

# Answers to Selected Exercises for *Fortran 90/95 for Scientists and Engineers* by Stephen J. Chapman

## Chapter 1. Introduction to Computers and the Fortran Language

- 1-1 (a)  $1010_2$  (c)  $1001101_2$
- 1-2 (a)  $72_{10}$  (c)  $255_{10}$
- 1-5 A 23-bit mantissa can represent approximately  $\pm 2^{22}$  numbers, or about six significant decimal digits. A 9-bit exponent can represent multipliers between  $2^{-255}$  and  $2^{255}$ , so the range is from about  $10^{-76}$  to  $10^{76}$ .
- 1-8 The sum of the two's complement numbers is:
- |                      |   |             |
|----------------------|---|-------------|
| $0010010010010010_2$ | = | $9362_{10}$ |
| $1111110011111100_2$ | = | $-772_{10}$ |
| $0010000110001110_2$ | = | $8590_{10}$ |
- The two answers agree with each other.

## Chapter 2. Basic Elements of Fortran

- 2-1 (a) Valid real constant (c) Invalid constant—numbers may not include commas (e) Invalid constant—need two single quotes to represent a quote within a string
- 2-4 (a) Legal: result = 0.888889 (c) Illegal—cannot raise a negative real number to a negative real power
- 2-12 The output of the program is:
- |           |            |            |     |      |      |
|-----------|------------|------------|-----|------|------|
| -3.141592 | 100.000000 | 200.000000 | 300 | -100 | -200 |
|-----------|------------|------------|-----|------|------|
- 2-13 The weekly pay program is shown below:

```
PROGRAM get_pay
!
! Purpose:
!   To calculate an hourly employee's weekly pay.
!
! Record of revisions:
!   Date       Programmer      Description of change
!   ====       =====
!   12/30/96   S. J. Chapman   Original code
!
IMPLICIT NONE

! List of variables:
```

```

REAL :: hours      ! Number of hours worked in a week.
REAL :: pay        ! Total weekly pay.
REAL :: pay_rate   ! Pay rate in dollars per hour.

! Get pay rate
WRITE (*,*) 'Enter employees pay rate in dollars per hour: '
READ (*,*) pay_rate

! Get hours worked
WRITE (*,*) 'Enter number of hours worked: '
READ (*,*) hours

! Calculate pay and tell user.
pay = pay_rate * hours
WRITE (*,*) "Employee's pay is $", pay

END PROGRAM

```

The result of executing this program is

```

C:\BOOK\F90\SOLN>get_pay
Enter employees pay rate in dollars per hour:
5.50
Enter number of hours worked:
39
Employee's pay is $      214.500000

```

2-17 A program to calculate the hypotenuse of a triangle from the two sides is shown below:

```

PROGRAM calc_hypotenuse
!
! Purpose:
!   To calculate the hypotenuse of a right triangle, given
!   the lengths of its two sides.
!
! Record of revisions:
!   Date      Programmer      Description of change
!   ====      =====
!   12/30/96   S. J. Chapman   Original code
!
IMPLICIT NONE

! List of variables:
REAL :: hypotenuse ! Hypotenuse of triangle
REAL :: side_1     ! Side 1 of triangle
REAL :: side_2     ! Side 2 of triangle

! Get lengths of sides.
WRITE (*,*) 'Program to calculate the hypotenuse of a right '
WRITE (*,*) 'triangle, given the lengths of its sides. '
WRITE (*,*) 'Enter the length side 1 of the right triangle: '
READ (*,*) side_1
WRITE (*,*) 'Enter the length side 2 of the right triangle: '
READ (*,*) side_2

! Calculate length of the hypotenuse.

```

```

hypotenuse = SQRT ( side_1**2 + side_2**2 )

! Write out results.
WRITE (*,*) 'The length of the hypotenuse is: ', hypotenuse

END PROGRAM

```

2-21 A program to calculate the hyperbolic cosine is shown below:

```

PROGRAM coshx
!
! Purpose:
!   To calculate the hyperbolic cosine of a number.
!
! Record of revisions:
!   Date      Programmer      Description of change
!   ====      =====
!   12/30/96   S. J. Chapman   Original code
!
IMPLICIT NONE

! List of variables:
REAL :: result      ! COSH(x)
REAL :: x           ! Input value

WRITE (*,*) 'Enter number to calculate cosh() of: '
READ (*,*) x

result = ( EXP(x) + EXP(-x) ) / 2.

WRITE (*,*) 'COSH(X) =', result

END PROGRAM

```

When this program is run, the result is:

```

C:\BOOK\F90\SOLN>coshx
Enter number to calculate cosh() of:
3.0
COSH(X) =      10.067660

```

The Fortran 90 intrinsic function `COSH()` produces the same answer.

### Chapter 3. Control Structures and Program Design

3-2 The statements to calculate  $y(t)$  for values of  $t$  between -9 and 9 in steps of 3 are:

```

IMPLICIT NONE
INTEGER :: i
REAL :: t, y

DO i = -9, 9, 3
    t = REAL(i)

```

```

      IF ( t >= 0. ) THEN
        y = -3.0 * t**2 + 5.0
      ELSE
        y = 3.0 * t**2 + 5.0
      END IF
      WRITE (*,*) 't = ', t, '      y(t) = ', y
    END DO
  END PROGRAM

```

- 3-6 The statements are incorrect. In an IF construct, the first branch whose condition is true is executed, and all others are skipped. Therefore, if the temperature is 104.0, then the second branch would be executed, and the code would print out 'Temperature normal' instead of 'Temperature dangerously high'. A correct version of the IF construct is shown below:

```

IF ( temp < 97.5 ) THEN
  WRITE (*,*) 'Temperature below normal'
ELSE IF ( TEMP > 103.0 ) THEN
  WRITE (*,*) 'Temperature dangerously high'
ELSE IF ( TEMP > 99.5 ) THEN
  WRITE (*,*) 'Temperature slightly high'
ELSE IF ( TEMP > 97.5 ) THEN
  WRITE (*,*) 'Temperature normal'
END IF

```

- 3-14 (a) This loop is executed 21 times, and afterwards `i res = 21`. (c) This outer loop is executed 4 times, the inner loop is executed 13 times, and afterwards `i res = 42`.
- 3-19 The legal values of  $x$  for this function are all  $x < 1.0$ , so the program should contain a while loop which calculates the function  $y(x) = \ln \frac{1}{1-x}$  for any  $x < 1.0$ , and terminates when  $x \geq 1.0$  is entered.

```

PROGRAM evaluate
!
! Purpose:
!   To evaluate the function ln(1./(1.-x)).
!
! Record of revisions:
!   Date           Programmer           Description of change
!   ====           =====
!   12/30/96       S. J. Chapman        Original code
!
IMPLICIT NONE

! Declare local variables:
REAL :: value      ! Value of function ln(1./(1.-x))
REAL :: x          ! Independent variable

! Loop over all valid values of x
DO

  ! Get next value of x.
  WRITE (*,*) 'Enter value of x: '
  READ (*,*) x

  ! Check for invalid value

```

```

        IF ( x >= 1. ) EXIT

        ! Calculate and display function
        value = LOG ( 1. / ( 1. - x ) )
        WRITE (*,*) 'LN(1./(1.-x)) = ', value

    END DO

END PROGRAM

```

- 3-28 A program to calculate the harmonic of an input data set is shown below. This problem gave the student the freedom to input the data in any way desired; I have chosen a **DO** loop for this example program.

```

PROGRAM harmon
!
! Purpose:
!   To calculate harmonic mean of an input data set, where each
!   input value can be positive, negative, or zero.
!
! Record of revisions:
!   Date           Programmer           Description of change
!   ====           =====
!   12/31/96       S. J. Chapman        Original code
!
IMPLICIT NONE

! List of variables:
REAL :: h_mean      ! Harmonic mean
INTEGER :: i         ! Loop index
INTEGER :: n         ! Number of input samples
REAL :: sum_rx = 0.  ! Sum of reciprocals of input values
REAL :: x = 0.       ! Input value

! Get the number of points to input.
WRITE (*,*) 'Enter number of points: '
READ  (*,*) n

! Loop to read input values.
DO i = 1, n

    ! Get next number.
    WRITE (*,*) 'Enter next number: '
    READ  (*,*) x

    ! Accumulate sums.
    sum_rx = sum_rx + 1.0 / x

END DO

! Calculate the harmonic mean
h_mean = REAL (n) / sum_rx

! Tell user.
WRITE (*,*) 'The harmonic mean of this data set is:', h_mean
WRITE (*,*) 'The number of data points is:          ', n

```

END PROGRAM

When the program is run with the sample data set, the results are:

C:\BOOK\F90\SOLN>harmon

Enter number of points:

4

Enter next number:

10.

Enter next number:

5.

Enter next number:

2.

Enter next number:

5.

The harmonic mean of this data set is: 4.000000

The number of data points is: 4

## Chapter 4. Basic I/O Concepts

4-3 (b) The result is printed out on the next line. It is:

A =	1.002000E+06	B =	.100010E+07	Sum =	.200210E+07	Diff =	1900.000000
----- ----- ----- ----- ----- ----- ----- -----							
	10	20	30	40	50	60	70 80

4-8 A program to calculate the average and standard deviation of an input data set stored in a file is shown below:

PROGRAM ave\_sd

!

! To calculate the average (arithmetic mean) and standard  
! deviation of an input data set found in a user-specified  
! file, with the data arranged so that there is one value  
! per line.

!

! Record of revisions:

Date	Programmer	Description of change
------	------------	-----------------------

====	=====	=====
------	-------	-------

01/04/97	S. J. Chapman	Original code
----------	---------------	---------------

!

IMPLICIT NONE

! Declare variables

REAL :: ave	! Average (arithmetic mean)
-------------	-----------------------------

CHARACTER(len=20) :: filename	! Name of file to open
-------------------------------	------------------------

INTEGER :: nvals = 0	! Number of values read in
----------------------	----------------------------

REAL :: sd	! Standard deviation
------------	----------------------

INTEGER :: status	! I/O status
-------------------	--------------

REAL :: sum_x = 0.0	! Sum of the input values
---------------------	---------------------------

REAL :: sum_x2 = 0.0	! Sum of input values squared
----------------------	-------------------------------

REAL :: value	! The real value read in
---------------	--------------------------

```

! Get the file name, and echo it back to the user.
WRITE (*,1000)
1000 FORMAT (1X,'This program calculates the average and standard ' &
           ,/,1X,'deviation of an input data set. Enter the name' &
           ,/,1X,'of the file containing the input data:' )
READ (*,*) filename

! Open the file, and check for errors on open.
OPEN (UNIT=3, FILE=filename, STATUS='OLD', ACTION='READ', &
      IOSTAT=status )
openif: IF ( status == 0 ) THEN

    ! OPEN was ok. Read values.
    readloop: DO
        READ (3,*,IOSTAT=status) value      ! Get next value
        IF ( status /= 0 ) EXIT              ! EXIT if not valid.
        nvals = nvals + 1                    ! Valid: increase count
        sum_x = sum_x + value                ! Sums
        sum_x2 = sum_x2 + value**2           ! Sum of squares
    END DO readloop

    ! The WHILE loop has terminated. Was it because of a READ
    ! error or because of the end of the input file?
    readif: IF ( status > 0 ) THEN ! a READ error occurred. Tell user.

        WRITE (*,1020) nvals + 1
        1020 FORMAT ('0','An error occurred reading line ', I6)

    ELSE ! the end of the data was reached. Calculate ave & sd.

        ave = sum_x / REAL(nvals)
        sd = SQRT( (REAL(nvals)*sum_x2-sum_x**2)/(REAL(nvals)*REAL(nvals-1)))
        WRITE (*,1030) filename, ave, sd, nvals
        1030 FORMAT ('0','Statistical information about data in file ',A, &
                   ,/,1X,' Average = ', F9.3, &
                   ,/,1X,' Standard Deviation = ', F9.3, &
                   ,/,1X,' No of points = ', I9 )
    END IF readif

ELSE openif
    WRITE (*,1040) status
    1040 FORMAT (' ','Error opening file: IOSTAT = ', I6 )
END IF openif

! Close file
CLOSE ( UNIT=8 )

END PROGRAM

```

## Chapter 5. Arrays

5-4 (a) 60 elements; valid subscript range is 1 to 60. (c) 105 elements; valid subscript range is (1,1) to (35,3).

- 5-5 (a) Valid. These statements declare and initialize the 100-element array **icount** to 1, 2, 3, ..., 100, and the 100-element array **jcount** to 2, 3, 4, ..., 101. (d) Valid. The **WHERE** construct multiplies the positive elements of array **info** by -1, and negative elements by -3. It then writes out the values of the array: -1, 9, 0, 15, 27, -3, 0, -1, -7.

$$z = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

- 5-9 (b) The **READ** statement here reads all values from the first line, then all the values from the second line, etc. until 16 values have been read. The values are stored in array **values** in row order. Therefore, array **values** will contain the following values

$$\text{values} = \begin{bmatrix} 27 & 17 & 10 & 8 \\ 6 & 11 & 13 & -11 \\ 12 & -21 & -1 & 0 \\ 0 & 6 & 14 & -16 \end{bmatrix}$$

- 5-25 A program to calculate the distance between two points  $(x_1, y_1, z_1)$  and  $(x_2, y_2, z_2)$  in three-dimensional space is shown below:

```
PROGRAM dist_3d
!
! Purpose:
!   To calculate the distance between two points (x1,y1,z1)
!   and (x2,y2,z2) in three-dimensional space.
!
! Record of revisions:
!   Date           Programmer           Description of change
!   ====           =====
!   01/05/97       S. J. Chapman       Original code
!
IMPLICIT NONE

! List of variables:
REAL :: dist      ! Distance between the two points.
REAL :: x1        ! x-component of first vector.
REAL :: x2        ! x-component of second vector.
REAL :: y1        ! y-component of first vector.
REAL :: y2        ! y-component of second vector.
REAL :: z1        ! z-component of first vector.
REAL :: z2        ! z-component of second vector.

! Get the first point in 3D space.
WRITE (*,1000)
1000 FORMAT (' Calculate the distance between two points ', &
            ' (X1,Y1,Z1) and (X2,Y2,Z2):' &
            ',1X,'Enter the first point (X1,Y1,Z1): ')
READ (*,*) x1, y1, z1

! Get the second point in 3D space.
WRITE (*,1010)
```

```

1010 FORMAT (' Enter the second point (X2,Y2,Z2): ')
      READ (*,*) x2, y2, z2

! Calculate the distance between the two points.
dist = SQRT ( (x1-x2)**2 + (y1-y2)**2 + (z1-z2)**2 )

! Tell user.
WRITE (*,1020) dist
1020 FORMAT (' The distance between the two points is ', F10.3)

END PROGRAM

```

When this program is run with the specified data values, the results are

```

C:\BOOK\F90\SOLN>dist_3d
Calculate the distance between two points (X1,Y1,Z1) and (X2,Y2,Z2):
Enter the first point (X1,Y1,Z1):
-1. 4. 6.
Enter the second point (X2,Y2,Z2):
1. 5. -2.
The distance between the two points is      8.307

```

## Chapter 6. Procedures and Structured Programming

- 6-1 A function is a procedure whose result is a single number, logical value, or character string, while a subroutine is a subprogram that can return one or more numbers, logical values, or character strings. A function is invoked by naming it in a Fortran expression, while a subroutine is invoked using the **CALL** statement.
- 6-6 Data is passed by reference from a calling program to a subroutine. Since only a pointer to the location of the data is passed, *there is no way for a subroutine with an implicit interface to know that the argument type is mismatched.* (However, some Fortran compilers are smart enough to recognize such type mismatches if both the calling program and the subroutine are contained in the same source file.)

The result of executing this program will vary from processor. When executed on a computer with IEEE standard floating-point numbers, the results are

```

C:\BOOK\F90\SOLN>min
I = -1063256064

```

- 6-10 According to the Fortran 90/95 standard, the values of all the local variables in a procedure become undefined whenever we exit the procedure. The next time that the procedure is invoked, the values of the local variables may or may not be the same as they were the last time we left it, depending on the particular processor being used. If we write a procedure that depends on having its local variables undisturbed between calls, it will work fine on some computers and fail miserably on other ones!

Fortran provides the **SAVE** attribute and the **SAVE** statement to guarantee that local variables are saved unchanged between invocations of a procedure. Any local variables declared with the **SAVE** attribute or listed in a **SAVE** statement will be saved unchanged. If no variables are listed in a **SAVE** statement, then all of the local variables will be saved unchanged. In addition, any local variables that are initialized in a type declaration statement will be saved unchanged between invocations of the procedure.

- 6-26 A subroutine to calculate the derivative of a discrete function is shown below.

```

SUBROUTINE derivative ( vector, deriv, nsamp, dx, error )
!
! Purpose:
!   To calculate the derivative of a sampled function f(x)
!   consisting of nsamp samples spaced a distance dx apart.
!   The resulting derivative is returned in array deriv, and
!   is nsamp-1 samples long. (Since calculating the derivative
!   requires both point i and point i+1, we can't find the
!   derivative for the last point in the input array.)
!
! Record of revisions:
!   Date       Programmer      Description of change
!   ====       =====
!   01/07/97   S. J. Chapman   Original code
!
IMPLICIT NONE

! List of calling arguments:
INTEGER, INTENT(IN) :: nsamp           ! Number of samples
REAL, DIMENSION(nsamp), INTENT(IN) :: vector ! Input data array
REAL, DIMENSION(nsamp-1), INTENT(OUT) :: deriv ! Input data array
REAL, INTENT(IN) :: dx                 ! sample spacing
INTEGER, INTENT(OUT) :: error          ! Flag: 0 = no error
!                                     ! 1 = dx <= 0

! List of local variables:
INTEGER :: i                          ! Loop index

! Check for legal step size.
IF ( dx > 0. ) THEN

    ! Calculate derivative.
    DO i = 1, nsamp-1
        deriv(i) = ( vector(i+1) - vector(i) ) / dx
    END DO
    error = 0

ELSE

    ! Illegal step size.
    error = 1

END IF

END SUBROUTINE

```

A test driver program for this subroutine is shown below. This program creates a discrete analytic function  $f(x) = \sin x$ , and calculates the derivative of that function using subroutine **DERV**. Finally, it compares the result of the subroutine to the analytical solution  $df(x)/dx = \cos x$ , and find the maximum difference between the result of the subroutine and the true solution.

```

PROGRAM test_derivative
!
! Purpose:
!   To test subroutine "derivative", which calculates the numerical

```

```

!   derivative of a sampled function f(x).  This program will take the
!   derivative of the function f(x) = sin(x), where nstep = 100, and
!   dx = 0.05.  The program will compare the derivative with the known
!   correct answer df/dx = cos(x)), and determine the error in the
!   subroutine.
!

```

```

!   Record of revisions:

```

Date	Programmer	Description of change
====	=====	=====
01/07/97	S. J. Chapman	Original code

```

!
IMPLICIT NONE

```

```

! List of named constants:

```

INTEGER, PARAMETER :: nsamp = 100	! Number of samples
REAL, PARAMETER :: dx = 0.05	! Step size

```

! List of local variables:

```

REAL, DIMENSION(nsamp-1) :: cderiv	! Analytically calculated deriv
REAL, DIMENSION(nsamp-1) :: deriv	! Derivative from subroutine
INTEGER :: error	! Error flag
INTEGER :: i	! Loop index
REAL :: max_error	! Max error in derivative
REAL, DIMENSION(nsamp) :: vector	! f(x)

```

! Calculate f(x)

```

```

DO i = 1, nsamp
    vector(i) = SIN ( REAL(i-1) * dx )
END DO

```

```

! Calculate analytic derivative of f(x)

```

```

DO i = 1, nsamp-1
    cderiv(i) = COS ( REAL(i-1) * dx )
END DO

```

```

! Call "derivative"

```

```

CALL derivative ( vector, deriv, nsamp, dx, error )

```

```

! Find the largest difference between the analytical derivative and
! the result of subroutine "derivative".

```

```

max_error = MAXVAL ( ABS( deriv - cderiv ) )

```

```

! Tell user.

```

```

WRITE (*,1000) max_error
1000 FORMAT (' The maximum error in the derivative is ', F10.4, '.')

```

```

END PROGRAM

```

When this program is run, the results are

```

C:\BOOK\F90\SOLN>test_derivative

```

```

The maximum error in the derivative is      .0250.

```

## Chapter 7. Character Variables

7-4 A character function version of `ucase` is shown below. In order to return a variable-length character string, this function must have an explicit interface, so it is embedded in a module here.

```
MODULE myprocs
CONTAINS
  FUNCTION ucase ( string )
    !
    ! Purpose:
    !   To shift a character string to upper case on any processor,
    !   regardless of collating sequence.
    !
    ! Record of revisions:
    !   Date           Programmer           Description of change
    !   ====           =====
    !   01/09/96      S. J. Chapman        Original code
    !
    IMPLICIT NONE

    ! Declare calling parameters:
    CHARACTER(len=*), INTENT(IN) :: string      ! Input string
    CHARACTER(len=LEN(string)) :: ucase         ! Function

    ! Declare local variables:
    INTEGER :: i                                ! Loop index
    INTEGER :: length                           ! Length of input string

    ! Get length of string
    length = LEN ( string )

    ! Now shift lower case letters to upper case.
    DO i = 1, length
      IF ( LGE(string(i:i), 'a') .AND. LLE(string(i:i), 'z') ) THEN
        ucase(i:i) = ACHAR ( IACHAR ( string(i:i) ) - 32 )
      ELSE
        ucase(i:i) = string(i:i)
      END IF
    END DO

    END FUNCTION ucase
END MODULE
```

A simple test driver program is shown below.

```
PROGRAM test_ucase
!
! Purpose:
!   To test function ucase.
!
! Record of revisions:
!   Date           Programmer           Description of change
```

```

!      ====      =====      =====
!      01/09/96   S. J. Chapman   Original code
!
USE myprocs
IMPLICIT NONE
CHARACTER(len=30) string
WRITE (*,*) 'Enter test string (up to 30 characters): '
READ (*,'(A30)') string
WRITE (*,*) 'The shifted string is: ', ucase(string)
END PROGRAM

```

When this program is executed, the results are:

```

C:\BOOK\F90\SOLN>test_ucase
Enter test string (up to 30 characters):
This is a Test! 12#$6*
The shifted string is: THIS IS A TEST! 12#$6*

```

- 7-11 A subroutine to return the positions of the first and last non-blank characters in a string is shown below. Note that this subroutine is designed to return `ibeg = iend = 1` whenever a string is completely blank. This choice is arbitrary.

```

SUBROUTINE string_limits ( string, ibeg, iend )
!
! Purpose:
!   To determine the limits of the non-blank characters within
!   a character variable. This subroutine returns pointers
!   to the first and last non-blank characters within a string.
!   If the string is completely blank, it will return ibeg =
!   iend = 1, so that any programs using these pointers will
!   not blow up.
!
! Record of revisions:
!   Date      Programmer      Description of change
!   ====      =====      =====
!   01/09/96   S. J. Chapman   Original code
!
IMPLICIT NONE
! List of dummy arguments:
CHARACTER(len=*),INTENT(IN) :: string ! Input string
INTEGER,INTENT(OUT) :: ibeg ! First non-blank character
INTEGER,INTENT(OUT) :: iend ! Last non-blank character
! List of local variables:
INTEGER :: length ! Length of input string
! Get the length of the input string.
length = LEN ( string )
! Look for first character used in string. Use a
! WHILE loop to find the first non-blank character.
ibeg = 0
DO
  ibeg = ibeg + 1
  IF ( ibeg > length ) EXIT

```

```

      IF ( string(ibeg:ibeg) /= ' ' ) EXIT
END DO

! If ibeg > length, the whole string was blank. Set
! ibeg = iend = 1. Otherwise, find the last non-blank
! character.
IF ( ibeg > length ) THEN
  ibeg = 1
  iend = 1
ELSE
  ! Find last nonblank character.
  iend = length + 1
  DO
    iend = iend - 1
    IF ( string(iend:iend) /= ' ' ) EXIT
  END DO
END IF

END SUBROUTINE

```

A test driver program for subroutine `string_limits` is shown below.

```

PROGRAM test_string_limits
!
! Purpose:
!   To test subroutine string_limits.
!
! Record of revisions:
!   Date       Programmer      Description of change
!   ====       =====
!   01/09/96    S. J. Chapman   Original code
!
IMPLICIT NONE

! List of local variables:
CHARACTER(len=30), DIMENSION(3) :: a ! Test strings
INTEGER :: i ! Loop index
INTEGER :: ibeg ! First non-blank char
INTEGER :: iend ! Last non-blank char

! Initialize strings
a(1) = 'How many characters are used?'
a(2) = ' ...and how about this one? '
a(3) = ' ! ! '

! Write results.
DO i = 1, 3
  WRITE (*, '(1X,A,I1,2A)') 'a(', i, ' ' = ', a(i)
  CALL string_limits ( a(i), ibeg, iend )
  WRITE (*, '(1X,A,I3)') 'First non-blank character = ', ibeg
  WRITE (*, '(1X,A,I3)') 'Last non-blank character = ', iend
END DO

END PROGRAM

```

When the program is executed, the results are:

C:\B00K\F90\SOLN>**test\_string\_limits**

```
a(1)                = How many characters are used?
First non-blank character = 1
Last non-blank character = 29

a(2)                = ...and how about this one?
First non-blank character = 4
Last non-blank character = 29

a(3)                = ! !
First non-blank character = 4
Last non-blank character = 8
```

## Chapter 8. Additional Data Types

8-1 “Kinds” are versions of the same basic data type that have differing characteristics. For the real data type, different kinds have different ranges and precisions. A Fortran compiler must support at least two kinds of real data: single precision and double precision.

8-5 (a) These statements are legal. They read ones into the double precision real variable **a** and twos into the single precision real variable **b**. Since the format descriptor is **F18.2**, there will 16 digits to the left of the decimal point. The result printed out by the **WRITE** statement is

```
1. 1111111111111111E+015    2. 222222E+15
```

(b) These statements are illegal. Complex values cannot be compared with the **>** relational operator.

8-10 A subroutine to accept a complex number **C** and calculate its amplitude and phase is shown below:

```
SUBROUTINE complex_2_amp_phase ( c, amp, phase )
!
! Purpose:
! Subroutine to accept a complex number C = RE + i IM and
! return the amplitude "amp" and phase "phase" of the number.
! This subroutine returns the phase in radians.
!
! Record of revisions:
!   Date       Programmer      Description of change
!   ====       =====
!   01/10/97    S. J. Chapman   Original code
!
IMPLICIT NONE

! List of dummy arguments:
COMPLEX, INTENT(IN) :: c           ! Input complex number
REAL, INTENT(OUT) :: amp           ! Amplitude
REAL, INTENT(OUT) :: phase         ! Phase in radians

! Get amplitude and phase.
amp = ABS ( c )
```

```
phase = ATAN2 ( AIMAG(c), REAL(c) )
```

```
END SUBROUTINE
```

A test driver program for this subroutine is shown below:

```
PROGRAM test
!
! Purpose:
!   To test subroutine complex_2_amp_phase, which converts an
!   input complex number into amplitude and phase components.
!
! Record of revisions:
!   Date       Programmer       Description of change
!   ====       =====
!   01/10/97    S. J. Chapman    Original code
!
IMPLICIT NONE

! Local variables:
REAL :: amp                ! Amplitude
COMPLEX :: c               ! Complex number
REAL :: phase              ! Phase

! Get input value.
WRITE (*,'(A)') ' Enter a complex number:'
READ (*,*) c

! Call complex_2_amp_phase
CALL complex_2_amp_phase ( c, amp, phase )

! Tell user.
WRITE (*,'(A,F10.4)') ' Amplitude = ', amp
WRITE (*,'(A,F10.4)') ' Phase      = ', phase

END PROGRAM
```

Some typical results from the test driver program are shown below. The results are obviously correct.

```
C:\B00K\F90\SOLN>test
Enter a complex number:
(1,0)
Amplitude =      1.0000
Phase      =      .0000
```

```
C:\B00K\F90\SOLN>test
Enter a complex number:
(0,1)
Amplitude =      1.0000
Phase      =      1.5708
```

```
C:\B00K\F90\SOLN>test
Enter a complex number:
(-1,0)
Amplitude =      1.0000
Phase      =      3.1416
```

```

C:\BOOK\F90\SOLN>test
Enter a complex number:
(0, -1)
Amplitude =      1.0000
Phase      =     -1.5708

```

8-16 The definitions of “point” and “line” are shown below:

```

! Declare type "point"
TYPE :: point
    REAL :: x           ! x position
    REAL :: y           ! y position
END TYPE point

! Declare type "line"
TYPE :: line
    REAL :: m           ! Slope of line
    REAL :: b           ! Y-axis intercept of line
END TYPE line

```

8-17 A function to calculate the distance between two points is shown below. Note that it is placed in a module to create an explicit interface.

```

MODULE geometry
!
! Purpose:
!   To define the derived data types "point" and "line".
!
! Record of revisions:
!   Date      Programmer      Description of change
!   ====      =====
!   01/11/97   S. J. Chapman   Original code
!
IMPLICIT NONE

! Declare type "point"
TYPE :: point
    REAL :: x           ! x position
    REAL :: y           ! y position
END TYPE point

! Declare type "line"
TYPE :: line
    REAL :: m           ! Slope of line
    REAL :: b           ! Y-axis intercept of line
END TYPE line

CONTAINS

FUNCTION distance(p1, p2)
!
! Purpose:
!   To calculate the distance between two values of type "point".
!
! Record of revisions:

```

```

!      Date      Programmer      Description of change
!      ====      =====
!      01/11/97   S. J. Chapman   Original code
!
IMPLICIT NONE

! List of dummy arguments:
TYPE (point), INTENT(IN) :: p1      ! First point
TYPE (point), INTENT(IN) :: p2      ! Second point
REAL :: distance                    ! Distance between points

! Calculate distance
distance = SQRT ( (p1%x - p2%x)**2 + (p1%y - p2%y)**2 )

END FUNCTION distance

END MODULE geometry

```

A test driver program for this function is:

```

PROGRAM test_distance
!
! Purpose:
!   To test function distance.
!
! Record of revisions:
!      Date      Programmer      Description of change
!      ====      =====
!      01/11/97   S. J. Chapman   Original code
!
USE geometry
IMPLICIT NONE

! Declare variables:
TYPE (point) :: p1, p2          ! Points

WRITE (*,*) 'Enter first point: '
READ (*,*) p1
WRITE (*,*) 'Enter second point: '
READ (*,*) p2
WRITE (*,*) 'The result is: ', distance(p1,p2)

END PROGRAM

```

When this program is executed, the results are.

```

C:\book\f90\SOLN>test_distance
Enter first point:
0 0
Enter second point:
3 4
The result is:      5.000000

C:\book\f90\SOLN>test_distance
Enter first point:
1 -1

```

Enter second point:

1 1

The result is: 2.000000

## Chapter 9. Advanced Features of Procedures and Modules

9-7 The **scope** of an object is the portion of a Fortran program over which it is defined. There are three levels of scope in a Fortran 90/95 program. They are:

1. **Global** — Global objects are objects which are defined throughout an entire program. The names of these objects must be unique within a program. Examples of global objects are the names of programs, external procedures, and modules.

2. **Local** — Local objects are objects which are defined and must be unique within a single **scoping unit**. Examples of scoping units are programs, external procedures, and modules. A local object within a scoping unit must be unique within that unit, but the object name, statement label, etc. may be reused in another scoping unit without causing a conflict. Local variables are examples of objects with local scope.

3. **Statement** — The scope of certain objects may be restricted to a single statement within a program unit. The only examples that we have seen of objects whose scope is restricted to a single statement are the implied **DO** variable in an array constructor and the index variables in a **FORALL** statement.

9-10 (a) This statement is legal. However, *y* and *z* should be initialized before the **CALL** statement, since they correspond to dummy arguments with **INTENT(IN)**. (c) This statement is illegal. Dummy argument **d** is not optional, and is missing in the **CALL** statement. (e) This statement is illegal. Dummy argument **b** is a non-keyword argument after a keyword argument, which is not allowed.

9-12 An interface block is a construct that creates an explicit interface for an external procedure. The interface block specifies all of the interface characteristics of an external procedure. An interface block is created by duplicating the calling argument information of a procedure within the interface. The form of an interface is

```
INTERFACE
    interface_body_1
    interface_body_2
    ...
END INTERFACE
```

Each *interface\_body* consists of the initial **SUBROUTINE** or **FUNCTION** statement of the external procedure, the type specification statements associated with its arguments, and an **END SUBROUTINE** or **END FUNCTION** statement. These statements provide enough information for the compiler to check the consistency of the interface between the calling program and the external procedure.

Alternatively, the *interface\_body* could be a **MODULE PROCEDURE** statement if the procedure is defined in a module.

An interface block would be needed when we want to create an explicit interface for older procedures written in earlier versions of Fortran, or for procedures written in other languages such as C.

9-23 Access to data items and procedures in a module can be controlled using the **PUBLIC** and **PRIVATE** attributes. The **PUBLIC** attribute specifies that an item will be visible outside a module in any program unit that uses the module, while the **PRIVATE** attribute specifies that an item will not be visible to any procedure outside of the module in which the item is defined. These attributes may also be specified in **PUBLIC** and **PRIVATE** statements. By default, all objects defined in a module have the **PUBLIC** attribute.

Access to items defined in a module can be further restricted by using the **ONLY** clause in the **USE** statement to specify the items from a module which are to be used in the program unit containing the **USE** statement.

## Chapter 10. Advanced I/O Concepts

- 10-1 The **ES** format descriptor displays a number in scientific notation, with one significant digit to the left of the decimal place, while the **EN** format descriptor displays a number in engineering notation, with an exponent which is a multiple of 3 and a mantissa between 1.0 and 999.9999. The difference between these two descriptors is illustrated by the following program:

```
PROGRAM test  
WRITE (*, '(1X, ES14. 6, /, 1X, EN14. 6)') 12345. 67, 12345. 67  
END PROGRAM
```

When this simple program is executed, the results are:

```
C: \book\f90\SOLN>test  
1. 234567E+04  
12. 345670E+03
```

- 10-10 Namelist I/O is a convenient way to write out a fixed list of variable names and values, or to read in a fixed list of variable names and values. A namelist is just a list of variable names that are always read or written as a group. A **NAMLIST** I/O statement looks like a formatted I/O statement, except that the **FMT=** clause is replaced by a **NML=** clause. When a namelist-directed **WRITE** statement is executed, the names of all of the variables in the namelist are printed out together with their values in a special order. The first item to be printed is an ampersand (&) followed by the namelist name. Next comes a series of output values in the form "**NAME=**value". Finally, the list is terminated by a slash (/).

When a namelist-directed **READ** statement is executed, the program searches the input file for the marker **&nl\_name**, which indicates the beginning of the namelist. It then reads all of the values in the namelist until a slash character (/) is encountered to terminate the **READ**. The values are assigned to the namelist variables according to the names given in the input list. The namelist **READ** statement does not have to set a value for every variable in the namelist. If some namelist variables are not included in the input file list, then their values will remain unchanged after the namelist **READ** executes.

Namelist-directed **READ** statements are very useful for initializing variables in a program. Suppose that you are writing a program containing 100 input variables. The variables will be initialized to their usual values by default in the program. During any particular run of the program, anywhere from 1 to 10 of these values may need to be changed, but the others would remain at their default values. In this case, you could include all 100 values in a namelist and include a namelist-directed **READ** statement in the program. Whenever a user runs the program, he or she can just list the few values to be changed in the namelist input file, and all of the other input variables will remain unchanged. This approach is much better than using an ordinary **READ** statement, since all 100 values would need to be listed in the ordinary **READ**'s input file, even if they were not being changed during a particular run.

- 10-15 The status of the file is '**UNKNOWN**'. It is a formatted file opened for sequential access, and the location of the file pointer is '**ASIS**', which is processor dependent. It is opened for both reading and writing, with a variable record length. List-directed character strings will be written to the file without delimiters. If the file is not found, the results of the **OPEN** are processor dependent; however, most processors will create a new file and open it. If an error occurs during the open process, the program will abort with a runtime error.
- 10-16 (b) The status of the file is '**REPLACE**'. If the file does not exist, it will be created. If it does exist, it will be deleted and a new file will be created. It is an unformatted file opened for direct access. List-directed i/o does not apply to

unformatted files, so the delimiter clause is meaningless for this file. It is opened for writing only. The length of each record is 80 processor-dependent units. If there is an error in the open process, the program containing this statement will continue, with **ISTAT** set to an appropriate error code.

## Chapter 11. Pointers and Dynamic Data Structures

- 11-2 An ordinary assignment statement assigns a value to a variable. If a pointer is included on the right-hand side of an ordinary assignment statement, then the value used in the calculation is the value stored in the variable pointed to by the pointer. If a pointer is included on the left-hand side of an ordinary assignment statement, then the result of the statement is stored in the variable pointed to by the pointer. By contrast, a pointer assignment statement assigns the *address* of a value to a pointer variable.

In the statement "**a** = **z**", the value contained in variable **z** is stored in the variable *pointed to* by **a**, which is the variable **x**. In the statement "**a** => **z**", the *address* of variable **z** is stored in the pointer **a**.

- 11-7 The statements required to create a 1000-element integer array and then point a pointer at every tenth element within the array are shown below:

```
INTEGER, DIMENSION(1000), TARGET :: my_data = (/ (i, i=1,1000) /)
INTEGER, DIMENSION(:), POINTER :: ptr
ptr => my_data(1:1000:10)
```

- 11-11 This program has several serious flaws. Subroutine **running\_sum** allocates a new variable on pointer **sum** each time that it is called, resulting in a memory leak. In addition, it does not initialize the variable that it creates. Since **sum** points to a different variable each time, it doesn't actually add anything up! A corrected version of this program is shown below:

```
MODULE my_sub
CONTAINS
  SUBROUTINE running_sum (sum, value)
    REAL, POINTER :: sum, value
    IF ( .NOT. ASSOCIATED(sum) ) THEN
      ALLOCATE(sum)
      sum = 0.
    END IF
    sum = sum + value
  END SUBROUTINE running_sum
END MODULE
PROGRAM sum_values
USE my_sub
IMPLICIT NONE
INTEGER :: istat
REAL, POINTER :: sum, value
ALLOCATE (sum, value, STAT=istat)
WRITE (*,*) 'Enter values to add: '
DO
  READ (*,*, IOSTAT=istat) value
  IF ( istat /= 0 ) EXIT
  CALL running_sum (sum, value)
  WRITE (*,*) ' The sum is ', sum
END DO
END PROGRAM
```

When this program is compiled and executed with the Lahey Fortran 90 compiler, the results are:

C:\book\f90\SOLN>**lf90 ex11\_11.f90**

Lahey Fortran 90 Compiler Release 3.00b S/N: F9354220

Copyright (C) 1994-1996 Lahey Computer Systems. All rights reserved.

Copyright (C) 1985-1996 Intel Corp. All rights reserved.

Registered to: Stephen J. Chapman

Options:

-nap	-nbind	-nc	-nchk	-co
-ndal	-ndbl	-ndll	-nf90	-nfix
-ng	-hed	-nin	-inln	-nlisk
-nlst	-nml	-ol	-out ex11_11.exe	
-pca	-sav	-stack 20000h	-stchk	-nswm
-nsyn	-t4	-ntrace	-ntrap	-nvax
-vm	-w	-nwin	-nwo	-wrap
-nxref				

Compiling file ex11\_11.f90.

Compiling program unit MY\_SUB at line 1.

Compiling program unit SUM\_VALUES at line 12.

Encountered 0 errors, 0 warnings in file ex11\_11.f90.

386|LIB: 7.0 -- Copyright (C) 1986-96 Phar Lap Software, Inc.

386|LINK: 8.0\_Lahey2 -- Copyright (C) 1986-96 Phar Lap Software, Inc.

C:\book\f90\SOLN>**ex11\_11**

Enter values to add: 4

The sum is 4.00000 2

The sum is 6.00000 5

The sum is 11.0000 7

The sum is 18.0000 4

The sum is 22.0000 ^D

- 11-17 A function that returns a pointer to the largest value in an input array is shown below. Note that it is contained in a module to produce an explicit interface.

**MODULE subs**

**CONTAINS**

**FUNCTION maxval (array) RESULT (ptr\_maxval)**

!

! Purpose:

! To return a pointer to the maximum value in a  
! rank one array.

!

! Record of revisions:

!	Date	Programmer	Description of change
!	====	=====	=====
!	01/25/97	S. J. Chapman	Original code

```

!
IMPLICIT NONE

! Declare calling arguments:
REAL, DIMENSION(:), TARGET, INTENT(IN) :: array ! Input array
REAL, POINTER :: ptr_maxval ! Pointer to max value

! Declare local variables:
INTEGER :: i ! Index variable
REAL :: max ! Maximum value in array

max = array(1)
ptr_maxval => array(1)
DO i = 2, UBOUND(array, 1)
    IF ( array(i) > max ) THEN
        max = array(i)
        ptr_maxval => array(i)
    END IF
END DO

END FUNCTION maxval
END MODULE

```

A test driver program for this function is shown below:

```

PROGRAM test_maxval
!
! Purpose:
!   To test function maxval.
!
! Record of revisions:
!   Date      Programmer      Description of change
!   ====      =====
!   01/25/97   S. J. Chapman   Original code
!
USE subs
IMPLICIT NONE

! Declare variables
REAL, DIMENSION(6), TARGET :: array = (/ 1., -34., 3., 2., 87., -50. /)
REAL, POINTER :: ptr

! Get pointer to max value in array
ptr => maxval(array)

! Tell user
WRITE (*, '(1X, A, F6.2)') 'The max value is: ', ptr

END PROGRAM

```

When this program is executed, the results are:

```

C:\book\f90\SOLN>test_maxval
The max value is: 87.00

```

## Chapter 12. Introduction to Numerical Methods

- 12-1 Two subroutines to calculate the derivative of a function using the central difference method and the forward difference method are shown below.

```

SUBROUTINE deriv_cen ( f, x0, dx, dfdx, error )
!
! Purpose:
!   To take the derivative of function f(x) at point x0
!   using the central difference method with step size dx.
!   This subroutine expects the function f(x) to be passed
!   as a calling argument.
!
! Record of revisions:
!   Date       Programmer       Description of change
!   ====       =====
!   01/25/97    S. J. Chapman    Original code
!
IMPLICIT NONE

! Declare dummy arguments:
REAL, EXTERNAL :: f           ! Function to differentiate
REAL, INTENT(IN) :: x0        ! Location to take derivative
REAL, INTENT(IN) :: dx        ! Desired step size
REAL, INTENT(OUT) :: dfdx     ! Derivative
INTEGER, INTENT(OUT) :: error ! Error flag: 0 = no error
!                               ! 1 = dx < 0.

! If dx <= 0., this is an error.
IF ( dx < 0. ) THEN
    error = 1
    RETURN

! If dx is specified, then calculate derivative using the
! central difference method and the specified dx.
ELSE IF ( dx > 0. ) THEN
    dfdx = ( f(x0+dx/2.) - f(x0-dx/2.) ) / dx
    error = 0
END IF

END SUBROUTINE

SUBROUTINE deriv_fwd ( f, x0, dx, dfdx, error )
!
! Purpose:
!   To take the derivative of function f(x) at point x0
!   using the forward difference method with step size dx.
!   This subroutine expects the function f(x) to be passed
!   as a calling argument.
!
! Record of revisions:
!   Date       Programmer       Description of change
!   ====       =====
!   01/25/97    S. J. Chapman    Original code

```

```

!
IMPLICIT NONE

! Declare dummy arguments:
REAL, EXTERNAL :: f           ! Function to differentiate
REAL, INTENT(IN) :: x0        ! Location to take derivative
REAL, INTENT(IN) :: dx        ! Desired step size
REAL, INTENT(OUT) :: dfdx     ! Derivative
INTEGER, INTENT(OUT) :: error ! Error flag: 0 = no error
                                !           1 = dx < 0.

! If dx <= 0., this is an error.
IF ( dx < 0. ) THEN
    error = 1
    RETURN

! If dx is specified, then calculate derivative using the
! central difference method and the specified dx.
ELSE IF ( dx > 0. ) THEN
    dfdx = (f(x0+dx) - f(x0) ) / dx
    error = 0
END IF

END SUBROUTINE

```

A test driver program that uses these two routines to calculate the derivative of the function  $f(x) = 1/x$  for the location  $x_0 = 0.15$  is shown below:

```

PROGRAM test_deriv
!
! Purpose:
!   To test subroutines deriv_cen and deriv_fwd by evaluating
!   the derivative of the function  $f(x) = 1. / x$  at
!    $x_0 = 0.15$  for a variety of step sizes. This program
!   compares the performance of the central difference
!   method and the forward difference method.
!
! Record of revisions:
!   Date      Programmer      Description of change
!   ====      =====
!   01/25/97   S. J. Chapman   Original code
!
IMPLICIT NONE

! List of external functions:
REAL, EXTERNAL :: f           ! Function to calculate deriv of

! List of variables:
REAL :: dfdx_cen              ! Derivative of function (central)
REAL :: dfdx_fwd              ! Derivative of function (forward)
REAL :: dx                    ! Step size
INTEGER :: error               ! Error flag
INTEGER :: i                   ! Loop index

! Calculate derivative for 3 step sizes, and tell user.

```

```

DO i = 1, 3
  dx = 1. / 10**I
  CALL deriv_cen ( f, 0.15, dx, dfdx_cen, error )
  CALL deriv_fwd ( f, 0.15, dx, dfdx_fwd, error )
  !
  WRITE (*,1000) dx, dfdx_cen, dfdx_fwd
  1000 FORMAT ( ' dx = ',F5.3,' Central Diff = ',F10.6, &
    ' Forward Diff = ',F10.6)
END DO

END PROGRAM

FUNCTION f(x)
!
! Purpose:
! To evaluate the function f(x) = 1. / x
!
! Record of revisions:
!      Date      Programmer      Description of change
!      ====      =====
!      01/25/97   S. J. Chapman   Original code
!
IMPLICIT NONE

! List of dummy arguments:
REAL, INTENT(IN) :: x      ! Independent variable
REAL :: f                 ! Function result

f = 1. / x

END FUNCTION

```

When this program is executed, the results are shown below. Recall from Chapter 8 that the derivative of the function is actually -44.44444... As you can see, the central difference method does a much better job of estimating the derivative than the forward difference method, regardless of step size.

```

C:\B00K\F90\SOLN>test_deriv
dx = .100 Central Diff = -49.999990 Forward Diff = -26.666660
dx = .010 Central Diff = -44.493820 Forward Diff = -41.666700
dx = .001 Central Diff = -44.444080 Forward Diff = -44.150350

```

## Chapter 13. Fortran Libraries

- 13-1 A *library* is a collection of compiled Fortran object modules organized into a single disk file, which is indexed for easy recovery. When a Fortran program invokes procedures that are located in a library, the linker searches the library to get the object modules for the procedures, and includes them in the final program. The procedures in a library do not have to be recompiled for use, so it is quicker to compile and link programs that use some components from libraries. In addition, libraries can be distributed in object form only, protecting the source code which someone invested so much time writing. Furthermore, libraries which are in the form of modules supply an explicit interface to catch common programming errors.

Libraries may be purchased to perform many special functions, such as mathematical or statistical calculations, graphics, signal processing, etc. They allow users to add features to their programs without having to write all of the specialized code themselves.

- 13-19 A program to calculate the derivative of  $f(x) = \sin x$  with automatic step size selection, comparing the results to the true analytical answer.

```

PROGRAM calc_derivative_1
!
! Purpose:
!   To calculate the derivative of the function sin(x) at 101
!   equally-spaced points between 0 and 2*pi using automatic
!   step sizes, comparing the results with the analytical
!   solution.
!
! Record of revisions:
!   Date       Programmer       Description of change
!   ====       =====
!   01/31/97   S. J. Chapman    Original code
!
USE booklib
IMPLICIT NONE

! List of named constants:
REAL, PARAMETER :: pi = 3.141593      ! Pi

! External function:
REAL, EXTERNAL :: fun                 ! Function to differentiate

! List of variables:
REAL :: dx = 2*pi / 100.              ! Step size
INTEGER :: error                      ! Error flag
INTEGER :: i                          ! Loop index
REAL :: step = 0.0                   ! Step size
REAL :: x0                           ! Point to take derivative
REAL, DIMENSION(101) :: y            ! y data points
REAL, DIMENSION(101) :: y_true       ! Analytical solution

! Calculate data points.
DO i = 1, 101
    x0 = (i-1) * dx
    step = 0.
    CALL deriv ( fun, x0, step, y(i), error )
    y_true(i) = cos(x0)
END DO

! Compare the resulting data.
WRITE (*,1000)
1000 FORMAT (' A comparison of the true answer versus the calculation', &
            ' with auto step sizes:')
WRITE (*,1010)
1010 FORMAT (T10, ' x ', T21, 'True', T32, ' Auto ' &
            /T10, '===', T21, '====', T32, '=====')

DO i = 1, 101

```

```

      x0 = (i-1) * dx
      WRITE (*,1020) x0, y_true(i), y(i)
      1020 FORMAT (3X,3F12.6)
END DO

```

```

END PROGRAM

```

```

FUNCTION fun(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
REAL :: fun

```

```

! Evaluate function.
fun = sin(x)

```

```

END FUNCTION

```

When this program is executed, the results are:

C:\BOOK\f90\SOLN>**calc\_derivative1**

A comparison of the true answer versus the calculation with auto step sizes:

x	True	Auto
===	=====	=====
.000000	1.000000	1.000000
.062832	.998027	.998030
.125664	.992115	.992106
.188496	.982287	.982269
.251327	.968583	.968565
.314159	.951057	.951057
...		
...		
3.015929	-.992115	-.991700
3.078761	-.998027	-.997610
3.141593	-1.000000	-.999582
3.204425	-.998027	-.997610
3.267256	-.992115	-.991700
3.330088	-.982287	-.981877
3.392920	-.968583	-.968179
3.455752	-.951056	-.950659
3.518584	-.929776	-.929388
3.581416	-.904827	-.904449
3.644248	-.876307	-.875941
3.707079	-.844328	-.843975
3.769911	-.809017	-.808679
3.832743	-.770513	-.770191
...		
...		
6.031858	.968583	.968969
6.094690	.982287	.982680
6.157522	.992115	.992513
6.220354	.998027	.998429
6.283185	1.000000	1.000404