

Answers to Selected Exercises for *Introduction to Fortran 90/95* 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}
- 1-3 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} .

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 apostrophes to represent an apostrophe 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 90/95 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 `ires = 21`. (b) This outer loop is executed 4 times, the inner loop is executed 3 times, and afterwards `ires = 43`.

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 (a) The result is printed out at the top of a new page. The numeric field will be displayed with 5 numbers, since the number of digits is specified in the format descriptor. The result is:

```

i = -00123
-----|-----|
          10      20

```

- (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

```

- (c) The result is printed out on the next line. It is:

```

Result =      F
-----|-----|-----|-----|-----|-----|-----|-----|
          10      20      30      40      50      60      70      80

```

- 4-11 There are many possible **FORMAT** statements that could perform the specified functions. One possible correct answer is shown here, but there are many others. Note in (b) that there are 7 significant digits, since one is before the decimal point.

```

(a) 1000 FORMAT ('1', T41, 'INPUT DATA')
(b) 1010 FORMAT ('0', 5X, I5, 4X, ES12.6)

```

- 4-13 The program to convert time in seconds since the beginning of the day into 24-hour **HH:MM SS** format is shown below:

```

PROGRAM hhmss
!
! Purpose:
!   To convert a time in seconds since the start of the day
!   into HH:MM SS format, using the 24 hour convention.
!
! Record of revisions:
!   Date      Programmer      Description of change
!   ====      =====
!   01/04/97  S. J. Chapman      Original code
!

```

```
IMPLICIT NONE
```

```
! List of named constants:
```

```
REAL :: sec_per_hour = 3600.    ! Seconds per hour  
REAL :: sec_per_minute = 60.    ! Seconds per minute
```

```
! List of variables
```

```
INTEGER :: hour                ! Number of hours  
INTEGER :: minute              ! Number of minutes  
INTEGER :: sec                 ! Remaining seconds  
REAL :: remain                 ! Remaining seconds  
REAL :: seconds                ! Seconds since start of day
```

```
WRITE (*,*) 'Enter the number of seconds since the start of day: '
```

```
READ (*,*) seconds
```

```
! Check for a valid number of seconds.
```

```
IF ( ( seconds < 0. ) .OR. ( seconds > 86400. ) ) THEN
```

```
! Tell user and quit.
```

```
WRITE (*,100) seconds
```

```
100 FORMAT (1X, 'Invalid time entered: ', F16.3, '/', &  
           1X, 'Time must be 0.0 <= seconds <= 86400.0 ')
```

```
ELSE
```

```
! Time ok. Calculate the number of hours, and the number of  
! seconds left over after the hours are calculated.
```

```
hour = INT ( seconds / sec_per_hour )  
remain = seconds - REAL (hour) * sec_per_hour
```

```
! Calculate the number of minutes left, and the number of  
! seconds left over after the hours are calculated.
```

```
minute = INT ( remain / sec_per_minute )  
remain = remain - REAL (minute) * sec_per_minute
```

```
! Get number of seconds left.
```

```
sec = NINT ( remain )
```

```
! Write out result.
```

```
WRITE (*,110) seconds, hour, minute, sec  
110 FORMAT (1X, F7.1, ' seconds = ', I2, ':', I2.2, ':', I2.2)
```

```
END IF
```

```
END PROGRAM
```

When the program is tested, the results are:

```
C:\BOOK\F90\SOLN>hhmss
```

```
Enter the number of seconds since the start of day:
```

```
1202
```

```
1202.0 seconds = 0:20:02
```

```
C:\BOOK\F90\SOLN>hhmss
```

Enter the number of seconds since the start of day:

30000

30000.0 seconds = 8:20:00

C:\BOOK\F90\SOLN>hhmmss

Enter the number of seconds since the start of day:

100000

Invalid time entered: 100000.000

Time must be 0.0 <= seconds <= 86400.0

- 4-15 Input files should be opened with `STATUS = 'OLD'` because the input data file must already exist and contain data. Output files may have one of two possible statuses. If we want to ensure that previous data is not overwritten, then the output file should be opened with `STATUS = 'NEW'`. If we don't care whether or not old data is overwritten, then it should be opened with `STATUS = 'REPLACE'`. A temporary file should be opened with `STATUS = 'SCRATCH'`.

Chapter 5. Arrays

- 5-2 An *array* is a group of variables, all of the same type, that are referred to by a single name, and that notionally occupy consecutive positions in the computer's memory. A *array element* is a single variable within the array; it is addressed by naming the array with a subscript. For example, if `array` is a ten-element array, then `array(2)` is the second array element in the array.
- 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.
- 5-8 (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

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

- 5-23 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.
- 6-4 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
```

6-8 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.

6-11 If we examine the ASCII character set shown in Appendix A, we can notice certain patterns. One is that the upper case letters 'A' through 'Z' are in consecutive sequence with no gaps, and the lower case letters 'a' through 'z' are in consecutive sequence with no gaps. Furthermore, each lower case letter is exactly 32 characters above the corresponding upper case letter. Therefore, the strategy to convert lower case letters to upper case without affecting any other characters in the string is:

1. First, determine if a character is between 'a' and 'z'. If it is, it is lower case.
2. If it is lower case, get its collating sequence and subtract 32. Then convert the new sequence number back into a character.
3. If the character is not lower case, just skip it!

```
SUBROUTINE ucase(string)
!
! Purpose:
!   To shift a character string to upper case.
!
! Record of revisions:
!   Date       Programmer       Description of change
!   =====
!   01/06/96   S. J. Chapman     Original code
!
IMPLICIT NONE

! Declare dummy arguments:
CHARACTER(len=*), INTENT(INOUT) :: string ! String to shift

! Declare local variables:
INTEGER :: i ! Loop index

! Shift lower case letters to upper case.
DO i = 1, LEN(string)
  IF ( string(i:i) >= 'a' .AND. string(i:i) <= 'z' ) THEN
    string(i:i) = ACHAR ( IACHAR ( string(i:i) ) - 32 )
  END IF
END DO
```

END SUBROUTINE ucase

6-25 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. Additional Data Types

7-1 “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.

7-5 (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 be 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.

7-9 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:\BOOK\F90\SOLN>test
Enter a complex number:
(1, 0)
Amplitude =      1.0000
Phase      =      .0000

```

```

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

```

```

C:\BOOK\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

```

7-14 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

```

7-15 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 8. Advanced Features of Procedures and Modules

8-4 Variables *x*, *y*, *i*, and *j* are declared in the main program, and variables *x* and *i* are re-declared in the internal function. Therefore, variables *y* and *j* are the same in both the main program and the internal function, while variables *x* and *i* are different in the two places. Initially, the values of the variables are *x* = 12.0, *y* = -3.0, *i* = 6, and

$j = 4$. In the call to function `exec`, the value of `y` is passed to dummy variable `x`, and the value of `i` is passed to dummy variable `i`, so the values of the variables are `x = -3.0`, `y = -3.0`, `i = 6`, and `j = 4`. Then `j` is set to 6 in the function, changing its value both in the function and the main program. After the function is executed, the values of the variables are `x = 12.0`, `y = -3.0`, `i = 6`, and `j = 6`.

```
C:\book\f90\SOLN>ex8_4
Before call: x, y, i, j = 12.0 -3.0 6 4
In exec:     x, y, i, j = -3.0 -3.0 6 4
The result is -6.000000E-01
After call:  x, y, i, j = 12.0 -3.0 6 6
```

8-7 (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.

8-9 A generic procedure is a procedure that is designed to work with more than one type of input and output arguments. Fortran has many built-in generic procedures, such as `SIN()`, `COS()`, `TAN()`, etc. Fortran 90/95 permits a programmer to create user-defined generic procedures using a generic interface block. The general form of a generic interface block is

```
INTERFACE generic_name
  specific_interface_body_1
  specific_interface_body_2
  ...
END INTERFACE
```

Each *specific_interface_body* in the generic interface block is a `MODULE PROCEDURE` statement if the procedure resides in a module. Each procedure in the block must be *unambiguously* distinguished from the others by the type and characteristics of its dummy arguments.

8-18 (a) These statements are illegal. Constant `pi` is declared in the module, but all data items are private, so `pi` is not available to be used in the main program. (b) These statements are illegal. Constant `two_pi` is declared in the module, and is re-declared in the main program. This double declaration is illegal.

Chapter 9. Dynamic Memory Allocation and Pointers

9-3 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`.

9-7 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)
```