f  tdvfrom.f
```

Source file: `tdrand48.f`

```
c===========================================================
c     Demonstrates use of the real*8 (pseudo-)random number
c     generator, 'drand48' available on the lnx machines via
c     the 'p410f' utility library.
c
c     usage: tdrand48 <nrand> [<seed>]
c
c     where <nrand> is the number of random numbers on the
c     interval (0..1) to be generated, and <seed> is the
c     optional, integer-valued "seed" for the random number
c     generator.
c
c     The program outputs the random numbers generated to
c     standard output (one per line), and their average to
c     standard error.  In the limit of very large <nrand>,
c     this average should approach 0.5 since 'drand48'
c     generates numbers uniformly distributed on the unit
c     interval.
c
c     Note carefully that for fixed seed, 'drand48' (and
c     most other pseudo-random number generators) will
c     ALWAYS RETURN THE SAME SEQUENCE OF NUMBERS!  In
c     fact, the purpose of seeding the generator is precisely
c     to produce variation ("randomness") in the sequence
c     of iterates produced.  If 'drand48' is not explicitly
c     seeded via 'srand48' a specific default seed value is
c     used.
c===========================================================
      program        tdrand48

      implicit       none

      integer        iargc,      i4arg
      real*8         drand48

      real*8         ranval,     sum
      integer        i,          nrand

      if( iargc() .lt. 1 ) go to 900
      nrand = i4arg(1,-1)
      if( nrand .le. 0 ) go to 900
      if( iargc() .gt. 1 ) then
         call srand48(i4arg(2,0))
      end if

      sum = 0.0d0
      do i = 1 , nrand
c-----------------------------------------------------------
c        Generate a random number
c-----------------------------------------------------------
         ranval = drand48()
         sum = sum + ranval
         write(*,*) ranval
      end do

      write(0,*)
      write(0,*) 'Average: ', sum / nrand

      stop

 900  continue
         write(0,*) 'usage: tdrand48 <nrand> [<seed>]'
      stop
      end
```

Source file: `tdrand48-output`

```
lnx1 1> make tdrand48
pgf77 -g -c tdrand48.f
pgf77 -g -L/usr/local/PGI/lib tdrand48.o -lp410f -o tdrand48

lnx1 2> tdrand48
 usage: trand <n> [<seed>]

############################################################
# Generate 10 random numbers using the default seed.
############################################################
lnx1 3> tdrand48 10
   3.9079850466805510E-014
   9.8539467465030839E-004
   4.1631001594613082E-002
   0.1766426425429160
   0.3646022483906073
   9.1330612112294318E-002
   9.2297647698675434E-002
   0.4872172239468284
   0.5267502797621084
   0.4544334237382444

 Average:   0.2235890474460977

############################################################
# Use different seeds ...
############################################################
lnx1 4> tdrand48 10 0
   0.1708280361062897
   0.7499019804849638
   9.6371655623567420E-002
   0.8704652270270756
   0.5773035067951078
   0.7857992588396741
   0.6921941534586402
   0.3687662699204211
   0.8739040768618089
   0.7450950984500651

 Average:   0.5930629263567614
lnx1 5> tdrand48 10 1024
   0.8723921096689118
   0.2505373659982695
   0.2646277000113848
   0.5637676199866704
   0.4796618003284330
   0.9366764961596203
   0.9770464197702857
   0.9566037275772814
   0.5051937973970873
   0.8319197539829020

 Average:   0.6638426790880846

############################################################
# Use a csh foreach loop to investigate the large-n limit
# of the average of the generated numbers.  Also note
# the use of the Unix 'time' command to report the CPU time
# usage (and other statistics, see 'man time') of an
# arbitrary command.
############################################################
lnx1 7> foreach n (10 100 1000 10000 100000 1000000)
foreach? time tdrand48 $n > /dev/null
foreach? end

 Average:   0.2235890474460977
0.000u 0.000s 0:00.00 0.0%      0+0k 0+0io 119pf+0w

 Average:   0.4741044494625102
0.000u 0.010s 0:00.00 0.0%      0+0k 0+0io 119pf+0w

 Average:   0.4914224988159484
0.010u 0.000s 0:00.01 100.0%    0+0k 0+0io 119pf+0w

 Average:   0.4988303692558301
0.100u 0.010s 0:00.10 110.0%    0+0k 0+0io 119pf+0w

 Average:   0.4995105666942632
1.030u 0.000s 0:01.02 100.9%    0+0k 0+0io 119pf+0w

 Average:   0.4997852648772501
10.270u 0.070s 0:11.17 92.5%    0+0k 0+0io 119pf+0w

############################################################
# Note that in this case, particularly for large values
# of n, the execution time is completely dominated by the
# formatted output to standard output.  EXERCISE: Repeat
# the timing test using a modified version of 'tdrand48'
# that does NOT output the random numbers to standard
# output, but which still outputs the average value of the
# generated iterates to standard error.
############################################################
```

Source file: `tsavedata.f`

```
c===========================================================
c     Demonstration main program and subroutine to
c     illustrate use of SAVE and DATA statements.
c===========================================================
      program         tsavedata

      implicit        none

      integer         i

      do i = 1 , 10
         call sub1()
      end do

      stop

      end

c===========================================================
c     Subprogram 'sub1': writes a message to standard
c     error the FIRST time it is called, and writes
c     the number of times it has been called so far to
c     standard output EVERY time it is called.
c===========================================================
      subroutine sub1()

         implicit     none

         logical      first
         integer      ncall
c-----------------------------------------------------------
c        Strict f77 statement ordering demands that
c        ANY DATA statements appear after ALL variable
c        declarations.  Note the use of '/' to delimit the
c        initialization value.
c-----------------------------------------------------------
         data         first / .true. /
c-----------------------------------------------------------
c        This 'save' statement guarantees that ALL local
c        storage is preserved between calls.
c-----------------------------------------------------------
         save

         if( first ) then
            ncall = 1
            write(0,*) 'First call to sub1'
            first = .false.
         end if

         write(*,*) 'sub1: Call ', ncall
         ncall = ncall + 1

         return

      end
```

Source file: `tsavedata-output`

```
lnx1 1> make tsavedata
pgf77 -g -c tsavedata.f
pgf77 -g -L/usr/local/PGI/lib tsavedata.o -o tsavedata

lnx1 2> tsavedata
 First call to sub1
 sub1: Call            1
 sub1: Call            2
 sub1: Call            3
 sub1: Call            4
 sub1: Call            5
 sub1: Call            6
 sub1: Call            7
 sub1: Call            8
 sub1: Call            9
 sub1: Call           10
```

Source file: `tsub.f`

```
c===========================================================
c     Demonstration main program, subroutines and functions
c     to illustrate argument passing (call by address) in
c     Fortran.
c===========================================================
      program        tsub

      real*8         r8side

      integer        n
      parameter     ( n = 6 )
      real*8         v1(n)
      real*8         a,            b

      a = -1.0d0
      b =  1.0d0
      write(*,*) 'Pre r8swap: a = ', a, ' b = ', b
      call r8swap(a,b)
      write(*,*) 'Post r8swap: a = ', a, ' b = ', b
      call prompt('Through r8swap')

      a = 10.0d0
      b = r8side(a)
      write(*,*) 'Post r8side: a = ', a, ' b = ', b
      call prompt('Through r8side')

c-----------------------------------------------------------
c     Load 'v1' with 0.0d0
c-----------------------------------------------------------
      call dvloadsc(v1,n,0.0d0)
      call dvstderr('v1 loaded with 0.0',v1,n)
      call prompt('Through dvloadsc')

c-----------------------------------------------------------
c     'v1' and 'v1(1)' have the SAME ADDRESS and thus
c     this call to 'dvloadsc' has precisely the same effect
c     as the previous one.
c-----------------------------------------------------------
      call dvloadsc(v1(1),n,0.0d0)
      call dvstderr('v1 loaded with 0.0',v1,n)
      call prompt('Through dvloadsc (second time)')

c-----------------------------------------------------------
c     Load v(2:n-1) with 1.0d0, values 'v(1)' and 'v(n)'
c     are unchanged
c-----------------------------------------------------------
      call dvloadsc(v1(2),n-2,1.0d0)
      call dvstderr('v1 loaded with 0.0 and 1.0',v1,n)
      call prompt('Through dvloadsc (third time)')

c-----------------------------------------------------------
c     It is actually a violation of strict F77 to pass
c     the same address more than once to a subroutine
c     or argument, but in many cases, such as this one
c     it is perfectly safe.  This sequence uses the
c     routine 'dvaddsc' to increment each value of 'v1'
c     by 2.0d0.
c-----------------------------------------------------------
      call dvaddsc(v1,v1,n,2.0d0)
      call dvstderr('v1 incremented by 2.0',v1,n)
      call prompt('Through dvaddsc')

      call prompt('Through tsub')

      stop
      end

c===========================================================
c     This routine swaps its two real*8 arguments
c===========================================================
      subroutine  r8swap(val1,val2)

         implicit     none

         real*8       val1,      val2
         real*8       temp

         temp = val1
         val1 = val2
```

```
         val2 = temp

         return

      end

c===========================================================
c     Real*8 function 'r8side' which has the 'side effect'
c     of overwriting its argument with 0.0d0.  As a general
c     matter of style, Fortran FUNCTION subprograms should
c     act like real functions (i.e. NO side-effects) where
c     possible.
c
c     Also note that the name of a Fortran
c     function is treated as a local variable in the
c     subprogram source code and MUST be assigned a value
c     before any 'return' statements are encountered.
c===========================================================
      real*8 function r8side(x)

         implicit     none

         real*8       x

         r8side = x * x * x
         x = 0.0d0

         return

      end

c===========================================================
c     Loads output real*8 vector 'v' with input scalar
c     value 'sc'.
c===========================================================
      subroutine dvloadsc(v,n,sc)

         implicit      none

         integer       n
         real*8        v(n)
         real*8        sc

         integer       i

         do i = 1 , n
            v(i) = sc
         end do

         return

      end

c===========================================================
c     Adds real*8 scalar to input real*8 vector 'v1',
c     and returns results in output real*8 vector 'v2'
c===========================================================
      subroutine dvaddsc(v1,v2,n,sc)

         implicit      none

         integer       n
         real*8        v1(n),      v2(n)
         real*8        sc

         integer       i

         do i = 1 , n
            v2(i) = v1(i) + sc
         end do

         return

      end

c===========================================================
c     Dumps 'string' and the real*8 vector 'v' to stderr.
c===========================================================
      subroutine dvstderr(string,v,n)

         implicit       none

         character*(*)  string
         integer        n
         real*8         v(n)

         integer        i

         write(0,*) string
         do i = 1 , n
            write(0,*) v(i)
         end do

         return

      end

c===========================================================
c     Prints a message on stdout and then waits for input
c     from stdin.
c===========================================================
      subroutine prompt(pstring)

         implicit      none

         character*(*) pstring
         integer       rc
         character*1   resp

         write(*,*) pstring
         write(*,*) 'Enter anything & <CR> to continue'
         read(*,*,iostat=rc,end=900)  resp
         return

 900     continue
         stop
      end
```

Source file: `tsub-output`

```
##############################################################
# Blank lines added for readibility ...
##############################################################
lnx1 1> make tsub
pgf77 -g -c tsub.f
pgf77 -g -L/usr/local/PGI/lib tsub.o -o tsub

lnx1 2> tsub
 Pre r8swap: a =   -1.000000000000000      b =    1.000000000000000
 Post r8swap: a =    1.000000000000000      b =   -1.000000000000000
 Through r8swap
 Enter anything & <CR> to continue
a

 Post r8side: a =   0.0000000000000000E+000 b =    1000.000000000000
 Through r8side
 Enter anything & <CR> to continue
a

 v1 loaded with 0.0
   0.0000000000000000E+000
   0.0000000000000000E+000
   0.0000000000000000E+000
   0.0000000000000000E+000
   0.0000000000000000E+000
   0.0000000000000000E+000
 Through dvloadsc
 Enter anything & <CR> to continue
a

 v1 loaded with 0.0
   0.0000000000000000E+000
   0.0000000000000000E+000
   0.0000000000000000E+000
   0.0000000000000000E+000
   0.0000000000000000E+000
   0.0000000000000000E+000
  Through dvloadsc (second time)
 Enter anything & <CR> to continue
a

 v1 loaded with 0.0 and 1.0
   0.0000000000000000E+000
    1.000000000000000
    1.000000000000000
    1.000000000000000
    1.000000000000000
   0.0000000000000000E+000
 Through dvloadsc (third time)
 Enter anything & <CR> to continue
a

 v1 incremented by 2.0
    2.000000000000000
    3.000000000000000
    3.000000000000000
    3.000000000000000
    3.000000000000000
    2.000000000000000
 Through dvaddsc
 Enter anything & <CR> to continue
a

 Through tsub
 Enter anything & <CR> to continue
a
```

Source file: `texternal.f`

```
c===========================================================
c     Demonstration main program and subprograms
c     illustrating the 'EXTERNAL' statement and how
c     subprograms may be passed as ARGUMENTS to other
c     subprograms.  This technique is often used to
c     pass "user-defined" functions to routines which
c     can do generic things with such functions (such
c     as integrating or differentiating them, for example).
c===========================================================
      program        texternal

c-----------------------------------------------------------
c     The 'external' statement tells the compiler that the
c     specified names are names of externally-defined
c     subprograms (i.e. subroutines or functions)
c-----------------------------------------------------------
      real*8         r8fcn
      external       r8fcn,       r8sub2

c-----------------------------------------------------------
c     Call 'r8fcncaller' which then invokes 'r8fcn'
c-----------------------------------------------------------
      call r8fcncaller(r8fcn)
c-----------------------------------------------------------
c     Call 'r8subcaller' which then invokes 'r8sub2'
c-----------------------------------------------------------
      call subcaller(r8sub2)

      stop
      end

c===========================================================
c     Input 'fcn' is the name of an externally defined
c     real*8 function.  This routine invokes that function
c     with argument 10.0d0 and writes the result on
c     standard error
c===========================================================
      subroutine  r8fcncaller(fcn)

         implicit     none

         real*8       fcn
         external     fcn

         real*8       fcnval

         fcnval = fcn(10.0d0)

         write(0,*) 'r8caller: ', fcnval

         return

      end

c===========================================================
c     Input 'sub' is the name of an externally defined
c     subroutine.  This routine invokes that subroutine
c     with arguments 10.0d0 and 20.0d0.
c===========================================================
      subroutine  subcaller(sub)

         implicit     none

         external     sub

         call sub(10.0d0,20.0d0)

         return

      end

c-----------------------------------------------------------
c     Demonstration real*8 function
c-----------------------------------------------------------
      real*8 function r8fcn(x)

         implicit     none

         real*8       x
```

```
            r8fcn = x**2

            return

         end

c===========================================================
c     Demonstration subroutine
c===========================================================
         subroutine r8sub2(x,y)

            implicit     none

            real*8       x,       y

            write(0,*) 'r8sub: x = ', x, ' y = ', y

            return

         end
```

Source file: `texternal-output`

```
lnx1 1> make texternal
pgf77 -g -c texternal.f
pgf77 -g -L/usr/local/PGI/lib texternal.o -o texternal

lnx1 2> texternal
 r8caller:    100.0000000000000
 r8sub: x =    10.00000000000000      y =    20.00000000000000
```

Source file: `tcommon.f`

```
c===========================================================
c     Demonstration main program and subroutine
c     to illustrate use of COMMON blocks for creating
c     'global' storage.  Common blocks should always
c     be labelled (named) and should be used sparingly.
c===========================================================
      program        tcommon

      implicit       none

c-----------------------------------------------------------
c     Declare variables to be placed in common block
c-----------------------------------------------------------
      character*16   string
      real*8         v(3),
     &               x,             y,            z
      integer        i
c-----------------------------------------------------------
c     Variables are stored in a common block in the
c     order in which they are specified in the 'common'
c     statement. ALWAYS order variables from longest to
c     shortest to avoid "alignment problems".  Don't
c     try to put a variable in more than one common block
c     and note that entire arrays (such as 'v') are placed
c     in the common block by simply specifying the name
c     of the array.  Finally, note that variables in a
c     common block CAN NOT be initialized with a 'data'
c     statement.
c-----------------------------------------------------------
      common   / coma /
     &               string,
     &               v,
     &               x,             y,            z,
     &               i

      string = 'foo'
      v(1) = 1.0d0
      v(2) = 2.0d0
      v(3) = 3.0d0
      x = 10.0d0
      y = 20.0d0
      z = 30.0d0
      i = 314

      call subcom()

      stop
      end

c===========================================================
c     This subroutine dumps information passed to it in
c     a common block.
c===========================================================
      subroutine  subcom()

c-----------------------------------------------------------
c     Overall layout of common block should be identical
c     in all program units which use the common block.
c-----------------------------------------------------------
         character*16   string
         real*8         v(3),
     &                  x,             y,            z
         integer        i

         common   / coma /
     &                  string,
     &                  v,
     &                  x,             y,            z,
     &                  i

         write(0,*) 'In subcom:'
         write(0,*) 'string = ', string
         write(0,*) 'v = ', v
         write(0,*) 'x = ', x, ' y = ', y, ' z = ', z
         write(0,*) 'i = ', i

         return

      end
```

Source file: `coma.inc`

```
c-----------------------------------------------------------
c     Defining the variables stored in a common block
c     (along with the common block itself) in a separate
c     'include file' minimizes the potential for the many
c     obscure and difficult to debug problems which can
c     arise from the use of common blocks.
c-----------------------------------------------------------
      character*16    string
      real*8          v(3),
     &                x,             y,             z
      integer         i

      common   / coma /
     &                string,
     &                v,
     &                x,             y,             z,
     &                i
```

Source file: `tcommon1.f`

```
c===========================================================
c     Demonstration main program, subroutines and functions
c     to illustrate RECOMMENDED use of common blocks
c     using 'include' statement.  Safe Fortran 77
c     extension.
c===========================================================
      program         tcommon1

      implicit        none

c-----------------------------------------------------------
c     By convention, I use the extension '.inc' for
c     Fortran source files which are to be included.
c-----------------------------------------------------------
      include        'coma.inc'

      string = 'foo'
      v(1) = 1.0d0
      v(2) = 2.0d0
      v(3) = 3.0d0
      x = 10.0d0
      y = 20.0d0
      z = 30.0d0
      i = 314

      call subcom()

      stop
      end

c===========================================================
c     This subroutine dumps information passed to it in
c     a common block.
c===========================================================
      subroutine  subcom()

         include       'coma.inc'

         write(0,*) 'In subcom:'
         write(0,*) 'string = ', string
         write(0,*) 'v = ', v
         write(0,*) 'x = ', x, ' y = ', y, ' z = ', z
         write(0,*) 'i = ', i

         return

      end
```

Source file: `tcommon-output`

```
lnx1 1> make tcommon
pgf77 -g -c tcommon.f
pgf77 -g -L/usr/local/PGI/lib tcommon.o -o tcommon

lnx1 2> tcommon
 In subcom:
 string = foo
 v =    1.000000000000000        2.000000000000000
    3.000000000000000
 x =    10.00000000000000      y =    20.00000000000000      z =
    30.00000000000000
 i =           314
```

Source file: `Makefile`

```
.IGNORE:

F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS)
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)

.f.o:
   $(F77_COMPILE) $*.f

EXECUTABLES = tdrand48 tsavedata tsub texternal tcommon tcommon1

all: $(EXECUTABLES)

tdrand48: tdrand48.o
   $(F77_LOAD) tdrand48.o -lp410f -o tdrand48

tsavedata: tsavedata.o
   $(F77_LOAD) tsavedata.o -o tsavedata

tsub: tsub.o
   $(F77_LOAD) tsub.o -o tsub

texternal: texternal.o
   $(F77_LOAD) texternal.o -o texternal

tcommon: tcommon.o
   $(F77_LOAD) tcommon.o -o tcommon

tcommon1.o: tcommon1.f coma.inc

tcommon1: tcommon1.o
   $(F77_LOAD) tcommon1.o -o tcommon1

clean:
   rm *.o
   rm $(EXECUTABLES)
```

Source file: `make-output`

```
lnx1 1> make
pgf77 -g -c tdrand48.f
pgf77 -g -L/usr/local/PGI/lib tdrand48.o -lp410f -o tdrand48
pgf77 -g -c tsavedata.f
pgf77 -g -L/usr/local/PGI/lib tsavedata.o -o tsavedata
pgf77 -g -c tsub.f
pgf77 -g -L/usr/local/PGI/lib tsub.o -o tsub
pgf77 -g -c texternal.f
pgf77 -g -L/usr/local/PGI/lib texternal.o -o texternal
pgf77 -g -c tcommon.f
pgf77 -g -L/usr/local/PGI/lib tcommon.o -o tcommon
pgf77 -g -c tcommon1.f
pgf77 -g -L/usr/local/PGI/lib tcommon1.o -o tcommon1
```

Source file: `arraydemo.f`

```
c===========================================================
c     arraydemo.f:  Program which demonstrates manipulation
c     of 'run-time' dimensioned arrays in Fortran.
c
c     The program accepts two integer arguments which
c     specify the bounds for the two-dimensional arrays
c     which are to be defined and manipulated.
c
c     The basic guidelines are as follows:
c
c         (1) To deal with run-time defined dimensions,
c             perform all array manipulation (including
c             input and output) in SUBPROGRAMS rather
c             that the main program.
c
c         (2) Always pass ALL bounds of an array, along
c             with the array itself, to subprograms which
c             are to manipulate the array.
c
c         (3) Declare sufficient storage in the main routine
c             to deal with the largest array(s) you
c             anticipate dealing with, but make sure that
c             you always check that the size of the storage
c             is sufficient
c
c         (4) An address of a location in a ONE dimensional
c             array can be passed to a subprogram expecting
c             a multi-dimensional array.
c===========================================================
      program        arraydemo

      implicit       none

      integer        iargc,        i4arg
c-----------------------------------------------------------
c     Single-dimensioned array which can be used to provide
c     storage for the multi-dimensional array manipulation.
c     ("Poor-man's memory allocation")
c-----------------------------------------------------------
      integer        maxq
      parameter    ( maxq = 100 000 )
      real*8         q(maxq)
c-----------------------------------------------------------
c     'Pointer' to next available location in 'q'
c-----------------------------------------------------------
      integer        qnext
c-----------------------------------------------------------
c     'Pointers' for three 2-D arrays ('a1', 'a2', and 'a3')
c-----------------------------------------------------------
      integer        narray
      parameter    ( narray = 3 )

      integer        a1,       a2,       a3
c-----------------------------------------------------------
c     Array bounds which are to be defined at run time
c-----------------------------------------------------------
      integer        n1,       n2

c-----------------------------------------------------------
c     Get the desired array bounds from the command-line
c     and check that there is sufficient 'main-storage'.
c-----------------------------------------------------------
      if( iargc() .ne. 2 ) go to 900
      n1 = i4arg(1,-1)
      n2 = i4arg(2,-1)
      if( n1 .le. 0  .or.  n2 .le.  0 ) go to 900
      if( narray * n1 * n2 .gt. maxq ) then
         write(0,*) 'arraydemo: Insufficient main storage'
         stop
      end if
c-----------------------------------------------------------
c     Initialize the main storage pointer ...
c-----------------------------------------------------------
      qnext = 1
c-----------------------------------------------------------
c     ... and set up the 'pointers' for the two arrays
c     with bounds (n1,n2).
c-----------------------------------------------------------
      a1 = qnext
```

```
      qnext = qnext + n1 * n2
      a2 = qnext
      qnext = qnext + n1 * n2
      a3 = qnext
c-----------------------------------------------------------
c     Define and manipulate the 2-d arrays using various
c     subroutines.
c-----------------------------------------------------------
      call load2d( q(a1), n1, n2,  1.0d0 )
      call load2d( q(a2), n1, n2, -1.0d0 )
      call add2d(  q(a1), q(a2), q(a3), n1, n2 )

c-----------------------------------------------------------
c     Dump the 3 arrays to standard error.
c-----------------------------------------------------------
      call dump2d( q(a1), n1, n2, 'a1' )
      call dump2d( q(a2), n1, n2, 'a2' )
      call dump2d( q(a3), n1, n2, 'a1 + a2' )

      stop

900   continue
         write(0,*) 'usage: arraydemo <n1> <n2>'
      stop

      end

c-----------------------------------------------------------
c     Loads a 2-D array with the values:
c
c     a(i,j) = sc * (100 * j + i)
c-----------------------------------------------------------
      subroutine load2d(a,d1,d2,sc)
         implicit      none

         integer       d1,        d2
         real*8        a(d1,d2)
         real*8        sc

         integer       i,         j

         do j = 1 , d2
            do i = 1 , d1
               a(i,j) = sc * (100.0d0 * j + i)
            end do
         end do

         return

      end

c-----------------------------------------------------------
c     Adds 2-D arrays 'a1' and 'a2' element-wise and returns
c     result in 'a3'
c-----------------------------------------------------------
      subroutine add2d(a1,a2,a3,d1,d2)
         implicit      none

         integer       d1,        d2
         real*8        a1(d1,d2), a2(d1,d2), a3(d1,d2)

         integer       i,         j

         do j = 1 , d2
            do i = 1 , d1
               a3(i,j) = a1(i,j) + a2(i,j)
            end do
         end do

         return

      end

c-----------------------------------------------------------
c     Dumps 2-d array labelled with 'label' on stderr
c-----------------------------------------------------------
      subroutine dump2d(a,d1,d2,label)
         implicit       none

         integer        d1,       d2
         real*8         a(d1,d2)
         character*(*)  label
         integer        i,        j,        st

         if( d1 .gt. 0  .and.  d2 .gt. 0 ) then
            write(0,100) label
100         format( /' <<< ',A,' >>>'/)
            do j = 1 , d2
               st = 1
110            continue
                  write(0,120) ( a(i,j) , i = st , min(st+7,d1))
120               format(' ',8F9.3)
                  st = st + 8
               if( st .le. d1 ) go to 110
               if( j .lt. d2 ) write(0,*)
            end do
         end if

         return

      end
```

Source file: `arraydemo-output`

```
############################################################
#     Sample output from 'arraydemo'
############################################################

lnx1 1> make arraydemo
pgf77 -g -c arraydemo.f
pgf77 -g -L/usr/local/PGI/lib arraydemo.o -lp410f -o arraydemo

lnx1 2> arraydemo
 usage: arraydemo <n1> <n2>

lnx1 3> arraydemo 3 4

 <<< a1 >>>

   101.000  102.000  103.000

   201.000  202.000  203.000

   301.000  302.000  303.000

   401.000  402.000  403.000

 <<< a2 >>>

  -101.000 -102.000 -103.000

  -201.000 -202.000 -203.000

  -301.000 -302.000 -303.000

  -401.000 -402.000 -403.000

 <<< a1 + a2 >>>

     0.000    0.000    0.000

     0.000    0.000    0.000

     0.000    0.000    0.000

     0.000    0.000    0.000
```

Source file: `nth-output`

```
############################################################
# Illustrates use of 'nth', a script/filter available on the
# machines for selecting columns from standard input
#############################################################

lnx1 1> cat powers
1    1     1      1
2    4     8      16
3    9     27     81
4    16    64     256
5    25    125    625
6    36    216    1296
7    49    343    2401
8    64    512    4096
9    81    729    6561
10   100   1000   10000

lnx1 2> nth 1 2 < powers
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100

lnx1 3> nth 1 3 1 < powers
1 1 1
2 8 2
3 27 3
4 64 4
5 125 5
6 216 6
7 343 7
8 512 8
9 729 9
10 1000 10
```

Source file: `arraydemo90.f`

```
!============================================================
!     arraydemo90.f:  Fortran 90 version of arraydemo.f
!
!     Fortran 90 implements run-time allocation of arrays
!     and other data objects.
!
!     Array names are identified as dynamically allocated
!     via the 'allocatable' attribute in the array
!     declaration.  Storage is allocated via the 'allocate'
!     statement, and freed with 'deallocate'.
!============================================================
      program        arraydemo90

      implicit       none

      integer        iargc,    i4arg

!------------------------------------------------------------
!     Identify a1, a2 and a3 as rank-2 allocatable arrays.
!
!     Alternate equivalent declaration:
!
!     real*8, allocatable:: a1(:,:), a2(:,:), a3(:,:)
!------------------------------------------------------------
      real*8,  allocatable, dimension( : , : ) ::
     &               a1,       a2,       a3

!------------------------------------------------------------
!     Run-time array bounds.
!------------------------------------------------------------
      integer        n1,       n2

!------------------------------------------------------------
!     Get the desired array bounds from the command-line
!     and perform superficial check for validity.
!------------------------------------------------------------
      if( iargc() .ne. 2 ) go to 900
      n1 = i4arg(1,-1)
      n2 = i4arg(2,-1)
      if( n1 .le. 0  .or.  n2 .le.  0 ) go to 900
!------------------------------------------------------------
!     Allocate the arrays
!------------------------------------------------------------
      allocate( a1(n1,n2) )
      allocate( a2(n1,n2) )
      allocate( a3(n1,n2) )

!------------------------------------------------------------
!     Define and manipulate the 2-d arrays using various
!     subroutines.
!------------------------------------------------------------
      call load2d( a1, n1, n2,  1.0d0 )
      call load2d( a2, n1, n2, -1.0d0 )
      call add2d(  a1, a2, a3, n1, n2 )

!------------------------------------------------------------
!     Dump the 3 arrays to standard error.
!------------------------------------------------------------
      call dump2d( a1, n1, n2, 'a1' )
      call dump2d( a2, n1, n2, 'a2' )
      call dump2d( a3, n1, n2, 'a1 + a2' )

!------------------------------------------------------------
!     Deallocate the arrays
!------------------------------------------------------------
      deallocate(a1)
      deallocate(a2)
      deallocate(a3)

      stop

900   continue
         write(0,*) 'usage: arraydemo90 <n1> <n2>'
      stop

      end

!------------------------------------------------------------
!     Loads a 2-D array with the values:
```

```
!
!     a(i,j) = sc * (100 * j + i)
!-----------------------------------------------------------
      subroutine load2d(a,d1,d2,sc)
         implicit      none

         integer       d1,        d2
         real*8        a(d1,d2)
         real*8        sc

         integer       i,         j

         do j = 1 , d2
            do i = 1 , d1
               a(i,j) = sc * (100.0d0 * j + i)
            end do
         end do

         return

      end

!-----------------------------------------------------------
!     Adds 2-D arrays 'a1' and 'a2' element-wise and returns
!     result in 'a3'
!-----------------------------------------------------------
      subroutine add2d(a1,a2,a3,d1,d2)
         implicit      none

         integer       d1,        d2
         real*8        a1(d1,d2), a2(d1,d2), a3(d1,d2)

         integer       i,         j

         do j = 1 , d2
            do i = 1 , d1
               a3(i,j) = a1(i,j) + a2(i,j)
            end do
         end do

         return

      end

!-----------------------------------------------------------
!     Dumps 2-d array labelled with 'label' on stderr
!-----------------------------------------------------------
      subroutine dump2d(a,d1,d2,label)
         implicit       none

         integer        d1,       d2
         real*8         a(d1,d2)
         character*(*)  label
         integer        i,        j,        st

         if( d1 .gt. 0  .and.  d2 .gt. 0 ) then
            write(0,100) label
 100        format( /' <<< ',A,' >>>'/)
            do j = 1 , d2
               st = 1
 110           continue
                  write(0,120) ( a(i,j) , i = st , min(st+7,d1))
 120              format(' ',8F9.3)
                  st = st + 8
               if( st .le. d1 ) go to 110
               if( j .lt. d2 ) write(0,*)
            end do
         end if

         return

      end
```

Source file: meps.f

```
c===========================================================
c     Computes and reports estimate of machine epsilon.
c
c     Recall: machine epsilon is smallest positive 'eps'
c     such that
c
c             (1.0d0 + eps ) .ne. (1.0d0)
c
c     Program accepts optional argument which specifies
c     division factor: values close to 1.0 will result
c     in more accurate estimate of machine epsilon.
c===========================================================
      program        meps

      implicit       none

c-----------------------------------------------------------
c     Note use of 'r8arg', available in 'libp410f.a' which
c     works exactly like 'i4arg' except that it returns
c     a real*8 value parsed from the specified command-line
c     argument
c-----------------------------------------------------------
      real*8         r8arg

      real*8         default_fac
      parameter     ( default_fac = 2.0d0 )

      real*8         eps,        neweps,       fac

      fac = r8arg(1,default_fac)
      write(0,*) 'meps: using division factor: ', fac

      eps    = 1.0d0
      neweps = 1.0d0
      do while( 1.0d0 .ne. (1.0d0 + neweps) )
         eps    = neweps
         neweps = neweps / fac
      end do

      write(*,*) eps

      stop

      end
```

Source file: meps-sgi-output

```
############################################################
# Output from 'meps' on Sun 4 (IEEE floating point)
############################################################

physics 41> make meps
f77 -g -c meps.f
meps.f:
 MAIN meps:
f77 -g -L/home/choptuik/lib meps.o -lp410f -o meps

physics 42> meps
 meps: using division factor:     2.0000000000000
     2.2204460492503D-16

physics 43> meps 1.01
 meps: using division factor:     1.0100000000000
     1.1104218387155D-16

physics 44> meps 1.0001
 meps: using division factor:     1.0001000000000
     1.1102645224602D-16
```

Source file: `meps-pclinux-output`

```
##############################################################
# Output from 'meps' on PC Linux machine (80 bit floating pt)
##############################################################

lnx1 1> make meps
pgf77 -g -c meps.f
pgf77 -g -L/usr/local/PGI/lib meps.o -lp410f -o meps

lnx1 2> meps
 meps: using division factor:    2.000000000000000
   1.0842021724855044E-019

lnx1 3> meps 1.01
 meps: using division factor:    1.010000000000000
   5.4364534909517435E-020

lnx1 4> meps 1.0001
 meps: using division factor:    1.000100000000000
   5.4212146310714582E-020
```

Source file: `catprec.f`

```
c===========================================================
c     Program illustrating "catastrophic" loss of precision
c     resulting from the subtraction of two nearly equal
c     floating point values.
c===========================================================
      program        catprec

      implicit       none

      real*8         x
      parameter     ( x = 0.2d0 )

      integer        i
      real*8         h,    dsinx

      write(*,*) '      h      d(sin) approx   '//
     &          'd(sin) exact    d(sin) err'
      write(*,*)

      h = 0.5d0
      do i = 1 , 16
c-----------------------------------------------------------
c        Algebraically, in the limit h -> 0, dsinx should
c        approach cos(x), but sin(x+h) -> sin(x) so
c        catastrophic loss of precision occurs.
c-----------------------------------------------------------
         dsinx = (sin(x+h) - sin(x)) / h
         write(*,1000) h,  dsinx,  cos(x),   dsinx - cos(x)
1000     format(1P,E12.3,2E16.8,E12.3)
         h = 0.125d0 * h
      end do

      stop

      end
```

Source file: `catprec-output`

```
############################################################
# Output from 'catprec' illustrating catastrophic precision
# loss due to subtraction of nearly-equal floating point
# values.
############################################################

lnx1 1>  make catprec
pgf77 -g -c catprec.f
pgf77 -g -L/usr/local/PGI/lib catprec.o -o catprec

lnx1 2> catprec
      h      d(sin) approx   d(sin) exact    d(sin) err

   5.000E-01  8.91096713E-01  9.80066578E-01  -8.897E-02
   6.250E-02  9.73222242E-01  9.80066578E-01  -6.844E-03
   7.813E-03  9.79280560E-01  9.80066578E-01  -7.860E-04
   9.766E-04  9.79969416E-01  9.80066578E-01  -9.716E-05
   1.221E-04  9.80054450E-01  9.80066578E-01  -1.213E-05
   1.526E-05  9.80065062E-01  9.80066578E-01  -1.516E-06
   1.907E-06  9.80066388E-01  9.80066578E-01  -1.895E-07
   2.384E-07  9.80066554E-01  9.80066578E-01  -2.368E-08
   2.980E-08  9.80066575E-01  9.80066578E-01  -2.960E-09
   3.725E-09  9.80066577E-01  9.80066578E-01  -3.702E-10
   4.657E-10  9.80066578E-01  9.80066578E-01  -5.373E-11
   5.821E-11  9.80066578E-01  9.80066578E-01  -1.701E-10
   7.276E-12  9.80066577E-01  9.80066578E-01  -8.686E-10
   9.095E-13  9.80066568E-01  9.80066578E-01  -1.018E-08
   1.137E-13  9.80066538E-01  9.80066578E-01  -3.998E-08
   1.421E-14  9.80066299E-01  9.80066578E-01  -2.784E-07
```

Source file: `dmroutines.f`

```
c===========================================================
c     Implements matrix-matrix multiply
c
c     c = a b
c
c     where a, b and c are n x n (square) real*8 matrices.
c===========================================================
      subroutine dmmmult(a,b,c,n)

         implicit         none

         integer          n
         real*8           a(n,n),    b(n,n),    c(n,n)

         integer          i,     j,     k

         do j = 1 , n
            do i = 1 , n
               c(i,j) = 0.0d0
               do k = 1 , n
                  c(i,j) = c(i,j) + a(i,k) * b(k,j)
               end do
            end do
         end do

         return

      end

c===========================================================
c     Writes a double precision matrix (two dimensional
c     array) to file 'fname'.  If 'fname' is the
c     string '-', the matrix is written to standard input.
c
c     This routine is modelled on 'dvto' previously
c     discussed in class: see ~phys410/f77/ex3/dvto.f
c===========================================================

      subroutine dmto(fname,a,d1,d2)
c-----------------------------------------------------------
c     Arguments:
c
c        fname:  (I)    File name
c        a:      (I)    Input matrix
c        d1:     (I)    First dimension of a
c        d2:     (I)    Second dimension of a
c-----------------------------------------------------------
         implicit         none

         integer          indlnb,      getu

         character*(*)    fname
         integer          d1,          d2
         real*8           a(d1,d2)

         integer          ustdout
         parameter      ( ustdout = 6 )

         integer          uto,         rc

c-----------------------------------------------------------
c        Parse fname: either "attach" 'uto' to stdout or
c        get a unit number using 'getu', and open the
c        file 'fname' for formatted I/O via 'uto'
c-----------------------------------------------------------
         if( fname .eq. '-' ) then
            uto = ustdout
         else
            uto = getu()
            open(uto,file=fname(1:indlnb(fname)),
     &           form='formatted',iostat=rc)
            if( rc .ne. 0 ) then
               write(0,*) 'dmto: Error opening ',
     &                    fname(1:indlnb(fname))
               return
            end if
         end if

c-----------------------------------------------------------
c        Write dimensions, then array elements
c-----------------------------------------------------------
         write(uto,*,iostat=rc) d1, d2
         if( rc .ne. 0 ) then
            write(0,*) 'dmto: Error writing dimensions'
         go to 500
         end if

         write(uto,*,iostat=rc) a
         if( rc .ne. 0 ) then
            write(0,*) 'dmto: Error reading matrix'
         end if

c-----------------------------------------------------------
c        Exit: Close file and return
c-----------------------------------------------------------
 500     continue
         if( uto .ne. ustdout ) then
            close(uto)
         end if

         return

         end

c===========================================================
c     Returns a double precision matrix (two dimensional
c     array) read from file 'fname'.  If 'fname' is the
c     string '-', the matrix is read from standard input.
c
c     The dimensions of the matrix must precede the matrix
c     elements themselves in the file.  Specifically, the
c     file should have been created using the following
c     list-directed, formatted READ statement
c     (or equivalent):
c
c        integer      d1,    d2
c        real*8       a(d1,d2)
c        integer      uout
c
c        write(uout,*) d1, d2
c        write(uout,*) a
c
c     This routine is modelled on 'dvfrom' previously
c     discussed in class: see ~phys410/f77/ex3/dvfrom.f
c
c     Note the use of helper routine 'dmfrom1' which
c     reads actual array values once bounds have been
c     extracted from file.
c===========================================================

      subroutine dmfrom(fname,a,d1,d2,asize)
c-----------------------------------------------------------
c     Arguments:
c
c        fname:  (I)    File name
c        a:      (O)    Return matrix
c        d1:     (O)    First dimension of a
c        d2:     (O)    Second dimension of a
c        asize:  (I)    Maximum size (d1 * d2) of a
c-----------------------------------------------------------
         implicit         none

         integer          indlnb,      getu

         character*(*)    fname
         integer          d1,          d2,          asize
         real*8           a(d1,d2)

         integer          ustdin
         parameter      ( ustdin = 5 )

         integer          ufrom,       rc

c-----------------------------------------------------------
c        Parse fname: either "attach" 'ufrom' to stdin or
c        get a unit number using 'getu', and open the
c        file 'fname' for formatted I/O via 'ufrom'
c-----------------------------------------------------------
         if( fname .eq. '-' ) then
            ufrom = ustdin
```

```
         else
            ufrom = getu()
            open(ufrom,file=fname(1:indlnb(fname)),
     &           form='formatted',iostat=rc,status='old')
            if( rc .ne. 0 ) then
               write(0,*) 'dmfrom: Error opening ',
     &                    fname(1:indlnb(fname))
               return
            end if
         end if

c------------------------------------------------------------
c        Read dimensions and abort if there is insufficient
c        storage for the entire matrix.  Note the 'go to'
c        to the 'exit block' since we've opened a file now
c        and should close it, even if there's an error.
c        Also, we set the dimensions to 0 for all error
c        conditions as a way of communicating failure to
c        the calling routine.
c------------------------------------------------------------
         read(ufrom,*,iostat=rc) d1, d2
         if( rc .ne. 0 ) then
            write(0,*) 'dmfrom: Error reading dimensions'
            d1 = 0
            d2 = 0
         go to 500
         end if
         if( (d1 * d2) .gt. asize ) then
            write(0,*) 'dmfrom: Insufficient storage'
            d1 = 0
            d2 = 0
         go to 500
         end if
c------------------------------------------------------------
c        Now that dimensions have been determined call
c        helper routine to read values
c------------------------------------------------------------
         call dmfrom1(ufrom,a,d1,d2,rc)
         if( rc .ne. 0 ) then
            write(0,*) 'dmfrom: Error reading matrix'
            d1 = 0
            d2 = 0
         end if
c------------------------------------------------------------
c        Exit: Close file and return
c------------------------------------------------------------
 500     continue
         if( ufrom .ne. ustdin ) then
            close(ufrom)
         end if

         return

      end

c============================================================
c     Helper routine for dmfrom: Reads array values, returns
c     I/O status to calling routine via 'rc'
c============================================================
      subroutine dmfrom1(ufrom,a,d1,d2,rc)

         implicit      none

         integer       d1,       d2,     ufrom,     rc
         real*8        a(d1,d2)

         read(ufrom,*,iostat=rc) a

         return

      end
```

Source file: `tdm.f`

```
c============================================================
c     Test program for subroutine 'dmfrom', 'dmto' and
c     'dmmmult' (see 'dmroutines.f')
c
c     Program expects one argument, the name of a file which
c     contains a real*8 square matrix written as descibed
c     in the documentation for 'dmfrom' in 'dmroutines.f'
c     Use '-' to read from stdin.  Program then computes
c     square of matrix and outputs result to stdout.
c============================================================

      program         tdm

      implicit        none

      integer         iargc

      character*256   fname

c------------------------------------------------------------
c     Maximum size for input and output arrays (matrices).
c------------------------------------------------------------
      integer         maxsize
      parameter     ( maxsize = 100 000 )
      real*8          a(maxsize),   asq(maxsize)
      integer         d1a,          d2a

      if( iargc() .ne. 1 ) go to 900
      call getarg(1,fname)

c------------------------------------------------------------
c     Read matrix ...
c------------------------------------------------------------
      call dmfrom(fname,a,d1a,d2a,maxsize)
      if( d1a .gt. 0  .and.  d2a .gt. 0 ) then
         if( d1a .eq. d2a ) then
c------------------------------------------------------------
c           Compute square ...
c------------------------------------------------------------
            call dmmmult(a,a,asq,d1a,d1a)
c------------------------------------------------------------
c           ... and output.
c------------------------------------------------------------
            call dmto('-',asq,d1a,d1a)
         else
            write(0,*) 'tdm: Input array not square'
         end if
      else
         write(0,*) 'tdm: dmfrom() failed'
      end if

      stop

 900  continue
         write(0,*) 'usage: tdm <file name>'
         write(0,*)
         write(0,*) '       Use ''tdm -'' to read ',
     &              'from standard input'

      stop

      end
```

Source file: `tdm-output`

```
############################################################
# Building 'tdm' and sample output
#############################################################

lnx1 1> make tdm
pgf77 -g -c tdm.f
pgf77 -g -c dmroutines.f
pgf77 -g -L/usr/local/PGI/lib tdm.o dmroutines.o -lp410f -o tdm

lnx1 2> tdm
 usage: tdm <file name>

        Use 'tdm -' to read from standard input

lnx1 3> tdm -
2 2
1 2 3 4
            2           2
    7.000000000000000        10.00000000000000        15.00000000000000
    22.00000000000000

lnx1 4> tdm -
2 3
1 2 3 4 5 6
 tdm: Input array not square
```

Source file: `Makefile`

```
.IGNORE:

F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS)
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)

.f.o:
   $(F77_COMPILE) $*.f

EXECUTABLES = meps catprec tdm

all: $(EXECUTABLES)

meps: meps.o
   $(F77_LOAD) meps.o -lp410f -o meps

catprec: catprec.o
   $(F77_LOAD) catprec.o -o catprec

tdm: tdm.o dmroutines.o
   $(F77_LOAD) tdm.o dmroutines.o -lp410f -o tdm

clean:
   rm *.o
   rm $(EXECUTABLES)
   rm core
```