Source file: `trand.f`

Source file: `trand-output`

```
Script started on Wed Sep 20 19:06:37 2000

sgi1 1> make trand
f77 -g -64 -c trand.f
f77 -g -64 -L/usr/local/lib trand.o -lp410f -o trand

sgi1 2> trand 10
   0.5138549804687500
   0.1757202148437500
   0.3086242675781250
   0.5345153808593750
   0.9476013183593750
   0.1717224121093750
   0.7022094726562500
   0.2264099121093750
   0.4947509765625000
   0.1246948242187500

 Average:   0.4200103759765625

sgi1 3> foreach n (10 100 1000 10000 100000)
foreach? trand $n > /dev/null
foreach? end

 Average:   0.4200103759765625

 Average:   0.5154736328125000

 Average:   0.5092929992675781

 Average:   0.5025000335693359

 Average:   0.5015412191772461
```

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
```