Instructor's Manual

to accompany

Chapman

Fortran 95/2003 for Scientists and Engineers

Third Edition

Stephen J. Chapman
BAE SYSTEMS Australia

Instructor's Manual to accompany *Fortran 95/2003 for Scientists and Engineers*, Third Edition, ISBN 978-0-07-319157-7

TABLE OF CONTENTS

PREFACE

TO THE INSTRUCTOR

This Instructor's Manual intended to accompany the second edition of *Fortran 95/2003 for Scientists and Engineers*. It contains solutions to every end-of-chapter exercise in the book.

The first edition of *Fortran 95/2003 for Scientists and Engineers* was conceived as a result of my experience writing and maintaining large Fortran programs in both the defense and geophysical fields. During my time in industry, it became obvious that the strategies and techniques required to write large, *maintainable* Fortran programs were quite different from what new engineers were learning in their Fortran programming classes at school. The incredible cost of maintaining and modifying large programs once they are placed into service absolutely demands that they be written to be easily understood and modified by people other than their original programmers. My goal for this book is to teach simultaneously both the fundamentals of the Fortran language and a programming style that results in good, maintainable programs. In addition, it is intended to serve as a reference for graduates working in industry.

It is quite difficult to teach undergraduates the importance of taking extra effort during the early stages of the program design process in order to make their programs more maintainable. Class programming assignments must by their very nature be simple enough for one person to complete in a short period of time, and they do not have to be maintained for years. Because the projects are simple, a student can often "wing it" and still produce working code. A student can take a course, perform all of the programming assignments, pass all of the tests, and still not learn the habits that are really needed when working on large projects in industry.

From the very beginning, this book teaches Fortran in a style suitable for use on large projects. It emphasizes the importance of going through a detailed design process before any code is written, using a top-down design technique to break the program up into logical portions that can be implemented separately. It stresses the use of procedures to implement those individual portions, and the importance of unit testing before the procedures are combined into a finished product. Finally, it emphasizes the importance of exhaustively testing the finished program with many different input data sets before it is released for use.

In addition, this book teaches Fortran as it is actually encountered by engineers and scientists working in industry and in laboratories. One fact of life is common in all programming environments: large amounts of old legacy code that have to be maintained. The legacy code at a particular site may have been originally written in Fortran IV (or an even earlier version!), and it may use programming constructs that are no longer common today. For example, such code may use arithmetic `IF` statements, or computed or assigned `GO TO` statements. Chapter 17 is devoted to those older features of the language which are no longer commonly used, but which are encountered in legacy code. The chapter emphasizes that these features should *never* be used in a new program, but also prepares the student to handle them when he or she encounters them. Students must be able to recognize and work with this code when they encounter it. On the other hand, we do *not* want students using these features in new programs, so all such older features are clearly labeled as undesirable. In addition, there are no end-of-chapter problems for Chapter 16—we don't want them to get used to using poor features.

CHANGES IN THIS EDITION

This edition build directly on the success of *Fortran 95/2003 for Scientists and Engineers*, 2/e. It preserves the structure of the previous edition, while weaving the new Fortran 2003 material throughout the text. Most of the material in this book applies to both Fortran 95 and Fortran 2003. Topics that are unique to Fortran 2003 are printed in a shaded background.

Most of the additions in Fortran 2003 are logical extensions of existing capabilities in Fortran 95, and they are integrated into the text in the proper chapters. However, the object-oriented programming capabilities of Fortran 2003 are completely new, and a new Chapter 16 has been created to cover that material.

The vast majority of Fortran courses are limited to one quarter or one semester, and the student is expected to pick up both the basics of the Fortran language and the concept of how to program. Such a course would cover Chapters 1 through 7 of this text, plus selected topics in Chapters 8 and 9 if there is time. This provides a good foundation for students to build on in their own time as they use the language in practical projects.

Advanced students and practicing scientists and engineers will need the material on `COMPLEX` numbers, derived data types, and pointers found in Chapters 11 through 15. Practicing scientists and engineers will almost certainly need the material on obsolete, redundant, and deleted Fortran features found in Chapter 17. These materials are rarely taught in the classroom, but they are included here to make the book a useful reference text when the language is actually used to solve real-world problems.

FEATURES OF THIS BOOK

Many features of this book are designed to emphasize the proper way to write reliable Fortran programs. These features should serve a student well as he or she is first learning Fortran, and should also be useful to the practitioner on the job. They include:

1. Emphasis on Modern Fortran 95/2003

The book consistently teaches the best current practice in all of its examples. Many Fortran 95/2003 features duplicate and supersede older features of the Fortran language. In those cases, the proper usage of the modern language is presented. Examples of older usage are largely relegated to Chapter 17, where their old / undesirable nature is emphasized. Examples of Fortran 95/2003 features that supersede older features are the use of modules to share data instead of `COMMON` blocks, the use of `DO … END DO` loops instead of `DO … CONTINUE` loops, the use of internal procedures instead of statement functions, and the use of `CASE` constructs instead of Computed `GOTO`s.

2. Emphasis on Strong Typing

The `IMPLICIT NONE` statement is used consistently throughout the book to force the explicit typing of every variable used in every program, and to catch common typographical errors at compilation time. In conjunction with the explicit declaration of every variable in a program, the book emphasizes the importance of creating a data dictionary that describes the purpose of each variable in a program unit.

3. Emphasis on Top-Down Design Methodology

The book introduces a top-down design methodology in Chapter 3, and then uses it consistently throughout the rest of the book. This methodology encourages a student to think about the proper design of a program *before* beginning to code. It emphasizes the importance of clearly defining the problem to be solved and the required inputs and outputs before any other work is begun. Once the problem is properly defined, it teaches the student to employ stepwise refinement to break the task down into successively smaller sub-tasks, and to implement the subtasks as separate subroutines or functions. Finally, it teaches the importance of testing at all stages of the process, both unit testing of the component routines and exhaustive testing of the final product. Several examples are given of programs that work properly for some data sets, and then fail for others.

The formal design process taught by the book may be summarized as follows:

1. *Clearly state the problem that you are trying to solve.*

2. *Define the inputs required by the program and the outputs to be produced by the program.*

3. *Describe the algorithm that you intend to implement in the program*. This step involves top-down design and stepwise decomposition, using pseudocode or flow charts.

4. *Turn the algorithm into Fortran statements*.

5. *Test the Fortran program*. This step includes unit testing of specific subprograms, and also exhaustive testing of the final program with many different data sets.

4. Emphasis on Procedures

The book emphasizes the use of subroutines and functions to logically decompose tasks into smaller subtasks. It teaches the advantages of procedures for data hiding. It also emphasizes the importance of unit testing procedures before they are combined into the final program. In addition, the book teaches about the common mistakes made with procedures, and how to avoid them (argument type mismatches, array length mismatches, etc.). It emphasizes the advantages associated with explicit interfaces to procedures, which allow the Fortran compiler to catch most common programming errors at compilation time.

5. Emphasis on Portability and Standard Fortran 95/2003.

The book stresses the importance of writing portable Fortran code, so that a program can easily be moved from one type of computer to another one. It teaches students to use only standard Fortran 95 statements in their programs, so that they will be as portable as possible. In addition, it teaches the use of features such as the `SELECTED_REAL_KIND` function to avoid precision and kind differences when moving from computer to computer, and the `ACHAR` and `IACHAR` functions to avoid problems when moving from ASCII to EBCDIC computers.

6. Good Programming Practice Boxes

These boxes highlight good programming practices when they are introduced for the convenience of the student. In addition, the good programming practices introduced in a chapter are summarized at the end of the chapter.

7. Programming Pitfalls Boxes

These boxes highlight common errors so that they can be avoided.

8. Emphasis on Pointers and Dynamic Data Structures

Chapter 15 contains a detailed discussion of Fortran pointers, including possible problems resulting from the incorrect use of pointers such as memory leaks and pointers to deallocated memory. Examples of dynamic data structures in the chapter include linked lists and binary trees.

Chapter 16 contains a discussion of Fortran objects and object-oriented programming, including the use of dynamic pointers to achieve polymorphic behavior.

9. Use of Sidebars

A number of sidebars are scattered throughout the book. These sidebars provide additional information of potential interest to the student. Some sidebars are historical in nature. For example, one sidebar in Chapter 1 describes the IBM Model 704, the first computer to ever run Fortran. Other sidebars reinforce lessons from the main text. For example, Chapter 9 contains a sidebar reviewing and summarizing the many different types of arrays found in Fortran 95/2003.

10. Completeness

Finally, the book endeavors to be a complete reference to the Fortran 95/2003 language, so that a practitioner can locate any required information quickly. Special attention has been paid to the index to make features easy to find. A special effort has also been made to cover such obscure and little understood features as passing procedure names by reference, and defaulting values in list-directed input statements.

PEDAGOGICAL FEATURES

The book includes several features designed to aid student comprehension. Each chapter begins with a list of the objectives that should be achieved in that chapter. A total of 26 quizzes appear scattered throughout the chapters, with answers to all questions included in Appendix E. These quizzes can serve as a useful self-test of comprehension. In addition, there are approximately 340 end-of-chapter exercises. Answers to selected exercises are available at the book's Web site, and of course answers to all exercises are included in the Instructor's Manual. Good programming practices are highlighted in all chapters with special Good Programming Practice boxes, and common errors are highlighted in Programming Pitfalls boxes. End of chapter materials include Summaries of Good Programming Practice and Summaries of Fortran Statements and Structures. Finally, a detailed description of every Fortran 95/2003 intrinsic procedure is included in Appendix B, and an extensive Glossary is included in Appendix D.

The book is accompanied by an Instructor's Manual, containing the solutions to all end-of-chapter exercises. Instructors can also download the solutions in the Instructor's Manual from the book's Web site. The source code for all examples in the book, plus other supplemental materials, can be downloaded by anyone from the book's Web site.

POSSIBLE SEQUENCE OF TOPICS FOR A ONE SEMESTER COURSE

This book contains much more information than can be covered in a typical one-semester introduction to Fortran course. The exact material covered by each instructor will vary depending on the goals of his or her course. As much as possible, the later chapters of the book have been structured so that they may be covered in any desired order. You may take advantage of this feature to select topics that meet the needs of your particular students. However, I do believe that Chapters 1 through 9 should be covered in the order included in the book. Each of them contains material that builds directly on the contents of the preceding chapters. All chapters after Chapter 9 are essentially independent, and may be selected in any desired order, except that Chapter 16 on object-oriented programming is dependent on the discussions of bound data types and operators in Chapters 12 and 13.

In my own classroom, I teach Chapters 1 through 9 consecutively, and then skip to selected topics in Chapters 11, 13, and 15. I find that this sequence fills an ambitious one-semester course. It also fulfills my ambition to introduce students to the full richness of the language, including derived data types, user-defined operators, and pointers.

A BRIEF NOTE ABOUT FORTRAN COMPILERS

Two Fortran 95 compilers were used during the preparation of this book: Intel Visual Fortran 9.1 and NAGWare Fortran 95 version 5.1. Both of these compilers have selected Fortran 2003 extensions. However, at the time of this writing, only the NAGWare Fortran compiler supports the object-oriented features of Fortran 2003. References to all three compiler vendors may be found at this book's World Wide Web site.

NOTE: At the current state of Fortran 2003 compiler development in May 2007, the exercises in Chapter 16 are not compiling properly. I will be releasing solutions to the problems in that chapter as soon as the next generation of compilers is released.

A FINAL NOTE TO THE INSTRUCTOR

No matter how hard I try to proofread a document like this book, it is inevitable that some typographical errors will slip through and appear in print. If you should spot any such errors, please drop me a note via the publisher, and I will do my best to get them eliminated from subsequent printings and editions. Thank you very much for your help in this matter.

I will maintain a complete list of errata and corrections at the book's World Wide Web site, which is http://www.mcgraw-hillengineeringcs.com. Please check that site for any updates and / or corrections.

Stephen J. Chapman
Melbourne, Australia
1 June 2007

Chapter 1. Introduction to Computers and the Fortran Language

1-1 *(a)* 1010_2 *(b)* 100000_2 *(c)* 1001101_2 *(d)* 111111_2

1-2 *(a)* 72_{10} *(b)* 137_{10} *(b)* 255_{10} *(d)* 5_{10}

1-3 *(a)* 127361_8 and $AEF1_{16}$ *(b)* 512_8 and $14A_{16}$ *(c)* 157_8 and $6F_{16}$ *(d)* 3755_8 and $7ED_{16}$

1-4 *(a)* 11111111_2 and 255_{10} *(b)* 110101000_2 and 424_{10} *(c)* 1001001_2 and 73_{10} *(d)* 111111111_2 and 511_{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}.

1-6 46-bit integer: From -2^{45} to -2^{45}-1, or -35,184,372,088,832 to 35,184,372,088,831. 64-bit integer: From -2^{63} to -2^{63}-1, or -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

1-7 *(a)* 110111_2 *(b)* 1111111111111011_2 *(c)* 10000000000_2 *(d)* 1111110000000000_2

1-8 The sum of the two's complement numbers is:

$$= \quad 9362_{10}$$

$$\underline{1111110011111100_2} \quad = \quad \underline{-772}_{10}$$

$$0010000110001110_2 \quad = \quad 8590_{10}$$

The two answers agree with each other.

1-9 $01111111_2 = 127_{10}$, and $10000000_2 = -128_{10}$. These results agree with Equations (1-1) and (1-2).

1-10 A 53-bit mantissa can represent approximately $\pm 2^{52}$ numbers, or about fifteen significant decimal digits. An 11-bit exponent can represent multipliers between 2^{-1023} and 2^{1023}, so the range of double precision numbers is from about 10^{-307} to 10^{307}.

Chapter 2. Basic Elements of Fortran

2-1 *(a)* Valid real constant *(b)* Valid character constant *(c)* Invalid constant—numbers may not include commas *(d)* Invalid constant—real numbers must include a decimal point[1] *(e)* Invalid constant—need two apostrophes to represent an apostrophe within a string *(f)* Invalid character constant—mismatched apostrophe and quotation mark *(g)* Valid character constant

2-2 *(a)* Different values—one is real and the other is integer *(b)* Different values *(c)* The same value *(d)* The same value.

2-3 *(a)* Valid *(b)* Invalid—name must begin with a letter *(c)* Invalid—question mark is not a legal character *(d)* Valid.

2-4 *(a)* Legal: result = 0.888889 *(b)* Legal: result = 30 *(c)* Illegal—cannot have two adjacent operators *(d)* Legal: result = 0.002 *(e)* Illegal—division by zero

2-5 *(a)* Legal: This expression is evaluated entirely with integer arithmetic: 58/4 = 14, and 4/58 = 0, so the result = 0. *(b)* Legal: The first part of this expression is evaluated with integer arithmetic and the second part of the expression is evaluated with real arithmetic. The final expression is evaluated with real arithmetic: 58/4 = 14, and 4/58. = 0.06896552, so the result = 0.9655172. *(c)* This expression is evaluated entirely with real arithmetic: 58./4 = 14.5, and 4/58. = 0.06896552, so the result = 1.000000. *(d)* Illegal: parentheses are unbalanced.

2-6 *(a)* 12 *(b)* 12 *(c)* 0 *(d)* 15.6 *(e)* 12.0 *(f)* 12 *(g)* 18 *(h)* 18 *(i)* 12

2-7 *(a)* 19683 *(b)* 729 *(c)* 19683

2-8 `i1` = 2, `i2` = -4, `i3` = -5, `i4` = -5, `a1` = 2.4, `a2` = 5.76

2-9 The program will run, but it will produce wrong answers, because the sine and cosine functions expect their arguments to have units of radians, not degrees.

2-10 The output of the program is:

```
      -3.141592       100.000000       200.000000           300          -100          -200
```

2-11 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
!      ====       ==========          =====================
!   05/01/2007   S. J. Chapman        Original code
!
```

[1] Many compilers will accept this form, even though it does not meet the strict definition of a real constant.

```
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 get_pay
```

The result of executing this program is

```
C:\book\f95_2003\soln>get_pay
Enter employees pay rate in dollars per hour:
7.90
Enter number of hours worked:
42
Employee's pay is $     331.800000
```

2-12 Assume that `energy`, `mass`, `grav`, `height`, and `velocity` are all real quantities. Then the total energy of an object in the Earth's gravitational field is given by the equation

```
energy = mass * grav * height + 0.5 * mass * velocity**2
```

2-13 Assume that `grav`, `height`, and `velocity` are all real quantities. Then the velocity of a ball when it hits the Earth is given by the equation

```
velocity = SQRT( 2.0 * grav * height )
```

2-14 A program to calculate the impact velocity of a ball dropped from a specified height is shown below.

```
PROGRAM calc_vel
!
!  Purpose:
!    To calculate the velocity of a ball when it hits the
!    Earth.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/01/2007   S. J. Chapman       Original code
!
IMPLICIT NONE

! List of constants
```

```
REAL,PARAMETER :: G = 9.81     ! Acc due to gravity (m/s**2)

! List of variables:
REAL :: height       ! Initial height of ball (m)
REAL :: vel          ! Velocity at impact (m/s)

! Get the height in meters
WRITE (*,*) 'Enter height in meters:'
READ (*,*) height

! Get velocity
vel = SQRT(2 * G * height)

! Write out answer
WRITE (*,*) 'The velocity at impact is ', vel, ' m/s.'

END PROGRAM calc_vel
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex2_14>calc_vel
 Enter height in meters:
1
 The velocity at impact is    4.429447      m/s.

C:\book\f95_2003\soln\ex2_14>calc_vel
 Enter height in meters:
10
 The velocity at impact is    14.00714      m/s.

C:\book\f95_2003\soln\ex2_14>calc_vel
 Enter height in meters:
100
 The velocity at impact is    44.29447      m/s.
```

2-15 To calculate the energy in joules, we must multiply the total time in seconds times the power supplied in watts. Therefore, the program must convert one year into seconds, and multiply it by 500,000,000 W. Then, the mass consumed is

$$m = \frac{E}{c^2}$$

The resulting program is shown below.

```
PROGRAM calc_mass
!
!  Purpose:
!    To calculate the mass converted to energy by a nuclear
!    generating station.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!   05/01/2007   S. J. Chapman        Original code
!
IMPLICIT NONE
```

```
! List of constants
REAL,PARAMETER :: C = 2.9979E8    ! Speed of light (m/s)

! List of variables:
REAL :: energy       ! Energy consumed over the period, in joules.
REAL :: mass         ! Mass consumed, in kilograms
REAL :: power        ! Power supplied in watts
REAL :: time         ! Time that power is supplied, in seconds.

! Get power produced, in watts
power = 400.0E6

! Get time in seconds.  This is a year converted to seconds:
! time = (365 days) * (24 hr/day) * (60 min/hr) * ( 60 s/min)
time = 365.0 * 24.0 * 60.0 * 60.0

! Get energy consumed in joules
energy = power * time

! Calculate mass
mass = energy / C**2

! Write out answer
WRITE (*,*) 'The mass consumed is ', mass, ' kg.'

END PROGRAM calc_mass
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln>calc_mass
 The mass consumed is   0.1403564      kg.
```

2-16 A modified version of the previous problem that allows the user to specify the output power level in MW and the duration in months is shown below. Note that we have converted months to seconds assuming that there are 30 days in a month.

```
PROGRAM calc_mass
!
!  Purpose:
!    To calculate the mass converted to energy by a nuclear
!    generating station.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/01/2007   S. J. Chapman        Original code
! 1. 05/01/2007   S. J. Chapman        Modified for user inputs
!
IMPLICIT NONE

! List of constants
REAL,PARAMETER :: C = 2.9979E8    ! Speed of light (m/s)

! List of variables:
REAL :: energy       ! Energy consumed over the period, in joules.
```

```
REAL :: mass         ! Mass consumed, in kilograms
REAL :: power        ! Power supplied in watts
REAL :: time         ! Time that power is supplied, in seconds.
REAL :: time_months  ! Time that power is supplied, in months.

! Get power produced, in watts
WRITE (*,*) 'Enter the output power of the station, in MW:'
READ (*,*) power
power = power * 1.0E6        ! Convert to watts

! Get the number of months of operation
WRITE (*,*) 'Enter the operating time, in months:'
READ (*,*) time_months

! Get time in seconds.  This is a month converted to seconds,
! assuming 30 days in a month:
! time = months * (30 days) * (24 hr/day) * (60 min/hr) * (60 s/min)
time = time_months * 30.0 * 24.0 * 60.0 * 60.0

! Get energy consumed in joules
energy = power * time

! Calculate mass
mass = energy / C**2

! Write out answer
WRITE (*,*) 'The mass consumed is ', mass, ' kg.'

END PROGRAM calc_mass
```

2-17 A program to calculate the period of a pendulum is shown below:

```
PROGRAM pendulum
!
!  Purpose:
!    To calculate the period of a pendulum in seconds, given
!    its length in meters.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!   05/02/2007   S. J. Chapman        Original code
!
IMPLICIT NONE

! List of constants:
REAL, PARAMETER :: GRAV = 9.81      ! 9.81 m/sec**2
REAL, PARAMETER :: PI = 3.141493    ! Pi

! List of variables:
REAL :: length        ! Length of pendulum, in meters
REAL :: period        ! Period, in seconds

! Get length of pendulum
WRITE (*,*) 'Enter the length of the pendulum in meters: '
READ (*,*) length
```

```
! Calculate period of the pendulum.
period = 2.0 * PI * SQRT ( length / GRAVv )

! Write out results.
WRITE (*,*) 'The period of the pendulum in seconds is: ', period

END PROGRAM pendulum
```

2-18 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
!      ====       ==========          =====================
!    05/02/2007  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 calc_hypotenuse
```

2-19 A program to calculate the logarithm of a number to the base *b* is shown below:

```
PROGRAM calc_log
!
!  Purpose:
!    To calculate the logarithm of a number to an
!    arbitrary base b.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
```

```
!    05/02/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of variables:
REAL :: base         ! Base of logarithm
REAL :: log_res      ! Resulting logarithm
REAL :: x            ! Input value

! Get the number to convert
WRITE (*,*) 'Enter the number to take LOG of: '
READ (*,*) x
WRITE (*,*) 'Enter the base of the logarithm: '
READ (*,*) base

! Calculate logarithm.
log_res = LOG10(x) / LOG10(base)

! Write out results.
WRITE (*,*) 'The logarithm is: ', log_res

END PROGRAM calc_log
```

To test this program, we will calculate the logarithm of 100 to the base *e*. Note that the value of *e* can be found from the function `EXP(1.0)` — it is 2.718282.

```
C:\book\f95_2003\soln>calc_log
 Enter the number to take LOG of:
100
 Enter the base of the logarithm:
2.718282
 The logarithm is:    4.605170
```

Using a calculator or by calling function `LOG(100)`, we can show that this value is correct.

2-20 This solution to this problem is computer and compiler dependent. You instructor will have to provide you with the correct result for your particular combination of computer and compiler.

2-21 A program to calculate the distance between two points on a Cartesian plane is shown below:

```
PROGRAM calc_distance
!
!  Purpose:
!    To calculate the distance between two points (X1,Y1)
!    and (X2,Y2) on a Cartesian coordinate plane.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/02/2007  S. J. Chapman        Original code
!
IMPLICIT NONE


! List of variables:
REAL :: distance       ! Distance between points.
```

```
REAL :: x1              ! x position of point 1
REAL :: x2              ! x position of point 2
REAL :: y1              ! y position of point 1
REAL :: y2              ! y position of point 2

! Get positions of points 1 and 2.
WRITE (*,*) 'Program to calculate the distance between two points'
WRITE (*,*) '(x1,y1) and (x2,y2) on a Cartesian coordinate plane.'
WRITE (*,*) 'Enter the position (x1,y1) of point 1:'
READ (*,*) x1, y1
WRITE (*,*) 'Enter the position (x2,y2) of point 2:'
READ (*,*) x2, y2

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

! Write out results.
WRITE (*,*) 'The distance between the points is: ', distance

END PROGRAM calc_distance
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln>calc_distance
Program to calculate the distance between two points
 (x1,y1) and (x2,y2) on a Cartesian coordinate plane.
 Enter the position (x1,y1) of point 1:
-1 1
 Enter the position (x2,y2) of point 2:
6 2
 The distance between the points is:    7.071068
```

2-22 One possible program is shown below:

```
PROGRAM calc_db
!
!  Purpose:
!    To calculate the power of a signal in dB referenced
!    to 1 milliwatt.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/02/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of constants:
REAL, PARAMETER :: P_REF = 0.001     ! Input power in mW

! List of variables:
REAL :: dbmw                   ! Power in dBmW
REAL :: power                  ! Input power in watts

! Get input power in watts.
WRITE (*,*) 'Program to power in dB referenced to 1 mW, given'
```

```
WRITE (*,*) 'an input power in watts.  Enter input power: '
READ (*,*) power

! Calculate distance between the points.
dbmw = 10.0 * LOG10 ( power / P_REF )

! Write out results.
WRITE (*,*) 'The power in dB(mW) is: ', dbmw

END PROGRAM calc_db
```

2-23 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
!      ====       ==========          =====================
!    05/02/2007  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 coshx
```

When this program is run, the result is:

```
C:\book\f95_2003\soln>coshx
Enter number to calculate cosh() of:
3.0
COSH(X) =      10.067660
```

The Fortran 95/2005 intrinsic function `COSH()` produces the same answer.

2-24 A program to calculate the resonant frequency of the radio is shown below:

```
PROGRAM calc_future_value
!
!  Purpose:
!    To calculate the future value of a sum of money held in
!    an account for a specified period of years at a specified
!    interest rate.
!
```

```
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/02/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of variables:
REAL :: apr                ! Annual percentage rate (%)
REAL :: future             ! Future value ($)
INTEGER :: m               ! Number of times compounded per year
INTEGER :: n               ! Number of years
REAL :: principal          ! Principal value ($)

! Get input data
WRITE (*,*) 'This program calculates the future value of an '
WRITE (*,*) 'account help for a specified number of years at '
WRITE (*,*) 'a specified interest rate. '

WRITE (*,*) 'Enter the initial value of the account: '
READ (*,*) principal

WRITE (*,*) 'Enter the annual percentage rate (%): '
READ (*,*) apr

WRITE (*,*) 'Enter the number of times per year that the interest '
WRITE (*,*) 'is compounded: '
READ (*,*) m

WRITE (*,*) 'Enter the number of years that the account is held: '
READ (*,*) n

! Calculate the future value
future = principal * (1 + (apr/(100*m))) ** (m*n)

! Tell the user
WRITE (*,*) 'The future value is $', future

END PROGRAM calc_future_value
```

When this program is run, the result is:

```
C:\book\f95_2003\soln\ex2_24>calc_future_value
 This program calculates the future value of an
 account help for a specified number of years at
 a specified interest rate.
 Enter the initial value of the account:
1000.00
 Enter the annual percentage rate (%):
5
 Enter the number of times per year that the interest
 is compounded:
1
 Enter the number of years that the account is held:
1
 The future value is $   1050.000
```

```
C:\book\f95_2003\soln\ex2_24>calc_future_value
 This program calculates the future value of an
 account help for a specified number of years at
 a specified interest rate.
 Enter the initial value of the account:
1000.00
 Enter the annual percentage rate (%):
5
 Enter the number of times per year that the interest
 is compounded:
2
 Enter the number of years that the account is held:
1
 The future value is $   1050.625

C:\book\f95_2003\soln\ex2_24>calc_future_value
 This program calculates the future value of an
 account help for a specified number of years at
 a specified interest rate.
 Enter the initial value of the account:
1000.00
 Enter the annual percentage rate (%):
5
 Enter the number of times per year that the interest
 is compounded:
12
 Enter the number of years that the account is held:
1
 The future value is $   1051.1630
```

The rate of compounding is not very important over a period of one year, but it makes a significant difference over a period of 10 years. Try the program and see.

2-25 A program to calculate the resonant frequency of the radio is shown below:

```
PROGRAM resonant_freq
!
!  Purpose:
!    To calculate the resonant frequency of a radio receiver.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/02/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of constants:
REAL, PARAMETER :: PI = 3.141593   ! Pi

! List of variables:
REAL :: capacitance        ! Capacitance (farads)
REAL :: freq               ! frequency (Hz)
REAL :: inductance         ! Inductance (henrys)
```

```
! Get input data
WRITE (*,*) 'Enter capacitance of circuit in farads: '
READ (*,*) capacitance
WRITE (*,*) 'Enter inductance of circuit in henrys: '
READ (*,*) inductance

! Calculate resonant frequency
freq = 1. / ( 2. * PI * SQRT (inductance * capacitance) )

! Write result
WRITE (*,*) 'The resonant frequency is ', freq, ' Hz.'

END PROGRAM resonant_freq
```

When this program is run, the result is:

```
C:\book\f95_2003\soln>resonant_freq
Enter capacitance of circuit in farads:
0.25E-9
Enter inductance of circuit in henrys:
0.1E-3
The resonant frequency is   1006584.     Hz.
```

2-26 A program to calculate the turning radius of an aircraft, given a velocity specified in mach numbers and a lateral acceleration specified in g's, is shown below:

```
PROGRAM turning_radius
!
!  Purpose:
!    To calculate the turning radius of an aircraft with a
!    given speed and lateral acceleration.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/02/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of constants:
REAL, PARAMETER :: G = 9.81        ! Accel of gravity (m/s**2)
REAL, PARAMETER :: MACH1 = 340.    ! Mach 1 (m/s)

! List of variables:
REAL :: acc                ! Lateral acceleration (m/s**2)
REAL :: radius             ! Turning radius (m)
REAL :: vel                ! Velocity (mach)

! Get input data
WRITE (*,*) 'Enter aircraft speed in Mach numbers: '
READ (*,*) vel
WRITE (*,*) 'Enter lateral acceleration (g): '
READ (*,*) acc

! Calculate turning radius
radius = (vel * MACH1)**2 / (acc * G)
```

```
! Write result
WRITE (*,*) 'The turning radius is ', radius, ' m.'

END PROGRAM turning_radius
```

We can use this program to answer the questions asked in the problem:

```
C:\book\f95_2003\soln\ex2_26>turning_radius
 Enter aircraft speed in Mach numbers:
0.80
 Enter lateral acceleration (g):
2.5
 The turning radius is    3016.677      m.

C:\book\f95_2003\soln\ex2_26>turning_radius
 Enter aircraft speed in Mach numbers:
1.50
 Enter lateral acceleration (g):
2.5
 The turning radius is    10605.50      m.

C:\book\f95_2003\soln\ex2_26>turning_radius
 Enter aircraft speed in Mach numbers:
1.50
 Enter lateral acceleration (g):
7
 The turning radius is    3787.680      m.
```

2-27 A program to calculate the escape velocity from a body with a given mass and radius is shown below. **Note:** There is an error in the equation on the first printing of the book, which will be corrected in later printings. The correct equation is:

$$v_{\text{esc}} = \sqrt{\frac{2GM}{R}}$$

```
PROGRAM escape_velocity
!
!  Purpose:
!    To calculate the turning escape velocity from a body
!    with a given mass and radius.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/02/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of constants:
REAL, PARAMETER :: G = 6.673E-11   ! Gravitational constant

! List of variables:
REAL :: mass                ! Mass of body (kg)
REAL :: radius              ! Radius of body (m)
```

```
REAL :: v_esc              ! Escape velocity (m/s)

! Get input data
WRITE (*,*) 'Enter the mass of the body in kg: '
READ (*,*) mass
WRITE (*,*) 'Enter the radius of the body in meters: '
READ (*,*) radius

! Calculate escape velocity
v_esc = SQRT( 2.0 * G * mass / radius )

! Write result
WRITE (*,*) 'The escape velocity ', v_esc, ' m/s.'

END PROGRAM escape_velocity
```

When this program is run, the result is:

```
C:\book\f95_2003\soln>escape_velocity
 Enter the mass of the body in kg:
6.0e24
 Enter the radius of the body in meters:
6.4e6
 The escape velocity    11185.65      m/s.

C:\book\f95_2003\soln>escape_velocity
 Enter the mass of the body in kg:
7.4e22
 Enter the radius of the body in meters:
1.7e6
 The escape velocity    2410.277      m/s.

C:\book\f95_2003\soln>escape_velocity
 Enter the mass of the body in kg:
8.7e20
 Enter the radius of the body in meters:
4.7e5
 The escape velocity    497.0342      m/s.

C:\book\f95_2003\soln>escape_velocity
 Enter the mass of the body in kg:
1.9e27
 Enter the radius of the body in meters:
7.1e7
 The escape velocity    59761.73      m/s.
```

Chapter 3. Program Design and Branching Structures

3-1 *(a)* Legal: result = `.TRUE.` *(b)* Legal: result = `.FALSE.` *(c)* Legal: result = `.FALSE.` *(d)* Illegal—relational operators require numerical data *(e)* Legal: result = `.TRUE.` *(f)* Legal: result = `.TRUE.` *(g)* Illegal—combinational logical operators require logical data, while 17.5 is numeric

3-2 The statements to evaluate tan θ are:

```
PROGRAM tan_theta
!
!  Purpose:
!    To calculate the tangent of an angle specified in degrees.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/03/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of constants
REAL, PARAMETER :: DEGREES_2_RADIANS = 0.01745329

! List of variables
REAL :: costh        ! Cosine of theta
REAL :: tanth        ! Tangent of theta
REAL :: theta        ! Angle in degrees

WRITE (*,*) 'Enter angle in degrees: '
READ (*,*) theta

! Calculate tan(theta)
costh = COS ( theta * DEGREES_2_RADIANS )
IF ( ABS(costh) > 1.0E-20 ) THEN
   tanth = SIN ( theta * DEGREES_2_RADIANS ) / costh
   WRITE (*,*) 'tan(theta) = ', tanth
ELSE
   WRITE (*,*) 'Error: cosine(theta) too small.'
END IF

END PROGRAM tan_theta
```

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

```
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
```

3-4 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
```

3-5 A program to calculate the cost of expressing a package of a given weight is shown below:

```
PROGRAM send
!
!  Purpose:
!    To calculate the cost of sending an express package.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/03/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of variables:
REAL :: cost          ! Cost to send package
REAL :: round_up      ! Weight rounded up to next pound
REAL :: weight        ! Weight of package

! Get package weight in pounds.
WRITE (*,*) 'Enter package weight in pounds: '
READ (*,*) weight

! Calculate postage.  First check for overweight packages.
IF ( WEIGHT > 100.0 ) THEN

   ! Too heavy.
   WRITE (*,*) 'Error -- Package weight > 100 lbs:', weight
   WRITE (*,*) 'Package cannot be sent.'

ELSE
```

```
   ! Calculate weight.  Note that "round_up" is required
   ! to make sure that we treat fractional weights properly.

   ! Base cost...
   cost = 12.00

   ! Charge for weight > 2 lbs...
   IF ( weight > 2.0 ) THEN
      round_up = CEILING(weight)
      cost = cost + 4.00 * ( round_up - 2.0 )
   END IF

   ! Excess weight charge for weight > 70 lbs...
   IF ( weight > 70.0 ) THEN
      cost = cost + 10.00
   END IF

   ! Tell user what total cost is.
   WRITE (*,*) 'Total cost = ', cost

END IF

END PROGRAM send
```

3-6 This code fragment is correct.

3-7 The modified program is shown below.

```
PROGRAM funxy
!
!  Purpose:
!    This program solves the function f(x,y) for a user-specified x and y,
!    where f(x,y) is defined as:
!                 _
!                |
!                |   X + Y              X >= 0 and Y >= 0
!                |   X + Y**2           X >= 0 and Y < 0
!       F(X,Y) = |   X**2 + Y           X < 0  and Y >= 0
!                |   X**2 + Y**2        X < 0  and Y < 0
!                |_
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    11/06/2006  S. J. Chapman        Original code
! 1. 05/03/2007  S. J. Chapman        Modified for nested IFs
!
IMPLICIT NONE

! Declare the variables used in this program.
REAL :: x                 ! First independent variable
REAL :: y                 ! Second independent variable
REAL :: fun               ! Resulting function

! Prompt the user for the values x and y
WRITE (*,*) 'Enter the coefficients x and y: '
```

```
READ  (*,*) x, y

! Write the coefficients of x and y.
WRITE (*,*) 'The coefficients x and y are: ', x, y

!  Calculate the function f(x,y) based upon the signs of x and y.
outer: IF ( x >= 0. ) THEN
   x_pos: IF ( y >= 0. ) THEN
      fun = x + y
   ELSE
      fun = x + y**2
   END IF x_pos
ELSE outer
   x_neg: IF ( y >= 0. ) THEN
      fun = x**2 + y
   ELSE
      fun = x**2 + y**2
   END IF x_neg
END IF outer

! Write the value of the function.
WRITE (*,*) 'The value of the function is: ', fun

END PROGRAM funxy
```

3-8 The code fragment below shows a CASE construct using character case selector, used to specify the processing to perform for each possible elective choice.

```
CHARACTER(len=12) :: choice

! Get user's elective choice
WRITE (*,*) 'Enter elective choice: '
READ (*,*) choice

! Process choice
SELECT CASE ( choice )
CASE( 'Englsh')

   (Process English selection...)

CASE( 'History')

   (Process History selection...)

CASE ( 'Astronomy' )

   (Process Astronomy selection...)

CASE ( 'Literature' )

   (Process Literature selection...)

CASE DEFAULT
   WRITE (*,*) 'Invalid choice entered!'
END CASE
```

3-9 A program to calculate Australian income tax is shown below:

```
PROGRAM income_tax
!
!  Purpose:
!    To calculate the income tax owed by a person working
!    in Austrlia.
!
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/03/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! Declare the variables used in this program.
REAL :: income           ! Total income in dollars
REAL :: tax              ! Tax

! Prompt the user for the total income
WRITE (*,*) 'Enter total income in dollars: '
READ  (*,*) income

! Calculate basic tax
IF ( income <= 6000.00 ) THEN
   tax = 0.
ELSE IF ( income <= 20000.00 ) THEN
   tax = 0.17 * (income - 6000.00)
ELSE IF ( income <= 50000.0 ) THEN
   tax = 0.30 * (income - 20000.00) + 2380.00
ELSE IF ( income <= 60000.0 ) THEN
   tax = 0.42 * (income - 50000.00) + 11380.00
ELSE
   tax = 0.47 * (income - 60000.00) + 15580.00
END IF

! Add medicare levy
tax = tax + 0.015 * income

! Write the tax owed.
WRITE (*,*) 'The tax owed is $', tax

END PROGRAM income_tax
```

3-10 A program to convert a value from any one of the four currencies to any other currency is shown below. This program works by first converting the input currency into US dollars, and then converting the US dollars into the specified output currency.

```
PROGRAM convert_value
!
!  Purpose:
!    To convert a an amount in one currency into an amount
!    in another currency.
!
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/03/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! Declare conversion rates to the US dollar
REAL,PARAMETER :: AUD_TO_USD = 0.74      ! Conversion rate AUD to USD
REAL,PARAMETER :: EURO_TO_USD = 1.21     ! Conversion rate EURO to USD
REAL,PARAMETER :: UK_TO_USD = 1.78       ! Conversion rate UK to USD

! Declare the variables used in this program.
INTEGER :: input_currency  ! Input currency (1=AUD;2=USD;3=EURO;4=UK)
REAL :: input_value        ! Total income in input currency
INTEGER :: output_currency ! Output currency (1=AUD;2=USD;3=EURO;4=UK)
REAL :: output_value       ! Total income in output currenct
REAL :: usd_value          ! Value in USD

! Prompt the user for the type of the input currency
WRITE (*,*) 'Specify input currency (1=AUD;2=USD;3=EURO;4=UK):'
READ  (*,*) input_currency

! Prompt for the input value, and convert to USD
WRITE (*,*) 'Enter amount:'
READ  (*,*) input_value

! First calculate the amount in USD
IF ( input_currency == 1 ) THEN
   usd_value = input_value * AUD_TO_USD
ELSE IF ( input_currency == 2 ) THEN
   usd_value = input_value
ELSE IF ( input_currency == 3 ) THEN
   usd_value = input_value * EURO_TO_USD
ELSE IF ( input_currency == 4 ) THEN
   usd_value = input_value * UK_TO_USD
ELSE
   WRITE (*,*) 'Illegal input currency type'
END IF

! Prompt the user for the type of the output currency
WRITE (*,*) 'Specify output currency (1=AUD;2=USD;3=EURO;4=UK):'
READ  (*,*) output_currency

! Now calculate amount in output currency
IF ( output_currency == 1 ) THEN
   output_value = usd_value / AUD_TO_USD
   WRITE (*,*) 'The amount in AUD = ', output_value
ELSE IF ( output_currency == 2 ) THEN
```

```
   output_value = usd_value
   WRITE (*,*) 'The amount in USD = ', output_value
ELSE IF ( output_currency == 3 ) THEN
   output_value = usd_value / EURO_TO_USD
   WRITE (*,*) 'The amount in EURO = ', output_value
ELSE IF ( output_currency == 4 ) THEN
   output_value = usd_value / UK_TO_USD
   WRITE (*,*) 'The amount in UK pounds = ', output_value
ELSE
   WRITE (*,*) 'Illegal output currency type'
END IF

END PROGRAM convert_value
```

3-11 The program to calculate the power level in dBmw, while trapping illegal values, is shown below:

```
PROGRAM calc_db
!
!  Purpose:
!    To calculate the power of a signal in dB referenced
!    to 1 milliwatt.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/02/2007  S. J. Chapman        Original code
! 1. 05/05/2007  S. J. Chapman        Modified to trap illegal values
!
IMPLICIT NONE

! List of constants:
REAL, PARAMETER :: P_REF = 0.001     ! Input power in mW

! List of variables:
REAL :: dbmw                    ! Power in dBmW
REAL :: power                   ! Input power in watts

! Get input power in watts.
WRITE (*,*) 'Program to power in dB referenced to 1 mW, given'
WRITE (*,*) 'an input power in watts.  Enter input power: '
READ (*,*) power

IF ( power > 0. ) THEN

   ! Calculate distance between the points.
   dbmw = 10.0 * LOG10 ( power / P_REF )

   ! Write out results.
   WRITE (*,*) 'The power in dB(mW) is: ', dbmw

ELSE

   ! Illegal value
   WRITE (*,*) 'Illegal value--the power must be > 0.0'

END IF
```

```
END PROGRAM calc_db
```

When this program is run, the result is

```
C:\book\f95_2003\soln>calc_db
 Program to power in dB referenced to 1 mW, given
 an input power in watts.  Enter input power:
-5
 Illegal value--the power must be > 0.0

C:\book\f95_2003\soln>calc_db
 Program to power in dB referenced to 1 mW, given
 an input power in watts.  Enter input power:
5
 The power in dB(mW) is:    36.98970
```

3-12 A program to calculate the angle of incidence θ_2 in medium 2 is shown below:

```
PROGRAM refraction
!
!  Purpose:
!    To calculate the angle of incidence at which a light ray
!    is refracted when passing from a medium with index of
!    refraction n1 into a medium with index of refraction n2.
!    The light ray is assumed to have an angle of incidence
!    theta1 in the first medium, and the program calculates
!    the angle of incidence theta2 in the second medium.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/03/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of named constants:
REAL, PARAMETER :: DEG_2_RAD = 0.01745329 ! Deg to radians

! List of variables:
REAL :: arg               ! Argument of the ASIN function
REAL :: n1                ! Index of refraction in medium 1
REAL :: n2                ! Index of refraction in medium 2
REAL :: theta1            ! Angle of incidence in medium 1
REAL :: theta2            ! Angle of incidence in medium 2

! Prompt user for the index of refraction of medium 1
WRITE (*,*) 'Enter index of refraction N1 of medium 1: '
READ (*,*) n1

!Prompt user for the index of refraction of medium 1
WRITE (*,*) 'Enter index of refraction N2 of medium 2: '
READ (*,*) n2

! Prompt user for the angle of incidence in medium 1
WRITE (*,*) 'Enter angle of incidence in medium 1 (degrees): '
```

```
READ (*,*) theta1

! Convert theta1 to radians.
theta1 = theta1 * DEG_2_RAD

! Calculate the argument of the arcsin function
arg = ( n1 / n2 ) * SIN( theta1 )

! Check for total reflection.
IF ( ABS(arg) > 1.0 ) THEN

   ! Tell user.
   WRITE (*,*) 'This light ray is totally reflected.'

ELSE

   ! Get theta2 in degrees.
   theta2 = ASIN ( arg ) / DEG_2_RAD

   ! Tell user.
   WRITE (*,*) 'The angle of incidence in medium 2 is',  &
               theta2, ' degrees.'

END IF

END PROGRAM refraction
```

When this program is run with the two test data sets, the results are:

```
C:\book\f95_2003\soln>refraction
Enter index of refraction N1 of medium 1:
1.0
Enter index of refraction N2 of medium 2:
1.7
Enter angle of incidence in medium 1 (degrees):
45.
The angle of incidence in medium 2 is      24.578850 degrees.

C:\book\f95_2003\soln>refraction
Enter index of refraction N1 of medium 1:
1.7
Enter index of refraction N2 of medium 2:
1.0
Enter angle of incidence in medium 1 (degrees):
45.
This light ray is totally reflected.
```

Chapter 4. Loops and Character Manipulation

4-1 *(a)* Legal: result = `.FALSE.` *(b)* Illegal—comparison of character string to an integer *(c)* Legal: result = `'A_H'` *(d)* Legal: result = `'o'`

4-2 The statements required to calculate and print out the squares of all even numbers between 0 and 50 are:

```
INTEGER :: i

DO i = 0, 50, 2
   WRITE (*,*) i, i**2
END DO
```

4-3 A program to evaluate the function $y(x) = x^2 - 3x + 2$ is shown below:

```
PROGRAM eval_function
!
!  Purpose:
!    To evaluate the function y(x) = x**2-3*x+2 for all values of
!    x between -1.0 and 3.0, in steps of 0.1.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/04/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of variables:
INTEGER :: i          ! Loop index
REAL :: x             ! Independent variable
REAL :: y             ! Dependent variable

! Calculate and print out Y(X).
DO i = -10, 30
   x = 0.1 * REAL(i)
   y = x**2 - 3.0 * x + 2.0
   WRITE (*,*) 'x = ', x, '   y(x) = ', y
END DO

END PROGRAM eval_function
```

4-4 The Fortran statements required to calculate *y(t)* from the equation

$$y(t) = \begin{cases} -3t^2 + 5 & t \geq 0 \\ 3t^2 + 5 & t < 0 \end{cases}$$

is shown below:

```
IF ( t >= 0 ) THEN
   y = -3 * t**2 + 5
ELSE
   y = 3 * t**2 + 5
END IF
```

4-5 A program to calculate the factorial function is shown below.

```
PROGRAM factorial
!
!  Purpose:
!    To evaluate the factorial function N! for N >= 0.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/04/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of variables:
INTEGER :: fact        ! factorial (N!)
INTEGER :: i           ! Loop index
INTEGER :: n           ! Input value

! Get number to calculate factorial of.
WRITE (*,*) 'Enter number to calculate factorial of: '
READ (*,*) n

! Calculate factorial.  First check for valid n.
error_check: IF ( n < 0 ) THEN

   ! Error.  Tell user and do not calculate.
   WRITE (*,*) 'Error -- N is < 0: ', n

ELSE

   ! N is valid.  Calculate N!.
   calc: IF ( n == 0 ) THEN
      ! Calculate 0!.
      fact = 1

   ELSE

      ! General case.
      fact = 1
      DO i = n, 1, -1
         fact = fact * i
      END DO
   END IF calc

   ! Write result.
   WRITE (*,*) n, '! = ', fact
END IF error_check
```

```
END PROGRAM factorial
```

The `IF ( n == 0 )` clause in this program is not actually needed. Why not?

4-6 When the `CYCLE` statement is executed, control returns directly to the top of the loop, the loop index is incremented, and the loop is executed again if the loop control parameters are still satisfied. When the `EXIT` statement is executed, the execution of the loop stops immediately, and control transfers to the first executable statement after the `END DO` statement.

4-7 The program `stats_2` modified to use a `DO WHILE` loop is:

```
PROGRAM stats_2
!
!  Purpose:
!    To calculate mean and the standard deviation of an input
!    data set containing an arbitrary number of input values.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    11/10/05    S. J. Chapman        Original code
! 1. 11/12/05    S. J. Chapman        Correct divide-by-0 error if
!                                     0 or 1 input values given.
! 2. 05/03/07    S. J. Chapman        Modified to use DO WHILE
!
IMPLICIT NONE

! Declare the variables used in this program.
INTEGER :: n =0      ! The number of input samples.
REAL :: std_dev = 0. ! The standard deviation of the input samples.
REAL :: sum_x = 0.   ! The sum of the input values.
REAL :: sum_x2 = 0.  ! The sum of the squares of the input values.
REAL :: x = 0.       ! An input data value.
REAL :: x_bar        ! The average of the input samples.

! Get first value.
WRITE (*,*) 'Enter number: '
READ  (*,*) x
WRITE (*,*) 'The number is ', x

! While Loop to read input values.
DO WHILE ( x >= 0. )

   ! Accumulate sums
   n      = n + 1
   sum_x  = sum_x + x
   sum_x2 = sum_x2 + x**2

   ! Read in next value
   WRITE (*,*) 'Enter number: '
   READ  (*,*) x
   WRITE (*,*) 'The number is ', x

END DO

! Check to see if we have enough input data.
```

```
IF ( n < 2 ) THEN ! Insufficient information

   WRITE (*,*) 'At least 2 values must be entered!'

ELSE ! There is enough information, so
     ! calculate the mean and standard deviation

   x_bar = sum_x / real(n)
   std_dev = sqrt( (real(n) * sum_x2 - sum_x**2) / (real(n)*real(n-1)))

   ! Tell user.
   WRITE (*,*) 'The mean of this data set is:', x_bar
   WRITE (*,*) 'The standard deviation is:   ', std_dev
   WRITE (*,*) 'The number of data points is:', n

END IF

END PROGRAM stats_2
```

4-8 *(a)* 65536 *(b)* 10 *(c)* 1 *(d)* 0 *(e)* 3 *(f)* 6 *(g)* This statement loops an indefinite number of times until some condition is true and an `EXIT` statement is executed.

4-9 *(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. *(c)* This outer loop is executed 4 times, the inner loop is executed 13 times, and afterwards `ires` = 42. *(d)* This outer loop is executed 1 time, the inner loop is executed 4 times, and afterwards `ires` = 12.

4-10 *(a)* This loop is executed 10 times, and afterwards `ires` = 10. *(b)* This loop is executed 3 times, and afterwards `ires` = 256. *(c)* This loop is *never* executed, since `ires` ≤ 200 the first time that the loop test is evaluated.

4-11 The modified program is shown below:

```
PROGRAM ball
!
!  Purpose:
!    To calculate distance traveled by a ball thrown at a specified
!    angle THETA and at a specified velocity VO from a point on the
!    surface of the earth, ignoring the effects of air friction and
!    the earth's curvature.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!     11/14/05    S. J. Chapman        Original code
!  1. 05/04/07    S. J. Chapman        Modified for variable gravity
!
IMPLICIT NONE

!  Declare parameters
REAL, PARAMETER :: DEGREES_2_RAD = 0.01745329 ! Deg ==> rad conv.

! Declare variables
REAL :: gravity          ! Accel. due to gravity (m/s)
INTEGER :: max_degrees   ! angle at which the max rng occurs (degrees)
REAL :: max_range        ! Maximum range for the ball at vel v0 (meters)
REAL :: range            ! Range of the ball at a particular angle (meters)
```

```
REAL :: radian          ! Angle at which the ball was thrown (in radians)
INTEGER :: theta        ! Angle at which the ball was thrown (in degrees)
REAL :: v0              ! Velocity of the ball (in m/s)

! Initialize variables.
max_range = 0.
max_degrees = 0
v0 = 20.

! Get gravitational acceleration
WRITE (*,*) 'Enter gravitational acceleration, in m/sec**2: '
READ (*,*) gravity

! Loop over all specified angles.

loop: DO theta = 0, 90

   ! Get angle in radians
   radian = REAL(theta) * DEGREES_2_RAD

   ! Calculate range in meters.
   range = (-2. * v0**2 / gravity) * sin(radian) * cos(radian)

   ! Write out the range for this angle.
     WRITE (*,*) 'Theta = ', theta, ' degrees; Range = ', range, &
                 ' meters'

   ! Compare the range to the previous maximum range.  If this
   ! range is larger, save it and the angle at which it occurred.
   IF ( range > max_range ) THEN
      max_range = range
      max_degrees = theta
   END IF

END DO loop

! Skip a line, and then write out the maximum range and the angle
! at which it occurred.
WRITE (*,*) ' '
WRITE (*,*) 'Max range = ', max_range, ' at ', max_degrees, ' degrees'

END PROGRAM ball
```

The maximum range and optimum angle θ are shown as a function of the acceleration due to gravity *g* in the table below. Note that the maximum range increases as the gravitational acceleration decreases, but the optimum angle is unaffected by the value of *g*.

g (m/s^2)	Max Range (m)	Max Angle θ
-9.8	40.816	45°
-9.7	41.237	45°
-9.6	41.667	45°

4-12 The modified program is shown below:

```
PROGRAM ball
```

```
!
!  Purpose:
!    To calculate distance traveled by a ball thrown at a specified
!    angle THETA and at a specified velocity VO from a point on the
!    surface of the earth, ignoring the effects of air friction and
!    the earth's curvature.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!     11/14/05    S. J. Chapman        Original code
!  1. 05/04/07    S. J. Chapman        Modified for variable gravity
!
IMPLICIT NONE

!  Declare parameters
REAL, PARAMETER :: DEGREES_2_RAD = 0.01745329 ! Deg ==> rad conv.
REAL, PARAMETER :: GRAVITY = -9.81            ! Accel. due to gravity (m/s)

! Declare variables
INTEGER :: max_degrees  ! angle at which the max rng occurs (degrees)
REAL :: max_range       ! Maximum range for the ball at vel v0 (meters)
REAL :: range           ! Range of the ball at a particular angle (meters)
REAL :: radian          ! Angle at which the ball was thrown (in radians)
INTEGER :: theta        ! Angle at which the ball was thrown (in degrees)
REAL :: v0              ! Velocity of the ball (in m/s)

! Initialize variables.
max_range = 0.
max_degrees = 0

! Get initial velocity
WRITE (*,*) 'Enter initial velocity, in m/sec: '
READ (*,*) v0

! Loop over all specified angles.

loop: DO theta = 0, 90

   ! Get angle in radians
   radian = real(theta) * DEGREES_2_RAD

   ! Calculate range in meters.
   range = (-2. * v0**2 / gravity) * sin(radian) * cos(radian)

   ! Write out the range for this angle.
     WRITE (*,*) 'Theta = ', theta, ' degrees; Range = ', range, &
                 ' meters'

   ! Compare the range to the previous maximum range.  If this
   ! range is larger, save it and the angle at which it occurred.
   IF ( range > max_range ) THEN
      max_range = range
      max_degrees = theta
   END IF
```

```
END DO loop

! Skip a line, and then write out the maximum range and the angle
! at which it occurred.
WRITE (*,*) ' '
WRITE (*,*) 'Max range = ', max_range, ' at ', max_degrees, ' degrees'

END PROGRAM ball
```

The maximum range and optimum angle θ are shown as a function of the initial velocity v_o in the table below. Note that the maximum range increases as the square of the initial velocity v_o, but the optimum angle is unaffected by the value of v_o.

v_o (m/s)	Max Range (m)	Max Angle θ
10	10.194	45°
20	40.775	45°
30	91.943	45°

4-13 The modified program to validate the input dates to the day-of-year program:

```
PROGRAM doy

!  Purpose:
!    This program calculates the day of year corresponding to a
!    specified date.  It illustrates the use CASE construct.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    11/13/05    S. J. Chapman        Original code
! 1. 05/03/07    S. J. Chapman        Modified to verify date
!
IMPLICIT NONE

! Declare the variables used in this program
INTEGER :: day         ! Day (dd)
INTEGER :: day_of_year ! Day of year
INTEGER :: i           ! Index variable
INTEGER :: leap_day    ! Extra day for leap year
INTEGER :: month       ! Month (mm)
LOGICAL :: valid       ! Valid date flag
INTEGER :: year        ! Year (yyyy)

! Get day, month, and year to convert
WRITE (*,*) 'This program calculates the day of year given the '
WRITE (*,*) 'current date.  Enter current month (1-12), day(1-31),'
WRITE (*,*) 'and year in that order:  '
READ (*,*) month, day, year

! Validate the year entered.
valid = .TRUE.
IF ( year <= 0 ) THEN
   valid = .FALSE.
END IF
```

```
! Validate the month entered.
IF ( month < 1 .OR. month > 12 ) THEN
   valid = .FALSE.
END IF

! Check for leap year, and add extra day if necessary
IF ( MOD(year,400) == 0 ) THEN
   leap_day = 1           ! Years divisible by 400 are leap years
ELSE IF ( MOD(year,100) == 0 ) THEN
   leap_day = 0           ! Other centuries are not leap years
ELSE IF ( MOD(year,4) == 0 ) THEN
   leap_day = 1           ! Otherwise every 4th year is a leap year
ELSE
   leap_day = 0           ! Other years are not leap years
END IF

! Validate the day entered, considering leap year status.
SELECT CASE (month)
CASE (1,3,5,7,8,10,12)
   IF ( day < 0 .OR. day > 31 ) THEN
      valid = .FALSE.
   END IF
CASE (4,6,9,11)
   IF ( day < 0 .OR. day > 30 ) THEN
      valid = .FALSE.
   END IF
CASE (2)
   IF ( day < 0 .OR. day > 28+leap_day ) THEN
      valid = .FALSE.
   END IF
END SELECT

! Is the date valid?  If so, calculate doy.  Otherwise,
! tell of invalid date ahd quit.
IF ( valid ) THEN

   ! Calculate day of year
   day_of_year = day
   DO i = 1, month-1

      ! Add days in months from January to last month
      SELECT CASE (i)
      CASE (1,3,5,7,8,10,12)
         day_of_year = day_of_year + 31
      CASE (4,6,9,11)
         day_of_year = day_of_year + 30
      CASE (2)
         day_of_year = day_of_year + 28 + leap_day
      END SELECT

   END DO

   ! Tell user
   WRITE (*,*) 'Day         = ', day
   WRITE (*,*) 'Month       = ', month
```

```
   WRITE (*,*) 'Year        = ', year
   WRITE (*,*) 'day of year = ', day_of_year

ELSE

   ! Invalid date entered.
   WRITE (*,*) 'Invalid date entered!'

END IF
```

4-14 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
!       ====       ==========         =====================
!    05/03/07    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 evaluate
```

4-15 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!

```
PROGRAM ucase
!
!  Purpose:
!    To shift a character string to upper case.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/03/07    S. J. Chapman         Original code
!
IMPLICIT NONE

! Declare named constants:
INTEGER, PARAMETER :: LEN_STR = 40  ! String length

! Declare variables:
INTEGER :: i                        ! Loop index
CHARACTER(len=LEN_STR) :: string    ! String

! Get string
WRITE (*,*) 'Enter string to shift to upper case: '
READ (*,*) string

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

! Write out string
WRITE (*,*) 'The converted string is: ', string

END PROGRAM ucase
```

When this program is executed, the results are as shown below. Lower case letters are converted to upper case, but no other characters are affected.

```
C:\book\f95_2003\soln\ex4_15>ucase
Enter string to shift to upper case:
"This is a test! 123$%^+\"
The converted string is: THIS IS A TEST! 123$%^+\
```

We will reexamine this program in Chapter 10 with an eye to designing an algorithm that works properly on all computers, not just those with an ASCII collating sequence.

4-16 A program to calculate the distance from the center of the Earth to a satellite in orbit as a function of the eccentricity of the orbit and the position in the orbit theta is given below:

```
PROGRAM orbit
!
!  Purpose:
```

```
!    To calculate the distance r from the center of the
!    Earth to a satellite in orbit, as a function of
!    the orbit's eccentricity and the size parameter p.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/04/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! Declare named constants:
REAL, PARAMETER :: DEG_2_RAD = 0.01745329 ! Degrees to radians

! Declare variables:
REAL :: ecc                         ! Eccentricity (0-1)
REAL :: p                           ! Size parameter (m)
REAL :: r                           ! Distance from centre of Earth to orbit
REAL :: theta                       ! Angle in orbit (deg)
INTEGER :: i                        ! Loop index

! Get size parameter
WRITE (*,*) 'Enter size parameter (m): '
READ (*,*) p

! Get eccentricity
WRITE (*,*) 'Enter eccentricity (0-1): '
READ (*,*) ecc

! Now calculate the minimum and maximum distances as a function
! of angle around the orbit.
DO i = 0, 360, 30

   ! Get angle theta (deg)
   theta = i

   ! Get range at this angle
   r = p / ( 1 - ecc * COS(theta * DEG_2_RAD) )

   ! Print out the results
   WRITE (*,*) 'Theta = ', theta, ', range = ', r

END DO

END PROGRAM orbit
```

When this program is executed, the results are as shown below.

```
C:\book\f95_2003\soln>orbit
Enter size parameter (m):
1200000
 Enter eccentricity (0-1):
0
 Theta =   0.0000000E+00 , range =    1200000.
 Theta =    30.00000     , range =    1200000.
 Theta =    60.00000     , range =    1200000.
```

```
 Theta =    90.00000     , range =    1200000.
 Theta =    120.0000     , range =    1200000.
 Theta =    150.0000     , range =    1200000.
 Theta =    180.0000     , range =    1200000.
 Theta =    210.0000     , range =    1200000.
 Theta =    240.0000     , range =    1200000.
 Theta =    270.0000     , range =    1200000.
 Theta =    300.0000     , range =    1200000.
 Theta =    330.0000     , range =    1200000.
 Theta =    360.0000     , range =    1200000.

C:\book\f95_2003\soln>orbit
 Enter size parameter (m):
1200000
 Enter eccentricity (0-1):
0.25
 Theta =   0.0000000E+00 , range =    1600000.
 Theta =    30.00000     , range =    1531602.
 Theta =    60.00000     , range =    1371429.
 Theta =    90.00000     , range =    1200000.
 Theta =    120.0000     , range =    1066667.
 Theta =    150.0000     , range =    986431.4
 Theta =    180.0000     , range =    960000.0
 Theta =    210.0000     , range =    986431.3
 Theta =    240.0000     , range =    1066667.
 Theta =    270.0000     , range =    1200000.
 Theta =    300.0000     , range =    1371428.
 Theta =    330.0000     , range =    1531601.
 Theta =    360.0000     , range =    1600000.

C:\book\f95_2003\soln>orbit
 Enter size parameter (m):
1200000
 Enter eccentricity (0-1):
0.5
 Theta =   0.0000000E+00 , range =    2400000.
 Theta =    30.00000     , range =    2116450.
 Theta =    60.00000     , range =    1600000.
 Theta =    90.00000     , range =    1200000.
 Theta =    120.0000     , range =    960000.1
 Theta =    150.0000     , range =    837396.7
 Theta =    180.0000     , range =    800000.0
 Theta =    210.0000     , range =    837396.6
 Theta =    240.0000     , range =    959999.8
 Theta =    270.0000     , range =    1200000.
 Theta =    300.0000     , range =    1600000.
 Theta =    330.0000     , range =    2116449.
 Theta =    360.0000     , range =    2400000.
```

If the eccentricity is 0.0, the orbit is a uniform circle with 1200 km radius. If the eccentricity is 0.25, the orbit is elliptical with a minimum size of 960 km and a maximum size of 1600 km. If the eccentricity is 0.5, the orbit is elliptical with a minimum size of 800 km and a maximum size of 2400 km.

4-17 A program that capitalizes the first letter in each word and shifts the remaining letters to lower case is shown below. Note that a new word is found if this is the first character in the string, or if the current letter is *not* a space and the previous letter was a space.

```
PROGRAM caps
!
!  Purpose:
!    To shift capitalize all the words in a character
!    string.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/04/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare named constants:
INTEGER, PARAMETER :: LEN_STR = 80  ! String length

! Declare variables:
INTEGER :: i                        ! Loop index
CHARACTER(len=LEN_STR) :: string  ! String

! Get string
WRITE (*,*) 'Enter string to shift to capitalize: '
READ (*,*) string

! Special case: capitalize the first character in the string.
IF ( string(1:1) >= 'a' .AND. string(1:1) <= 'z' ) THEN
   string(1:1) = ACHAR ( IACHAR ( string(1:1) ) - 32 )
END IF

! Now capitalize all other characters
DO i = 2, LEN_STR

   ! Check for start of word.
   IF ( string(i-1:i-1) == ' ' .AND. string(i:i) /= ' ' ) THEN

      ! This is the start of a word.  Capitalize it.
      IF ( string(i:i) >= 'a' .AND. string(i:i) <= 'z' ) THEN
         string(i:i) = ACHAR ( IACHAR ( string(i:i) ) - 32 )
      END IF

   ELSE

      ! This is NOT the start of a word.  Shift to lower case.
      ! This is the start of a word.  Capitalize it.
      IF ( string(i:i) >= 'A' .AND. string(i:i) <= 'Z' ) THEN
         string(i:i) = ACHAR ( IACHAR ( string(i:i) ) + 32 )
      END IF

   END IF
END DO

! Write out string
WRITE (*,*) 'The converted string is: ', string

END PROGRAM caps
```

When this program is executed, the results are as shown below. Lower case letters are converted to upper case, but no other characters are affected.

```
C:\book\f95_2003\soln>caps
 Enter string to shift to capitalize:
"This IS a TEST! 123$%^+\"
 The converted string is:
 This Is A Test! 123$%^+\
```

The program capitalized the first letter of each word, shifted all remaining letter to lower case, and left any non-alphabetic characters alone.

4-18 The program to calculate the current through the diode is shown below:

```
PROGRAM diode
!
!  Purpose:
!    Program to calculate the current flowing through a diode
!    as a function of the voltage across it.  The program must
!    allow the user to specify the temperature of the diode.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/04/2007    S. J. Chapman      Original code
!
! List of named constants:
REAL, PARAMETER :: I0 = 2.00E-6   ! Diode leakage current (amps)
REAL, PARAMETER :: K = 1.38E-23   ! Boltzmann's constant
REAL, PARAMETER :: Q = 1.06E-19   ! Charge of an electron (coulombs)

! List of variables:
INTEGER :: i                      ! Loop index
REAL :: i_diode                   ! Diode current (amps)
REAL :: temp_f                    ! Temperature (degrees F)
REAL :: temp_k                    ! Temperature (kelvins)
REAL :: v_diode                   ! Diode voltage (volts)

! Get the junction temperature of the diode.
WRITE (*,*) 'Enter temperature of the diode in degrees F: '
READ (*,*) temp_f

! Convert temperature to kelvins.
temp_k = (5./9.) * (temp_f - 32.) + 273.15


! Calculate currents versus diode voltage
DO i = -10, 6

   ! Get diode voltage.
   v_diode = REAL(i) / 10.

   ! Calculate current flow.
   i_diode = I0 * ( EXP ((Q*v_diode)/(K*temp_k)) - 1.0 )

   ! Write out voltage and current.
```

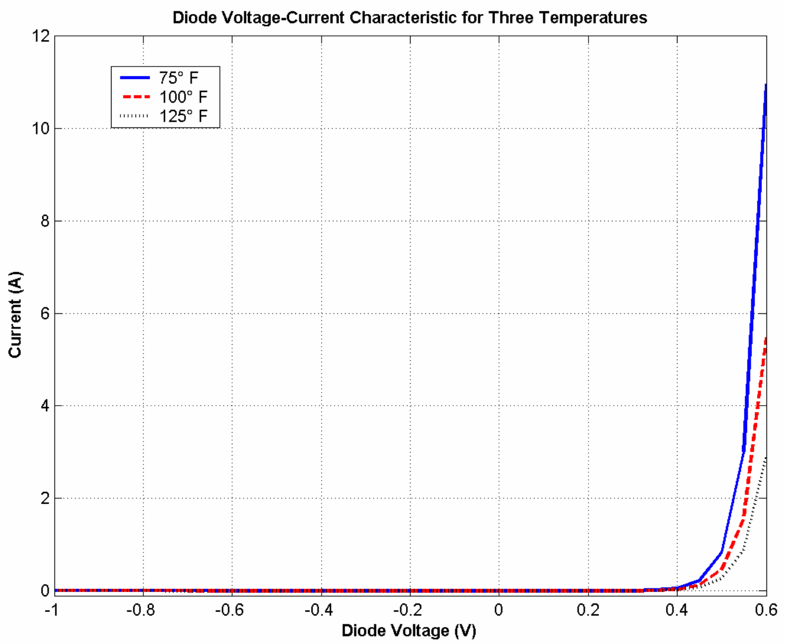
```
   WRITE (*,*) 'VD = ', v_diode, '    ID = ', i_diode

END DO

END PROGRAM diode
```

The results of this program for diode temperatures of 75° F, 100° F, and 125° F are shown below.

```
C:\book\f95_2003\soln>diode
Enter temperature of the diode in degrees F:
75
VD =       -1.000000    ID =   -2.000000E-06
VD =   -9.000000E-01    ID =   -2.000000E-06
VD =   -8.000000E-01    ID =   -2.000000E-06
VD =   -7.000000E-01    ID =   -2.000000E-06
VD =   -6.000000E-01    ID =   -2.000000E-06
VD =   -5.000000E-01    ID =   -1.999995E-06
VD =   -4.000000E-01    ID =   -1.999936E-06
VD =   -3.000000E-01    ID =   -1.999145E-06
VD =   -2.000000E-01    ID =   -1.988652E-06
VD =   -1.000000E-01    ID =   -1.849345E-06
VD =    0.000000E+00    ID =    0.000000E+00
VD =    1.000000E-01    ID =    2.455074E-05
VD =    2.000000E-01    ID =    3.504710E-04
VD =    3.000000E-01    ID =    4.677185E-03
VD =    4.000000E-01    ID =    6.211591E-02
VD =    5.000000E-01    ID =    8.246362E-01
VD =    6.000000E-01    ID =       10.947380

C:\book\f95_2003\soln>diode
Enter temperature of the diode in degrees F:
100
VD =       -1.000000    ID =   -2.000000E-06
VD =   -9.000000E-01    ID =   -2.000000E-06
VD =   -8.000000E-01    ID =   -2.000000E-06
VD =   -7.000000E-01    ID =   -2.000000E-06
VD =   -6.000000E-01    ID =   -1.999999E-06
VD =   -5.000000E-01    ID =   -1.999991E-06
VD =   -4.000000E-01    ID =   -1.999898E-06
VD =   -3.000000E-01    ID =   -1.998791E-06
VD =   -2.000000E-01    ID =   -1.985702E-06
VD =   -1.000000E-01    ID =   -1.830898E-06
VD =    0.000000E+00    ID =    0.000000E+00
VD =    1.000000E-01    ID =    2.165436E-05
VD =    2.000000E-01    ID =    2.777643E-04
VD =    3.000000E-01    ID =    3.306823E-03
VD =    4.000000E-01    ID =    3.913203E-02
VD =    5.000000E-01    ID =    4.628431E-01
VD =    6.000000E-01    ID =        5.474152

C:\book\f95_2003\soln>diode
Enter temperature of the diode in degrees F:
125
VD =       -1.000000    ID =   -2.000000E-06
VD =   -9.000000E-01    ID =   -2.000000E-06
VD =   -8.000000E-01    ID =   -2.000000E-06
```

```
VD =   -7.000000E-01    ID =   -2.000000E-06
VD =   -6.000000E-01    ID =   -1.999999E-06
VD =   -5.000000E-01    ID =   -1.999985E-06
VD =   -4.000000E-01    ID =   -1.999844E-06
VD =   -3.000000E-01    ID =   -1.998340E-06
VD =   -2.000000E-01    ID =   -1.982339E-06
VD =   -1.000000E-01    ID =   -1.812058E-06
VD =    0.000000E+00    ID =    0.000000E+00
VD =    1.000000E-01    ID =    1.928314E-05
VD =    2.000000E-01    ID =    2.244860E-04
VD =    3.000000E-01    ID =    2.408166E-03
VD =    4.000000E-01    ID =    2.564595E-02
VD =    5.000000E-01    ID =    2.729323E-01
VD =    6.000000E-01    ID =        2.904449
```

The effect of the temperature on the diode operating characteristic can best be shown in a graph. The voltage versus current characteristics for each of the three temperatures is shown below:



4-19 The program to convert a binary number into an equivalent decimal value is shown below. Note that this program checks to confirm that the string is valid before it converts it.

```
PROGRAM binary_to_decimal
!
!  Purpose:
!    Program to convert a binary number into its decimal
!    equivalent.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/04/2007    S. J. Chapman       Original code
!
! Declare named constants:
```

```
INTEGER, PARAMETER :: LEN_STR = 10  ! String length

! Declare variables:
INTEGER :: i                       ! Loop index
INTEGER :: iend                    ! End of valid values
INTEGER :: istart                  ! Start of valid values
INTEGER :: ival                    ! Value of a digit in this position
CHARACTER(len=LEN_STR) :: string  ! String
LOGICAL :: valid                   ! Test for valid string
INTEGER :: value                   ! Final value

! Get string to convert.
WRITE (*,*) 'Enter binary string to convert (up to 10 characters): '
READ (*,*) string

! Find beginning of valid values
DO i = 1, LEN_STR
   IF ( string(i:i) /= ' ' ) THEN
      istart = i
      EXIT
   END IF
END DO

! Find end of valid values
DO i = LEN_STR, 1, -1
   IF ( string(i:i) /= ' ' ) THEN
      iend = i
      EXIT
   END IF
END DO

! Are the characters between the start and the end all valid?
valid = .TRUE.
DO i = istart, iend
   IF ( string(i:i) < '0' .OR. string(i:i) > '1' ) THEN
      valid = .FALSE.
   END IF
END DO

! If the string only contains valid characters, convert it.
IF ( valid ) THEN

   ! Convert each value starting with the smallest and
   ! working upward.  The variable "ival" contains the
   ! value of a "1" in the current position.
   ival = 1
   value = 0
   DO i = iend, istart, -1
      IF ( string(i:i) == '1' ) THEN
         value = value + ival
      END IF
      ival = ival * 2
   END DO

   ! Write out the result.
   WRITE (*,*) 'The value is ', value
```

```
ELSE

   ! This is an illegal string.
   WRITE (*,*) 'Illegal string!'

END IF

END PROGRAM binary_to_decimal
```

The results of this program for the four specified inputs and an illegal string are shown below.

```
C:\book\f95_2003\soln>binary_to_decimal
Enter binary string to convert (up to 10 characters):
0010010010
 The value is          146

C:\book\f95_2003\soln>binary_to_decimal
 Enter binary string to convert (up to 10 characters):
1111111111
 The value is         1023

C:\book\f95_2003\soln>binary_to_decimal
 Enter binary string to convert (up to 10 characters):
1000000001
 The value is          513

C:\book\f95_2003\soln>binary_to_decimal
 Enter binary string to convert (up to 10 characters):
0111111110
 The value is          510

C:\book\f95_2003\soln>binary_to_decimal
 Enter binary string to convert (up to 10 characters):
12112
 Illegal string!
```

4-20 The program to convert a decimal number into an equivalent binary value is shown below. This program works by seeing if the input value is over 512. If it is, it set the first bit to 1 and subtracts 512 from the original value. If not, it sets the first bit to 0. Then it sees if the input value is over 256. If it is, it set the second bit to 1 and subtracts 256 from the original value. If not, it sets the second bit to 0. This process is repeated until the tenth bit is reached.

```
PROGRAM decimal_to_binary
!
!  Purpose:
!    Program to convert a decimal number into its binary
!    equivalent.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/04/2007    S. J. Chapman      Original code
!

! Declare named constants:
INTEGER, PARAMETER :: LEN_STR = 10  ! String length
```

```
! Declare variables:
INTEGER :: i                        ! Loop index
INTEGER :: iend                     ! End of valid values
INTEGER :: istart                   ! Start of valid values
INTEGER :: ival                     ! Value of a digit in this position
CHARACTER(len=LEN_STR) :: string    ! String
LOGICAL :: valid                    ! Test for valid string
INTEGER :: value                    ! Number to convert value

! Get string to convert.
WRITE (*,*) 'Enter decimal number to convert (0-1023): '
READ (*,*) value

IF ( value < 0 .OR. value > 1023 ) THEN

   ! This is an illegal value.
   WRITE (*,*) 'Illegal value entered!'

ELSE

   ! Start with the largest bit and work downwards.  The
   ! largest bit is worth 2**10, or 512.  If the number
   ! is greater than that, set that bit to one and subtract
   ! 512 from the value.  Then try the bit at 2**9, or 512,
   ! and so forth.
   ival = 512
   string = ' ';
   DO i = 1, LEN_STR

      IF ( value >= ival ) THEN
         string(i:i) = '1'
         value = value - ival
      ELSE
         string(i:i) = '0'
      END IF
      ival = ival / 2
   END DO

   ! Write out result.
   WRITE (*,*) 'The value is ', string

END IF

END PROGRAM decimal_to_binary
```

The results of this program for the four specified inputs are shown below.

```
C:\book\f95_2003\soln>decimal_to_binary
 Enter decimal number to convert (0-1023):
256
 The value is 0100000000

C:\book\f95_2003\soln>decimal_to_binary
 Enter decimal number to convert (0-1023):
63
```

```
 The value is 0000111111

C:\book\f95_2003\soln>decimal_to_binary
 Enter decimal number to convert (0-1023):
140
 The value is 0010001100

C:\book\f95_2003\soln>decimal_to_binary
 Enter decimal number to convert (0-1023):
768
 The value is 1100000000
```

4-21 The program to convert an octal number into an equivalent decimal value is shown below. Note that this program checks to confirm that the string is valid before it converts it.

```
PROGRAM octal_to_decimal
!
!  Purpose:
!    Program to convert an octal number into its decimal
!    equivalent.
!
!  Record of revisions:
!      Date       Programmer         Description of change
!      ====       ==========         =====================
!    05/04/2007   S. J. Chapman      Original code
!

! Declare named constants:
INTEGER, PARAMETER :: LEN_STR = 10  ! String length

! Declare variables:
INTEGER :: i                        ! Loop index
INTEGER :: iend                     ! End of valid values
INTEGER :: istart                   ! Start of valid values
INTEGER :: ival                     ! Value of a digit in this position
CHARACTER(len=LEN_STR) :: string    ! String
LOGICAL :: valid                    ! Test for valid string
INTEGER :: value                    ! Final value

! Get string to convert.
WRITE (*,*) 'Enter binary string to convert (up to 10 characters): '
READ (*,*) string

! Find beginning of valid values
DO i = 1, LEN_STR
   IF ( string(i:i) /= ' ' ) THEN
      istart = i
      EXIT
   END IF
END DO

! Find end of valid values
DO i = LEN_STR, 1, -1
   IF ( string(i:i) /= ' ' ) THEN
      iend = i
      EXIT
```

```
   END IF
END DO

! Are the characters between the start and the end all valid?
valid = .TRUE.
DO i = istart, iend
   IF ( string(i:i) < '0' .OR. string(i:i) > '7' ) THEN
      valid = .FALSE.
   END IF
END DO

! If the string only contains valid characters, convert it.
IF ( valid ) THEN

   ! Convert each value starting with the smallest and
   ! working upward.  The variable "ival" contains the
   ! value of a "1" in the current position.
   ival = 1
   value = 0
   DO i = iend, istart, -1
      SELECT CASE (string(i:i))
      CASE ('0')
         value = value
      CASE ('1')
         value = value + ival
      CASE ('2')
         value = value + 2*ival
      CASE ('3')
         value = value + 3*ival
      CASE ('4')
         value = value + 4*ival
      CASE ('5')
         value = value + 5*ival
      CASE ('6')
         value = value + 6*ival
      CASE ('7')
         value = value + 7*ival
      END SELECT
      ival = ival * 8
   END DO

   ! Write out the result.
   WRITE (*,*) 'The value is ', value

ELSE

   ! This is an illegal string.
   WRITE (*,*) 'Illegal string!'

END IF

END PROGRAM octal_to_decimal
```

The results of this program for the four specified inputs are shown below.

```
C:\book\f95_2003\soln>octal_to_decimal
```

```
 Enter binary string to convert (up to 10 characters):
377
 The value is          255

C:\book\f95_2003\soln>octal_to_decimal
 Enter binary string to convert (up to 10 characters):
11111
 The value is         4681

C:\book\f95_2003\soln>octal_to_decimal
 Enter binary string to convert (up to 10 characters):
70000
 The value is        28672

C:\book\f95_2003\soln>octal_to_decimal
 Enter binary string to convert (up to 10 characters):
77777
 The value is        32767
```

4-22 A program to calculate the tension on the cable is shown below:

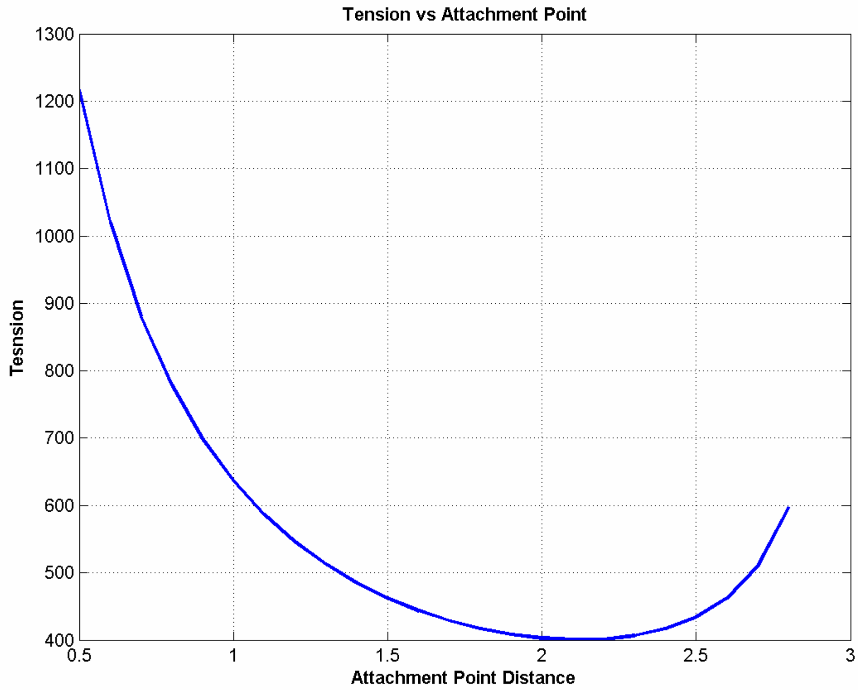
```
PROGRAM calc_tension
!
!  Purpose:
!    Program to calculate the minimum tension on a cable
!    supporting a 200 kg weight attached to a wall.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/05/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
REAL :: dist                       ! Distance to attachment (m)
INTEGER :: i                       ! Index variable
REAL :: lc = 3.                    ! Cable length (m)
REAL :: lp  = 3.                   ! Pole length (m)
REAL :: saved_dist                 ! Saved attachment distance
REAL :: saved_tension = 999999.    ! Saved tension
REAL :: tension                    ! Tension on cable
REAL :: weight = 200.              ! Weight of object (kg)

! Calculate tension at all attachment points between 1 and 7 ft
DO i = 5, 28

   ! Get attachment point
   dist = REAL(i) / 10.

   ! Calculate tension
   tension = weight * lc * lp / ( dist * SQRT(lp**2 - dist**2) )

   ! Write results
   WRITE (*,*) 'dist = ', dist, ' tension = ', tension
```

```
   ! Check for minimum tension
   IF ( tension < saved_tension ) THEN
      saved_tension = tension
     saved_dist = dist
   END IF

END DO

! Tell user of minimum tension
WRITE (*,*) 'Minimum at d = ', saved_dist, ' tension = ', saved_tension

END PROGRAM calc_tension
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln>calc_tension
dist =   0.5000000      tension =    1217.022
 dist =   0.6000000      tension =    1020.621
 dist =   0.7000000      tension =    881.4744
 dist =   0.8000000      tension =    778.1788
 dist =   0.9000000      tension =    698.8566
 dist =    1.000000      tension =    636.3961
 dist =    1.100000      tension =    586.2881
 dist =    1.200000      tension =    545.5447
 dist =    1.300000      tension =    512.1185
 dist =    1.400000      tension =    484.5718
 dist =    1.500000      tension =    461.8802
 dist =    1.600000      tension =    443.3121
 dist =    1.700000      tension =    428.3542
 dist =    1.800000      tension =    416.6667
 dist =    1.900000      tension =    408.0605
 dist =    2.000000      tension =    402.4922
 dist =    2.100000      tension =    400.0800
 dist =    2.200000      tension =    401.1466
 dist =    2.300000      tension =    406.3102
 dist =    2.400000      tension =    416.6667
 dist =    2.500000      tension =    434.1763
 dist =    2.600000      tension =    462.5675
 dist =    2.700000      tension =    509.8128
 dist =    2.800000      tension =    596.8778
 Minimum at d =    2.100000      tension =    400.0800
```

A plot of tension versus attachment point is shown below:



4-23 It would be safe to connect the cable anywhere from 0.9 m to 2.6 m.

4-24 The program to calculate bacterial growth rates is

```
PROGRAM growth
!
!  Purpose:
!    Program to calculate rate of bacterial growth in
!    different culture media.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/05/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
REAL :: doubling_time_1 = 1.5  ! Doubling time of medium 1 (hrs)
REAL :: doubling_time_2 = 2.0  ! Doubling time of medium 2 (hrs)
INTEGER :: i                   ! Index variables
REAL :: n_bacteria_1           ! Number of bacteria in med 1
REAL :: n_bacteria_2           ! Number of bacteria in med 2
REAL :: time                   ! Time in hours

! Print heading.
WRITE (*,*) 'The rates of colony growth are: '

! Calculate the sizes of each colony.
DO i = 0, 24, 3

   ! Calculate time in hours
```

```
   time = REAL(i)

   ! Calculate colony size
   n_bacteria_1 = 2 ** (time / doubling_time_1 )
   n_bacteria_2 = 2 ** (time / doubling_time_2 )

   ! Tell user
   WRITE (*,*) time, n_bacteria_1, n_bacteria_2

END DO

END PROGRAM growth
```

When this program is executed, the result is

```
C:\book\f95_2003\soln>growth
The rates of colony growth are:
  0.0000000E+00   1.000000       1.000000
   3.000000       4.000000       2.828427
   6.000000       16.00000       8.000000
   9.000000       64.00000       22.62742
   12.00000       256.0000       64.00000
   15.00000       1024.000       181.0193
   18.00000       4096.000       512.0000
   21.00000       16384.00       1448.155
   24.00000       65536.00       4096.000
```

After 24 hours, Medium A has 16 times more bacteria than Medium B.

4-25 The program to calculate the power level in dB is

```
PROGRAM db_calc
!
!  Purpose:
!    To calculate the decibel level corresponding to power levels
!    between 1 and 20 watts, referenced to 1 watt.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/05/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
REAL, PARAMETER :: P_REF = 1.0 ! Reference power (1 W)

! List of variables:
REAL :: db                    ! Power level in dB (ref. to 1 W)
INTEGER :: i                  ! Loop index
REAL :: power                 ! Input power level (W)

DO i = 1, 40

   ! Get power level.
   power = REAL(i/2)
```

```
   ! Calculate power in dB.
   db = 10.0 * LOG10 ( power / P_REF )

   ! Write out power and dB level.
   WRITE (*,*) 'Power = ', power, ' W    dB = ', db

END DO

END PROGRAM db_calc
```

When this program is run, the result is

```
C:\book\f95_2003\soln>db_calc
 Power =   0.5000000      W    dB =   -3.010300
 Power =    1.000000      W    dB =   0.0000000E+00
 Power =    1.500000      W    dB =    1.760913
 Power =    2.000000      W    dB =    3.010300
 Power =    2.500000      W    dB =    3.979400
 Power =    3.000000      W    dB =    4.771213
 Power =    3.500000      W    dB =    5.440681
 Power =    4.000000      W    dB =    6.020600
 Power =    4.500000      W    dB =    6.532125
 Power =    5.000000      W    dB =    6.989700
 Power =    5.500000      W    dB =    7.403627
 Power =    6.000000      W    dB =    7.781513
 Power =    6.500000      W    dB =    8.129133
 Power =    7.000000      W    dB =    8.450980
 Power =    7.500000      W    dB =    8.750612
 Power =    8.000000      W    dB =    9.030900
 Power =    8.500000      W    dB =    9.294189
 Power =    9.000000      W    dB =    9.542425
 Power =    9.500000      W    dB =    9.777236
 Power =    10.00000      W    dB =    10.00000
 Power =    10.50000      W    dB =    10.21189
 Power =    11.00000      W    dB =    10.41393
 Power =    11.50000      W    dB =    10.60698
 Power =    12.00000      W    dB =    10.79181
 Power =    12.50000      W    dB =    10.96910
 Power =    13.00000      W    dB =    11.13943
 Power =    13.50000      W    dB =    11.30334
 Power =    14.00000      W    dB =    11.46128
 Power =    14.50000      W    dB =    11.61368
 Power =    15.00000      W    dB =    11.76091
 Power =    15.50000      W    dB =    11.90332
 Power =    16.00000      W    dB =    12.04120
 Power =    16.50000      W    dB =    12.17484
 Power =    17.00000      W    dB =    12.30449
 Power =    17.50000      W    dB =    12.43038
 Power =    18.00000      W    dB =    12.55272
 Power =    18.50000      W    dB =    12.67172
 Power =    19.00000      W    dB =    12.78754
 Power =    19.50000      W    dB =    12.90035
 Power =    20.00000      W    dB =    13.01030
```

4-26 The program to evaluate the sine using a truncated infinite series is shown below:

```
PROGRAM sines
!
!  Purpose:
!    To calculate the sine of an input value, first using the
!    intrinsic function sin() and then using the truncated
!    infinite series approximation to the sine.  The truncated
!    series should be calculate with first with 1 term, then
!    with two terms, and so forth up to 10 terms.  Note that
!    the input value for this programs should be in DEGREES.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/05/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
REAL, PARAMETER :: DEG_2_RAD = 0.01745329  ! Conv factor

! List of variables:
REAL :: fact               ! Factorial
INTEGER :: i, j            ! Loop indices
REAL :: sine               ! Sine
REAL :: theta              ! Angle

! Get theta
WRITE (*,*) 'Program to calculate the sine of a number using the'
WRITE (*,*) 'truncated infinite series approximation.'
WRITE (*,*) 'Please enter the desired angle (in degrees): '
READ (*,*) theta

! Convert to radians.
theta = theta * DEG_2_RAD

! Calculate sin(theta) using intrinsic function, and tell user.
WRITE (*,*) 'Intrinsic function:  SIN(THETA) = ', SIN(theta)

! Calculate series approximation, writing out the result after
! each term is calculated.
sine = 0.
DO i = 1, 10

   ! First, calculate (2*i-1)!.
   fact = 1.0
   DO j = 2*i-1, 1, -1
      fact = fact * REAL(j)
   END DO

   ! Next, add this term to the series.
   sine = sine + (-1)**(i-1) * theta**(2*i-1) / fact

   ! Display the result so far.
   WRITE (*,*) i, '   term:  SIN(THETA) = ', sine
```

```
END DO

END PROGRAM sines
```

When this program is executed on a PC, the results are:

```
C:\book\f95_2003\soln>sines
Program to calculate the sine of a number using the
truncated infinite series approximation.
Please enter the desired angle (in degrees):
45
Intrinsic function:  SIN(THETA) =    7.071067E-01
          1   term:  SIN(THETA) =    7.853981E-01
          2   term:  SIN(THETA) =    7.046526E-01
          3   term:  SIN(THETA) =    7.071430E-01
          4   term:  SIN(THETA) =    7.071064E-01
          5   term:  SIN(THETA) =    7.071067E-01
          6   term:  SIN(THETA) =    7.071067E-01
          7   term:  SIN(THETA) =    7.071067E-01
          8   term:  SIN(THETA) =    7.071067E-01
          9   term:  SIN(THETA) =    7.071067E-01
         10   term:  SIN(THETA) =    7.071067E-01
```

On this particular computer, the infinite series converges to the correct answer after 5 or 6 terms. Calculating more terms is just a waste of computing time.

4-27 A program to calculate the arithmetic and geometric means is shown below:

```
PROGRAM means
!
!  Purpose:
!    To calculate the average (arithmetic mean) and geometric
!    mean of a set of input data values.  All input values
!    must be positive or zero; a negative number terminates
!    further input.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/05/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
REAL :: ave           ! Average of input samples
REAL :: gmean         ! Geometric mean
INTEGER :: n = 0      ! Number of input samples
REAL :: prod_x = 1.   ! Product of input values
REAL :: sum_x = 0.    ! Sum of input values
REAL :: x             ! An input value

DO

   ! Get number
   WRITE (*,*) 'Enter number: '
```

```
   READ  (*,*) x
   WRITE (*,*) 'The number is ', x

   ! Test for exit.
   IF ( x < 0. ) EXIT

   ! Accumulate sums and products
   n      = n + 1
   sum_x  = sum_x + x
   prod_x = prod_x * x

END DO

! Calculate the arithmetic mean and geometric mean.
! Note that taking the Nth root of a number is the
! equivalent of raising the number to the 1/Nth power.
ave = sum_x / REAL(n)
gmean = prod_x ** ( 1. / REAL(n) )

! Tell user.
WRITE (*,*) 'The average of this data set is:', ave
WRITE (*,*) 'The geometric mean is:          ', gmean
WRITE (*,*) 'The number of data points is:   ', n

END PROGRAM means
```

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

```
C:\book\f95_2003\soln>means
Enter first number:
10
The number is       10.000000
Enter next number:
5
The number is        5.000000
Enter next number:
2
The number is        2.000000
Enter next number:
5
The number is        5.000000
Enter next number:
-1
The number is       -1.000000
The average of this data set is:       5.500000
The geometric mean is:                 4.728708
The number of data points is:             4
```

4-28 A program to calculate the rms average of an input data set is shown below:

```
PROGRAM rms_ave
!
!  Purpose:
!    To calculate rms average of an input data set, where each
!    input value can be positive, negative, or zero.
!
```

```
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/05/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of variables:
INTEGER :: i          ! Loop index
INTEGER :: n          ! Number of input samples
REAL :: rms           ! rms average
REAL :: sum_x2 = 0.   ! Sum of squares of inputs
REAL :: x = 0.        ! Input data value

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

DO i = 1, n

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

  ! Accumulate sums.
  sum_x2 = sum_x2 + x**2

END DO

! Calculate the rms average
rms = SQRT ( sum_x2 / REAL(n) )

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

END PROGRAM rms_ave
```

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

```
C:\book\f95_2003\soln>rms_ave
Enter number of points:
4
Enter next number:
10.
Enter next number:
5.
Enter next number:
2.
Enter next number:
5.
The rms average of this data set is:       6.204837
The number of data points is:                 4
```

4-29 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
!       ====       ==========         =====================
!    05/05/2007    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 harmon
```

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

```
C:\book\f95_2003\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
```

4-30 A program to calculate all of the means of an input data set is shown below:

```
PROGRAM all_means
!
!  Purpose:
!    To calculate the average (arithmetic mean), rms average,
!    geometric mean, and 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
!       ====       ==========         =====================
!    05/05/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
REAL :: ave            ! Average (arithmetic mean)
REAL :: g_mean         ! Geometric mean
REAL :: h_mean         ! Harmonic mean
INTEGER :: i           ! Index variable
INTEGER :: n           ! Number of input values
REAL :: prod_x = 1.0   ! Product of the input values
REAL :: rms            ! Rms average
REAL :: sum_x = 0.0    ! Sum of the input values
REAL :: sum_x2 = 0.0   ! Sum of input values squared
REAL :: sum_rx = 0.0   ! Sum of reciprocal of input values
REAL :: x = 0.0        ! Input value

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

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

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

   ! Accumulate sums.
   prod_x = prod_x * x
   sum_x  = sum_x + x
   sum_x2 = sum_x2 + x**2
   sum_rx = sum_rx + 1.0 / x

END DO

! Calculate the means
ave    = sum_x / REAL(n)
```

```
g_mean = prod_x ** ( 1. / REAL(n) )
h_mean = REAL(n) / sum_rx
rms    = SQRT ( sum_x2 / REAL(n) )

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

END PROGRAM all_means
```

(a) When the program is run with the first sample data set, the results are:

```
C:\book\f95_2003\soln>all_means
Enter number of points:
7
Enter next number:
4
Enter next number:
4
Enter next number:
4
Enter next number:
4
Enter next number:
4
Enter next number:
4
Enter next number:
4
The average of this data set is:          4.000000
The geometric mean is:                    4.000000
The harmonic mean of this data set is:    4.000000
The rms average of this data set is:      4.000000
The number of data points is:                 7
```

(b) When the program is run with the second sample data set, the results are:

```
The average of this data set is:          4.000000
The geometric mean is:                    3.926918
The harmonic mean of this data set is:    3.853211
The rms average of this data set is:      4.070802
The number of data points is:                 7
```

(c) When the program is run with the third sample data set, the results are:

```
The average of this data set is:          4.000000
The geometric mean is:                    3.158510
The harmonic mean of this data set is:    2.305882
The rms average of this data set is:      4.598136
The number of data points is:                 7
```

(d) When the program is run with the fourth sample data set, the results are:

```
The average of this data set is:            4.000000
The geometric mean is:                      3.380015
The harmonic mean of this data set is:      2.699724
The rms average of this data set is:        4.472136
The number of data points is:                  7
```

4-31 A program to calculate the overall MTBF of a system consisting of "ncomp" series components is:

```
PROGRAM calc_mtbf
!
!  Purpose:
!    To calculate the mean time between failures of a system
!    consisting of "ncomp" series components, each of which has
!    a known MTBF.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/06/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
INTEGER :: i             ! Loop index
REAL :: mtbf_i           ! MTBF of ith component
REAL :: mtbf_total       ! MTBF of whole system
INTEGER :: ncomp         ! Number of components
REAL :: sum_recip = 0.   ! Sum of reciprocals of MTBFs

! Get the number of components in series.
WRITE (*,*) 'This program calculates the MTBF of a system'
WRITE (*,*) 'consisting of "ncomp" series components. '
WRITE (*,*) 'Enter number of components: '
READ  (*,*) ncomp

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

   ! Get next number.
   WRITE (*,*) 'Enter MTBF of component', i,': '
   READ  (*,*) mtbf_i

   ! Accumulate sums.
   sum_recip = sum_recip + 1.0 / mtbf_i

END DO

! Calculate the total MTBF
mtbf_total = 1. / sum_recip

! Tell user.
WRITE (*,*) 'The MTBF of the overall system is: ', mtbf_total

END PROGRAM calc_mtbf
```

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

```
C:\book\f95_2003\soln>calc_mtbf
This program calculates the MTBF of a system
consisting of "ncomp" series components.
Enter number of components:
4
Enter MTBF of component          1:
2000
Enter MTBF of component          2:
800
Enter MTBF of component          3:
3000
Enter MTBF of component          4:
5000
The MTBF of the overall system is:      437.956200
```

4-32 A program to calculate the volume of an ideal gas as a function of pressure is shown below:

```
PROGRAM ideal_gas1
!
!  Purpose:
!    To calculate the volume of one mole of an ideal gas as
!    pressure is varied from 1 to 1001 kPa in steps of 100 kPa.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/06/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Constants
REAL,PARAMETER :: R = 8.314   ! Ideal gas constant (L kPa/mol K)

! List of variables:
INTEGER :: i              ! Loop index
REAL :: n = 1.0           ! Number of atoms (mol)
REAL :: p                 ! Pressure (kPa)
REAL :: t                 ! Temperature (K)
REAL :: v                 ! volume (L)

! Get temperature
WRITE (*,*) 'Enter gas temperature in kelvins:'
READ (*,*) t

! Calculate the volume as a function pressure
DO i = 1, 1001, 100

   ! Get pressure
   p = i

   ! Calculate the volume
   v = n * R * t / p

   ! Write out volume
   WRITE (*,*) 'Pressure = ', p, ', Volume = ', v
```

```
END DO

END PROGRAM ideal_gas1
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln>ideal_gas1
 Enter gas temperature in kelvins:
273
 Pressure =    1.000000     , Volume =    2269.722
 Pressure =    101.0000     , Volume =    22.47250
 Pressure =    201.0000     , Volume =    11.29215
 Pressure =    301.0000     , Volume =    7.540605
 Pressure =    401.0000     , Volume =    5.660155
 Pressure =    501.0000     , Volume =    4.530383
 Pressure =    601.0000     , Volume =    3.776576
 Pressure =    701.0000     , Volume =    3.237834
 Pressure =    801.0000     , Volume =    2.833611
 Pressure =    901.0000     , Volume =    2.519114
 Pressure =    1001.000     , Volume =    2.267455

C:\book\f95_2003\soln>ideal_gas1
 Enter gas temperature in kelvins:
300
 Pressure =    1.000000     , Volume =    2494.200
 Pressure =    101.0000     , Volume =    24.69505
 Pressure =    201.0000     , Volume =    12.40896
 Pressure =    301.0000     , Volume =    8.286379
 Pressure =    401.0000     , Volume =    6.219950
 Pressure =    501.0000     , Volume =    4.978443
 Pressure =    601.0000     , Volume =    4.150083
 Pressure =    701.0000     , Volume =    3.558060
 Pressure =    801.0000     , Volume =    3.113858
 Pressure =    901.0000     , Volume =    2.768258
 Pressure =    1001.000     , Volume =    2.491708
```

4-33 A program to calculate the pressure of an ideal gas as a function of pressure for a fixed volume is shown below:

```
PROGRAM ideal_gas2
!
!  Purpose:
!    To calculate the pressure of one mole of an ideal gas in
!    a fixed volume of 10 L as a function of temperature.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/06/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Constants
REAL,PARAMETER :: R = 8.314   ! Ideal gas constant (L kPa/mol K)

! List of variables:
```

```
INTEGER :: i              ! Loop index
REAL :: n = 1.0           ! Number of atoms (mol)
REAL :: p                 ! Pressure (kPa)
REAL :: t                 ! Temperature (K)
REAL :: v = 10.           ! Volume (L)

! Calculate the volume as a function pressure
DO i = 250, 400, 50

   ! Get temperature
   t = i

   ! Calculate the volume
   p = n * R * t / v

   ! Write out volume
   WRITE (*,*) 'Temperature = ', t, 'Pressure = ', p

END DO

END PROGRAM ideal_gas2
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln>ideal_gas2
 Temperature =    250.0000     Pressure =    207.8500
 Temperature =    300.0000     Pressure =    249.4200
 Temperature =    350.0000     Pressure =    290.9900
 Temperature =    400.0000     Pressure =    332.5600
```

4-34 A program to calculate the weights required to lift an object with a lever as a function of the length of the lever arm is shown below:

```
PROGRAM lever
!
!  Purpose:
!    To calculate the weights required to lift a load as a
!    function of the length of the lever arm used.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/06/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of variables:
INTEGER :: i              ! Loop index
REAL :: d1                ! Length of lever arm for applied force (m)
REAL :: d2 = 1.0          ! Length of lever arm for load (m)
REAL :: f_app             ! Applied force on lever arm
REAL :: wt = 600          ! Weight to be lifted

! Calculate the required force as a function of lever arn length
DO i = 5, 30
```

```
    ! Length of arm
    d1 = REAL(i) / 10.

    ! Calculate the applied force
    f_app = wt * d2 / d1

    ! Write out applied force
    WRITE (*,*) 'Arm length = ', d1, 'F_app = ', f_app

END DO

END PROGRAM lever
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln>ideal_gas2
 Arm length =   0.5000000     F_app =    1200.000
 Arm length =   0.6000000     F_app =    999.9999
 Arm length =   0.7000000     F_app =    857.1429
 Arm length =   0.8000000     F_app =    750.0000
 Arm length =   0.9000000     F_app =    666.6667
 Arm length =    1.000000     F_app =    600.0000
 Arm length =    1.100000     F_app =    545.4545
 Arm length =    1.200000     F_app =    500.0000
 Arm length =    1.300000     F_app =    461.5385
 Arm length =    1.400000     F_app =    428.5714
 Arm length =    1.500000     F_app =    400.0000
 Arm length =    1.600000     F_app =    375.0000
 Arm length =    1.700000     F_app =    352.9412
 Arm length =    1.800000     F_app =    333.3333
 Arm length =    1.900000     F_app =    315.7895
 Arm length =    2.000000     F_app =    300.0000
 Arm length =    2.100000     F_app =    285.7143
 Arm length =    2.200000     F_app =    272.7273
 Arm length =    2.300000     F_app =    260.8696
 Arm length =    2.400000     F_app =    250.0000
 Arm length =    2.500000     F_app =    240.0000
 Arm length =    2.600000     F_app =    230.7692
 Arm length =    2.700000     F_app =    222.2222
 Arm length =    2.800000     F_app =    214.2857
 Arm length =    2.900000     F_app =    206.8965
 Arm length =    3.000000     F_app =    200.0000
```

If the maximum amount of available weights to apply the force is 400 kg, then the lever arm must be longer than 1.5 m in order to lift the load.

Chapter 5. Basic I/O Concepts

5-1 A format specifies the exact manner in which data should be written out or read into a Fortran program. Formats may be defined in one of three ways: in `FORMAT` statements, in character variables, or in character constants.

5-2 *(a)* Advance to new page and print contents of buffer. *(b)* Advance one line and print contents of buffer. *(c)* Advance two lines and print contents of buffer. *(d)* Do not advance (remain in current line) and print contents of buffer. *(e)* Results undefined—usually the same as having a blank in the control character.

Note: The use of printer control characters is no longer a part of the standard as of Fortran 2003, but it will be supported by all compilers for the indefinite future for backwards compatibility reasons.

5-3 *(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
```

5-4 The result is printed out on the next line. Note that the result of the `ES` format descriptor is easier to interpret than the results of the `E` format descriptor.

```
 1.6020000E-19  .5729578E+02 -.1000000E+01
---------|---------|---------|---------|---------|
        10        20        30        40        50
```

5-5 After these statements are executed, `A`, `B`, and `C` will contain the data shown below. `A` is only five characters long, and it was read using an `A10` descriptor, so the rightmost 5 characters in the field are stored in `A`. `B` is ten characters long, and it was read using an `A10` descriptor, so the entire contents of the field are stored in `B`. `C` is 15 characters long, and it was read using an `A10` descriptor, so the entire contents of the field are stored left-justified in `C`, and the rest of the variable is padded with blanks.

```
A = 'is a '
B = 'test of re'
C = 'ading char     '
     ---------|---------|
             10        20
```

5-6 *(a)* The first READ statement uses all of input line 1 and the first item on line 2. The rest of line 2 is skipped, and the second READ statement begins with the first item of line 3. The contents of the variables after the read statements are: item1 = -300, item2 = -250, item3 = -210, item4 = -160, item5 = -135, item6 = -105, item7 = 17, item8 = 55, item9 = 102, item10 = 165

(b) The first READ statement uses the first four items on of input line 1 and the first two items on line 2. The rest of line 2 is skipped, and the second READ statement begins with the first item of line 3. The contents of the variables after the read statements are: item1 = -300, item2 = -250, item3 = -210, item4 = -160, item5 = -105, item6 = -70, item7 = -17, item8 = 55, item9 = 102, item10 = 165

5-7 A program generating a table of Base-10 logarithms between 1.0 and 10.0 is shown below.

```
PROGRAM logs
!
!  Purpose:
!    To generate a table of the base-10 logarithms between 1.0
!    and 10.0, in steps of 0.1.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/06/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of variables:
REAL :: base     ! Base value for each line of the log table
INTEGER :: i, j  ! Loop indices

! Write out title.
WRITE (*,100)
100 FORMAT ('1',19X,'Table of Base-10 Logarithms Between 1.0 and 10.0',//)

! Write out the column headings.
WRITE (*,110) (j, j=0, 9)
110 FORMAT (6X,10(4X,'0.',I1))
WRITE (*,120)
120 FORMAT (6X,10('-------'))

! Write out the table.
DO i = 1, 9
   base = REAL(i)
   WRITE (*,130) base, (LOG10(base+j/10.), j=0,9)
  130 FORMAT (1X,F5.1,10(2X,F5.3))
END DO

! Write the very last value.
base = 10.
WRITE (*,130) base, LOG10(base)
```

```
END PROGRAM logs
```

The output from this program is shown below.

```
                   Table of Base-10 Logarithms Between 1.0 and 10.0


        0.0    0.1    0.2    0.3    0.4    0.5    0.6    0.7    0.8    0.9
     ----------------------------------------------------------------------
  1.0   .000   .041   .079   .114   .146   .176   .204   .230   .255   .279
  2.0   .301   .322   .342   .362   .380   .398   .415   .431   .447   .462
  3.0   .477   .491   .505   .519   .531   .544   .556   .568   .580   .591
  4.0   .602   .613   .623   .633   .643   .653   .663   .672   .681   .690
  5.0   .699   .708   .716   .724   .732   .740   .748   .756   .763   .771
  6.0   .778   .785   .792   .799   .806   .813   .820   .826   .833   .839
  7.0   .845   .851   .857   .863   .869   .875   .881   .886   .892   .898
  8.0   .903   .908   .914   .919   .924   .929   .934   .940   .944   .949
  9.0   .954   .959   .964   .968   .973   .978   .982   .987   .991   .996
 10.0  1.000
```

5-8 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
!       ====       ==========         =====================
!    05/05/2007    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 ave_sd
```

5-9 The smallest field width *w* that will always be able to display the value of `length` is 11 (i.e., the field should be `F11.4`). The worst case occurs when the number is `-10000.0000`, which requires 11 characters to display.

5-10 The characters will be printed in columns 29 through 36. Although the format descriptor starts writing in column 30, the first character is the control character, so the first character of `'Rubbish!'` is printed out in column 29 of the output file.

5-11 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)`

5-12 In general, the field width *w* necessary to display any real data value in `E` or `ES` format is

$$w \geq d + 7 \tag{5-1}$$

where d is the number of digits to the right of the decimal point. Therefore, the minimum field width to display 6 significant digits is 13 characters (`E13.6`) . The 13 characters are used as follows:

```
±0.ddddddE±ee                (E)
```

Since the `ES` descriptor replaces the zero with a significant digit, it can use a 12-character field (`ES12.6`):

```
±d.dddddE±ee                 (ES)
```

Note that the `ES` descriptor gets an extra significant digit out of the same field width.

5-13 The program to convert time in seconds since the beginning of the day into 24-hour `HH:MM:SS` format is shown below:

```
PROGRAM hhmmss
!
!  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
!       ====       ==========         =====================
!    05/06/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
REAL :: SEC_PER_HOUR = 3600.     ! Seconds per hour
REAL :: SEC_PER_MINUTE = 60.     ! Secinds 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 hhmmss
```

When the program is tested, the results are:

```
C:\book\f95_2003\soln>hhmmss
Enter the number of seconds since the start of day:
1202
 1202.0 seconds =  0:20:02

C:\book\f95_2003\soln>hhmmss
Enter the number of seconds since the start of day:
30000
30000.0 seconds =  8:20:00

C:\book\f95_2003\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
```

5-14 A program to calculate the acceleration due to gravity at various heights above the surface of the Earth is shown below:

```
PROGRAM gravity
!
!  Purpose:
!    To calculate the acceleration due to Earth's gravity
!    for various heights above the surface of the Earth.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/06/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
REAL, PARAMETER :: G = 6.672E-11     ! Gravitational constant
                                     ! (N * m**2 / kg**2)
REAL, PARAMETER :: M_EARTH = 5.98E24 ! Mass of earth (kg)
```

```
REAL, PARAMETER :: R_EARTH = 6371.E3 ! Radius of the earth (m)

! List of variables
REAL :: grav                    ! Gravitational accel (m/sec**2)
REAL :: height                  ! Height above surface (m)
INTEGER :: i                    ! Index variable

! Tell user about program, and set up table headings.
WRITE (*,*) "This program displays the acceleration due to gravity "
WRITE (*,*) "as a function of height above the Earth's surface: "
WRITE (*,1000)
1000 FORMAT (//,'      Height     Acceleration', &
              /,'       (km)       (m/sec**2) ', &
              /,'      ======     ============')

DO i = 0, 40000000, 500000

    ! Get height
    height = REAL(i)

    ! Calculate acceleration
    grav = - G * M_EARTH / ( R_EARTH + height ) ** 2

    ! Write out results
    WRITE (*,'(5X,F7.0,6X,F8.3)') height/1000., grav

END DO

END PROGRAM gravity
```

When the program is tested, the results are:

```
C:\book\f95_2003\soln>gravity
This program displays the acceleration due to gravity
as a function of height above the Earth's surface:


     Height     Acceleration
      (km)       (m/sec**2)
     ======     ============
         0.        -9.830
       500.        -8.451
      1000.        -7.344
      1500.        -6.440
      2000.        -5.694
      2500.        -5.070
      3000.        -4.543
      3500.        -4.095
      4000.        -3.710
      4500.        -3.376
      5000.        -3.086
      5500.        -2.831
      6000.        -2.607
      6500.        -2.408
      7000.        -2.232
      7500.        -2.074
```

8000.	-1.932
8500.	-1.804
9000.	-1.689
9500.	-1.584
10000.	-1.489
10500.	-1.402
11000.	-1.322
11500.	-1.249
12000.	-1.182
12500.	-1.120
13000.	-1.063
13500.	-1.010
14000.	-.961
14500.	-.916
15000.	-.874
15500.	-.834
16000.	-.797
16500.	-.763
17000.	-.730
17500.	-.700
18000.	-.672
18500.	-.645
19000.	-.620
19500.	-.596
20000.	-.574
20500.	-.553
21000.	-.533
21500.	-.514
22000.	-.496
22500.	-.479
23000.	-.463
23500.	-.447
24000.	-.433
24500.	-.419
25000.	-.405
25500.	-.393
26000.	-.381
26500.	-.369
27000.	-.358
27500.	-.348
28000.	-.338
28500.	-.328
29000.	-.319
29500.	-.310
30000.	-.302
30500.	-.293
31000.	-.286
31500.	-.278
32000.	-.271
32500.	-.264
33000.	-.257
33500.	-.251
34000.	-.245
34500.	-.239
35000.	-.233
35500.	-.228

```
36000.        -.222
36500.        -.217
37000.        -.212
37500.        -.207
38000.        -.203
38500.        -.198
39000.        -.194
39500.        -.190
40000.        -.186
```

5-15 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'`.

5-16 Input files should be opened with `ACTION = 'READ'`, since we will be reading data from them, and the choice of `ACTION = 'READ'` will prevent us from accidentally overwriting the input data. Output files should be opened with `ACTION = 'WRITE'`, since we intend to write out data to the files. Scratch files should be opened with `ACTION = 'READWRITE'`, since data be both written to and read from them.

5-17 `CLOSE` statements are not always required in Fortran programs that use disk files. A Fortran program automatically closes all open files when it ends. A file may be closed before the end of the program by using a `CLOSE` statement. If this is done, then the i/o unit that the file was attached to may be reused, and the file is made available for other users sooner.

5-18 The program to perform the specified functions is shown below:

```
PROGRAM ex_5_18
!
!  Purpose:
!    To open two files, and copy all positive values from file
!    1 into file 2.
!
!  Record of revisions:
!      Date       Programmer         Description of change
!      ====       ==========         =====================
!    05/06/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
INTEGER :: istat          ! I/O Status of READs.
INTEGER :: istat1         ! I/O Status of input file OPEN.
INTEGER :: istat2         ! I/O Status of output file OPEN.
REAL :: value             ! Value read from input file.

! Open files.
OPEN ( 98, FILE='input.dat', STATUS='OLD', IOSTAT=istat1 )
OPEN ( 99, FILE='newout.dat', STATUS='NEW', IOSTAT=istat2 )

! Process data if both files opened correctly.
IF ( ( istat1 == 0 ) .AND. ( istat2 == 0 ) ) THEN

   DO
      READ (98, *, IOSTAT=istat ) value
      IF ( istat /= 0 ) EXIT
```

```
      IF ( value > 0. ) THEN
         WRITE (99,*) value
      END IF
   END DO

   ! Close files
   CLOSE (UNIT=98)
   CLOSE (UNIT=99)

ELSE

   ! Open error on files.
   WRITE (*,1000) istat1, istat2
   1000 FORMAT (' Open error: istat1 = ', I6, ' istat2 = ', I6)

END IF

END PROGRAM ex_5_18
```

5-19 A program to read real data from an input file, round off the values, and write them to an output file is shown below:

```
PROGRAM round
!
!  Purpose:
!    To read in real values from a user-specified input file,
!    round them off to the nearest integer, and write the
!    integers out to a user-specified output file.  The program
!    requires that the specified input file already exist, and
!    that the specified output file NOT already exist.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/06/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
CHARACTER(len=36) :: filename1   ! Input file
CHARACTER(len=36) :: filename2   ! Output file
INTEGER :: istat                 ! I/O Status of READs
INTEGER :: istat1                ! I/O Status of input file OPEN
INTEGER :: istat2                ! I/O Status of output file OPEN
REAL :: value                    ! Input value

! Get the name of the file containing the input data.
WRITE (*,*) 'round -- Round values to nearest integer.'
WRITE (*,'(1X,A)') 'Enter the input file name: '
READ (*,'(A36)') filename1

! Get the name of the file to write the output data to.
WRITE (*,'(1X,A)') 'Enter the output file name: '
READ (*,'(A36)') filename2

! Open input data file.  Status is OLD because the input data
! must already exist.
```

```
OPEN ( UNIT=8, FILE=filename1, STATUS='OLD', IOSTAT=istat1 )

! Is open OK?
in_ok: IF ( istat1 /= 0 ) THEN
   WRITE (*,1010) istat1
   1010 FORMAT (1X,'Open failed on input file: iostat = ',I6)
ELSE

   ! Input file opened successfully.  Open output data file.
   ! Status is NEW so that we don't overwrite existing data.
   OPEN ( UNIT=9, FILE=filename2, STATUS='NEW', IOSTAT=istat2 )

   ! Is open OK?
   out_ok: IF ( istat2 /= 0 ) THEN
      WRITE (*,1020) istat2
      1020 FORMAT (1X,'Open failed on output file: iostat = ',I6)
   ELSE

      ! Both files were opened successfully.  Read values from
      ! the input file, round them, and write them into the
      ! output file.
      loop: DO
         READ (8,*,IOSTAT=istat) value
         IF ( istat /= 0 ) EXIT
         WRITE (9,*,IOSTAT=istat) NINT(value)
      END DO loop

      ! Close output file.
      CLOSE (UNIT=9)

   END IF out_ok

   ! Close input file.
   CLOSE (UNIT=8)

END IF in_ok

END PROGRAM round
```

5-20 The program shown below opens a scratch file and writes the numbers 1 to 10 into separate lines in the file. After writing the 10 lines, the file pointer points to just *after* record 10. Then, the program backspaces 6 times, moving the file pointer from just *after* record 10 to just before record 5. The program then reads record 5 into x, and simultaneously moves the file pointer to just before record 6. Next it backspaces 3 times, placing the file pointer just before record 3. The program reads record 3 into y, and simultaneously moves the file pointer to just before record 4. Then it multiplies x and y and displays the results.

```
PROGRAM ex_5_20
!
!  Purpose:
!    This program performs the following functions:
!    1.  Open a scratch file, and write the numbers 1 to
!        10 in the first 10 records of the file.
!    2.  Backspace 6 records in the file.
!    3.  Read the value at that record into x.
!    4.  Backspace 3 records in the file.
!    5.  Read the value at that record into y.
```

```
!    6.  Calculate and display x * y.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/06/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
INTEGER :: i               ! Loop index
INTEGER :: istat           ! I/O Status.
INTEGER :: x               ! First value read from file.
INTEGER :: y               ! Second value read from file.

! Open file.
OPEN ( UNIT=27, STATUS='SCRATCH', IOSTAT=ISTAT )

! Write data to file.
DO i = 1, 10
   WRITE (27,*) i
END DO

! Backspace 6 records in the file.
DO i = 1, 6
   BACKSPACE ( UNIT=27, IOSTAT=istat )
END DO

! Read current record into x.
READ (27,*) x

! Backspace 3 records in the file.
DO i = 1, 3
   BACKSPACE ( UNIT=27, IOSTAT=ISTAT )
END DO
! Read current record into y.
READ (27,*) y

! Calculate and display the product x*y.
WRITE (*,1000) x, y, x*y
1000 FORMAT (' X   = ',I3,/,1X,'Y   = ',I3,/,1X,'X*Y = ',I3)

END PROGRAM ex_5_20
```

When the program executes, the results are:

```
C:\book\f95_2003\soln>ex_5_20
X   =   5
Y   =   3
X*Y =  15
```

5-21 *(a)* These statements are incorrect. The file INFO.DAT is a newly created and empty file, and yet we are trying to read from it. *(b)* These statements are incorrect. It is illegal to name a scratch file. *(c)* These statements are correct. *(d)* These statements are incorrect. Here, we open a file on logical unit `unit`, and then read a new value into `unit`. When we try to close the file, the value of `unit` has changed, so the file close will fail. *(e)* These statements are correct. They will work fine if OUTPUT.DAT does not already exist.

5-22 A program to print a table containing the sine and cosine of θ for θ between 0° and 90°, in 1° increments is shown below:

```
PROGRAM sincos
!
!  Purpose:
!    To generate a table of SIN(theta) and COS(theta) for
!    theta between 0 and 90 degrees in 1-degree increments.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/06/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of named constants:
REAL, PARAMETER :: DEG_2_RADIANS = 0.01745329

! List of variables:
INTEGER :: i                      ! Loop index
REAL :: theta                     ! Angle theta

! Write title.
WRITE (*,1000)
1000 FORMAT (6X,'Table sines and cosines for angles ',/, &
             7X,'     between 0 and 90 degrees',//)

! Write the column headings.
WRITE (*,1010)
1010 FORMAT (T8,'theta',T18,'SIN(theta)',T33,'COS(theta)')
WRITE (*,1020)
1020 FORMAT (7X,5('-------'))

! Write the table.
DO i = 0, 90
   theta = REAL(i)
   WRITE (*,1030) theta, SIN(theta*DEG_2_RADIANS), COS(theta*DEG_2_RADIANS)
   1030 FORMAT (6X,F5.1,5X,F10.7,5X,F10.7)
END DO

END PROGRAM sincos
```

When the program executes, the results are:

```
C:\book\f95_2003\soln \ex5_22>sincos
      Table sines and cosines for angles
            between 0 and 90 degrees


       theta     SIN(theta)     COS(theta)
       -----------------------------------
        0.0      0.0000000      1.0000000
        1.0      0.0174524      0.9998477
        2.0      0.0348995      0.9993908
        3.0      0.0523360      0.9986295
```

4.0	0.0697565	0.9975641
5.0	0.0871557	0.9961947
6.0	0.1045284	0.9945219
7.0	0.1218693	0.9925461
8.0	0.1391731	0.9902681
9.0	0.1564344	0.9876884
10.0	0.1736482	0.9848077
11.0	0.1908090	0.9816272
12.0	0.2079117	0.9781476
13.0	0.2249510	0.9743701
14.0	0.2419219	0.9702957
15.0	0.2588190	0.9659258
16.0	0.2756373	0.9612617
17.0	0.2923717	0.9563048
18.0	0.3090170	0.9510565
19.0	0.3255681	0.9455186
20.0	0.3420201	0.9396926
21.0	0.3583679	0.9335805
22.0	0.3746065	0.9271839
23.0	0.3907311	0.9205049
24.0	0.4067366	0.9135455
25.0	0.4226182	0.9063078
26.0	0.4383711	0.8987941
27.0	0.4539905	0.8910065
28.0	0.4694715	0.8829476
29.0	0.4848096	0.8746197
30.0	0.4999999	0.8660254
31.0	0.5150380	0.8571673
32.0	0.5299192	0.8480482
33.0	0.5446390	0.8386706
34.0	0.5591928	0.8290376
35.0	0.5735764	0.8191521
36.0	0.5877852	0.8090171
37.0	0.6018150	0.7986355
38.0	0.6156614	0.7880108
39.0	0.6293203	0.7771460
40.0	0.6427876	0.7660445
41.0	0.6560590	0.7547097
42.0	0.6691306	0.7431449
43.0	0.6819983	0.7313538
44.0	0.6946583	0.7193398
45.0	0.7071067	0.7071068
46.0	0.7193397	0.6946585
47.0	0.7313536	0.6819984
48.0	0.7431448	0.6691307
49.0	0.7547095	0.6560591
50.0	0.7660444	0.6427877
51.0	0.7771459	0.6293204
52.0	0.7880107	0.6156616
53.0	0.7986354	0.6018151
54.0	0.8090169	0.5877854
55.0	0.8191520	0.5735765
56.0	0.8290375	0.5591930
57.0	0.8386705	0.5446391
58.0	0.8480480	0.5299194
59.0	0.8571672	0.5150382

```
        60.0      0.8660253      0.5000001
        61.0      0.8746197      0.4848097
        62.0      0.8829476      0.4694717
        63.0      0.8910065      0.4539906
        64.0      0.8987940      0.4383713
        65.0      0.9063078      0.4226184
        66.0      0.9135454      0.4067368
        67.0      0.9205048      0.3907312
        68.0      0.9271838      0.3746067
        69.0      0.9335804      0.3583681
        70.0      0.9396926      0.3420203
        71.0      0.9455186      0.3255683
        72.0      0.9510565      0.3090171
        73.0      0.9563047      0.2923718
        74.0      0.9612616      0.2756375
        75.0      0.9659258      0.2588192
        76.0      0.9702957      0.2419220
        77.0      0.9743700      0.2249512
        78.0      0.9781476      0.2079118
        79.0      0.9816272      0.1908091
        80.0      0.9848077      0.1736483
        81.0      0.9876883      0.1564346
        82.0      0.9902681      0.1391733
        83.0      0.9925461      0.1218695
        84.0      0.9945219      0.1045286
        85.0      0.9961947      0.0871559
        86.0      0.9975640      0.0697566
        87.0      0.9986295      0.0523361
        88.0      0.9993908      0.0348997
        89.0      0.9998477      0.0174526
        90.0      1.0000000      0.0000002
```

5-23 A program to calculate the speed of a ball as a function of the distance fallen is shown below:

```
PROGRAM falling_speed
!
!  Purpose:
!    To generate a table of the speed of a ball falling from
!    rest as a function of the distance the ball has fallen.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/06/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of constants
REAL,PARAMETER :: G = 9.81       ! Accel due to gravity (m/s**2)

! List of variables:
INTEGER :: i                     ! Loop index
REAL :: dh                       ! Delta h (m)

! Write title.
WRITE (*,1000)
```

```
1000 FORMAT (3X,'Table of Ball Speed vs Distance Fallen')

! Write the column headings.
WRITE (*,1010)
1010 FORMAT (T8,'Distance (m)',T25,'Speed (m/s)')
WRITE (*,1020)
1020 FORMAT (7X,5('======'))

! Write the table.
DO i = 0, 200, 10
   dh = REAL(i)
   WRITE (*,1030) dh, SQRT(2 * G * dh)
   1030 FORMAT (10X,F5.1,6X,F10.1)
END DO

END PROGRAM falling_speed
```

When the program executes, the results are:

```
C:\book\f95_2003\soln\ex5_23> falling_speed
   Table of Ball Speed vs Distance Fallen
       Distance (m)     Speed (m/s)
       ==============================
            0.0             0.0
           10.0            14.0
           20.0            19.8
           30.0            24.3
           40.0            28.0
           50.0            31.3
           60.0            34.3
           70.0            37.1
           80.0            39.6
           90.0            42.0
          100.0            44.3
          110.0            46.5
          120.0            48.5
          130.0            50.5
          140.0            52.4
          150.0            54.2
          160.0            56.0
          170.0            57.8
          180.0            59.4
          190.0            61.1
          200.0            62.6
```

5-24 A program to calculate the potential energy, kinetic energy, and total energy of a falling ball is shown below:

```
PROGRAM pe_ke
!
!  Purpose:
!    To generate a table of the total potential and kinetic
!    energy of a ball as it falls from a height of 100 m to
!    the ground.
!
!  Record of revisions:
!       Date       Programmer         Description of change
```

```
!       ====       ==========         ====================
!    05/06/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of constants
REAL,PARAMETER :: G = 9.81       ! Accel due to gravity (m/s**2)

! List of variables:
INTEGER :: i                     ! Loop index
REAL :: dh                       ! Delta h (m)
REAL :: height                   ! Height (m)
REAL :: ke                       ! Kinetic energy (J)
REAL :: m = 1.0                  ! Mass (kg)
REAL :: pe                       ! Potential energy (J)
REAL :: total_energy             ! Total energy (J)
REAL :: v                        ! velocity of the ball (m/s)

! Write title.
WRITE (*,1000)
1000 FORMAT (10X,'Table of PE, KE, and Total Energy vs Height')

! Write the column headings.
WRITE (*,1010)
1010 FORMAT (T8,'Height (m)',T24,'PE (J)',T38,'KE (J)',T49,'Total (J)')
WRITE (*,1020)
1020 FORMAT (7X,5('=========='))

! Write the table.
DO i = 100, 0, -10

   ! Get height
   height = REAL(i)

   ! Calculate ball speed
   dh = 100 - height
   v = SQRT(2 * G * dh)

   ! Get potential energy
   pe = m * G * height

   ! Get kinetic energy
   ke = 0.5 * m * v**2

   ! Get total energy
   total_energy = pe + ke


   WRITE (*,1030) height, pe, ke, total_energy
   1030 FORMAT (10X,F5.1,6X,F8.1,6X,F8.1,6X,F8.1)
END DO

END PROGRAM pe_ke
```

When the program executes, the results are:

```
C:\book\f95_2003\soln\ex5_24> pe_ke
          Table of PE, KE, and Total Energy vs Height
       Height (m)      PE (J)        KE (J)     Total (J)
       ==================================================
          100.0         981.0           0.0         981.0
           90.0         882.9          98.1         981.0
           80.0         784.8         196.2         981.0
           70.0         686.7         294.3         981.0
           60.0         588.6         392.4         981.0
           50.0         490.5         490.5         981.0
           40.0         392.4         588.6         981.0
           30.0         294.3         686.7         981.0
           20.0         196.2         784.8         981.0
           10.0          98.1         882.9         981.0
            0.0           0.0         981.0         981.0
```

5-25 A program to calculate the future value of an account based on a given present value and annual interest rate is shown below:

```
PROGRAM compound
!
!  Purpose:
!    To calculate the value of an account that compounds
!    monthly with a given annual interest rate.  The program
!    will prompt the user for the present value and apr, and
!    will calculate a table containing the value of the account
!    for the next 48 months.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/06/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

!  List of variables:
REAL :: apr                  ! Annual percentage rate
REAL :: future_value         ! Future value of account
INTEGER :: i                 ! Index variable
REAL :: present_value        ! Present value of account

! Write out title.
WRITE (*,1000)
1000 FORMAT (1X,'Program to calculate the future value of a ', &
                'bank account for the next 60 months.',/, &
             1X,'This program assumes that interest is ', &
                'compounded monthly on the account.')
! Get present value of account.
WRITE (*,1010)
1010 FORMAT (1X,'Enter present value of account:')
READ (*,*) present_value

! Get interest rate.
WRITE (*,1020)
1020 FORMAT (1X,'Enter annual interest rate (apr, in %):')
READ (*,*) apr
```

```
! Write out title.
WRITE (*,1030) present_value, apr
1030 FORMAT (//,7X,'Table of Future Values for an Account with a',/,&
                7X,'Present Value of $',F9.2,' and a ',F5.2,'% apr')

!     Write the column headings.
WRITE (*,1040)
1040 FORMAT ('0',T22,'Month',T35,'Value')
WRITE (*,1050)
1050 FORMAT (20X,3('-------'))

!     Write the table.
DO i = 0, 48
    future_value = present_value * (1.0 + apr/1200.0)**i
     WRITE (*,1060) i, future_value
    1060 FORMAT (21X,I3,6X,F10.2)
END DO

END PROGRAM compound
```

An example compounding table is shown below:

```
C:\book\f95_2003\soln\ex5_25>compound
Program to calculate the future value of a bank account for th

 This program assumes that interest is compounded monthly on th
 Enter present value of account:
1000.00
 Enter annual interest rate (apr, in %):
7.75

       Table of Future Values for an Account with a
       Present Value of $  1000.00 and a  7.75% apr
0                    Month        Value
                    ---------------------
                      0         1000.00
                      1         1006.46
                      2         1012.96
                      3         1019.50
                      4         1026.08
                      5         1032.71
                      6         1039.38
                      7         1046.09
                      8         1052.85
                      9         1059.65
                     10         1066.49
                     11         1073.38
                     12         1080.31
                     13         1087.29
                     14         1094.31
                     15         1101.38
                     16         1108.49
                     17         1115.65
                     18         1122.86
                     19         1130.11
```

20	1137.41
21	1144.75
22	1152.14
23	1159.59
24	1167.07
25	1174.61
26	1182.20
27	1189.83
28	1197.52
29	1205.25
30	1213.04
31	1220.87
32	1228.75
33	1236.69
34	1244.68
35	1252.71
36	1260.81
37	1268.95
38	1277.14
39	1285.39
40	1293.69
41	1302.05
42	1310.46
43	1318.92
44	1327.44
45	1336.01
46	1344.64
47	1353.32
48	1362.06

5-26 This program needs to open an input data file and read from it until the last data sample has been reached. It should compare each data point with the largest and smallest previous values, and update the extreme values if the current point is a new high or a new low. (Note that the input file should be opened with STATUS = 'OLD' since it must already exist if it is to contain input data.)

```
PROGRAM minmax
!
!  Purpose:
!    To find the minimum and maximum values in an input data
!    file.  The file will contain an arbitrary number of real
!    values, arranged with one value per record.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/06/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER, PARAMETER :: IN1 = 99      ! Unit for file i/o

! List of variables:
CHARACTER(len=24) :: filename       ! input file name
INTEGER :: error                    ! Error flag
REAL :: maxval                      ! Maximum value found
```

```
REAL :: minval                      ! Minimum value found
REAL :: x                           ! An input value

! Prompt user and get the name of the input file.
WRITE (*,1000)
1000 FORMAT (1X,'This program finds the minimum and maximum values ',/,&
             1X,'in an input data set.  Enter the name of the file ',/,&
             1X,'containing the input data:' )
READ (*,'(A)') filename

! Open the input file
OPEN (UNIT=IN1, FILE=filename, STATUS='OLD', IOSTAT=error )

! Check to see of the OPEN failed.
openok: IF ( error > 0 ) THEN
    WRITE (*,1020) filename, error
    1020 FORMAT (1X,'ERROR: Open error on file ',A,': IOSTAT = ',I6)

ELSE

   ! Read the first data value from the input file.
   READ (IN1,*,IOSTAT=error) x

   ! If the first read is successful, initialize minval and maxval.
   IF ( error == 0 ) THEN
      minval = x
      maxval = x
   END IF

   ! Process remaining values
   loop: DO
      READ (IN1,*,IOSTAT=error) x
      IF ( error /= 0 ) EXIT

      ! Check for new minima and maxima.
      minval = MIN ( minval, x )
      maxval = MAX ( maxval, x )
   END DO loop

   ! Write out the minimum and maximum values in the input data set.
   WRITE (*,1030) minval
   1030 FORMAT (1X,'The minimum value in the file is ', ES13.6, '.')
   WRITE (*,1040) maxval
   1040 FORMAT (1X,'The maximum value in the file is ', ES13.6, '.')

   ! Close input file, and quit.
   CLOSE (UNIT=IN1)

END IF openok

END PROGRAM minmax
```

To test this program, we will create an input file `input_file` containing the following data

```
0
30000
```

```
-5
9
2
-4400
7
```

An example compounding table is shown below:

```
C:\book\f95_2003\soln\ex5_26>minmax
 This program finds the minimum and maximum values
 in an input data set.  Enter the name of the file
 containing the input data:
input_file
 The minimum value in the file is -4.400000E+03.
 The maximum value in the file is  3.000000E+04.
```

5-27 A program to calculate the average (arithmetic mean), rms average, geometric mean, and harmonic mean of an input data set contained in a user-specified file is shown below:

```
PROGRAM all_means
!
!  Purpose:
!    To calculate the average (arithmetic mean), rms average,
!    geometric mean, and harmonic mean of an input data set,
!    where each input value can be positive, negative, or zero.
!    This program reads the input data from a file.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/07/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of variables:
REAL :: ave            ! Average (arithmetic mean)
CHARACTER(len=24) :: filename ! input file name
REAL :: g_mean         ! Geometric mean
REAL :: h_mean         ! Harmonic mean
INTEGER :: n = 0       ! Number of input values
REAL :: prod_x = 1.0   ! Product of the input values
REAL :: rms            ! Rms average
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 :: sum_rx = 0.0   ! Sum of reciprocal of input values
REAL :: x = 0.0        ! Input value

! Prompt user and get the name of the input file.
WRITE (*,1000)
1000 FORMAT (1X,'This program calculates the average (arithmetic mean),',/,&
             1X,'geometric mean, harmonic mean, and rms average of an ',/,&
             1X,'input data set.  Enter name of file containing the ',/,&
             1X,'input data:' )
READ (*,'(A)') filename
```

```
! Open the input file
OPEN (UNIT=10, FILE=filename, STATUS='OLD', IOSTAT=status )

! Check to see of the OPEN failed.
openok: IF ( status > 0 ) THEN
    WRITE (*,1020) filename, status
    1020 FORMAT (1X,'ERROR: Open error on file ',A,': IOSTAT = ',I6)

ELSE

   ! Read the values.
   loop: DO
      READ (10,*,IOSTAT=status) x
      IF ( status /= 0 ) EXIT

      ! Accumulate sums.
      n = n + 1
      prod_x = prod_x * x
      sum_x  = sum_x + x
      sum_x2 = sum_x2 + x**2
      sum_rx = sum_rx + 1.0 / x

   END DO loop

   ! Calculate the means
   ave    = sum_x / REAL(n)
   g_mean = prod_x ** ( 1. / REAL(n) )
   h_mean = REAL(n) / sum_rx
   rms    = SQRT ( sum_x2 / REAL(n) )

   ! Tell user.
   WRITE (*,1030) 'The average of this data set is:      ', ave
   WRITE (*,1030) 'The geometric mean is:                ', g_mean
   WRITE (*,1030) 'The harmonic mean of this data set is:', h_mean
   WRITE (*,1030) 'The rms average of this data set is:  ', rms
   WRITE (*,1040) 'The number of data points is:         ', n
   1030 FORMAT (1X,A,F10.4)
   1040 FORMAT (1X,A,I10)

END IF openok

END PROGRAM all_means
```

If we place the data values 1.0, 2.0, 5.0, 4.0, 3.0, 2.1, 4.7, and 3.0 into file INPUT.DAT and run the program on that file, the results are

```
C:\book\f95_2003\soln\ex5_27>all_means
This program calculates the average (arithmetic mean),
geometric mean, harmonic mean, and rms average of an
input data set.  Enter name of file containing the
input data:
input.dat
The average of this data set is:          3.1000
The geometric mean is:                    2.7786
The harmonic mean of this data set is:    2.4201
The rms average of this data set is:      3.3634
```

```
The number of data points is:                 8
```

5-28 A program that converts angles in radians to degrees, minutes, and seconds is shown below:

```
PROGRAM dms
!
!  Purpose:
!    To read angles in radians from a disk file, and convert
!    them into degrees, minutes, and seconds.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/07/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
REAL, PARAMETER :: RAD_2_DEG = 57.2957795  ! Radians to degrees

! List of variables:
REAL :: angle_rad                  ! Input angle (radians)
REAL :: angle_deg                  ! Input angle (degrees)
CHARACTER(len=24) :: filename      ! input file name
INTEGER :: degrees                 ! Degrees
INTEGER :: error                   ! Error flag
INTEGER :: minutes                 ! Minutes
REAL :: residual                   ! Residual part of angle
INTEGER :: seconds                 ! Seconds

! Prompt user and get the name of the input file.
WRITE (*,1000)
1000 FORMAT (1X,'This program reads angles in radians from a disk file ',/,&
             1X,'and writes them out in degrees, minutes, and seconds. ',/,&
             1X,'Enter file name containing the input data:' )
READ (*,'(A)') filename

! Open the input file
OPEN (UNIT=20, FILE=filename, STATUS='OLD', IOSTAT=error )

! Check to see of the OPEN failed.
openok: IF ( error > 0 ) THEN
    WRITE (*,1020) filename, error
    1020 FORMAT (1X,'ERROR: Open error on file ',A,': IOSTAT = ',I6)

ELSE

   ! Read angle from file, and convert it
   loop: DO
      READ (20,*,IOSTAT=error) angle_rad
      IF ( error /= 0 ) EXIT

      ! Convert to degrees
      angle_deg = angle_rad * RAD_2_DEG

      ! Get degrees, minutes, and seconds
```

```
      degrees = INT(angle_deg)
      residual = angle_deg - REAL(degrees)
      minutes = INT(residual * 60. )
      residual = residual - REAL(minutes) / 60.
      seconds = NINT(residual * 3600. )

      ! Tell user
      WRITE (*,1030) angle_rad, degrees, minutes, seconds
      1030 FORMAT (1X,F10.6,' radians = ',I3,' deg ',I2.2,' min ',I2.2,' sec')

   END DO loop

   ! Close file
   CLOSE (UNIT=20)

END IF openok

END PROGRAM dms
```

When the specified values are placed in file `in.dat` and the program is executed, the results are:

```
C:\book\f95_2003\soln>dms
This program reads angles in radians from a disk file
and writes them out in degrees, minutes, and seconds.
Enter file name containing the input data:
in.dat
   .000000 radians =   0 deg 00 min 00 sec
  1.000000 radians =  57 deg 17 min 45 sec
  3.141593 radians = 180 deg 00 min 00 sec
  6.000000 radians = 343 deg 46 min 29 sec
```

5-29 The least-squares fitting program will fail with a divide by zero error if the number of data points is less than 2. A program which avoids this problem is shown below:

```
PROGRAM least_squares_fit
!
!  Purpose:
!    To perform a least-squares fit of an input data set
!    to a straight line, and print out the resulting slope
!    and intercept values.  The input data for this fit
!    comes from a user-specified input data file.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    11/19/06    S. J. Chapman        Original code
! 1. 05/07/07    S. J. Chapman        Modified to avoid divide
!                                     by 0 errors for < 2 points
!
IMPLICIT NONE

! List of parameters:
INTEGER, PARAMETER :: LU = 18 ! Unit for disk I/O

! List of variables.  Note that cumulative variables are all
! initialized to zero.
```

```
CHARACTER(len=24) :: filename ! Input file name (<= 24 chars)
INTEGER :: ierror             ! Status flag from I/O statements
INTEGER :: n = 0              ! Number of input data pairs (x,y)
REAL :: slope                 ! Slope of the line
REAL :: sum_x = 0.            ! Sum of all input X values
REAL :: sum_x2 = 0.           ! Sum of all input X values squared
REAL :: sum_xy = 0.           ! Sum of all input X*Y values
REAL :: sum_y = 0.            ! Sum of all input Y values
REAL :: x                     ! An input X value
REAL :: x_bar                 ! Average X value
REAL :: y                     ! An input Y value
REAL :: y_bar                 ! Average Y value
REAL :: y_int                 ! Y-axis intercept of the line

! Prompt user and get the name of the input file.
WRITE (*,1000)
1000 FORMAT (1X,'This program performs a least-squares fit of an ',/, &
             1X,'input data set to a straight line. Enter the name',/ &
             1X,'of the file containing the input (x,y) pairs:  ' )
READ (*,'(A)') filename

! Open the input file
OPEN (UNIT=LU, FILE=filename, STATUS='OLD', ACTION='READ', &
      IOSTAT=ierror )

! Check to see of the OPEN failed.
errorcheck: IF ( ierror > 0 ) THEN

   WRITE (*,1020) filename
   1020 FORMAT (1X,'ERROR: File ',A,' does not exist!')

ELSE

   ! File opened successfully. Read the (x,y) pairs from
   ! the input file.
   loop: DO
      READ (LU,*,IOSTAT=ierror) x, y   ! Get pair
      IF ( ierror /= 0 ) EXIT
      n      = n + 1                   !
      sum_x  = sum_x + x               ! Calculate
      sum_y  = sum_y + y               !   statistics
      sum_x2 = sum_x2 + x**2           !
      sum_xy = sum_xy + x * y          !
   END DO loop

   ! Now calculate the slope and intercept if enough data
   ! is available.
   enough: IF ( n > 1 ) THEN
      x_bar = sum_x / real(n)
      y_bar = sum_y / real(n)
      slope = (sum_xy - sum_x * y_bar) / ( sum_x2 - sum_x * x_bar)
      y_int = y_bar - slope * x_bar

      ! Tell user.
      WRITE (*, 1030 ) slope, y_int, N
      1030 FORMAT ('0','Regression coefficients for the least-squares line:',&
```

```
             /,1X,'  slope (m)     = ', F12.3,&
             /,1X,'  Intercept (b) = ', F12.3,&
             /,1X,'  No of points  = ', I12 )

   ELSE enough

      ! Tell user not enough data
      WRITE (*,1040)
      1040 FORMAT (' ERROR--at least 2 input data points required.')

   END IF enough

   ! Close input file, and quit.
   CLOSE (UNIT=LU)

END IF errorcheck

END PROGRAM least_squares_fit
```

5-30 A modified version of the ideal gas law program with neat output is shown below:

```
PROGRAM ideal_gas2
!
!  Purpose:
!    To calculate the volume of one mole of an ideal gas as
!    pressure is varied from 1 to 1001 kPa in steps of 100 kPa.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/06/2007    S. J. Chapman       Original code
! 1. 05/08/2007    S. J. Chapman       Modified to create neat output
!
IMPLICIT NONE

! Constants
REAL,PARAMETER :: R = 8.314   ! Ideal gas constant (L kPa/mol K)

! List of variables:
INTEGER :: i             ! Loop index
REAL :: n = 1.0          ! Number of atoms (mol)
REAL :: p                ! Pressure (kPa)
REAL :: t                ! Temperature (K)
REAL :: v                ! volume (L)

! Get temperature
WRITE (*,*) 'Enter gas temperature in kelvins:'
READ (*,*) t

! Write headings
WRITE (*,*) '     Pressure (kPa)    Volume (L)    '
WRITE (*,*) '   ================================  '

! Calculate the volume as a function pressure
DO i = 1, 1001, 100
```

```
   ! Get pressure
   p = i

   ! Calculate the volume
   v = n * R * t / p

   ! Write out volume
   WRITE (*,'(8X,F7.1,10X,F8.2)') p, v

END DO

END PROGRAM ideal_gas2
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln>ideal_gas2
 Enter gas temperature in kelvins:
273
      Pressure (kPa)    Volume (L)
    ================================
           1.0          2269.72
         101.0            22.47
         201.0            11.29
         301.0             7.54
         401.0             5.66
         501.0             4.53
         601.0             3.78
         701.0             3.24
         801.0             2.83
         901.0             2.52
        1001.0             2.27
```

5-31 A program to print out the gain of a microwave antenna as a function of the angle with respect to the antenna boresight is shown below:

```
PROGRAM antenna_gain
!
!  Purpose:
!    To calculate and print out the gain of a microwave antenna
!    as a function of the angle from the antenna boresight.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/08/2003    S. J. Chapman      Original code
!
IMPLICIT NONE

! Named constants
REAL, PARAMETER :: DEG_2_RAD = 0.01745329  ! Degrees to radians

! List of variables.
REAL :: arg                   ! Argument of sinc function
REAL :: gain                  ! Antenna gain
INTEGER :: i                  ! Loop index
REAL :: theta                 ! Angle from boresight (deg)
```

```
! Create title and headings
WRITE (*,'(/,2X,A,/)') 'Antenna Gain vs Angle (deg)'
WRITE (*,*) '   Angle (deg)       Gain    '
WRITE (*,*) '  ========================== '

! Calculate gain
DO i = 0, 90

   ! Get angle in degrees
   theta = i

   ! Calculate gain
   arg = 6. * theta * DEG_2_RAD
   IF ( arg /= 0. ) THEN
      gain = SIN( arg ) / arg
   ELSE
      gain = 1.0
   END IF

   ! Write out gain
   WRITE (*,'(5X,F6.1,7X,F9.5)') theta, gain

END DO

END PROGRAM antenna_gain
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln>antenna_gain

  Antenna Gain vs Angle (deg)

    Angle (deg)       Gain
   ==========================
        0.0         1.00000
        1.0         0.99817
        2.0         0.99271
        3.0         0.98363
        4.0         0.97101
        5.0         0.95493
        6.0         0.93549
        7.0         0.91282
        8.0         0.88706
        9.0         0.85839
       10.0         0.82699
       11.0         0.79307
       12.0         0.75683
       13.0         0.71851
       14.0         0.67836
       15.0         0.63662
       16.0         0.59356
       17.0         0.54945
       18.0         0.50455
       19.0         0.45914
       20.0         0.41350
```

21.0	0.36788
22.0	0.32257
23.0	0.27781
24.0	0.23387
25.0	0.19099
26.0	0.14939
27.0	0.10929
28.0	0.07091
29.0	0.03442
30.0	0.00000
31.0	-0.03220
32.0	-0.06204
33.0	-0.08942
34.0	-0.11424
35.0	-0.13642
36.0	-0.15591
37.0	-0.17270
38.0	-0.18675
39.0	-0.19809
40.0	-0.20675
41.0	-0.21277
42.0	-0.21624
43.0	-0.21722
44.0	-0.21584
45.0	-0.21221
46.0	-0.20646
47.0	-0.19874
48.0	-0.18921
49.0	-0.17804
50.0	-0.16540
51.0	-0.15148
52.0	-0.13647
53.0	-0.12056
54.0	-0.10394
55.0	-0.08681
56.0	-0.06936
57.0	-0.05177
58.0	-0.03423
59.0	-0.01692
60.0	0.00000
61.0	0.01636
62.0	0.03202
63.0	0.04684
64.0	0.06069
65.0	0.07346
66.0	0.08504
67.0	0.09537
68.0	0.10436
69.0	0.11196
70.0	0.11814
71.0	0.12287
72.0	0.12614
73.0	0.12795
74.0	0.12834
75.0	0.12732
76.0	0.12496

```
      77.0         0.12131
      78.0         0.11643
      79.0         0.11043
      80.0         0.10337
      81.0         0.09538
      82.0         0.08654
      83.0         0.07698
      84.0         0.06682
      85.0         0.05617
      86.0         0.04516
      87.0         0.03392
      88.0         0.02256
      89.0         0.01122
      90.0         0.00000
```

5-32 A program to calculate the torque, speed, and power from a motor as it starts up is shown below.

```
PROGRAM motor
!
!  Purpose:
!    To calculate the torque, speed, and power from a motor
!    as it starts up.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/08/2003    S. J. Chapman       Original code
!
IMPLICIT NONE

! Declare variables:
INTEGER :: i                       ! Loop index
REAL :: power                      ! Output power (W)
REAL :: speed                      ! Speed (rad/s)
REAL :: time                       ! Time (s)
REAL :: torque                     ! Torque (N-m)

! Set up headings for table
WRITE (*,'(/,10X,A,/)') 'Speed, Torque, and Power vs Time'
WRITE (*,*) '      Time       Speed      Torque       Power '
WRITE (*,*) '       (s)       (m/s)       (N-m)        (W)  '
WRITE (*,*) '  =============================================== '

! Now calculate the minimum and maximum distances as a function
! of angle around the orbit.
DO i = 0, 40

   ! Get current time
   time = REAL(i) / 4.0

   ! Get speed (rad/s)
   speed = 188.5 * ( 1 - EXP(-0.2*time) )

   ! Get torque (N-m)
   torque = 10.0 * EXP(-0.2*time)
```

```
   ! Get power (W)
   power = speed * torque

   ! Print out the results
   WRITE (*,'(5X,F6.2,5X,F7.1,5X,F7.2,5X,F7.1)') time, speed, torque, power

END DO

END PROGRAM motor
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln>motor

          Speed, Torque, and Power vs Time

       Time        Speed        Torque        Power
        (s)        (m/s)         (N-m)         (W)
   ===============================================
       0.00          0.0         10.00          0.0
       0.25          9.2          9.51         87.4
       0.50         17.9          9.05        162.3
       0.75         26.3          8.61        226.0
       1.00         34.2          8.19        279.8
       1.25         41.7          7.79        324.7
       1.50         48.9          7.41        361.9
       1.75         55.7          7.05        392.3
       2.00         62.1          6.70        416.6
       2.25         68.3          6.38        435.5
       2.50         74.2          6.07        449.9
       2.75         79.7          5.77        460.1
       3.00         85.0          5.49        466.8
       3.25         90.1          5.22        470.3
       3.50         94.9          4.97        471.2
       3.75         99.5          4.72        469.8
       4.00        103.8          4.49        466.4
       4.25        107.9          4.27        461.3
       4.50        111.9          4.07        454.8
       4.75        115.6          3.87        447.1
       5.00        119.2          3.68        438.3
       5.25        122.5          3.50        428.8
       5.50        125.8          3.33        418.6
       5.75        128.8          3.17        407.9
       6.00        131.7          3.01        396.7
       6.25        134.5          2.87        385.3
       6.50        137.1          2.73        373.7
       6.75        139.6          2.59        362.0
       7.00        142.0          2.47        350.2
       7.25        144.3          2.35        338.4
       7.50        146.4          2.23        326.8
       7.75        148.5          2.12        315.2
       8.00        150.4          2.02        303.7
       8.25        152.3          1.92        292.5
       8.50        154.1          1.83        281.4
       8.75        155.7          1.74        270.6
       9.00        157.3          1.65        260.1
```

```
        9.25        158.9        1.57       249.8
        9.50        160.3        1.50       239.8
        9.75        161.7        1.42       230.0
       10.00        163.0        1.35       220.6
```

5-33 This program is almost identical to the Exercise 4-15, except that here we will format the results in a neat table.

```
PROGRAM orbit
!
!  Purpose:
!    To calculate the distance r from the center of the
!    Earth to a satellite in orbit, as a function of
!    the orbit's eccentricity and the size parameter p.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/04/2007    S. J. Chapman      Original code
! 1. 05/08/2007    S. J. Chapman      Modified for neat output
!
IMPLICIT NONE

! Declare named constants:
REAL, PARAMETER :: DEG_2_RAD = 0.01745329 ! Degrees to radians

! Declare variables:
REAL :: ecc                        ! Eccentricity (0-1)
REAL :: p                          ! Size parameter (m)
REAL :: r                          ! Distance from centre of Earth to orbit
REAL :: theta                      ! Angle in orbit (deg)
INTEGER :: i                       ! Loop index

! Get size parameter
WRITE (*,*) 'Enter size parameter (m): '
READ (*,*) p

! Get eccentricity
WRITE (*,*) 'Enter eccentricity (0-1): '
READ (*,*) ecc

! Set up headings for table
WRITE (*,*) '   Angle (deg)    Range (m)  '
WRITE (*,*) '  ========================== '

! Now calculate the minimum and maximum distances as a function
! of angle around the orbit.
DO i = 0, 360, 30

   ! Get angle theta (deg)
   theta = i

   ! Get range at this angle
   r = p / ( 1 - ecc * COS(theta * DEG_2_RAD) )

   ! Print out the results
   WRITE (*,'(5X,F7.1,5X,F10.1)') theta, r
```

```
END DO

END PROGRAM orbit
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln\ex5_33>orbit
Enter size parameter (m):
10000000
Enter eccentricity (0-1):
0
    Angle (deg)    Range (m)
   ==========================
         0.0     10000000.0
        30.0     10000000.0
        60.0     10000000.0
        90.0     10000000.0
       120.0     10000000.0
       150.0     10000000.0
       180.0     10000000.0
       210.0     10000000.0
       240.0     10000000.0
       270.0     10000000.0
       300.0     10000000.0
       330.0     10000000.0
       360.0     10000000.0

C:\book\f95_2003\soln\ex5_33>orbit
 Enter size parameter (m):
10000000
 Enter eccentricity (0-1):
0.25
    Angle (deg)    Range (m)
   ==========================
         0.0     13333333.0
        30.0     12763345.0
        60.0     11428572.0
        90.0     10000000.0
       120.0      8888889.0
       150.0      8220261.5
       180.0      8000000.0
       210.0      8220261.0
       240.0      8888888.0
       270.0      9999999.0
       300.0     11428570.0
       330.0     12763344.0
       360.0     13333333.0

C:\book\f95_2003\soln\ex5_33>orbit
 Enter size parameter (m):
10000000
 Enter eccentricity (0-1):
0.5
    Angle (deg)    Range (m)
   ==========================
```

```
  0.0     20000000.0
 30.0     17637080.0
 60.0     13333334.0
 90.0     10000001.0
120.0      8000000.5
150.0      6978305.5
180.0      6666666.5
210.0      6978304.5
240.0      7999998.5
270.0      9999997.0
300.0     13333329.0
330.0     17637074.0
360.0     20000000.0
```

5-34 The following program calculates the apogee and perigee of a satellite orbit as a function of semi-major axis and eccentricity. Note that the height above the surface of the Earth is given by

$$h - \frac{p}{1-\varepsilon\cos\theta} - R$$

The apogee will occur when $\cos\theta$ is 1.0, and the perigee will occur when $\cos\theta$ is –1.0. The program takes advantage of this fact to calculate the apogee and perigee. (*Note:* The value for *p* was incorrectly given to be 1000 km in the first printing; it should be 10,000 km.)

```
PROGRAM apogee_and_perigee
!
!  Purpose:
!    To calculate the apogee and perigee of a satellite
!    orbit.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/08/2003    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare named constants:
REAL, PARAMETER :: DEG_2_RAD = 0.01745329 ! Degrees to radians
REAL, PARAMETER :: EARTH_RADIUS = 6.371E6 ! Reart radius (m)

! Declare variables:
REAL :: apogee                    ! Apogee (m)
REAL :: ecc                       ! Eccentricity (0-1)
INTEGER :: i                      ! Loop index
REAL :: p                         ! Size parameter (m)
REAL :: perigee                   ! Perigee (m)
REAL :: r                         ! Distance from centre of Earth to orbit
REAL :: theta                     ! Angle in orbit (deg)

! Get size parameter
WRITE (*,*) 'Enter size parameter (m): '
READ (*,*) p

! Set up headings for table
WRITE (*,*) '       Ecc       Apogee (m)     Perigee (m) '
```

```
WRITE (*,*) '  =========================================='

! Now calculate the apogee and perigee as a function of eccentricity.
DO i = 0, 50, 5

   ! Get eccentricity
   ecc = REAL(i) / 100.0

   ! Get range at apogee
   apogee = p / ( 1 - ecc ) - EARTH_RADIUS

   ! Get range at perigee
   perigee = p / ( 1 + ecc ) - EARTH_RADIUS

   ! Print out the results
   WRITE (*,'(5X,F7.2,5X,F10.1,5X,F10.1)') ecc, apogee, perigee

END DO

END PROGRAM apogee_and_perigee
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln>apogee_and_perigee
 Enter size parameter (m):
10000000
        Ecc       Apogee (m)     Perigee (m)
   ==========================================
        0.00      3629000.0      3629000.0
        0.05      4155315.8      3152809.5
        0.10      4740111.0      2719909.0
        0.15      5393706.0      2324652.2
        0.20      6129000.0      1962333.4
        0.25      6962333.5      1629000.0
        0.30      7914714.5      1321307.8
        0.35      9013615.0      1036407.4
        0.40     10295667.0       771857.1
        0.45     11810818.0       525551.8
        0.50     13629000.0       295666.7
```

5-35 A program that dynamically modifies the a data format depending on the value being displayed is shown below:

```
PROGRAM modify_format
!
!  Purpose:
!    To display a number in either F or ES format depending
!    on its value, according to the following rules:
!    1.  If value == 0, display in F14.6 format.
!    2.  If 0.01 <= ABS(value) <= 1000.0, display in F14.6 format.
!    3.  Otherwise, display in ES14.6 format.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/08/2003    S. J. Chapman      Original code
!
```

```
IMPLICIT NONE

! List of variables:
CHARACTER(len=24) :: format       ! Format string
REAL :: value                     ! Value to display

! Prompt user and get value to display.
WRITE (*,*) 'Enter value to display: '
READ (*,*) value

! Set format string
IF ( value == 0. ) THEN
   format = "(' value = ',F14.6)"
ELSE IF ( 0.01 <= ABS(value) .AND. ABS(value) <= 1000. ) THEN
   format = "(' value = ',F14.6)"
ELSE
   format = "(' value = ',ES14.6)"
END IF

! Display value
WRITE (*,format) value

END PROGRAM modify_format
```

To test this program, we must supply it with examples of values in all possible ranges. We will test it with the 5 values 0.0, -0.0005., 12.3456, -12345.6:

```
C:\book\f95_2003\soln\ex5_35>modify_format
Enter value to display:
0.0
value =        .000000

C:\book\f95_2003\soln\ex5_35>modify_format
Enter value to display:
-0.00005
value =  -5.000000E-05

C:\book\f95_2003\soln\ex5_35>modify_format
Enter value to display:
12.3456
value =      12.345600

C:\book\f95_2003\soln\ex5_35>modify_format
Enter value to display:
-12345.6
value =  -1.234560E+04
```

5-36 A least-squares fitting program that also calculates the correlation coefficient is shown below:

```
PROGRAM lsq_corr_coef
!
!  Purpose:
!    To perform a least-squares fit of an input data set
!    to a straight line, and print out the resulting slope
!    and intercept values.  This program also calculates
!    the correlation coefficient associated with the fit.
```

```
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    11/19/06    S. J. Chapman        Original code
! 1. 05/07/07    S. J. Chapman        Modified to avoid divide
!                                     by 0 errors for < 2 points
! 2. 05/08/07    S. J. Chapman        Added corr coefficient.
!
IMPLICIT NONE

! List of parameters:
INTEGER, PARAMETER :: LU = 18 ! Unit for disk I/O

! List of variables.  Note that cumulative variables are all
! initialized to zero.
CHARACTER(len=24) :: filename ! Input file name (<= 24 chars)
REAL :: correl                ! Correlation coefficient (-1 to 1)
INTEGER :: ierror             ! Status flag from I/O statements
INTEGER :: n = 0              ! Number of input data pairs (x,y)
REAL :: slope                 ! Slope of the line
REAL :: sum_x = 0.            ! Sum of all input X values
REAL :: sum_x2 = 0.           ! Sum of all input X values squared
REAL :: sum_xy = 0.           ! Sum of all input X*Y values
REAL :: sum_y = 0.            ! Sum of all input Y values
REAL :: sum_y2 = 0.           ! Sum of all input Y values squared
REAL :: x                     ! An input X value
REAL :: x_bar                 ! Average X value
REAL :: y                     ! An input Y value
REAL :: y_bar                 ! Average Y value
REAL :: y_int                 ! Y-axis intercept of the line

! Prompt user and get the name of the input file.
WRITE (*,1000)
1000 FORMAT (' This program performs a least-squares fit of an ',/,&
             ' input data set to a straight line.  It also ',/,&
             ' calculates the correlation coefficient of the fit.',/, &
             ' Enter the name of the file containing the input', &
             ' (x,y) pairs:' )
READ (*,'(A)') filename

! Open the input file
OPEN (UNIT=LU, FILE=filename, STATUS='OLD', ACTION='READ', &
      IOSTAT=ierror )

! Check to see of the OPEN failed.
errorcheck: IF ( ierror > 0 ) THEN

   WRITE (*,1020) filename
   1020 FORMAT (1X,'ERROR: File ',A,' does not exist!')

ELSE

   ! File opened successfully. Read the (x,y) pairs from
   ! the input file.
   loop: DO
```

```
      READ (LU,*,IOSTAT=ierror) x, y ! Get pair
      IF ( ierror /= 0 ) EXIT
      n      = n + 1                 !
      sum_x  = sum_x + x             ! Calculate
      sum_y  = sum_y + y             !   statistics
      sum_x2 = sum_x2 + x**2         !
      sum_xy = sum_xy + x * y        !
      sum_y2 = sum_y2 + y**2         !
   END DO loop

   ! Now calculate the slope and intercept if enough data
   ! is available.
   enough: IF ( n > 1 ) THEN
      x_bar = sum_x / real(n)
      y_bar = sum_y / real(n)
      slope = (sum_xy - sum_x * y_bar) / ( sum_x2 - sum_x * x_bar)
      y_int = y_bar - slope * x_bar

      ! Calculate correlation coefficient.
      correl = ( REAL(n)*sum_xy - sum_x*sum_y ) &
         / SQRT ((REAL(n)*sum_x2-sum_x**2) * (REAL(n)*sum_y2-sum_y**2))

      ! Tell user.
      WRITE (*, 1030 ) slope, y_int, correl, n
      1030 FORMAT ('0','Regression coefficients for the least-squares line:',&
             /,1X,'  Slope (m)                   = ', F12.3,&
             /,1X,'  Intercept (b)               = ', F12.3,&
             /,1X,'  Correlation coefficient (r) = ', F12.3,&
             /,1X,'  No of points                = ', I12 )

      IF ( ABS(correl) < 0.3 ) THEN
         WRITE (*, 1040 )
         1040 FORMAT ('   WARNING: Small correlation coeficient!')
      END IF

   ELSE enough

      ! Tell user not enough data
      WRITE (*,1050)
      1050 FORMAT (' ERROR--at least 2 input data points required.')

   END IF enough

   ! Close input file, and quit.
   CLOSE (UNIT=LU)

END IF errorcheck

END PROGRAM lsq_corr_coef
```

If the values (0.0,0.0), (1.0, 0.5), (2.0, 2.4), (3.1, 2.9), and (4.0, 4.2) are placed in file `in5_34.dat`, and the program is run against this data set, the results are:

```
C:\book\f95_2003\soln\ex5_36>lsq_corr_coef
This program performs a least-squares fit of an
input data set to a straight line.  It also
```

```
calculates the correlation coefficient of the fit.
Enter the name of the file containing the input (x,y) pairs:
in5_34.dat

Regression coefficients for the least-squares line:
  Slope (m)                  =       1.067
  Intercept (b)              =       -.155
  Correlation coefficient (r) =       .981
  No of points               =          5
```

5-37 *(a)* A program to create a table of turning radius versus speed for a constant acceleration is shown below:

```
PROGRAM turning_radius1
!
!  Purpose:
!    To calculate the turning radius of an aircraft as a
!    function of speed for a given acceleration.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/08/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of constants:
REAL, PARAMETER :: G = 9.81        ! Accel of gravity (m/s**2)
REAL, PARAMETER :: MACH1 = 340.    ! Mach 1 (m/s)

! List of variables:
REAL :: acc               ! Lateral acceleration (m/s**2)
INTEGER :: i              ! Loop index
REAL :: radius            ! Turning radius (m)
REAL :: vel               ! Velocity (mach)

! Get input data
WRITE (*,*) 'Enter lateral acceleration (g): '
READ (*,*) acc

! Write headings
WRITE (*,'(/,4X,A,/)') 'Turning Radius vs Speed'
WRITE (*,*) "     Mach     Radius (m)  "
WRITE (*,*) "   ===================== "

! Calculate turning radius vs speed
DO i = 5, 20

   ! Get velocity
   vel = REAL(i) / 10.0

   ! Calculate turning radius
   radius = (vel * MACH1)**2 / (acc * G)

   ! Write result
   WRITE (*,'(3X,F7.1,5X,F8.1)') vel, radius
```

```
END DO

END PROGRAM turning_radius1
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln>turning_radius1
 Enter lateral acceleration (g):
2

    Turning Radius vs Speed

      Mach     Radius (m)
    ======================
       0.5       1473.0
       0.6       2121.1
       0.7       2887.1
       0.8       3770.8
       0.9       4772.5
       1.0       5891.9
       1.1       7129.3
       1.2       8484.4
       1.3       9957.4
       1.4      11548.2
       1.5      13256.9
       1.6      15083.4
       1.7      17027.7
       1.8      19089.9
       1.9      21269.9
       2.0      23567.8
```

(b) A program to create a table of turning radius versus speed for a constant acceleration is shown below:

```
PROGRAM turning_radius2
!
!  Purpose:
!    To calculate the turning radius of an aircraft as a
!    function of acceleration speed for a given speed.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/08/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of constants:
REAL, PARAMETER :: G = 9.81        ! Accel of gravity (m/s**2)
REAL, PARAMETER :: MACH1 = 340.    ! Mach 1 (m/s)

! List of variables:
REAL :: acc                ! Lateral acceleration (m/s**2)
INTEGER :: i               ! Loop index
REAL :: radius             ! Turning radius (m)
REAL :: vel                ! Velocity (mach)
```

```
! Get input data
WRITE (*,*) 'Enter speed in mach numbers: '
READ (*,*) vel

! Write headings
WRITE (*,'(/,4X,A,/)') 'Turning Radius vs Acc'
WRITE (*,*) "    Acc (g)   Radius (m)  "
WRITE (*,*) "   ====================== "

! Calculate turning radius vs speed
DO i = 20, 80, 5

   ! Get acceleration in g
   acc = REAL(i) / 10.0

   ! Calculate turning radius
   radius = (vel * MACH1)**2 / (acc * G)

   ! Write result
   WRITE (*,'(3X,F7.1,5X,F8.1)') acc, radius

END DO

END PROGRAM turning_radius2
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln>turning_radius2
 Enter speed in mach numbers:
.85

    Turning Radius vs Acc

     Acc (g)   Radius (m)
    ======================
        2.0       4256.9
        2.5       3405.5
        3.0       2838.0
        3.5       2432.5
        4.0       2128.5
        4.5       1892.0
        5.0       1702.8
        5.5       1548.0
        6.0       1419.0
        6.5       1309.8
        7.0       1216.3
        7.5       1135.2
        8.0       1064.2
```

Chapter 6. Introduction to Arrays

6-1 An array may be declared by specifying the rank and extent of the array using the `DIMENSION` attribute in a type declaration statement. For example, a 10-element integer array `status` would be declared as

```
INTEGER, DIMENSION(10) :: status
```

6-2 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.

6-3 The answer to this problem is processor dependent. Each instructor must supply the appropriate answer for his/her compiler / computer combination.

6-4 *(a)* 60 elements; valid subscript range is 1 to 60. *(b)* 225 elements; valid subscript range is 32 to 256. *(c)* 105 elements; valid subscript 1 range is 1 to 3 and valid subscript 2 range is 1 to 35.

6-5 *(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. *(b)* Valid. The statements print out the words `'Value = '` at the top of a new page, and then the ten values in the array, with one value per line. The values are printed out in the following order: 5.00, 10.00, 4.00, 9.00, 3.00, 8.00, 2.00, 7.00, 1.00, 6.00. *(c)* Valid. The expression "`a < b`" produces a 6-element logical array, so the output of the `WRITE` statement is: `T F F F F T`.

6-6 *(a)* The *size* of an array is the total number of elements in the array. *(b)* The *shape* of an array is the combination of the rank of the array and the number of elements in each dimension. *(c)* The *extent* in any given dimension of an array is the total number of elements in that dimension. *(d)* The *rank* of an array is the total number of dimensions (the total number of subscripts) in the array. *(e)* Two arrays are *conformable* if they have the same shape. A scalar is also conformable with an array.

6-7 The specified array sections are given below:

(a) This array is invalid, since the smallest valid subscript is –2.

(b) `my_array(-2,2)` = $\begin{bmatrix} -3 & -2 & -1 & 0 & 1 \end{bmatrix}$

(c) `my_array(1:5:2,:)` = $\begin{bmatrix} 0 & 2 & 4 \end{bmatrix}$

(d) `my_array(list)` = $\begin{bmatrix} -3 & 0 & 1 & 3 & 1 \end{bmatrix}$

6-8 The first `WRITE` statement is in a `DO` loop. It will be executed twice, and 4 values will be printed out each time. The second `WRITE` statement uses an implied `DO` loop to print out all 8 values at once. Since the format contains 6 descriptors, six values will be printed on one line and the remaining two on the following line. The output will be:

```
C:\book\f95_2003\soln>test_output
    1    2    3    4
    5    6    7    8
```

```
    1     2     3     4     5     6
    7     8
```

6-9 *(a)* The READ statement here is executed 4 times. Each time, it reads the first four values from the current line into four consecutive locations in the array. Therefore, array values will contain the following values

$$\texttt{values} = \begin{bmatrix} 27 & 17 & 10 & 8 & 11 & 13 & -11 & 12 & -1 & 0 & 0 & 6 & -16 & 11 & 21 & 26 \end{bmatrix}$$

(b) The READ statement here is executed one time. It 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 consecutive sequence. Therefore, array values will contain the following values

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

(c) The READ statement here is executed one time. It reads four values from the first line, then four the values from the second line, etc. until 16 values have been read. The values are stored in array values in consecutive sequence. Therefore, array values will contain the following values

$$\texttt{values} = \begin{bmatrix} 27 & 17 & 10 & 8 & 11 & 13 & -11 & 12 & -1 & 0 & 0 & 6 & -16 & 11 & 21 & 26 \end{bmatrix}$$

6-10 A program to convert two-dimensional vectors from polar form into rectangular form is shown below. Note that the input angles must be in units of degrees.

```
PROGRAM polar_to_rect
!
!  Purpose:
!    To read in a two-dimensional vector in magnitude & angle form,
!    and convert it into rectangular form.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/09/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of named constants:
REAL, PARAMETER :: DEG_2_RAD = 0.01745329   ! Degrees to radians

! List of variables:
REAL,DIMENSION(2) :: polar   ! Array containing magnitude / angle
                             !   polar(1) contains magnitude
                             !   polar(2) contains angle in degrees
REAL,DIMENSION(2) :: rect    ! Array containing rectangular comps

! Get vector in polar form.
WRITE (*,'(2A)') ' Enter the magnitude and angle (in degrees)', &
                 ' of the vector: '
READ (*,*) polar

! Convert to rectangular form.
rect(1) = polar(1) * COS ( polar(2) * DEG_2_RAD )
rect(2) = polar(1) * SIN ( polar(2) * DEG_2_RAD )

! Write out result.
```

```
WRITE (*,110) rect
110 FORMAT (' The rectangular form of the vector is ', &
              F9.5,'i + ',F9.5,'j')

END PROGRAM polar_to_rect
```

When the program is tested with the specified data values, the results are:

```
C:\book\f95_2003\soln>polar_to_rect
Enter the magnitude and angle (in degrees) of the vector:
5.0 -36.87
The rectangular form of the vector is   3.99999i +  -3.00001j

C:\book\f95_2003\soln>polar_to_rect
Enter the magnitude and angle (in degrees) of the vector:
10.0  45.0
The rectangular form of the vector is   7.07107i +   7.07107j

C:\book\f95_2003\soln>polar_to_rect
Enter the magnitude and angle (in degrees) of the vector:
25.0  233.13
The rectangular form of the vector is -15.00000i + -20.00000j
```

6-11 A program to convert two-dimensional vectors from rectangular form into polar form is shown below. Note that the output angles are in units of degrees. This program uses intrinsic function ATAN2 to calculate the angles, since that function works correctly in all quadrants.

```
PROGRAM rect_to_polar
!
!  Purpose:
!    To read in a two-dimensional vector in magnitude & angle form,
!    and convert it into rectangular form.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/09/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of named constants:
REAL, PARAMETER :: DEG_2_RAD = 0.01745329   ! Degrees to radians

! List of variables:
REAL,DIMENSION(2) :: polar   ! Array containing magnitude / angle
                             !   polar(1) contains magnitude
                             !   polar(2) contains angle in degrees
REAL,DIMENSION(2) :: rect    ! Array containing rectangular comps

! Get vector in polar form.
WRITE (*,'(2A)') ' Enter the vector in rectangular coordinates: '
READ (*,*) rect

! Get magnitude
polar(1) = SQRT ( rect(1)**2 + rect(2)**2 )
```

```
! Get angle in degrees.
polar(2) = ATAN2 ( rect(2), rect(1) ) / DEG_2_RAD

! Write out result.
WRITE (*,110) polar
110 FORMAT (' The polar form of the vector is ', &
            F9.5,' at an angle of ',F9.3,' degrees.')

END PROGRAM rect_to_polar
```

When the program is tested with the specified data values, the results are:

```
C:\book\f95_2003\soln>rect_to_polar
Enter the vector in rectangular coordinates:
3.  4.
The polar form of the vector is   5.00000 at an angle of    53.130 degrees.

C:\book\f95_2003\soln>rect_to_polar
Enter the vector in rectangular coordinates:
5.0   5.0
The polar form of the vector is   7.07107 at an angle of    45.000 degrees.

C:\book\f95_2003\soln>rect_to_polar
Enter the vector in rectangular coordinates:
-5.   12.
The polar form of the vector is  13.00000 at an angle of   112.620 degrees.
```

6-12 The statements required to count the positive, negative, and zero values in the array without using array intrinsic functions are:

```
REAL, DIMENSION(-50:50) :: values   ! Values
INTEGER :: i                        ! Loop index
INTEGER :: n_neg = 0                ! Number negative
INTEGER :: n_pos = 0                ! Number positive
INTEGER :: n_zero = 0               ! Number zero

DO i = -50, 50
   IF ( values(i) < 0.0 ) THEN
      n_neg = n_neg + 1
   ELSE IF ( values(i) == 0.0 ) THEN
      n_zero = n_zero + 1
   ELSE
      n_pos = n_pos + 1
   END IF
END DO

! Write summary statistics.
WRITE (*,1000) n_neg, n_zero, n_pos
1000 FORMAT (1X,'The distribution of values is:',/, &
             1X,'  Number of negative values = ', I3,/, &
             1X,'  Number of zero values     = ', I3,/, &
             1X,'  Number of positive values = ', I3)
```

6-13 The statements required to print out every fifth value in the array `values` with a `DO` loop are:

```
DO i = -50, 50, 5
```

```
   WRITE (*,100) i, values(i)
   100 FORMAT (7X,'values(',I3,') = ',F8.4)
END DO
```

6-14 A program to calculate the dot product of two three-dimensional vectors is shown below:

```
PROGRAM calc_dot_product
!
!  Purpose:
!    To calculate the dot product of two three-dimensional
!    vectors.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/09/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER, PARAMETER :: SIZE = 3  ! Size of vectors.

! List of variables:
REAL :: dot_product             ! Dot product
REAL, DIMENSION(SIZE) :: v1     ! Vector 1
REAL, DIMENSION(SIZE) :: v2     ! Vector 2

! Get the first vector.
WRITE (*,1000)
1000 FORMAT (' Calculate the dot product of 2 vectors. ', &
             /,1X,'Enter first vector (three terms): ')
READ (*,*) v1

! Get the second vector.
WRITE (*,1010)
1010 FORMAT (' Enter second vector (three terms): ')
READ (*,*) v2

! Calculate the dot product of the two vectors.
dot_product = v1(1) * v2(1) + v1(2) * v2(2) + v1(3) * v2(3)

! Tell user.
WRITE (*,1020) dot_product
1020 FORMAT (' The dot product of the two vectors is ', F12.4)

END PROGRAM calc_dot_product
```

When this program is tested with the data given in the problem, the results are

```
C:\book\f95_2003\soln\ex6_14>calc_dot_product
Calculate the dot product of 2 vectors.
Enter first vector (three terms):
5.   -3.    2.
Enter first vector (three terms):
2.    3.    4.
The dot product of the two vectors is       9.0000
```

6-15 Running the dot product program from Exercise 6-14 yields the result:

```
C:\book\f95_2003\soln\ex6_15>calc_dot_product
Calculate the dot product of 2 vectors.
Enter first vector (three terms):
4. 3. -2.
Enter first vector (three terms):
4. -2. 1.
The dot product of the two vectors is       8.0000
```

The total power supplied to the object is 8 W.

6-16 A program to calculate the cross product of two three-dimensional vectors is shown below:

```
PROGRAM calc_cross_product
!
!  Purpose:
!    To calculate the cross product of two three-dimensional
!    vectors.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/09/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER, PARAMETER :: SIZE = 3      ! Size of vectors

! List of variables:
REAL, DIMENSION(SIZE) :: v1         ! First vector
REAL, DIMENSION(SIZE) :: v2         ! Second vector
REAL, DIMENSION(SIZE) :: vcross     ! Cross product

! Get the first vector.
WRITE (*,1000)
1000 FORMAT (' Calculate the cross product of 2 vectors. ', &
           /,' Enter first vector (three terms): ')
READ (*,*) v1

! Get the second vector.
WRITE (*,1010)
1010 FORMAT (' Enter second vector (three terms): ')
READ (*,*) v2

! Calculate the cross product of the two vectors.
vcross(1) = v1(2) * v2(3) - v2(2) * v1(3)
vcross(2) = v1(3) * v2(1) - v2(3) * v1(1)
vcross(3) = v1(2) * v2(1) - v2(2) * v1(1)

! Tell user.
WRITE (*,1020) vcross
1020 FORMAT (1X,'The cross product of the two vectors is ', &
      F10.1, 'i + ', F10.1, 'j + ', F10.1, 'k')
```

```
END PROGRAM calc_cross_product
```

When this program is tested with the data given in the problem, the results are

```
C:\book\f95_2003\soln\ex6_16>calc_cross_product
Calculate the cross product of 2 vectors.
Enter first vector (three terms):
5.  -3.  2.
Enter first vector (three terms):
2.  3.  4.
The cross product of the two vectors is      -18.0i +      -16.0j +       21.0k
```

6-17 Using the program of the previous exercise, we get:

```
C:\book\f95_2003\soln\ex6_17>calc_cross_product
Calculate the cross product of 2 vectors.
Enter first vector (three terms):
300000.  400000.  50000.
Enter second vector (three terms):
-6.E-3  2.E-3  -9.E-4
The cross product of the two vectors is     -460.0i +      -30.0j +    -3000.0k
```

The velocity of the satellite is **v** = -460 **i** - 30 **j** -3000 **k** meters per second.

6-18 A modified form of the `stat_4` program that properly handles invalid values in the input data file is shown below:

```
PROGRAM stat_4a
!
!  Purpose:
!    To calculate mean, median, and standard deviation of an input
!    data set read from a file.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    11/17/06    S. J. Chapman        Original code
! 1. 05/09/07    S. J. Chapman        Modified to avoid problem
!                                     with invalid input data.
!
IMPLICIT NONE

! List of parameters:
INTEGER, PARAMETER :: MAX_SIZE = 100

! List of variables:
REAL, DIMENSION(MAX_SIZE) :: a   ! Data array to sort
LOGICAL :: exceed = .FALSE.      ! Logical indicating that array
                                 ! limits are exceeded.
CHARACTER(len=20) :: filename    ! Input data file name
INTEGER :: i                     ! Loop index
INTEGER :: iptr                  ! Pointer to smallest value
INTEGER :: j                     ! Loop index
REAL :: median                   ! The median of the input samples
INTEGER :: nvals = 0             ! Number of data values to sort
INTEGER :: status                ! I/O status: 0 for success
REAL :: std_dev                  ! Standard deviation of input samples
```

```
REAL :: sum_x = 0.                ! Sum of input values
REAL :: sum_x2 = 0.               ! Sum of input values squared
REAL :: temp                      ! Temporary variable for swapping
REAL :: x_bar                     ! Average of input values

! Get the name of the file containing the input data.
WRITE (*,1000)
1000 FORMAT (1X,'Enter the file name with the data to be sorted: ')
READ (*,'(A20)') filename

! Open input data file.  Status is OLD because the input data must
! already exist.
OPEN ( UNIT=9, FILE=filename, STATUS='OLD', ACTION='READ', &
       IOSTAT=status )

! Was the OPEN successful?
fileopen: IF ( status == 0 ) THEN          ! Open successful

   ! The file was opened successfully, so read the data to sort
   ! from it, sort the data, and write out the results.
   ! First read in data.
   DO
      READ (9, *, IOSTAT=status) temp      ! Get value
      readstat: IF ( status < 0 ) THEN
         EXIT                              ! Exit on end of data
      ELSE IF ( status > 0 ) THEN
         WRITE (*,1005) nvals + 1          ! Read error
         1005 FORMAT (' WARNING--Invalid data on line ', I6)
      ELSE
         nvals = nvals + 1                 ! Read ok
         size: IF ( nvals <= MAX_SIZE ) THEN ! Too many values?
            a(nvals) = temp                ! No: Save value in array
         ELSE
            exceed = .TRUE.                ! Yes: Array overflow
         END IF size
      END IF readstat
   END DO

   ! Was the array size exceeded?  If so, tell user and quit.
   toobig: IF ( exceed ) THEN
      WRITE (*,1010) nvals, MAX_SIZE
      1010 FORMAT (' Maximum array size exceeded: ', I6, ' > ', I6 )
   ELSE

      ! Limit not exceeded: sort the data.
      outer: DO i = 1, nvals-1

         ! Find the minimum value in a(i) through a(nvals)
         iptr = i
         inner: DO j = i+1, nvals
            minval: IF ( a(j) < a(iptr) ) THEN
               iptr = j
            END IF minval
         END DO inner

         ! iptr now points to the minimum value, so swap A(iptr)
```

```
         ! with a(i) if i /= iptr.
         swap: IF ( i /= iptr ) THEN
            temp    = a(i)
            a(i)    = a(iptr)
            a(iptr) = temp
         END IF swap

      END DO outer

      ! The data is now sorted.  Accumulate sums to calculate
      ! statistics.
      sums: DO i = 1, nvals
         sum_x  = sum_x + a(i)
         sum_x2 = sum_x2 + a(i)**2
      END DO sums

      ! Check to see if we have enough input data.
      enough: IF ( nvals < 2 ) THEN

         ! Insufficient data.
         WRITE (*,*) ' At least 2 values must be entered.'

      ELSE

         ! Calculate the mean, median, and standard deviation
         x_bar   = sum_x / real(nvals)
         std_dev = sqrt( (real(nvals) * sum_x2 - sum_x**2) &
                 / (real(nvals) * real(nvals-1)) )
         even: IF ( mod(nvals,2) == 0 ) THEN
            median = ( a(nvals/2) + a(nvals/2+1) ) / 2.
         ELSE
            median = a(nvals/2+1)
         END IF even

         ! Tell user.
         WRITE (*,*) 'The mean of this data set is:  ', x_bar
         WRITE (*,*) 'The median of this data set is:', median
         WRITE (*,*) 'The standard deviation is:     ', std_dev
         WRITE (*,*) 'The number of data points is:  ', nvals

      END IF enough

   END IF toobig

ELSE fileopen

   ! Else file open failed.  Tell user.
   WRITE (*,1050) status
   1050 FORMAT (1X,'File open failed--status = ', I6)

END IF fileopen

END PROGRAM stat_4a
```

If we create an input file IN5_24.DAT containing the 4 values 16.0, 4.z, 11.0, and 10.0, and feed the data into the program, the results are:

```
C:\book\f95_2003\soln\ex6_18>stat_4a
Enter the file name with the data to be processed:
in6_18.dat
 WARNING--Invalid data in input file on line     2!
 The mean of this data set is:        12.333330
 The median of this data set is:      11.000000
 The standard deviation is:            3.214550
 The number of data points is:            3
```

6-19 A program two read two sets of integers from a file, and to calculate the union and the intersection of the sets is shown below. To calculate the union of the two sets, the program compares each number in set 1 and set 2 to the union list. If the number is already in that list, then it does nothing. If the number is *not* in the list, then the program adds it to the list. To calculate the intersection of the two sets, the program compares each number in set 1to every number in set 2. If two numbers match, it checks to see if that number is in the intersection list. If the matching number is already in that list, then it does nothing. If the number is *not* in the list, then the program adds it to the list.

```
PROGRAM sets
!
!  Purpose:
!    To read in two sets of integer values into separate
!    arrays, and to calculate the sunion and the intersection
!    of the two arrays.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/09/2007  S. J. Chapman        Original code
!
IMPLICIT NONE

! List of parameters:
INTEGER, PARAMETER :: SIZE = 100 ! max size of arrays

! List of variables:
INTEGER, DIMENSION(SIZE) :: a1   ! First array
INTEGER, DIMENSION(SIZE) :: a2   ! Second array
INTEGER, DIMENSION(SIZE) :: a_u  ! Union of sets
INTEGER, DIMENSION(SIZE) :: a_i  ! Intersection of sets
CHARACTER(len=20) :: filename    ! Input data file name
INTEGER :: i                     ! Loop index
INTEGER :: j                     ! Loop index
LOGICAL :: in_set                ! Flag if element is in set
LOGICAL :: match                 ! Flag if two elements match
INTEGER :: nvals1 = 0            ! No of vals in array 1
INTEGER :: nvals2 = 0            ! No of vals in array 2
INTEGER :: nvals_u = 0           ! No of vals in union
INTEGER :: nvals_i = 0           ! No of vals in intersection
INTEGER :: status                ! I/O status: 0 for success
REAL :: temp                     ! Temp value for reading data

! Get the name of the file containing the first set.
WRITE (*,'(1X,A)') 'Enter the file name with the first set: '
READ (*,'(A20)') filename

! Open input data file.  Status is OLD because the input data must
```

```
! already exist.
OPEN ( UNIT=9, FILE=filename, STATUS='OLD', ACTION='READ', &
       IOSTAT=status )

! Was the OPEN successful?
IF ( status == 0 ) THEN          ! Open successful

   ! The file was opened successfully, so read the data to sort
   ! from it, sort the data, and write out the results.
   ! First read in data.
   DO
      READ (9, *, IOSTAT=status) temp      ! Get value
      IF ( status < 0 ) THEN
         EXIT                              ! Exit on end of data
      ELSE
         nvals1 = nvals1 + 1               ! Read ok
         a1(nvals1) = temp                 ! No: Save value in array
      END IF
   END DO

   ! Close file
   CLOSE(UNIT=9)
END IF

! Get the name of the file containing the second set.
WRITE (*,'(1X,A)') 'Enter the file name with the second set: '
READ (*,'(A20)') filename

! Open input data file.  Status is OLD because the input data must
! already exist.
OPEN ( UNIT=9, FILE=filename, STATUS='OLD', ACTION='READ', &
       IOSTAT=status )

! Was the OPEN successful?
IF ( status == 0 ) THEN          ! Open successful

   ! The file was opened successfully, so read the data to sort
   ! from it, sort the data, and write out the results.
   ! First read in data.
   DO
      READ (9, *, IOSTAT=status) temp      ! Get value
      IF ( status < 0 ) THEN
         EXIT                              ! Exit on end of data
      ELSE
         nvals2 = nvals2 + 1               ! Read ok
         a2(nvals2) = temp                 ! No: Save value in array
      END IF
   END DO

   ! Close file
   CLOSE(UNIT=9)
END IF

! Calculate the union by starting with set 1, and checking each
! successive value to see if it is already in the list.  If so,
! fine.  If not, add it to the list.
```

```
nvals_u = 0
DO i = 1, nvals1
   in_set = .FALSE.
   DO j = 1, nvals_u
      IF ( a1(i) == a_u(j) ) THEN
         in_set = .TRUE.
         EXIT
      END IF
   END DO

   ! If this element is not in the set, add it.
   IF ( .NOT. in_set ) THEN
      nvals_u = nvals_u + 1
      a_u(nvals_u) = a1(i)
   END IF
END DO

! Repeat this process for set 2.
DO i = 1, nvals2
   in_set = .FALSE.
   DO j = 1, nvals_u
      IF ( a2(i) == a_u(j) ) THEN
         in_set = .TRUE.
         EXIT
      END IF
   END DO

   ! If this element is not in the set, add it.
   IF ( .NOT. in_set ) THEN
      nvals_u = nvals_u + 1
      a_u(nvals_u) = a2(i)
   END IF
END DO

! Calculate the intersection by starting with each element
! in set 1 and comparing it to each element in set 2.  If
! the match, check to see if that number is already in the
! intersection.  If not, add it.
nvals_i = 0
DO i = 1, nvals1
   match = .FALSE.
   DO j = 1, nvals2
      IF ( a1(i) == a2(j) ) THEN
         match = .TRUE.
         EXIT
      END IF
   END DO

   ! This element is found in both set 1 and set 2.  If it
   ! is not in the intersection, add it.
   IF ( match ) THEN
      in_set = .FALSE.
      DO j = 1, nvals_i
         IF ( a1(i) == a_i(j) ) THEN
            in_set = .TRUE.
            EXIT
```

```
         END IF
      END DO

      ! If not in set, add it
      IF ( .NOT. in_set ) THEN
         nvals_i = nvals_i + 1
         a_i(nvals_i) = a1(i)
      END IF
   END IF
END DO

! Write out the results
WRITE (*,'(1X,A,20I5)') 'Set 1        = ', (a1(i), i=1,nvals1)
WRITE (*,'(1X,A,20I5)') 'Set 2        = ', (a2(i), i=1,nvals2)
WRITE (*,'(1X,A,20I5)') 'Union        = ', (a_u(i), i=1,nvals_u)
WRITE (*,'(1X,A,20I5)') 'Intersection = ', (a_i(i), i=1,nvals_i)

END PROGRAM sets
```

If we create two input files `inputA.dat` and `inputB.dat` as specified in the exercise, and run this program, the results are:

```
C:\book\f95_2003\soln\ex6_19>sets
 Enter the file name with the first set:
inputA.dat
 Enter the file name with the second set:
inputB.dat
 Set 1        =     0    1   -3    5  -11    6    8   11   17   15
 Set 2        =     0   -1    3    7   -6   16    5   12   21
 Union        =     0    1   -3    5  -11    6    8   11   17   15   -1    3
7   -6   16   12   21
 Intersection =     0    5
```

6-20 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
!      ====       ==========         =====================
!    05/09/2007  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 dist_3d
```

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

```
C:\book\f95_2003\soln\ex6_20>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 7. Introduction to Procedures

7-1 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.

7-2 Input data and results are passed between a subroutine and its calling program unit using dummy and actual arguments. The first dummy argument of the subroutine is associated with the first actual argument of the `CALL` statement, etc. using the pass-by-reference scheme. In this scheme, the program passes pointers to the locations of the calling arguments, instead of passing the arguments themselves.

7-3 The principal advantage of the pass-by-reference scheme is that it is efficient. It is much easier and quicker to pass a pointer to an array instead of passing all the values in the array, so subroutines called using this scheme execute faster.

The major disadvantage of the pass-by-reference scheme is that with an *implicit interface*, the programmer must ensure that the values in the calling argument list match the subroutine's calling parameters in number, type, and order. If there is a mismatch, the Fortran procedure will not be able to recognize that fact, and it will misuse the parameters without informing you of the problem. (This problem can be overcome in Fortran 95/2003 by using an explicit interface.)

7-4 For explicit-shape dummy arrays, both the array and all of its bounds are passed as arguments when the subroutine is called, and the array is declared to be of shape specified by the calling arguments. When an explicit-shape dummy array is used, the procedure has complete information about the array, and all array intrinsic functions may be used with it. Bounds checkers will also work with the array. The principal disadvantage is that all of the calling bounds must be included as calling arguments to the subroutine. An example of an explicit-shape dummy array is:

```
SUBROUTINE test1 (array, l1, u1)
INTEGER,INTENT(IN) :: l1, u1
REAL,DIMENSION(l1:u1) :: array           ! Explicit-shape
...
```

Assumed-shape dummy arrays pass the same information to a procedure without having to explicitly pass all of the array boundaries. Instead, the same information is passed through an explicit interface. When an assumed-shape dummy array is used, the procedure has complete information about the array, and all array intrinsic functions may be used with it. Bounds checkers will also work with the array. The principal disadvantage is that there *must* be an explicit interface to the procedure. An example of an assumed-shape dummy array is:

```
SUBROUTINE test2 (array)
REAL,DIMENSION(:) :: array               ! Assumed-shape
...
```

Assumed-size dummy arrays do not pass the final array boundary to the procedure, either explicitly via calling arguments or implicitly via an explicit interface. *The procedure does not know the actual size and shape of the array*. Many array intrinsic functions will not work with the array, and bounds checkers will not work with the array. With such arrays, it is easy for a procedure to access elements of an array that don't really exist. **Assumed-size dummy arrays should never be used in any modern program**. An example of an assumed-size dummy array is:

```
SUBROUTINE test3 (array)
REAL,DIMENSION(*) :: array                 ! Assumed-size
...
```

7-5 What happens will vary from processor to processor. When the subroutine attempts to write to element `a(16)`, it is addressing memory that was not allocated to the array. If that memory is being used for other variables, then some other variable in the program will be corrupted by the write. If that memory is not being used by the program to store other variables, then the program will probably abort with a memory protection violation.

7-6 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 PC compatible, the results are

```
C:\book\f95_2003\soln>main
 I = -1063256064
```

7-7 The program can be modified by placing the subroutine in a module, and then using that module in the main program. This will create an explicit interface to the subroutine, which will allow the compiler to automatically detect argument type mismatches.

7-8 The `INTENT` attribute specifies the intended use of each dummy argument in a procedure. There are three possible intents: `IN` for input-only arguments, `OUT` for output-only arguments, and `INOUT` for arguments used in both directions. The `INTENT` attribute should be included in the type declaration statement for each dummy argument. When the `INTENT` of an attribute is specified, the Fortran compiler can check to ensure that the argument is used properly within the procedure. If the procedure has an explicit interface, the compiler can also check that the actual arguments are consistent with the specified intent. For example, if a constant is used as an actual argument where the corresponding dummy argument is `INTENT(OUT)` the compiler will catch the error.

7-9 *(a)* Incorrect. Dummy argument `res` in subroutine `test_sub` is a real, while the corresponding actual argument `result` is an integer. Local variable `i` is declared with the `INTENT` attribute, which is only legal for dummy arguments. Also, `res` is never initialized in the subroutine. *(b)* Correct. This subroutine searches for the largest (highest collating sequence) character in an input character string, and returns that character to the calling program unit.

7-10 These statements are incorrect. The program attempts to assign a value to the named constant `g`, which is defined in the module and made accessible by use association.

7-11 The selection sort subroutine modified to sort in descending order is shown below:

```
SUBROUTINE sortd (arr, n)
!
!  Purpose:
!    To sort real array "arr" into descending order using a selection
!    sort.
!
IMPLICIT NONE

! Declare calling parameters:
INTEGER, INTENT(IN) :: n                 ! Number of values
REAL, DIMENSION(n), INTENT(INOUT) :: arr  ! Array to be sorted

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

```
INTEGER :: iptr               ! Pointer to smallest value
INTEGER :: j                  ! Loop index
REAL :: temp                  ! Temp variable for swaps

! Sort the array
outer: DO i = 1, n-1

   ! Find the maximum value in arr(i) through arr(n)
   iptr = i
   inner: DO j = i+1, n
      minval: IF ( arr(j) > arr(iptr) ) THEN
         iptr = j
      END IF minval
   END DO inner

   ! iptr now points to the maximum value, so swap arr(iptr)
   ! with arr(i) if i /= iptr.
   swap: IF ( i /= iptr ) THEN
      temp      = arr(i)
      arr(i)    = arr(iptr)
      arr(iptr) = temp
   END IF swap

END DO outer

END SUBROUTINE sortd
```

7-12 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
!       ====       ==========          =====================
!    05/09/2007    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
```

7-13 A test driver program for the statistical subroutines from Example 7-3 is shown below:

```
PROGRAM test_stat_subs
!
!  Purpose:
!    Test driver program for the statistical subroutines of
!    Example 7-3.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    08/03/2003    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER,PARAMETER :: LU = 27
INTEGER,PARAMETER :: MAXSIZ = 100

! List of variables:
REAL,DIMENSION(MAXSIZ) :: a          ! Input data array
REAL :: max_value                    ! Maximum value in array A
REAL :: min_value                    ! Minimum value in array A
REAL :: ave                          ! Average value in A
INTEGER :: error                     ! Error flag
CHARACTER(len=30) :: filename        ! Input file name
INTEGER :: imax                      ! Position of max value in A
INTEGER :: imin                      ! Position of min value in A
INTEGER :: istat                     ! I/o status
REAL :: med_value                    ! Median value in A
INTEGER :: nvals = 0                 ! Number of values in A
REAL :: std_dev                      ! Standard deviation of A
REAL :: temp                         ! Temp variable

! Get the name of the disk file containing the array.
WRITE (*,1000)
1000 FORMAT (' Enter the file name containing the array: ')
READ (*,'(A30)') filename

! Open input data file.  Status is OLD because the input data must
! already exist.
OPEN (UNIT=LU,FILE=filename,STATUS='OLD',ACTION='READ',IOSTAT=istat)

! Was the OPEN successful?
IF ( istat == 0 ) THEN

   ! The file was opened successfully, so read the data from it.
```

```
   DO
      READ (LU,*, IOSTAT=istat) temp
      IF ( istat /= 0 ) EXIT
      nvals = nvals + 1
      a(nvals) = temp
   END DO

   ! Now, calculate statistics on data.
   CALL rmax   ( a, nvals, max_value, imax )
   CALL rmin   ( a, nvals, min_value, imin )
   CALL ave_sd ( a, nvals, ave, std_dev, error )
   CALL median ( a, nvals, med_value )

   ! Tell user.
   WRITE (*,1030) imax, max_value
   1030 FORMAT (' The maximum value was A(',I3,') = ',F10.4)
   WRITE (*,1040) imin, min_value
   1040 FORMAT (' The minimum value was A(',I3,') = ',F10.4)
   WRITE (*,1050) ave
   1050 FORMAT (' The average value was          ',F10.4)
   WRITE (*,1060) std_dev
   1060 FORMAT (' The standard deviation was     ',F10.4)
   WRITE (*,1070) med_value
   1070 FORMAT (' The median value was           ',F10.4)
   WRITE (*,1080) error
   1080 FORMAT (' The error status from ave_sd was',I9)

END IF

END PROGRAM test_stat_subs
```

When the subroutines are tested with a series of data sets, they work fine as long as there are at least two point in the data set. If there is only one point, then subroutine `ave_sd` returns an error. It should also be noted that when a peak value occurs more than once in a data set, subroutines `rmin` and `rmax` pick the *first* occurrence only. An example test data set is shown below:

```
1.
2.
16.
-12.
10.
0.
16.
```

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

```
C:\book\f95_2003\soln\ex7_13>test_stat_subs
Enter the file name containing the array:
in7_13.dat
The maximum value was A(  3) =    16.0000
The minimum value was A(  4) =   -12.0000
The average value was              4.7143
The standard deviation was        10.0451
The median value was               2.0000
The error status from ave_sd was        0
```

7-14 A subroutine that uses `random0` to generate a set of uniform random numbers in the range [-1., 1.) is shown below.

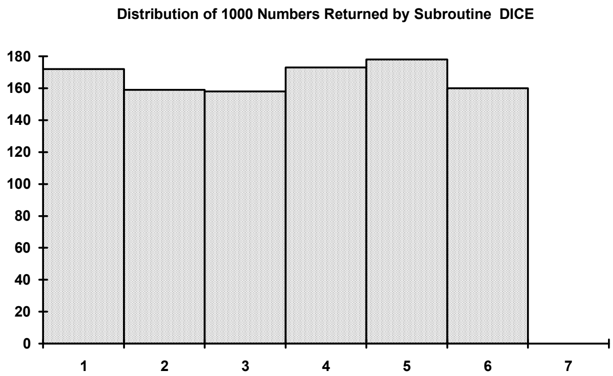
```
SUBROUTINE random1 ( value )
!
!  Purpose:
!    To generate uniform random numbers in the range [-1., 1.)
!    using subroutine random0.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/09/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of calling arguments:
REAL, INTENT(INOUT) :: value

! Call random0.
CALL random0 ( value )

! Map to the proper output range.
value = 2.0 * value - 1.0

END SUBROUTINE random1
```

7-15 A subroutine that uses `random0` to simulate the throw of a die is shown below:

```
SUBROUTINE dice ( ival )
!
!  Purpose:
!    To simulate the throw of a die, returning an integer
!    value between 1 and 6.  Do this by dividing the range
!    between 0 and 1 into 6 equal bins, and assigning a
!    particular number to "ival" if the value returned by
!    random0 falls into the corresponding bin.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/09/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of calling arguments:
INTEGER,INTENT(OUT) :: ival      ! Random value in range 1-6

! List of local variables:
REAL :: value                    ! Result of call to random0

! Get a random number
CALL random0 ( value )

! Map to the proper output range.
IF ( value < 0.1666667 ) THEN
   ival = 1
```

```
ELSE IF ( value < 0.3333333 ) THEN
   ival = 2
ELSE IF ( value < 0.5 ) THEN
   ival = 3
ELSE IF ( value < 0.6666667 ) THEN
   ival = 4
ELSE IF ( value < 0.8333333 ) THEN
   ival = 5
ELSE
   ival = 6
END IF

END SUBROUTINE dice
```

If this subroutine does indeed simulate a fair die, then each of the possible numbers 1 through 6 should occur with equal frequency. To test the subroutine, we can call it 1000 times, and plot the frequency of occurrence of each number. The results are shown in the chart below. As you can see, the number of occurrences of each number are very nearly equal.



7-16 A subroutine to calculate a value of the Poisson distribution for a specific *k*, *t*, and λ is shown below:

```
FUNCTION poisson( k, t, lamda )
!
!  Purpose:
!    To calculate a sample value from the poisson
!    distribution for specific values of k, t, and lamda.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/09/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of calling arguments:
INTEGER,INTENT(IN) :: k
REAL,INTENT(IN) :: t
REAL,INTENT(IN) :: lamda
REAL :: poisson                 ! Sample of distribution

! List of local variables:
REAL :: fact                    ! Factorial function
INTEGER :: i                    ! Loop index
```

```
! Calculate k!
fact = 1.
DO i = 2, k
   fact = fact * k
END DO

! Calculate value from poission distribution.
poisson = EXP(-lamda*t) * (lamda*t)**k / fact

END FUNCTION poisson
```

For the specified road traffic problem, k is the number of cars going by in interval t, given rate λ. We wish to determine the probability of $k = 0, 1, 2, 3, 4, 5$ cars going by in time $t = 1$ minute, given a rate λ of 1.6 cars/minute. A program to calculate this probability is:

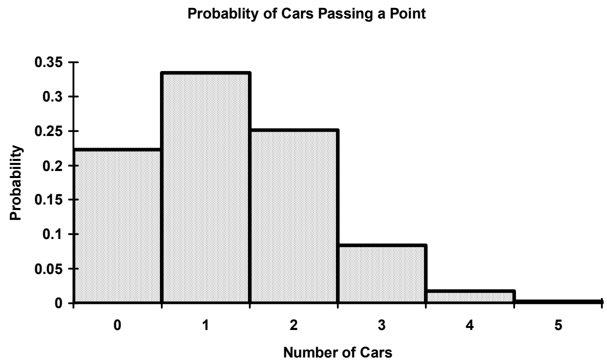
```
PROGRAM traffic_density
!
!  Purpose:
!    To calculate the probability of k cars passing a
!    given point in a specified interval of time.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    08/04/2003    S. J. Chapman       Original code
!
IMPLICIT NONE

! Declare external function:
REAL, EXTERNAL :: poisson       ! Sample of distribution

! List of variables:
INTEGER  :: k                   ! Number of cars
REAL :: lamda = 1.5             ! Average rate (cars/min)
REAL :: t = 1.                  ! Unit of time (min)
REAL :: probability             ! Probability

! Calculate probabilities:
WRITE (*,*) 'Number of cars vs probability: '
DO k = 0, 5
   probability = poisson( k, t, lamda )
   WRITE (*,100) k, probability
   100 FORMAT (4X,I3,4X,F12.7)
END DO

END PROGRAM traffic_density
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln>traffic_density
Number of cars vs probability:
     0        .2231302
     1        .3346952
     2        .2510214
     3        .0836738
```

```
     4        .0176499
     5        .0027110
```

This probability is plotted below:



7-17 Two purposes of modules are to share data between program units, and to provide an explicit interface to procedures placed in modules. If a procedure is placed in a module and accessed via USE association, the procedure has an explicit interface, permitting the Fortran compiler to detect most common errors associated with using procedures.

7-18 The functions to calculate sinh(x), cosh(x), and tanh(x) are shown below:

```
FUNCTION sinh1(x)
!
!  Purpose:
!    To calculate the hyperbolic sine function.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/11/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of calling arguments:
REAL,INTENT(IN) :: x          ! Input value
REAL :: sinh1                 ! Function result

! Calculate the hyperbolic sine function.
sinh1 = ( EXP(x) - EXP(-x) ) / 2.

END FUNCTION sinh1


FUNCTION cosh1(x)
!
!  Purpose:
!    To calculate the hyperbolic cosine function.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/11/2007    S. J. Chapman       Original code
!
```

```
IMPLICIT NONE

! List of calling arguments:
REAL,INTENT(IN) :: x          ! Input value
REAL :: cosh1                 ! Function result

! Calculate the hyperbolic sine function.
cosh1 = ( EXP(x) + EXP(-x) ) / 2.

END FUNCTION cosh1


FUNCTION tanh1 ( x )
!
!  Purpose:
!    To calculate the hyperbolic tangent function.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of calling arguments:
REAL,INTENT(IN) :: x          ! Input value
REAL :: tanh1                 ! Function result

! Calculate the hyperbolic sine function.
tanh1 = ( EXP(x) - EXP(-x) ) / ( EXP(x) + EXP(-x) )

END FUNCTION tanh1
```

A test driver program to test the hyperbolic functions is shown below

```
PROGRAM hyperbolic_test
!
!  Purpose:
!    To test the hyperbolic sine, cosine, and tangent functions.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of external functions:
REAL, EXTERNAL :: sinh1          ! Hyperbolic sine
REAL, EXTERNAL :: cosh1          ! Hyperbolic cosine
REAL, EXTERNAL :: tanh1          ! Hyperbolic tangent

! List of local variables:
INTEGER :: i                     ! Loop index
REAL,DIMENSION(11) :: test_vals  ! Test values
```

```
! Set test values
test_vals = (/ -2.0, -1.5, -1.0, 0.5, 0.25, 0.0, 0.25, 0.5, &
                1.0, 1.5, 2.0 /)

! Display heading.
WRITE (*,1010)
1010 FORMAT (/,4X,'SINH1(X)',T17,'SINH(X)',T29,'COSH1(X)',T41, &
              'COSH(X)',T53,'TANH1(X)',T65,'TANH(X)')

! Calculate and display values.
DO i = 1, 11
   WRITE (*,1020) sinh1(test_vals(i)), sinh(test_vals(i)), &
                  cosh1(test_vals(i)), cosh(test_vals(i)), &
                  tanh1(test_vals(i)), tanh(test_vals(i))
   1020 FORMAT (6(1X,F11.7))
END DO

END PROGRAM hyperbolic_test
```

The resulting table of values is shown below. It is clear that our functions produce the same answers as the corresponding intrinsic functions.

```
C:\book\f95_2003\soln\ex7_18>hyperbolic_test

   SINH1(X)    SINH(X)     COSH1(X)    COSH(X)     TANH1(X)    TANH(X)
 -3.6268600  -3.6268600   3.7621960   3.7621960   -.9640276   -.9640276
 -2.1292790  -2.1292790   2.3524100   2.3524100   -.9051483   -.9051483
 -1.1752010  -1.1752010   1.5430810   1.5430810   -.7615942   -.7615942
   .5210953    .5210953   1.1276260   1.1276260    .4621172    .4621172
   .2526123    .2526123   1.0314130   1.0314130    .2449187    .2449187
   .0000000    .0000000   1.0000000   1.0000000    .0000000    .0000000
   .2526123    .2526123   1.0314130   1.0314130    .2449187    .2449187
   .5210953    .5210953   1.1276260   1.1276260    .4621172    .4621172
  1.1752010   1.1752010   1.5430810   1.5430810    .7615942    .7615942
  2.1292790   2.1292790   2.3524100   2.3524100    .9051483    .9051483
  3.6268600   3.6268600   3.7621960   3.7621960    .9640276    .9640276
```

7-19 A subroutine to calculate the cross product of two three-dimensional vectors is shown below.

```
SUBROUTINE cross_product (v1, v2, vcross )
!
!  Purpose:
!    To calculate the cross product of two three-dimensional
!    vectors.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER, PARAMETER :: SIZE = 3     ! Three-dimensional vectors

! List of calling arguments:
```

```
REAL,DIMENSION(SIZE),INTENT(IN) :: v1      ! First input vector
REAL,DIMENSION(SIZE),INTENT(IN) :: v2      ! Second input vector
REAL,DIMENSION(SIZE),INTENT(OUT) :: vcross ! Cross product

! Calculate the cross product of the two vectors.
vcross(1) = v1(2) * v2(3) - v2(2) * v1(3)
vcross(2) = v1(3) * v2(1) - v2(3) * v1(1)
vcross(3) = v1(1) * v2(2) - v2(1) * v1(2)

END SUBROUTINE cross_product
```

A test driver program for this subroutine is shown below.

```
PROGRAM test_cross_product
!
!  Purpose:
!    To calculate the cross product of two three-dimensional
!    vectors.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER, PARAMETER :: size = 3     ! Three-dimensional vectors

! List of variables:
REAL,DIMENSION(size) :: v1         ! First input vector
REAL,DIMENSION(size) :: v2         ! Second input vector
REAL,DIMENSION(size) :: vcross     ! Cross product

! Get the first vector.
WRITE (*,1000)
1000 FORMAT (' Calculate the cross product of 2 vectors. ',&
           /,' Enter first vector (three terms): ')
READ (*,*) v1

! Get the second vector.
WRITE (*,1010)
1010 FORMAT (' Enter second vector (three terms): ')
READ (*,*) v2

! Calculate the cross product of the two vectors.
CALL cross_product ( v1, v2, vcross )

! Tell user.
WRITE (*,1020) vcross
1020 FORMAT (1X,'The cross product of the two vectors is ',&
      F10.1, 'i + ', F10.1, 'j + ', F10.1, 'k')

END PROGRAM test_cross_product
```

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

```
C:\book\f95_2003\soln>test_cross_product
Calculate the cross product of 2 vectors.
Enter first vector (three terms):
-2., 4., .5
Enter second vector (three terms):
.5, 3., 2.
The cross product of the two vectors is        6.5i +        4.3j +       -8.0k
```

7-20 A subroutine to sort a real array into ascending order while carrying along a second array is shown below.

```
SUBROUTINE sort2 (arr1, arr2, n)
!
!  Purpose:
!    To sort real array "arr1" into ascending order while carrying
!    along array "arr2", using a selection sort.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/11/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! Declare calling parameters:
INTEGER, INTENT(IN) :: n                  ! Number of values
REAL, DIMENSION(n), INTENT(INOUT) :: arr1 ! Array to be sorted
REAL, DIMENSION(n), INTENT(INOUT) :: arr2 ! Array carried along

! Declare local variables:
INTEGER :: i                  ! Loop index
INTEGER :: iptr               ! Pointer to smallest value
INTEGER :: j                  ! Loop index
REAL :: temp                  ! Temp variable for swaps

! Sort the array
outer: DO i = 1, n-1

   ! Find the minimum value in arr1(i) through arr1(n)
   iptr = i
   inner: DO j = i+1, n
      minval: IF ( arr1(j) < arr1(iptr) ) THEN
         iptr = j
      END IF minval
   END DO inner

   ! iptr now points to the minimum value, so swap arr1(iptr)
   ! with arr1(i) and arr2(iptr) with arr2(i) if i /= iptr.
   swap: IF ( i /= iptr ) THEN
      temp       = arr1(i)
      arr1(i)    = arr1(iptr)
      arr1(iptr) = temp
      temp       = arr2(i)
      arr2(i)    = arr2(iptr)
      arr2(iptr) = temp
   END IF swap
```

```
END DO outer

END SUBROUTINE sort2
```

A test driver program for this subroutine is shown below.

```
PROGRAM test_sort2
!
!  Purpose:
!    To test subroutine sort2, which sorts one array into ascending
!    order while carrying along a second array.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
REAL,DIMENSION(9) :: a              ! First array
REAL,DIMENSION(9) :: b              ! Second array
INTEGER :: i                        ! Index variable

a = (/  1., 11., -6., 17.,-23.,  0.,  5.,  1., -1. /)
b = (/ 31.,101., 36.,-17.,  0., 10., -8., -1., -1. /)

! Display arrays before sorting.
WRITE (*,'(1X,A)') 'Arrays before sorting: '
DO i = 1, 9
   WRITE (*,1000) i, a(i), i, b(i)
   1000 FORMAT (1X,'a(',I2,') = ',F8.2,'   b(',I2,') = ',F8.2)
END DO

!Sort arrays.
CALL sort2 ( a, b, 9 )

! Display arrays after sorting.
WRITE (*,'(/1X,A)') 'Arrays after sorting: '
DO i = 1, 9
   WRITE (*,1000) i, a(i), i, b(i)
END DO

END PROGRAM test_sort2
```

When this program is run, the results are

```
C:\book\f95_2003\soln>test_sort2
Arrays before sorting:
A( 1) =     1.00   B( 1) =    31.00
A( 2) =    11.00   B( 2) =   101.00
A( 3) =    -6.00   B( 3) =    36.00
A( 4) =    17.00   B( 4) =   -17.00
A( 5) =   -23.00   B( 5) =      .00
A( 6) =      .00   B( 6) =    10.00
```

```
A( 7) =     5.00   B( 7) =    -8.00
A( 8) =     1.00   B( 8) =    -1.00
A( 9) =    -1.00   B( 9) =    -1.00

Arrays after sorting:
A( 1) =   -23.00   B( 1) =      .00
A( 2) =    -6.00   B( 2) =    36.00
A( 3) =    -1.00   B( 3) =    -1.00
A( 4) =      .00   B( 4) =    10.00
A( 5) =     1.00   B( 5) =    31.00
A( 6) =     1.00   B( 6) =    -1.00
A( 7) =     5.00   B( 7) =    -8.00
A( 8) =    11.00   B( 8) =   101.00
A( 9) =    17.00   B( 9) =   -17.00
```

7-21 A subroutine to find the maximum and minimum values of a function within a specified range is shown below.

```
SUBROUTINE minmax ( func, first_value, last_value, num_steps, &
                    xmin, min_value, xmax, max_value )
!
!  Purpose:
!    To locate the position and value of the minimum and maximum
!    values of function func over the range first_value <= x
!    <= last_value.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of calling arguments:
REAL,EXTERNAL :: func                ! Function to be evaluated
REAL,INTENT(IN) :: first_value       ! First value of x to search
REAL,INTENT(IN) :: last_value        ! Last value of x to search
INTEGER,INTENT(IN) :: num_steps      ! Number of steps to search
REAL,INTENT(OUT) :: xmin             ! Value of x where min was found
REAL,INTENT(OUT) :: min_value        ! Minimum value
REAL,INTENT(OUT) :: xmax             ! Value of x where max was found
REAL,INTENT(OUT) :: max_value        ! Maximum value

! List of local variables:
INTEGER :: i                         ! Loop index
REAL :: step_size                    ! Step size for search
REAL :: x                            ! Position to evaluate func at
REAL :: value                        ! func(x)

! Calculate step size.
step_size = ( last_value - first_value ) / REAL(num_steps)

! Get value of function at first_value.
value   = func( first_value )
xmin    = first_value
min_value = value
xmax    = first_value
```

```
max_value = value

! Search over all other steps.
DO i = 1, num_steps
   x     = first_value + step_size * REAL(i)
   value = func( x )

   IF ( value > max_value ) THEN
      xmax      = x
      max_value = value
   END IF

   IF ( value < min_value ) THEN
      xmin      = x
      min_value = value
   END IF

END DO

END SUBROUTINE minmax
```

7-22 The test driver program is shown below.

```
PROGRAM test_minmax
!
!  Purpose:
!    To test subroutine minmax, which finds the minimum and maximum
!    values of a function over a user-specified search interval.  The
!    function to be evaluated is passed as a calling argument to
!    subroutine minmax.  For this test driver, the function func is
!    defined as
!        func(x) = x**3 - 5.*x**2 + 5.*x + 2.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare external function:
REAL,EXTERNAL :: func

! Declare variables:
REAL :: first_value = -1.  ! First value of x to search
REAL :: last_value = 3.    ! Last value of x to search
INTEGER :: num_steps = 200 ! Number of steps to search
REAL :: xmin               ! Value of x where min was found
REAL :: min_value          ! Minimum value
REAL :: xmax               ! Value of x where max was found
REAL :: max_value          ! Maximum value

! Call minmax.
CALL minmax ( func, first_value, last_value, num_steps, &
              xmin, min_value, xmax, max_value )
```

```
! Tell user.
WRITE (*,1000) xmin, min_value
1000 FORMAT (' The minimum value was func(',F10.5,') = ',F12.5)
WRITE (*,1010) xmax, max_value
1010 FORMAT (' The maximum value was func(',F10.5,') = ',F12.5)

END PROGRAM test_minmax

REAL FUNCTION func(x)
REAL,INTENT(IN) :: x
func = x**3 - 5.*x**2 + 5.*x + 2.
END FUNCTION func
```

When this program is executed, the results are

```
C:\book\f95_2003\soln>test_minmax
The minimum value was func(  -1.00000) =     -9.00000
The maximum value was func(    .62000) =      3.41633
```

7-23 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
!      ====       ==========         =====================
!    05/11/2007    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 derivative
```

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 `derivative`. 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 = cox(x)), and determine the error in the
!    subroutine.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    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 test_derivative
```

When this program is run, the results are

```
C:\book\f95_2003\soln>test_derivative
The maximum error in the derivative is      .0250.
```

7-24 To determine the effects of input noise on the quality of a numerical derivative, we will generate an input vector `sine1` containing 100 values of the function sin x starting at $x = 0$, and using a step size Δx of 0.05. Next, we will use subroutine `random0` to generate a uniform random noise with an amplitude of ±0.02, and use that to generate an input vector `sine2` containing the sinusoid plus 2% random noise. Then we will take the derivative of both functions, and determine how much their values differ from the analytic derivative of sin x:

$$\frac{d}{dx}\sin x = \cos x$$

The code to perform these steps is shown below.

```
PROGRAM deriv_with_noise
!
!  Purpose:
!    To examine the effects of noise on the calculation of the numerical
!    derivative of a function.  This routine generates two input data
!    sets, one a pure sinusoid and the other corrupted by a uniform random
!    noise whose peak amplitude is 2% if the peak amplitude of the
!    sinusoid.  It takes the derivative of both data sets, and compares
!    the numerical derivative with the known correct answer for a sine
!    function (df/dx = cox(x)).
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/11/2007   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     ! cos(x)
REAL,DIMENSION(nsamp-1) :: dsine1     ! Derivative of sine1
REAL,DIMENSION(nsamp-1) :: dsine2     ! Derivative of sine2
INTEGER :: error                      ! Error flag
INTEGER :: i                          ! Loop index
```

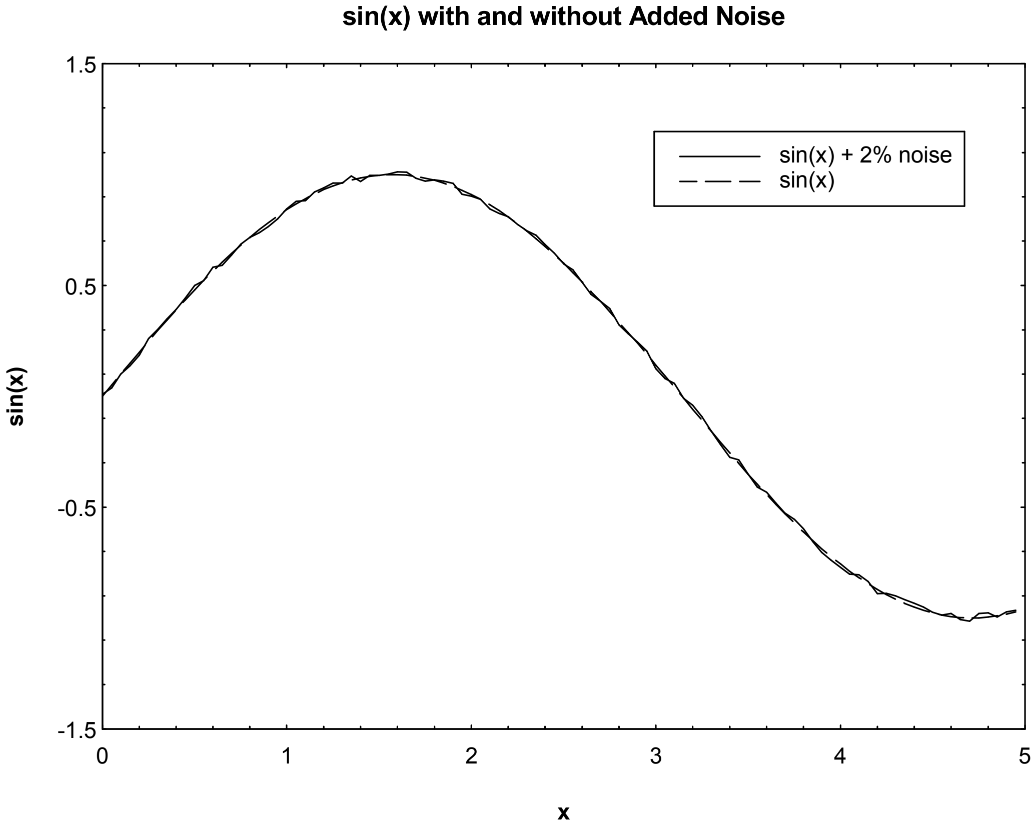
```
REAL :: max_error1                      ! Max error in dsine1
REAL :: max_error2                      ! Max error in dsine2
REAL,DIMENSION(nsamp) :: sine1          ! f(x) = sin(x)
REAL,DIMENSION(nsamp) :: sine2          ! f(x) = sin(x) + noise
REAL :: value                           ! Value from random0

! Calculate sine1 and sine2
DO i = 1, nsamp
   sine1(i) = SIN ( REAL(i-1) * dx )
   CALL random0 ( value )
   sine2(i) = sine1(i) + (0.04 * value - 0.02)
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 ( sine1, dsine1, nsamp, dx, error )
CALL derivative ( sine2, dsine2, nsamp, dx, error )

! Find the largest difference between the analytical derivative and
! the results of subroutine "derivative" with and without noise.
max_error1 = MAXVAL ( ABS( dsine1 - cderiv ) )
max_error2 = MAXVAL ( ABS( dsine2 - cderiv ) )

! Tell user.
WRITE (*,1010) max_error1
1010 FORMAT (' The max error in the numerical derivative is ', &
             'of the pure function is  ',F6.4,'.')
WRITE (*,1020) max_error2
1020 FORMAT (' The max error in the numerical derivative is ', &
             'of the noisy function is ',F6.4,'.')

END PROGRAM deriv_with_noise
```

When this program is run, the results are

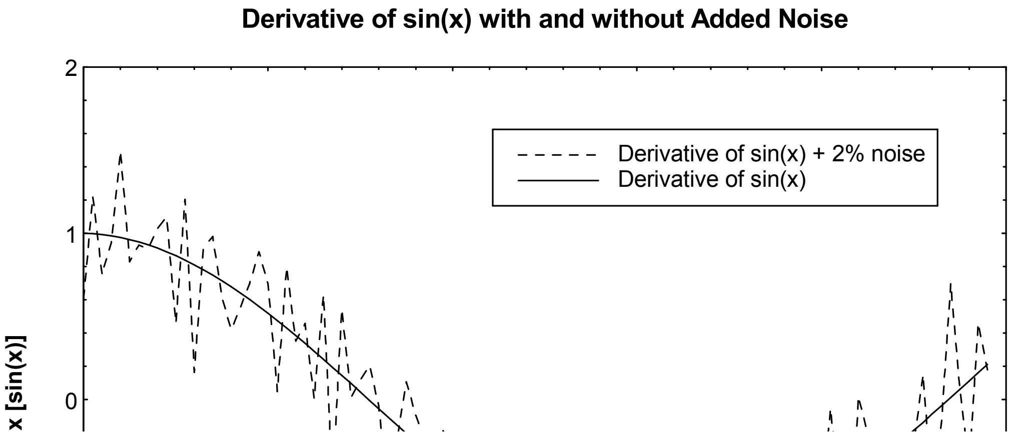
```
C:\book\f95_2003\soln>derivative_with_noise
The max error in the numerical derivative is of the pure function is   .0250.
The max error in the numerical derivative is of the noisy function is  .7419.
```

The maximum error in the numerical derivative of the pure sinusoid was about 2.5%, while the maximum error in the numerical derivative of a sinusoid corrupted by 2% noise was 74.2%! It is clear that taking a derivative magnifies the effect of any noise in the input data set.

This effect is illustrated in the following plots. The first plot shows the function $f(x) = \sin x$, and $f(x) = \sin x + 2\%$ noise. As you can see, there is little difference between the two plots. The second plot shows the derivative of the pure sine wave, and the derivative of the sine wave contaminated by 2% noise. The noise is greatly amplified by the process of taking the derivative.



(a) Plot of function $f(x)$ = sin x and $f(x)$ = sin x + 2% noise



(b) Plot of derivative of function $f(x)$ = sin x and $f(x)$ = sin x + 2% noise

7-25 *(a)* The subroutine shown below accepts 2 two's complement binary numbers stored as character strings containing eight 1's and 0's, and returns a string containing the sum of the two numbers. This subroutine works by doing a bit-by-bit sum, keeping track of the carries when they occur.

```
SUBROUTINE binary_add( val1, val2, sum )
!
!  Purpose:
!    Subroutine to perform two's complement addition using
!    values stored in character strings.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/11/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! Data dictionary: declare calling parameter types & definitions
CHARACTER(len=8),INTENT(IN) :: val1       ! Input value 1
CHARACTER(len=8),INTENT(IN) :: val2       ! Input value 2
CHARACTER(len=8),INTENT(OUT) :: sum       ! Result

! Declare local variables.
LOGICAL :: carry               ! Carry flag
INTEGER :: i                   ! Loop index

! Perform sum
carry = .FALSE.
DO i = 8, 1, -1

   ! Case 1: Both bits are 0
   IF ( val1(i:i) == '0' .AND. val2(i:i) == '0' ) THEN

      IF ( carry ) THEN
         sum(i:i) = '1'
         carry = .FALSE.
      ELSE
         sum(i:i) = '0'
         carry = .FALSE.
      END IF

   ! Case 2: One bit 1 and one bit 0
   ELSE IF ( val1(i:i) == '1' .AND. val2(i:i) == '0' ) THEN

      IF ( carry ) THEN
         sum(i:i) = '0'
         carry = .TRUE.
      ELSE
         sum(i:i) = '1'
         carry = .FALSE.
      END IF

   ! Case 3: One bit 0 and one bit 1
   ELSE IF ( val1(i:i) == '0' .AND. val2(i:i) == '1' ) THEN

      IF ( carry ) THEN
```

```
         sum(i:i) = '0'
         carry = .TRUE.
      ELSE
         sum(i:i) = '1'
         carry = .FALSE.
      END IF

   ! Case 4: Both bits are 1
   ELSE IF ( val1(i:i) == '1' .AND. val2(i:i) == '1' ) THEN

      IF ( carry ) THEN
         sum(i:i) = '1'
         carry = .TRUE.
      ELSE
         sum(i:i) = '0'
         carry = .TRUE.
      END IF

   END IF
END DO

END SUBROUTINE binary_add
```

(b) The subroutine shown below accepts two two's complement binary numbers stored as character strings containing eight 1's and 0's, and returns a string containing the difference between the two numbers. This subroutine works by doing a bit-by-bit subtraction, keeping track of the borrows when they occur.

```
SUBROUTINE binary_sub( val1, val2, diff )
!
!  Purpose:
!    Subroutine to perform two's complement subtraction using
!    values stored in character strings.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/11/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! Data dictionary: declare calling parameter types & definitions
CHARACTER(len=8),INTENT(IN) :: val1       ! Input value 1
CHARACTER(len=8),INTENT(IN) :: val2       ! Input value 2
CHARACTER(len=8),INTENT(OUT) :: diff      ! Result

! Declare local variables.
LOGICAL :: borrow              ! Borrow flag
INTEGER :: i                   ! Loop index

! Perform diff
borrow = .FALSE.
DO i = 8, 1, -1

   ! Case 1: Both bits are 0
   IF ( val1(i:i) == '0' .AND. val2(i:i) == '0' ) THEN
```

```
      IF ( borrow ) THEN
         diff(i:i) = '1'
         borrow = .TRUE.
      ELSE
         diff(i:i) = '0'
         borrow = .FALSE.
      END IF

   ! Case 2: One bit 1 and one bit 0
   ELSE IF ( val1(i:i) == '1' .AND. val2(i:i) == '0' ) THEN

      IF ( borrow ) THEN
         diff(i:i) = '0'
         borrow = .FALSE.
      ELSE
         diff(i:i) = '1'
         borrow = .FALSE.
      END IF

   ! Case 3: One bit 0 and one bit 1
   ELSE IF ( val1(i:i) == '0' .AND. val2(i:i) == '1' ) THEN

      IF ( borrow ) THEN
         diff(i:i) = '0'
         borrow = .TRUE.
      ELSE
         diff(i:i) = '1'
         borrow = .TRUE.
      END IF

   ! Case 4: Both bits are 1
   ELSE IF ( val1(i:i) == '1' .AND. val2(i:i) == '1' ) THEN

      IF ( borrow ) THEN
         diff(i:i) = '1'
         borrow = .TRUE.
      ELSE
         diff(i:i) = '0'
         borrow = .FALSE.
      END IF

   END IF
END DO

END SUBROUTINE binary_sub
```

(c) A subroutine to convert an 8-bit 2's complement binary number into an integer is shown below.

```
SUBROUTINE bin_to_int( val1, int1 )
!
!  Purpose:
!    Subroutine to convert a 2's complement number in the
!    range [-128,127) into an integer.
!
!  Record of revisions:
!       Date       Programmer         Description of change
```

```
!       ====       ==========         ====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Data dictionary: declare calling parameter types & definitions
CHARACTER(len=8),INTENT(IN) :: val1       ! Output value
INTEGER,INTENT(OUT) :: int1               ! Input integer

! Declare local variables.
INTEGER :: i                  ! Loop index
INTEGER :: ibit               ! Value corresponding to a particular bit
INTEGER :: ival               ! Working value
CHARACTER(len=8) :: sum       ! Sum (in 2's complelment calc)
CHARACTER(len=8) :: value     ! Local copy of val1

! If the number was negative, complement each bit and add one
! to get the 2's complement.
value = val1
IF ( value(1:1) == '1' ) THEN

   ! Take complement...
   DO i = 1, 8
      IF (value(i:i) == '0') THEN
         value(i:i) = '1'
      ELSE
         value(i:i) = '0'
      END IF
   END DO

   ! ... and add one.
   CALL binary_add( value, '00000001', sum )
   value = sum

END IF

! Now convert the string into an integer int1
ival = 1
int1 = 0
DO i = 8, 1, -1
   IF ( value(i:i) == '1' ) THEN
      int1 = int1 + ival
   END IF
   ival = ival * 2
END DO

! If the original number was negative, add it back here
IF ( val1(1:1) == '1' ) THEN
   int1 = -int1
END IF

END SUBROUTINE bin_to_int
```

(d) A subroutine to convert an integer into an 8-bit 2's complement binary number is shown below.

```
SUBROUTINE int_to_bin( int1, val1 )
!
!  Purpose:
!    Subroutine to convert an integer in the range [-128,127)
!    into a 2's complement number.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Data dictionary: declare calling parameter types & definitions
INTEGER,INTENT(IN) :: int1               ! Input integer
CHARACTER(len=8),INTENT(OUT) :: val1     ! Output value

! Declare local variables.
INTEGER :: i                   ! Loop index
INTEGER :: ibit                ! Value corresponding to a particular bit
INTEGER :: ival                ! Working value
CHARACTER(len=8) :: sum        ! Sum (in 2's complelment calc)

! Limit range to [-128,127)
ival = MAX( int1, -128 )
ival = MIN( ival,  127 )

! Convert bits using absolute value, and then convert to 2's
! complement for negative numbers.
ival = ABS(ival)

! Start with the largest bit and work downwards.  The
! largest bit is worth 2**7, or 128.  If the number
! is greater than that, set that bit to one and subtract
! 1 from the value.  Then try the bit at 2**6, or 64,
! and so forth.
ibit = 128
val1 = ' ';
DO i = 1, 8

   IF ( ival >= ibit ) THEN
      val1(i:i) = '1'
      ival = ival - ibit
   ELSE
      val1(i:i) = '0'
   END IF
   ibit = ibit / 2
END DO

! If the number was negative, complement each bit and add one
! to get the 2's complement.
IF ( int1 < 0 ) THEN

   ! Take complement...
   DO i = 1, 8
```

```
      IF (val1(i:i) == '0') THEN
         val1(i:i) = '1'
      ELSE
         val1(i:i) = '0'
      END IF
   END DO

   ! ... and add one.
   CALL binary_add( val1, '00000001', sum )
   val1 = sum

END IF

END SUBROUTINE int_to_bin
```

(e) A program that implements a dual decimal / binary calculator is shown below. The user can enter data in either binary or decimal format, and specify either addition or subtraction. The results are displayed in both binary and decimal format.

```
PROGRAM calculator
!
!  Purpose:
!    To perform calculations in both decimal and binary
!    arithmetic, and to display both results.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare variables:
INTEGER :: i                          ! Loop index
INTEGER :: ival                       ! Integer value
CHARACTER(len=8) :: conv1             ! Conversion from int
CHARACTER(len=8) :: diff              ! Difference
CHARACTER(len=8) :: sum               ! Sum
CHARACTER(len=8) :: bres              ! Binary result
CHARACTER(len=8) :: bval1             ! First binary value
CHARACTER(len=8) :: bval2             ! Second binary value
INTEGER :: ires                       ! Integer result
INTEGER :: ival1                      ! First integer value
INTEGER :: ival2                      ! Second integer value
CHARACTER(len=1) :: op                ! Operation
INTEGER :: type                       ! Type of input data

! Select type of input data
WRITE (*,*) 'Specify type of input data (1=decimal;2=binary):'
READ (*,*) type

! Get input data
IF ( type == 1 ) THEN

   WRITE (*,*) 'Enter first decimal number in range (-128 to 127):'
   READ (*,*) ival1
```

```
   WRITE (*,*) 'Enter second decimal number in range (-128 to 127):'
   READ (*,*) ival2
   CALL int_to_bin ( ival1, bval1 )
   CALL int_to_bin ( ival2, bval2 )

ELSE

   WRITE (*,*) 'Enter first binary number in range (00000000 to 11111111):'
   READ (*,*) bval1
   WRITE (*,*) 'Enter second binary number in range (00000000 to 11111111):'
   READ (*,*) bval2
   CALL bin_to_int ( bval1, ival1 )
   CALL bin_to_int ( bval2, ival2 )

END IF

! Select operation
WRITE (*,*) 'Select operation (+ or -):'
READ (*,*) op

! Now do math
IF ( op == '+' .AND. type == 1 ) THEN

   ! Decimal addition
   ires = ival1 + ival2
   CALL int_to_bin(ires, bres)

ELSE IF ( op == '+' .AND. type == 2 ) THEN

   ! Binary addition
   CALL binary_add( bval1, bval2, bres )
   CALL bin_to_int (bres, ires)

ELSE IF ( op == '-' .AND. type == 1 ) THEN

   ! Decimal addition
   ires = ival1 - ival2
   CALL int_to_bin(ires, bres)

ELSE IF ( op == '-' .AND. type == 2 ) THEN

   ! Binary addition
   CALL binary_sub( bval1, bval2, bres )
   CALL bin_to_int (bres, ires)

END IF

! Display results
WRITE (*,*) 'Value 1 = ', bval1, ival1
WRITE (*,*) 'Value 2 = ', bval2, ival2
WRITE (*,*) 'Result  = ', bres,  ires

END PROGRAM calculator
```

When this program is run, the results are

```
C:\book\f95_2003\soln\ex7_25>calculator
 Specify type of input data (1=decimal;2=binary):
1
 Enter first decimal number in range (-128 to 127):
8
 Enter second decimal number in range (-128 to 127):
-6
 Select operation (+ or -):
+
 Value 1 = 00001000          8
 Value 2 = 11111010         -6
 Result  = 00000010          2

C:\book\f95_2003\soln\ex7_25>calculator
 Specify type of input data (1=decimal;2=binary):
2
 Enter first binary number in range (00000000 to 11111111):
00100000
 Enter second binary number in range (00000000 to 11111111):
11111111
 Select operation (+ or -):
+
 Value 1 = 00100000         32
 Value 2 = 11111111         -1
 Result  = 00011111         31

C:\book\f95_2003\soln\ex7_25>calculator
 Specify type of input data (1=decimal;2=binary):
2
 Enter first binary number in range (00000000 to 11111111):
00100000
 Enter second binary number in range (00000000 to 11111111):
11111111
 Select operation (+ or -):
-
 Value 1 = 00100000         32
 Value 2 = 11111111         -1
 Result  = 00100001         33
```

7-26 A linear least-squares fit subroutine is shown below:

```
SUBROUTINE lsqfit ( x, y, nvals, slope, y_int, error )
!
!  Purpose:
!    To perform a least-squares fit of an input data set
!    to the line y(x) = slope * x + y_int and return the
!    resulting coefficients.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/11/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of calling arguments:
```

```
INTEGER,INTENT(IN) :: nvals             ! No. of values
REAL,DIMENSION(nvals),INTENT(IN) :: x   ! Array of x values
REAL,DIMENSION(nvals),INTENT(IN) :: y   ! Array of y values
REAL,INTENT(OUT) :: slope               ! Slope of fitted line
REAL,INTENT(OUT) :: y_int               ! y-axis intercept of line
INTEGER,INTENT(OUT) :: error            ! Error flag: 0 = no error
                                        !  1 = not enough input values
! List of local variables:
INTEGER :: i                ! Index variable
REAL :: sum_x               ! The sum of all input x values
REAL :: sum_x2              ! The sum of all input x values squared
REAL :: sum_xy              ! The sum of all input x*y values
REAL :: sum_y               ! The sum of all input y values
REAL :: xbar                ! The average x value
REAL :: ybar                ! The average y value

! First, check to make sure that we have enough input data.
IF ( nvals < 2 ) THEN

   ! Insufficient data.  Set error = 1, and get out.
   error = 1

ELSE

   ! Reset error flag.
   error = 0

   ! Zero the sums used to build the equations.
   sum_x  = 0.
   sum_x2 = 0.
   sum_xy = 0.
   sum_y  = 0.

   ! Build the sums required to solve the equations.
   DO i = 1, nvals
      sum_x  = sum_x + x(i)
      sum_y  = sum_y + y(i)
      sum_x2 = sum_x2 + x(i)**2
      sum_xy = sum_xy + x(i) * y(i)
   END DO

   ! Now calculate the slope and intercept.
   xbar  = sum_x / REAL(nvals)
   ybar  = sum_y / REAL(nvals)
   slope = (sum_xy - sum_x * ybar) / ( sum_x2 - sum_x * xbar)
   y_int  = ybar - slope * xbar

END IF

END SUBROUTINE lsqfit
```

A test driver program for this subroutine is shown below.

```
PROGRAM test_lsqfit
!
!  Purpose:
```

```
!    To test subroutine lsqfit, which performs a least squares
!    fit to a straight line, and returns the slope and intercept
!    of the best-fit line.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/11/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER,PARAMETER :: MAXVALS = 1000   ! Max number of (x,y) pairs
INTEGER,PARAMETER :: LU = 12          ! I/O unit

! List of variables:
LOGICAL :: exceed = .FALSE.       ! Flag for too much data
INTEGER :: error                  ! Status flag for i/o
CHARACTER(len=30) :: filename     ! The input file name
INTEGER :: nvals = 0              ! Number of input data points
REAL :: slope                     ! Slope of line
REAL :: tempx                     ! Temporary x value
REAL :: tempy                     ! Temporary y value
REAL,DIMENSION(MAXVALS) :: x      ! Array of x input values
REAL,DIMENSION(MAXVALS) :: y      ! Array of x input values
REAL :: y_int                     ! Y-axis intercept of line

! Prompt user and get the name of the input file.
WRITE (*,1000)
1000 FORMAT (' This program performs a least-squares fit of an ',/,&
             ' input data set to a straight line.  Enter the name',/,&
             ' of the file containing the input (x,y) pairs:' )
READ (*,'(A30)') filename

! Open the input file
OPEN (UNIT=LU,FILE=filename,STATUS='OLD',ACTION='READ',IOSTAT=error )

! Check to see of the OPEN failed.
open_ok: IF ( error > 0 ) THEN
   WRITE (*,1020) filename, error
   1020 FORMAT (' ERROR: File ',A,' open failed: IOSTAT = ',I6)
ELSE open_ok

   readloop: DO
      ! Read the first (x,y) pair from the input file.
      READ (LU,*,IOSTAT=error) tempx, tempy
      IF ( error /= 0 ) EXIT

      ! Bump the point count.
      nvals = nvals + 1

      ! If nvals <= maxval, then save these values.  Otherwise,
      ! set the exceed flag.
      IF ( nvals <= MAXVALS ) THEN
         x(nvals) = tempx
         y(nvals) = tempy
```

```
      ELSE
         exceed = .TRUE.
      END IF
   END DO readloop

   ! Now calculate the slope and intercept.
   calc: IF ( .NOT. exceed ) THEN
      CALL lsqfit ( x, y, nvals, slope, y_int, error )

      ! Tell user about fit.
      WRITE (*, 1030 ) slope, y_int, nvals
      1030 FORMAT ('0Regression coefficients for the least-squares line:',&
                /,'   Slope (m)     = ', F12.3,&
                /,'   Intercept (b) = ', F12.3,&
                /,'   No of points  = ', I12 )
   ELSE calc

      WRITE (*,1040) MAXVALS
      1040 FORMAT (' Too many input values: max = ', I5)

   END IF calc

   ! Close input file.
   CLOSE (UNIT=LU)

END IF open_ok

END PROGRAM test_lsqfit
```

When the data specified in the problem is placed into file IN6_28.DAT, and this program is run against that file, the results are

```
C:\book\f95_2003\soln\ex7_26>test_lsqfit
This program performs a least-squares fit of an
input data set to a straight line.  Enter the name
of the file containing the input (x,y) pairs:
in7_26.dat

Regression coefficients for the least-squares line:
  Slope (m)     =        1.844
  Intercept (b) =         .191
  No of points  =           20
```

7-27 A modified linear least squares fit subroutine that calculates a correlation coefficient along with the slope and intercept is shown below.

```
SUBROUTINE lsqfit_cor ( x, y, nvals, slope, y_int, correl, error )
!
!  Purpose:
!    To perform a least-squares fit of an input data set
!    to the line y(x) = slope * x + y_int and return the
!    resulting coefficients.  This subroutine also calculates
!    the correlation coefficient associated with the fit.
!
!  Record of revisions:
!      Date       Programmer         Description of change
```

```
!      ====      ==========        ====================
!   05/11/2007   S. J. Chapman     Original code
! 1. 05/12/2007  S. J. Chapman     Modified to add corr. coef.
!
IMPLICIT NONE

! List of calling arguments:
INTEGER,INTENT(IN) :: nvals            ! No. of values
REAL,DIMENSION(nvals),INTENT(IN) :: x  ! Array of x values
REAL,DIMENSION(nvals),INTENT(IN) :: y  ! Array of y values
REAL,INTENT(OUT) :: slope              ! Slope of fitted line
REAL,INTENT(OUT) :: y_int              ! y-axis intercept of line
REAL,INTENT(OUT) :: correl             ! correlation coefficient
INTEGER,INTENT(OUT) :: error           ! Error flag: 0 = no error
                                       !  1 = not enouth input values
! List of local variables:
INTEGER :: i                ! Index variable
REAL :: sum_x               ! The sum of all input x values
REAL :: sum_x2              ! The sum of all input x values squared
REAL :: sum_xy              ! The sum of all input x*y values
REAL :: sum_y               ! The sum of all input y values
REAL :: sum_y2              ! The sum of all input y values squared
REAL :: xbar                ! The average x value
REAL :: ybar                ! The average y value

! First, check to make sure that we have enough input data.
IF ( nvals < 2 ) THEN

   ! Insufficient data.  Set error = 1, and get out.
   error = 1

ELSE

   ! Reset error flag.
   error = 0

   ! Zero the sums used to build the equations.
   sum_x  = 0.
   sum_x2 = 0.
   sum_xy = 0.
   sum_y  = 0.
   sum_y2 = 0.

   ! Build the sums required to solve the equations.
   DO i = 1, nvals
      sum_x  = sum_x + x(i)
      sum_y  = sum_y + y(i)
      sum_x2 = sum_x2 + x(i)**2
      sum_xy = sum_xy + x(i) * y(i)
      sum_y2 = sum_y2 + y(i)**2
   END DO

   ! Now calculate the slope and intercept.
   xbar  = sum_x / REAL(nvals)
   ybar  = sum_y / REAL(nvals)
   slope = (sum_xy - sum_x * ybar) / ( sum_x2 - sum_x * xbar)
```

```
   y_int  = ybar - slope * xbar

   ! Calculate the correlation coefficient.
   correl = ( REAL(nvals)*sum_xy - sum_x*sum_y ) &
          / SQRT ( (REAL(nvals)*sum_x2-sum_x**2) &
          * (REAL(nvals)*sum_y2-sum_y**2) )

END IF

END SUBROUTINE lsqfit_cor
```

A test driver program for this subroutine is shown below.

```
PROGRAM test_lsqfit_cor
!
!  Purpose:
!    To test subroutine lsqcor, which performs a least squares
!    fit to a straight line, and returns the slope, intercept
!    and correlation coefficient of the best-fit line.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
! 1. 05/12/2007    S. J. Chapman      Modified to add corr. coef.
!
IMPLICIT NONE

! List of named constants:
INTEGER,PARAMETER :: MAXVALS = 1000   ! Max number of (x,y) pairs
INTEGER,PARAMETER :: LU = 12          ! I/O LU

! List of variables:
REAL :: correl                    ! Correlation coefficient
LOGICAL :: exceed = .FALSE.       ! Flag for too much data
INTEGER :: error                  ! Status flag for i/o
CHARACTER(len=30) :: filename     ! The input file name
INTEGER :: nvals = 0              ! Number of input data points
REAL :: slope                     ! Slope of line
REAL :: tempx                     ! Temporary x value
REAL :: tempy                     ! Temporary y value
REAL,DIMENSION(MAXVALS) :: x      ! Array of x input values
REAL,DIMENSION(MAXVALS) :: y      ! Array of x input values
REAL :: y_int                     ! Y-axis intercept of line

! Prompt user and get the name of the input file.
WRITE (*,1000)
1000 FORMAT (' This program performs a least-squares fit of an ',/,&
             ' input data set to a straight line.  Enter the name',/,&
             ' of the file containing the input (x,y) pairs:' )
READ (*,'(A30)') filename

! Open the input file
OPEN (UNIT=LU,FILE=filename,STATUS='OLD',ACTION='READ',IOSTAT=error )

! Check to see of the OPEN failed.
```

```
open_ok: IF ( error > 0 ) THEN
   WRITE (*,1020) filename, error
   1020 FORMAT (' ERROR: File ',A,' open failed: IOSTAT = ',I6)
ELSE open_ok

   readloop: DO
      ! Read the first (x,y) pair from the input file.
      READ (LU,*,IOSTAT=error) tempx, tempy
      IF ( error /= 0 ) EXIT

      ! Bump the point count.
      nvals = nvals + 1

      ! If nvals <= MAXVALS, then save these values.  Otherwise,
      ! set the exceed flag.
      IF ( nvals <= MAXVALS ) THEN
         x(nvals) = tempx
         y(nvals) = tempy
      ELSE
         exceed = .TRUE.
      END IF
   END DO readloop

   ! Now calculate the slope and intercept.
   calc: IF ( .NOT. exceed ) THEN
      CALL lsqfit_cor ( x, y, nvals, slope, y_int, correl, error )

      ! Tell user about fit.
      WRITE (*, 1030 ) slope, y_int, correl, nvals
      1030 FORMAT ('0Regression coefficients for the least-squares line:',&
                /,'   Slope (m)     = ', F12.3,&
                /,'   Intercept (b) = ', F12.3,&
                /,'   Correlation   = ', F12.3,&
                /,'   No of points  = ', I12 )
   ELSE calc

      WRITE (*,1040) MAXVALS
      1040 FORMAT (' Too many input values: max = ', I5)

   END IF calc

   ! Close input file.
   CLOSE (UNIT=LU)

END IF open_ok

END PROGRAM test_lsqfit_cor
```

When this program is run with the data set from the previous problem, the results are:

```
C:\book\f95_2003\soln\ex7_27>test_lsqcor
This program performs a least-squares fit of an
input data set to a straight line.  Enter the name
of the file containing the input (x,y) pairs:
in7_26.dat
```

```
Regression coefficients for the least-squares line:
  Slope (m)     =       1.844
  Intercept (b) =        .191
  Correlation   =        .948
  No of points  =          20
```

7-28 We will take advantage of the intrinsic subroutine RANDOM_NUMBER in this function. If an array is supplied to the subroutine, it will place a random value in each element of the array, allowing us to generate all "n" random birthdays at once. In addition, we will use automatic arrays to adjust the size of the arrays to the number of people specified in the calling argument. The automatic array declarations are shown in bold face below:

```
FUNCTION birthday(n)
!
!  Purpose:
!    To calculate the probability of two or more of n people
!    having the same birthday, where "n" is a calling argument.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of calling arguments:
INTEGER,INTENT(IN) :: n                ! Number of people
REAL :: birthday                       ! probability

! List of named constants:
INTEGER,PARAMETER :: N_TRIALS = 10000 ! Number of trials

! List of local variables:
INTEGER,DIMENSION(n) :: birthdays     ! Array of birthdays
INTEGER :: i, j, k                    ! Loop index
INTEGER :: n_match                    ! No. of trials with
                                      !   matching birthdays
REAL,DIMENSION(n) :: random           ! Array of random values

! Initialize variables
n_match = 0

! To determine this probability, we will generate an array
! of "n" random birthdays, and check to see if two or more
! are the same.  We will repeat this process "n_trials" times,
! and calculate the probability from the result.
trials: DO i = 1, N_TRIALS

   ! Generate random birthdays using the computer's random
   ! number generator.  Note that RANDOM_NUMBER returns a
   ! value in the range 0 <= value < 1.0, so we must divide
   ! that range into 365 equal intervals.  (We are ignoring
   ! leap years!)
   CALL RANDOM_NUMBER ( random )        ! n random numbers
   birthdays = INT( 365 * random ) + 1  ! Range is 1 to 365

   ! Now check for matches
```

```
   outer: DO j = 1, n-1
      inner: DO k = j+1, n
         IF ( birthdays(j) == birthdays(k) ) THEN
            n_match = n_match + 1
            EXIT outer
         END IF
      END DO inner
   END DO outer

END DO trials

! Now calculate probability
birthday = REAL(n_match) / REAL(n_trials)

END FUNCTION birthday
```

A test driver program for this function is:

```
PROGRAM test_birthday
!
!  Purpose:
!    To test function birthday, which calculates the probability
!    that at least two of "n" people in a random crowd will have
!    the same birthday.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! External functions:
REAL, EXTERNAL :: birthday        ! Birthday function

! List of variables:
INTEGER :: i                      ! Index variable

DO i= 2, 40
   WRITE (*,1000) i, birthday(i)
   1000 FORMAT (' The probability of at least 2 of ', I2, ' people', &
                ' having the same birthday is ',F5.3)
END DO

END PROGRAM test_birthday
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex7_28>test_birthday
The probability of at least 2 of  2 people having the same birthday is  .002
The probability of at least 2 of  3 people having the same birthday is  .009
The probability of at least 2 of  4 people having the same birthday is  .017
The probability of at least 2 of  5 people having the same birthday is  .029
The probability of at least 2 of  6 people having the same birthday is  .038
The probability of at least 2 of  7 people having the same birthday is  .057
The probability of at least 2 of  8 people having the same birthday is  .074
```

```
The probability of at least 2 of  9 people having the same birthday is  .095
The probability of at least 2 of 10 people having the same birthday is  .118
The probability of at least 2 of 11 people having the same birthday is  .140
The probability of at least 2 of 12 people having the same birthday is  .176
The probability of at least 2 of 13 people having the same birthday is  .194
The probability of at least 2 of 14 people having the same birthday is  .225
The probability of at least 2 of 15 people having the same birthday is  .251
The probability of at least 2 of 16 people having the same birthday is  .282
The probability of at least 2 of 17 people having the same birthday is  .312
The probability of at least 2 of 18 people having the same birthday is  .343
The probability of at least 2 of 19 people having the same birthday is  .376
The probability of at least 2 of 20 people having the same birthday is  .419
The probability of at least 2 of 21 people having the same birthday is  .442
The probability of at least 2 of 22 people having the same birthday is  .477
The probability of at least 2 of 23 people having the same birthday is  .505
The probability of at least 2 of 24 people having the same birthday is  .540
The probability of at least 2 of 25 people having the same birthday is  .577
The probability of at least 2 of 26 people having the same birthday is  .599
The probability of at least 2 of 27 people having the same birthday is  .630
The probability of at least 2 of 28 people having the same birthday is  .649
The probability of at least 2 of 29 people having the same birthday is  .676
The probability of at least 2 of 30 people having the same birthday is  .701
The probability of at least 2 of 31 people having the same birthday is  .736
The probability of at least 2 of 32 people having the same birthday is  .755
The probability of at least 2 of 33 people having the same birthday is  .777
The probability of at least 2 of 34 people having the same birthday is  .796
The probability of at least 2 of 35 people having the same birthday is  .817
The probability of at least 2 of 36 people having the same birthday is  .834
The probability of at least 2 of 37 people having the same birthday is  .850
The probability of at least 2 of 38 people having the same birthday is  .865
The probability of at least 2 of 39 people having the same birthday is  .883
The probability of at least 2 of 40 people having the same birthday is  .887
```

Note that the number of trials is set to 10,000 in this function. The number of trials can be increased to improve the accuracy of the estimated probability, or decreased to get extra speed if you computer is too slow. However, for reasonable accuracy, you should have at least a power of 10 more trials than the number of significant digits you are trying to calculate. In this case, we are attempting to calculate and display 3 significant digits (10^3), so we are using 10^4 trials.

7-29 A set of elapsed time subroutines is shown below. Note that they use a module to share data between the subroutines. **WARNING: If you use the Microsoft Fortran Powerstation 4.0, these procedures will give inaccurate results for short timing durations. The run-time library of Microsoft Fortran Powerstation 4.0 Compiler has a bug, and does not return any information in the millisecond field, so short-duration timings are not accurate.** Compaq Visual Fortran, Intel Fortran, Lahey Fortran, and NAGWare Fortran 95 do not appear to have this problem.

```
MODULE time_info
!
! Purpose:
!   To store information used by the timing subroutines.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/11/2007    S. J. Chapman       Original code
!
```

```
IMPLICIT NONE
SAVE                            ! Save all values

! Declare named constants:
INTEGER :: SEC_2_MS = 1000      ! Seconds to ms
INTEGER :: MIN_2_MS = 60000     ! Minutes to ms
INTEGER :: HR_2_MS = 3600000    ! Hours to ms
INTEGER :: DAY_2_MS = 86400000  ! Days to ms


! Declare variables for subroutine DATE_AND_TIME:
CHARACTER(len=8) :: date        ! Date string
CHARACTER(len=10) :: time       ! Time string
CHARACTER(len=5) :: zone        ! zone string
INTEGER,DIMENSION(8) :: values  ! Integer values

! Declare times in elapsed milliseconds since start of year.
INTEGER :: current_time         ! Current time in ms
INTEGER :: saved_time           ! Saved time in ms

END MODULE


SUBROUTINE set_timer
!
!  Purpose:
!    To start (or reset) the elapsed time counter.  Elapsed
!    time is returned by subsequent calls to subroutine
!    elapsed_time.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
USE time_info
IMPLICIT NONE

! Call DATE_AND_TIME
CALL DATE_AND_TIME (date, time, zone, values)

! Convert to ms and save.
saved_time = values(5) * HR_2_MS  + values(6) * MIN_2_MS &
           + values(7) * SEC_2_MS + values(8)

END SUBROUTINE set_timer


SUBROUTINE elapsed_time ( elapsed_seconds )
!
!  Purpose:
!    To calculate the elapsed time in seconds since the
!    last call to set_timer.  This timer is valid as
!    long as the timing exercise does not cross midnight.
!
!  Record of revisions:
```

```
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
USE time_info
IMPLICIT NONE

! Declare calling argument:
REAL,INTENT(OUT) :: elapsed_seconds  ! Elapsed time in seconds

! Declare local variables:
INTEGER :: elapsed_ms                ! Elapsed time in ms

! Call DATE_AND_TIME
CALL DATE_AND_TIME (date, time, zone, values)

! Convert to ms and save.
current_time = values(5) * HR_2_MS  + values(6) * MIN_2_MS &
             + values(7) * SEC_2_MS + values(8)

! Calculate elpased_time
elapsed_ms = current_time - saved_time
elapsed_seconds = REAL(elapsed_ms) / 1000.

END SUBROUTINE elapsed_time
```

7-30 A test program to determine the time required to sort 100, 1000, and 10000 element arrays is shown below. This program uses subroutine `random0` to generate arrays of random numbers for sorting, and the selection sort subroutine `sort` to do the sorting. Because the sort is so fast for small arrays, this program performs the sort 1000 times for the 100-element array, 100 times for the 1000-element array and 2 times for the 10000-element array, averaging the results. You may need to adjust the number of repetitions to get valid results for your computer.

```
PROGRAM test_sort
!
!  Purpose:
!    To determine the time required to sort random arrays
!    100, 1000, and 10000 elements long.  This program uses
!    the elapsed time routines from Exercise 6-31 to time the
!    sorting process.  To make the timing more accurate, the
!    short array will each be sorted many times, and the average
!    time will be calculated.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER, PARAMETER :: SIZE = 10000   ! Max array size

! Declare variables:
REAL,DIMENSION(SIZE) :: array         ! Array to sort
REAL,DIMENSION(SIZE) :: saved_array   ! Saved copy of array
INTEGER :: i, j, k                    ! Loop index
```

```
INTEGER :: nvals                      ! No, of values to sort
INTEGER,DIMENSION(2:4) :: n_loops = (/1000,100,2/) ! Number of loops
REAL :: sec                           ! Elapsed time

! Loop over arrays of various sizes.
DO i = 2, 4

   ! Get number of values.
   nvals = 10**i

   ! Calculate random variables.
   DO j = 1, nvals
      CALL random0 ( saved_array(j) )
   END DO

   ! Start the timer.
   CALL set_timer

   ! Main loop.  Sort the data n_loops(i) times, and calculate the
   ! average sorting time.  (This is especially needed for small
   ! array sizes.
   DO k = 1, n_loops(i)

      ! Copy array for sorting
      array = saved_array

      ! Sort array.
      CALL sort ( array, nvals )

   END DO

   ! Get elapsed time...
   CALL elapsed_time ( sec )

   ! Write out average time.
   WRITE (*,1000) nvals, sec / REAL(n_loops(i))
   1000 FORMAT (' Avg sort time for ',I6,' values = ', F10.4)

END DO

END PROGRAM test_sort
```

When this program is run on a 1.8 GHz Core 2 Duo system, the results are

```
C:\book\f95_2003\soln\ex7_30>test_sort
 Avg sort time for    100 values =     0.0000
 Avg sort time for   1000 values =     0.0012
 Avg sort time for  10000 values =     0.0935
```

The 100-element sort was so fast that the computer's system clock could not measure it accurately. Notice that the time required to sort the array increased approximately as the *square* of the size of the array to be sorted. If the size of the array increases by a factor of ten, then the time to do the sort increases by a factor of 100! Using this fact, we can predict that a 100,000 element array would take about 10 seconds to sort with this algorithm.

Fortunately, there are faster sorting algorithms available. We will meet an important one (the heapsort) in a later problem.

This problem also provides an interesting measure of the progress that PC's have made in the last 10 years. When the Solutions Manual for the first edition was prepared, this exercise was run on a 133 MHz Pentium system, with the following results:

```
C:\book\f95_2003\soln\ex7_30>test_sort
Avg sort time for    100 values =     0.0014
Avg sort time for   1000 values =     0.0769
Avg sort time for  10000 values =     7.9100
```

An inexpensive PC is about 85 times faster now than it was 10 years ago!

7-31 A Fortran function that calculates e^x using the first 12 terms of the infinite series $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$ is shown below.

```
FUNCTION exp1 ( x )
!
!  Purpose:
!    To calculate EXP(X) using an infinite series.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of calling arguments:
REAL,INTENT(IN) :: x              ! Input argument
REAL :: exp1                      ! Exponent

! List of local variables
INTEGER :: fact                   ! Factorial
INTEGER :: i                      ! Index variable
REAL :: xi                        ! x**i

! Calculate the first term of the series:
!   x**0 / 0! = 1.0
exp1 = 1.

! Calculate the next 11 terms of exp1(x).
xi   = 1.
fact = 1
DO i = 1, 11
   xi    = xi * x
   fact = fact * i
   exp1  = exp1 + xi / REAL(fact)
END DO

END FUNCTION exp1
```

A test driver program for this function is shown below.

```
PROGRAM test_exp1
!
!  Purpose:
!    To test function exp1(), which calculates the value of
!    E**X using a truncated infinite series, and compares the
```

```
!    result with output of intrinsic function EXP().
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of external functions:
REAL, EXTERNAL :: exp1

! List of variables
INTEGER :: i                  ! INdex variable
REAL,DIMENSION(8) :: x        ! Values to test

x = (/-10.,-5.,-1.,0.,1.,5.,10.,15./)

! Calculate E**X.
DO i = 1, 8
   WRITE (*,1000) x(i), EXP1(x(i)), x(i), EXP(x(i))
   1000 FORMAT (' EXP1(',F4.0,') = ',ES15.7,4X,'EXP(',F4.0,') = ', &
                ES15.7)
END DO

END PROGRAM test_exp1
```

When this program is run, the results are

```
C:\book\f95_2003\soln\ex7_31>test_exp1
EXP1(-10.) =  -1.1626235E+03    EXP(-10.) =   4.5399931E-05
EXP1( -5.) =  -3.5920841E-01    EXP( -5.) =   6.7379470E-03
EXP1( -1.) =   3.6787945E-01    EXP( -1.) =   3.6787945E-01
EXP1(  0.) =   1.0000000E+00    EXP(  0.) =   1.0000000E+00
EXP1(  1.) =   2.7182817E+00    EXP(  1.) =   2.7182817E+00
EXP1(  5.) =   1.4760385E+02    EXP(  5.) =   1.4841316E+02
EXP1( 10.) =   1.5347516E+04    EXP( 10.) =   2.2026465E+04
EXP1( 15.) =   6.0395681E+05    EXP( 15.) =   3.2690173E+06
```

Notice that the 12-term truncated infinite series approximation of e^x is very good for small $|x|$, but gets worse as $|x|$ gets very large. The approximation could be improved for large $|x|$ by including more terms in the series.

7-32 A program to calculate the average and standard deviation of an array of 10,000 uniform random values is shown below.

```
PROGRAM test_ave_sd
!
!  Purpose:
!    To verify the quality of the random numbers produced by subroutine RANO
!    by determining the standard deviation of 10,000 such numbers.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
```

```
! List of named constants:
INTEGER,PARAMETER :: NSAMP = 10000  ! Number of samples

! List of variables:
REAL :: ave                        ! Average
INTEGER :: error                   ! Error
INTEGER :: i                       ! Loop index
REAL :: std_dev                    ! Standard deviation
REAL,DIMENSION(NSAMP) :: value     ! Array of random values

! Calculate random values
 DO i = 1, nsamp
    CALL random0 ( value(i) )
 END DO

! Get average and standard deviation.
 CALL ave_sd ( value, nsamp, ave, std_dev, error )

!     Tell user.
WRITE (*,1000) ave, 0.5
1000 FORMAT (' Average value      = ',F10.6,'  Theoretical value =',&
              F10.6 )
WRITE (*,1010) std_dev, 1. / SQRT(12.)
1010 FORMAT (' Standard deviation = ',F10.6,'  Theoretical value =',&
              F10.6 )

END PROGRAM test_ave_sd
```

When this program is executed, the results are very close to the theoretical values:

```
C:\book\f95_2003\soln\ex7_32>test_ave_sd
Average value      =    .496803  Theoretical value =   .500000
Standard deviation =    .287766  Theoretical value =   .288675
```

7-33 A function to generate a normal Gaussian distribution with a zero average and a standard deviation of 1.0 is shown below.

```
REAL FUNCTION random_n( )
!
!  Purpose:
!    Function to generate a gaussian normal distribution with
!    a mean of 0.0 and a standard deviation of 1.0.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare local variables.
REAL :: r                 ! SQRT(ran(1)**2+ran(2)**2)
REAL :: v1                ! Unif random var [-1,1)
REAL :: v2                ! Unif random var [-1,1)
REAL :: x1                ! Unif random var [0,1)
REAL :: x2                ! Unif random var [0,1)
```

```
! Get 2 uniform random variables in the range [0.,1.) such
! that the square root of the sum of their squares < 1.
! Keep trying until we come up with such a combination.
DO
   CALL random_number( x1 )   ! Uniform random var [0,1)
   CALL random_number( x2 )   ! Uniform random var [0,1)
   v1 = 2. * x1 - 1.          ! Uniform random var [-1,1)
   v2 = 2. * x2 - 1.          ! Uniform random var [-1,1)
   r = v1**2 + v2**2
   IF ( r < 1. ) EXIT
END DO

! Calculate a Gaussian random variable from the uniform
! variables:
random_n = SQRT( -2. * LOG(r) / r ) * v1

END FUNCTION random_n
```

A test driver program is shown below:

```
PROGRAM test_random_n
!
!  Purpose:
!    Program to test function random_n.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! External functions:
REAL,EXTERNAL :: random_n         ! Normal distribution

! List of variables:
REAL :: ave                       ! Average of data set
INTEGER :: error                  ! Error flag
INTEGER :: i                      ! Index variable
REAL,DIMENSION(1000) :: x         ! Random values
REAL :: std_dev                   ! Std dev of data set

! Get 1000 random values
DO i = 1, 1000
   x(i) = random_n()
END DO

! Calculate average and standard deviation.
CALL ave_sd ( x, 1000, ave, std_dev, error )

! Tell user
WRITE (*,1000) ave, std_dev
1000 FORMAT (' Average            = ', F10.5,/, &
             ' Standard Deviation = ', F10.5 )
```

```
END PROGRAM test_random_n
```

When this program is executed, the results are very close to the theoretical values:

```
C:\book\f95_2003\soln\ex7_33>test_random_n
Average            =    -.01409
Standard Deviation =     .99891
```

7-34 A function to calculate the gravitational force between two masses is shown below:

```
FUNCTION force(range, mass1, mass2)
!
!  Purpose:
!    Function to calculate the gravitational force between two
!    objects of masses "mass1" and "mass2" separated by a distance
!    "range"
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments.
REAL,INTENT(IN) :: range      ! Range bewteen masses in meters
REAL,INTENT(IN) :: mass1      ! Mass of first object, in kg
REAL,INTENT(IN) :: mass2      ! Mass of second object, in kg
REAL :: force                 ! Force of gravity, in newtons

! Declare named constants:
REAL,PARAMETER :: GRAV_CONST = 6.672E-11    ! Gravitational constant

! Calculate the force
force = GRAV_CONST * mass1 * mass2 / range**2

END FUNCTION force
```

A test driver program for this function is shown below:

```
PROGRAM test_force
!
!  Purpose:
!    Program to test function force.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! External functions:
REAL,EXTERNAL :: force            ! Calculate gravitational force

! List of variables:
REAL :: range                 ! Range bewteen masses in meters
```

```
REAL :: mass1                  ! Mass of first object, in kg
REAL :: mass2                  ! Mass of second object, in kg

! Get the masses
WRITE (*,*) 'Enter mass 1, in kg: '
READ (*,*) mass1
WRITE (*,*) 'Enter mass 2, in kg: '
READ (*,*) mass2

! Get range
WRITE (*,*) 'Enter distance between objects: '
READ (*,*) range

! Tell user
WRITE (*,'(A,F12.6,A)') ' The resulting force is ', &
                        force(range,mass1,mass2), ' newtons.'

END PROGRAM test_force
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex7_34>test_force
Enter mass 1, in kg:
800
Enter mass 2, in kg:
5.98E24
Enter distance between objects:
38000000.
The resulting force is   221.044600 newtons.
```

7-35 The following program compares the time required to perform a selection sort with the time required to perform a heapsort on the same 5,000 element real array. Note that the heapsort was performed 500 times in order to get an average execution time, because each individual execution was too fast for accurate measurement.

```
PROGRAM test_sorts
!
!  Purpose:
!    To determine the time required to sort an array containing
!    5,000 random values using the selection sort algorithm and
!    the heapsort algorithm.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER,PARAMETER :: SIZE = 5000   ! Size of array

! List of variables:
REAL,DIMENSION(SIZE) :: copy1      ! Copy of array for selection sort
REAL,DIMENSION(SIZE) :: copy2      ! Copy of array for heapsort
INTEGER :: error                   ! Error flag
INTEGER :: i                       ! Loop index
REAL,DIMENSION(SIZE) :: orig       ! Array to sort
```

```
REAL :: sec1                        ! Time for selection sort
REAL :: sec2                        ! Time for heapsort

! Calculate random variables, and make copies.
CALL RANDOM_NUMBER ( orig )

! Start the timer for the selection sort.
CALL set_timer

! Sort array copy1 using the selection sort technique.
copy1 = orig
CALL sort ( copy1, SIZE )

! Get elapsed time.
CALL elapsed_time ( sec1 )

! Start the timer for the heapsort.
CALL set_timer

! Average 500 copies
DO i = 1, 500

   ! Make a copy of the array to sort
   copy2 = orig

   ! Sort array copy2 using the heapsort technique.
   CALL heapsort ( copy2, SIZE, error )
END DO

! Get elapsed time.
CALL elapsed_time ( sec2 )
sec2 = sec2 / 500.

! Tell user.
WRITE (*,1000) 'selection sort', sec1
WRITE (*,1000) 'heapsort      ', sec2
1000 FORMAT (' Sort time for ',A,' = ', F10.4)

END PROGRAM test_sorts
```

When this program executes, the results are:

```
C:\book\f95_2003\soln\ex7_35>test_sorts
 Sort time for selection sort =     0.0310
 Sort time for heapsort       =     0.0011
```

The heapsort algorithm is *much* faster than the selection sort algorithm.

Chapter 8. Additional Features of Arrays

8-1 *(a)* 180 elements; valid subscript range is (1,1) to (3,60). *(b)* 441 elements; valid subscript range is (-10,0) to (10,20). *(c)* 161051 (11 × 11 × 11 × 11 × 11) elements; valid subscript range is `(-5:-5:-5:-5:-5)` to `(5:5:5:5:5)`.

8-2 *(a)* Invalid. This code transposes the elements of array `b`, and in the process assigns values to nonexistent array elements. *(b)* 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. *(c)* Valid. The expression "`info` < 0" produces an 8-element logical array, so the output of the `WRITE` statement is: `F T T T T F T F`. *(d)* Valid. These statements produce the following array:

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

8-3 The specified array sections are given below:

(a) `my_array(3,:) =` $\begin{bmatrix} 11 & 12 & 13 & 14 & 15 \end{bmatrix}$

(b) `my_array(:,2) =` $\begin{bmatrix} 2 \\ 7 \\ 12 \\ 17 \\ 22 \end{bmatrix}$

(c) `my_array(1:5:2,:) =` $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 11 & 12 & 13 & 14 & 15 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}$

(d) `my_array(:,2:5:2) =` $\begin{bmatrix} 2 & 4 \\ 7 & 9 \\ 12 & 14 \\ 17 & 19 \\ 22 & 24 \end{bmatrix}$

(e) `my_array(1:5:2,1:5:2) =` $\begin{bmatrix} 1 & 3 & 5 \\ 11 & 13 & 15 \\ 21 & 23 & 25 \end{bmatrix}$

(f) `my_array(:,list)` = $\begin{bmatrix} 1 & 2 & 4 \\ 6 & 7 & 9 \\ 11 & 12 & 14 \\ 16 & 17 & 19 \\ 21 & 22 & 24 \end{bmatrix}$

8-4 The first `WRITE` statement is in a `DO` loop. It will be executed twice, and 4 values will be printed out each time. The second `WRITE` statement uses an implied `DO` loop to print out all 8 values at once. Since the format contains 6 descriptors, six values will be printed on one line and the remaining two on the following line. The output will be:

```
C:\book\f95_2003\soln>test_output1
   1    2    3    4
   5    6    7    8
   1    2    3    4    5    6
   7    8
```

8-5 *(a)* The `READ` statement here is executed 4 times. Each time, it reads the first four values from the current line into the corresponding row of the array. Therefore, array `values` will contain the following values

$$\texttt{values} = \begin{bmatrix} 27 & 17 & 10 & 8 \\ 11 & 13 & -11 & 12 \\ -1 & 0 & 0 & 6 \\ -16 & 11 & 21 & 26 \end{bmatrix}$$

(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

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

(c) The `READ` statement here is executed 4 times. Each time, it reads the first four values from the current line into the corresponding row of the array. Therefore, array `values` will contain the following values

$$\texttt{values} = \begin{bmatrix} 27 & 17 & 10 & 8 \\ 11 & 13 & -11 & 12 \\ -1 & 0 & 0 & 6 \\ -16 & 11 & 21 & 26 \end{bmatrix}$$

(d) 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 *column* order. Therefore, array `values` will contain the following values

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

8-6 This program declares a 6 × 11 array `info`, and then get information about the array using intrinsic functions. When it is executed, the results are:

```
C:\book\f95_2003\soln>test
The shape of the array is:            6    11
The size of the array is:            66
The lower bounds of the array are:    5     5
The upper bounds of the array are:   10    15
```

8-7 *(a)* The statements required to count the positive, negative, and zero values in the array without using array intrinsic functions are:

```
REAL, DIMENSION(-50:50) :: values   ! Values
INTEGER :: i                        ! Loop index
INTEGER :: n_neg = 0                ! Number negative
INTEGER :: n_pos = 0                ! Number positive
INTEGER :: n_zero = 0               ! Number zero

DO i = -50, 50
   IF ( values(i) < 0.0 ) THEN
      n_neg = n_neg + 1
   ELSE IF ( values(i) == 0.0 ) THEN
      n_zero = n_zero + 1
   ELSE
      n_pos = n_pos + 1
   END IF
END DO

! Write summary statistics.
WRITE (*,1000) n_neg, n_zero, n_pos
1000 FORMAT (1X,'The distribution of values is:',/, &
             1X,'  Number of negative values = ', I3,/, &
             1X,'  Number of zero values     = ', I3,/, &
             1X,'  Number of positive values = ', I3)
```

(b) If array intrinsic function `COUNT` is used, the code can become:

```
REAL, DIMENSION(-50:50) :: values   ! Values
INTEGER :: n_neg = 0                ! Number negative
INTEGER :: n_pos = 0                ! Number positive
INTEGER :: n_zero = 0               ! Number zero

n_neg  = COUNT ( values < 0. )
n_pos  = COUNT ( values > 0. )
n_zero = COUNT ( values == 0. )

! Write summary statistics.
WRITE (*,1000) n_neg, n_zero, n_pos
1000 FORMAT (1X,'The distribution of values is:',/, &
             1X,'  Number of negative values = ', I3,/, &
             1X,'  Number of zero values     = ', I3,/, &
             1X,'  Number of positive values = ', I3)
```

8-8 The following program reads in a rank-2 array from an input disk file, calculates the sums all the data in each row and each column in the array, and displays the results.

```
PROGRAM sum_rows_and_cols
!
!  Purpose:
!    To read in a array, and find the sums of all rows and
!    columns in the array.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER, PARAMETER :: MAXSIZ = 10    ! Max size of array

! List of variables:
REAL, DIMENSION(MAXSIZ,MAXSIZ) :: a     ! Array
CHARACTER(len=30) :: filename           ! Input file name
INTEGER :: i, j                         ! Loop index
INTEGER :: istat                        ! I/o status
INTEGER :: ncol                         ! No. of cols used in a
INTEGER :: nrow                         ! No. of rows used in a
REAL, DIMENSION(MAXSIZ) :: sum_col = 0. ! Sum of each column.
REAL, DIMENSION(MAXSIZ) :: sum_row = 0. ! Sum of each row.

! Get the name of the disk file containing the array.
WRITE (*,100)
100 FORMAT (' Enter the file name containing the array: ')
READ (*,'(A30)') filename

! Open input data file.  Status is OLD because the input data must
! already exist.
OPEN (UNIT=1, FILE=filename, STATUS='OLD', ACTION='READ', IOSTAT=istat)

! Was the OPEN successful?
openok: IF ( istat == 0 ) THEN

   ! The file was opened successfully, so read the size of array A.
   READ (1,*) nrow, ncol

   ! If the sizes are <= MAXSIZ, read A in and process it.
   sizeok: IF ( (nrow <= MAXSIZ ) .AND. (ncol <= MAXSIZ ) ) THEN
      DO i = 1, nrow
         READ (1,*) (a(i,j), j=1,ncol)
      END DO

      ! Sum the rows and columns.
      DO i = 1, nrow
         sum_row(i) = SUM ( a(i,:) )
      END DO
      DO j = 1, ncol
         sum_col(j) = SUM ( a(:,j) )
      END DO
```

```
      ! Write results.
      DO i = 1, nrow
         WRITE (*,110) i, sum_row(i)
         110 FORMAT (' Sum of row ',I2,' = ',F12.4)
      END DO
      DO j = 1, ncol
         WRITE (*,120) j, sum_col(j)
         120 FORMAT (' Sum of col ',I2,' = ',F12.4)
      END DO
   END IF sizeok

ELSE openok

   WRITE (*,130) filename, istat
   130 FORMAT (' ERROR: Open error on file ',A,': IOSTAT = ',I6)

END IF openok

END PROGRAM sum_rows_and_cols
```

8-9 When this program is run with the data set given in the problem, the results are

```
C:\book\f95_2003\soln>sum_rows_and_cols
Enter the file name containing the array:
in8_8.dat
Sum of row  1 =      35.6000
Sum of row  2 =      -3.4000
Sum of row  3 =      -8.4000
Sum of row  4 =      -8.6000
Sum of col  1 =      31.7000
Sum of col  2 =     -18.7000
Sum of col  3 =       2.5000
Sum of col  4 =       8.8000
Sum of col  5 =      -9.1000
```

8-10 A version of the row and column summation program from Exercise 5-16 that uses allocatable arrays is shown below:

```
PROGRAM sum_rows_and_cols
!
!  Purpose:
!    To read in a array, and find the sums of all rows and
!    columns in the array.  This version of the program uses
!    allocatable arrays to adjust to problems of any size.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/11/2007    S. J. Chapman      Original code
! 1. 05/11/2007    S. J. Chapman      Use allocatable arrays
!
IMPLICIT NONE

! List of named constants:
INTEGER, PARAMETER :: MAXSIZ = 10    ! Max size of array
```

```
! List of variables:
REAL, ALLOCATABLE, DIMENSION(:,:) :: a  ! Array
CHARACTER(len=30) :: filename           ! Input file name
INTEGER :: i, j                         ! Loop index
INTEGER :: istat                        ! I/o status
INTEGER :: ncol                         ! No. of cols used in a
INTEGER :: nrow                         ! No. of rows used in a
REAL, DIMENSION(MAXSIZ) :: sum_col = 0. ! Sum of each column.
REAL, DIMENSION(MAXSIZ) :: sum_row = 0. ! Sum of each row.

! Get the name of the disk file containing the array.
WRITE (*,100)
100 FORMAT (' Enter the file name containing the array: ')
READ (*,'(A30)') filename

! Open input data file.  Status is OLD because the input data must
! already exist.
OPEN (UNIT=1, FILE=filename, STATUS='OLD', ACTION='READ', IOSTAT=istat)

! Was the OPEN successful?
openok: IF ( istat == 0 ) THEN

   ! The file was opened successfully, so read the size of array A.
   READ (1,*) nrow, ncol

   ! Allocate array a with the right size.
   ALLOCATE ( a(nrow,ncol), STAT=istat )

   ! If allocation is ok, read the data.
   alloc_ok: IF ( istat == 0 ) THEN
      DO i = 1, nrow
         READ (1,*) (a(i,j), j=1,ncol)
      END DO

      ! Sum the rows and columns.
      DO i = 1, nrow
         sum_row(i) = SUM ( a(i,:) )
      END DO
      DO j = 1, ncol
         sum_col(j) = SUM ( a(:,j) )
      END DO

      ! Write results.
      DO i = 1, nrow
         WRITE (*,110) i, sum_row(i)
         110 FORMAT (' Sum of row ',I2,' = ',F12.4)
      END DO
      DO j = 1, ncol
         WRITE (*,120) j, sum_col(j)
         120 FORMAT (' Sum of col ',I2,' = ',F12.4)
      END DO

      ! Deallocate array
      DEALLOCATE ( a, STAT=istat )

   END IF alloc_ok
```

```
ELSE openok

   WRITE (*,130) filename, istat
   130 FORMAT (' ERROR: Open error on file ',A,': IOSTAT = ',I6)

END IF openok

END PROGRAM sum_rows_and_cols
```

8-11 This is a placeholder for a solution that will be added when the Fortran 95/2003 compilers mature more.

8-12 A set of Fortran statements that would search a rank-3 array `arr` and limit the maximum value of any array element to be less than or equal to 1000 is:

```
DO i = 1, 1000
   DO j = 1, 10
      DO k = 1, 30
         IF ( arr(i,j,k) > 1000. ) THEN
            arr(i,j,k) = 1000.
         END IF
      END DO
   END DO
END DO
```

A `WHERE` construct to do the same job is:

```
WHERE ( arr > 1000. )
   arr = 1000.
END WHERE
```

8-13 The raw data for this problem has been placed in placed in a file IN8_13.DAT. The contents of this file are

```
        90.0 W  90.5 W  91.0 W  91.5 W  92.0 W  92.5 W
30.0 N  68.2    72.1    72.5    74.1    74.4    74.2
30.5 N  69.4    71.1    71.9    73.1    73.6    73.7
31.0 N  68.9    70.5    70.9    71.5    72.8    73.0
31.5 N  68.6    69.9    70.4    70.8    71.5    72.2
32.0 N  68.1    69.3    69.8    70.2    70.9    71.2
32.5 N  68.3    68.8    69.6    70.0    70.5    70.9
----|----|----|----|----|----|----|----|----|----|----|
    5   10   15   20   25   30   35   40   45   50   55
```

The program will read the data from this file, and perform the calculations on it. In addition, it will read the row and column labels, so that it can properly label each of the resulting calculations. The labels will be stored in character arrays until they are needed.

```
PROGRAM ave_temp
!
!  Purpose:
!    To read in average temperature values from a meteorological
!    experiment, and calculate average temperatures at each
!    latitude, each longitude, and over the whole region.
!
!  Record of revisions:
```

```
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER, PARAMETER :: N_LAT = 6     ! No. of latitudes
INTEGER, PARAMETER :: N_LONG = 6    ! No. of longitudes

! List of variables:
REAL :: ave_global                  ! Global ave temp
REAL, DIMENSION(N_LAT) :: ave_lat   ! Ave temp for latitude
REAL, DIMENSION(N_LONG) :: ave_long ! Ave temp for longitude
CHARACTER(len=30) :: filename       ! Input file name
INTEGER :: i, j                     ! Loop index
INTEGER :: istat                    ! I/o status
CHARACTER(len=6),DIMENSION(N_LAT) :: lat
                                    ! Names of each latitude
CHARACTER(len=6),DIMENSION(N_LONG) :: long
                                    ! Names of each longitude
REAL,DIMENSION(N_LAT,N_LONG) :: temp ! Temperatures

! Get the name of the file containing the input data.
WRITE (*,1000)
1000 FORMAT (' Enter the file name with the data to be processed: ')
READ (*,'(A30)') filename

! Open input data file.  Status is OLD because the input data must
! already exist.
OPEN (UNIT=9, FILE=filename, STATUS='OLD', ACTION='READ', IOSTAT=istat)

! Was the OPEN successful?
openok: IF ( istat == 0 ) THEN

   ! The file was opened successfully, so read the longitude labels.
   READ (9,'(8X,6(A6,2X))') long

   ! Get the latitude labels and the temperature data.
   DO i = 1, N_LAT
      READ (9,1030) lat(i), (temp(i,j), j=1, N_LONG)
      1030 FORMAT (A6,2X,6F8.2)
   END DO

   ! Calculate average temperature at each latitude.
   DO i = 1, N_LAT
      ave_lat(i) = SUM ( temp(i,:) ) / N_LAT
   END DO

   ! Calculate average temperature at each longitude.
   DO j = 1, N_LONG
      ave_long(j) = SUM ( temp(:,j) ) / N_LONG
   END DO

   ! Calculate overall average temperature.
   ave_global = SUM ( temp ) / ( N_LAT * N_LONG )
```

```
   ! Tell user the results.
   DO i = 1, N_LAT
      WRITE (*,1040) lat(i), ave_lat(i)
      1040 FORMAT (' The average temperature at latitude  ',&
                   A,' is ', F5.2,' degrees.')
   END DO

   DO j = 1, N_LONG
      WRITE (*,1050) long(j), ave_long(j)
      1050 FORMAT (' The average temperature at longitude ',&
                   A,' is ', F5.2,' degrees.')
   END DO

   WRITE (*,1060) ave_global
   1060 FORMAT (' The average for the whole area is ',&
                F5.2,' degrees.')

   ! Close input file.
   CLOSE (UNIT=9)

ELSE openok

   ! If we get here, the file failed to open properly.
   WRITE (*,1070) filename, istat
   1070 FORMAT (' Open failure on file ',A,' IOSTAT = ',I6)

END IF openok

END PROGRAM ave_temp
```

When this program is run with the given data, the results are:

```
C:\book\f95_2003\soln\ex8_13>ave_temp
Enter the file name with the data to be processed:
in8_13.dat
The average temperature at latitude  30.0 N is 72.58 degrees.
The average temperature at latitude  30.5 N is 72.13 degrees.
The average temperature at latitude  31.0 N is 71.27 degrees.
The average temperature at latitude  31.5 N is 70.57 degrees.
The average temperature at latitude  32.0 N is 69.92 degrees.
The average temperature at latitude  32.5 N is 69.68 degrees.
The average temperature at longitude 90.0 W is 68.58 degrees.
The average temperature at longitude 90.5 W is 70.28 degrees.
The average temperature at longitude 91.0 W is 70.85 degrees.
The average temperature at longitude 91.5 W is 71.62 degrees.
The average temperature at longitude 92.0 W is 72.28 degrees.
The average temperature at longitude 92.5 W is 72.53 degrees.
The average for the whole area is 71.03 degrees.
```

8-14 A program to multiply two matrices together is shown below:

```
PROGRAM mat_mult
!
!  Purpose:
!    To read in two matrices and multiply them if they are of
!    compatible sizes.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/11/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
REAL,ALLOCATABLE,DIMENSION(:,:) :: a ! First array
REAL,ALLOCATABLE,DIMENSION(:,:) :: b ! Second array
REAL,ALLOCATABLE,DIMENSION(:,:) :: c ! Result array
CHARACTER(len=30) :: filename1       ! File containing a
CHARACTER(len=30) :: filename2       ! File containing b
INTEGER :: i, j, k                   ! Index variables
INTEGER :: istat1                    ! I/o status for a
INTEGER :: istat2                    ! I/o status for b
INTEGER :: ncol1                     ! No. cols in a
INTEGER :: ncol2                     ! No. cols in b
INTEGER :: nrow1                     ! No. rows in a
INTEGER :: nrow2                     ! No. rows in b

! Get the name of the disk file containing array A.
WRITE (*,1000)
1000 FORMAT (' Enter the file name containing array A: ')
READ (*,'(A30)') filename1

! Get the name of the disk file containing array B.
WRITE (*,1010)
1010 FORMAT (' Enter the file name containing array B: ')
READ (*,'(A30)') filename2

! Open input data files.  Status is OLD because the input data
! must already exist.
OPEN (UNIT=1, FILE=filename1, STATUS='OLD', ACTION='READ', IOSTAT=istat1)
OPEN (UNIT=2, FILE=filename2, STATUS='OLD', ACTION='READ', IOSTAT=istat2)

! Were the OPENs successful?
openok: IF ( (istat1 == 0) .AND. (istat2 == 0) ) THEN

   ! The files were opened successfully.  Read the size of array A.
   READ (1,*) nrow1, ncol1

   ! Read the size of array B.
   READ (2,*) nrow2, ncol2

   ! Allocate arrays
   ALLOCATE (a(nrow1,ncol1), b(nrow2,ncol2), c(nrow1,ncol2), STAT=istat1)

   ! Is allocation ok?
   sizecheck: IF ( istat1 /= 0 ) THEN
```

```
      WRITE (*,1020) istat1
      1020 FORMAT (' Error--Array allocation error:',I6)

   ! If ncol1 <> nrow2, tell user and quit.
   ELSE IF ( ncol1 /= nrow2 ) THEN sizecheck

      WRITE (*,1030) nrow1, ncol1, nrow2, ncol2
      1030 FORMAT (' Error--Incompatible sizes: A is ',I2, ' x ', &
                   I2,', and B is ', I2, ' x ', I2,'.')
   ELSE sizecheck

      !  Ok--Read matrices A and B.
      DO i = 1, nrow1
         READ (1,*) (a(i,j), j=1,ncol1)
      END DO

      DO i = 1, nrow2
         READ (2,*) (b(i,j), j=1,ncol2)
      END DO

      ! Multiply elements
      DO i = 1, nrow1
         DO j = 1, ncol2
            c(i,j) = 0.
            DO k = 1, ncol1
               c(i,j) = c(i,j) + a(i,k) * b(k,j)
            END DO
         END DO
      END DO

      ! Write out the result.
      WRITE (*,*) 'The resulting matrix C is:'
      DO i = 1, nrow1
         WRITE (*,1050) (c(i,j), j=1,ncol2)
         1050 FORMAT (1X,8(F9.2,1X))
      END DO

      ! Deallocate arrays
      DEALLOCATE (a, b, c)

   END IF sizecheck

   ! Close files
   CLOSE (UNIT=1)
   CLOSE (UNIT=2)

ELSE openok

   ! If we get here, there was an error opening one of the files.
   ! Tell user, and quit.
   IF ( istat1 /= 0 ) THEN
      WRITE (*,1060) filename1, istat1
      1060 FORMAT (' Error opening file ',A,': IOSTAT = ',I6)
   END IF
   IF ( istat2 /= 0 ) THEN
```

```
      WRITE (*,1060) filename2, istat2
   END IF
END IF openok

END PROGRAM mat_mult
```

If matrices A and B from this problem are placed in files `in8_14.a` and `in8_14.b` respectively, then when program `mat_mult` is tested, the results are:

```
C:\book\f95_2003\soln\ex8_14>mat_mult
Enter the file name containing array A:
in8_14.a
Enter the file name containing array B:
in8_14.b
The resulting matrix C is:
     1.00     15.00
     5.00     -2.00
```

8-15 First, place matrix A in file `in8_15.a` and matrix B in file `in8_15.b`. The contents of these two files are:

```
in8_15.a:
         2         4
       1.0      -5.0       4.0       2.0
      -6.0      -4.0       2.0       2.0

in8_15.b:
         4         3
       1.0      -2.0      -1.0
       2.0       3.0       4.0
       0.0      -1.0       2.0
       0.0      -3.0       1.0
```

When program `mat_mult` is run with this input data, the results are:

```
C:\book\f95_2003\soln\ex8_15>mat_mult
Enter the file name containing array A:
in8_15.a
Enter the file name containing array B:
in8_15.b
The resulting matrix C is:
    -9.00    -27.00    -11.00
   -14.00     -8.00     -4.00
```

There are two rows and three columns in the resulting matrix *C*.

8-16 A rewritten program using the `MATMUL` intrinsic function is shown below:

```
PROGRAM mat_mult
!
!  Purpose:
!    To read in two matrices and multiply them if they are of
!    compatible sizes.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
```

```
!    05/11/2007    S. J. Chapman      Original code
! 1. 05/12/2007    S. J. Chapman      Modified to use MATMUL
!
IMPLICIT NONE

! List of variables:
REAL,ALLOCATABLE,DIMENSION(:,:) :: a ! First array
REAL,ALLOCATABLE,DIMENSION(:,:) :: b ! Second array
REAL,ALLOCATABLE,DIMENSION(:,:) :: c ! Result array
CHARACTER(len=30) :: filename1       ! File containing a
CHARACTER(len=30) :: filename2       ! File containing b
INTEGER :: i, j                      ! Index variables
INTEGER :: istat1                    ! I/o status for a
INTEGER :: istat2                    ! I/o status for b
INTEGER :: ncol1                     ! No. cols in a
INTEGER :: ncol2                     ! No. cols in b
INTEGER :: nrow1                     ! No. rows in a
INTEGER :: nrow2                     ! No. rows in b

! Get the name of the disk file containing array A.
WRITE (*,1000)
1000 FORMAT (' Enter the file name containing array A: ')
READ (*,'(A30)') filename1

! Get the name of the disk file containing array B.
WRITE (*,1010)
1010 FORMAT (' Enter the file name containing array B: ')
READ (*,'(A30)') filename2

! Open input data files.  Status is OLD because the input data
! must already exist.
OPEN (UNIT=1, FILE=filename1, STATUS='OLD', ACTION='READ', IOSTAT=istat1)
OPEN (UNIT=2, FILE=filename2, STATUS='OLD', ACTION='READ', IOSTAT=istat2)

! Were the OPENs successful?
openok: IF ( (istat1 == 0) .AND. (istat2 == 0) ) THEN

   ! The files were opened successfully.  Read the size of array A.
   READ (1,*) nrow1, ncol1

   ! Read the size of array B.
   READ (2,*) nrow2, ncol2

   ! Allocate arrays
   ALLOCATE (a(nrow1,ncol1), b(nrow2,ncol2), c(nrow1,ncol2), STAT=istat1)

   ! Is allocation ok?
   sizecheck: IF ( istat1 /= 0 ) THEN

      WRITE (*,1020) istat1
      1020 FORMAT (' Error--Array allocation error:',I6)

   ! If ncol1 <> nrow2, tell user and quit.
   ELSE IF ( ncol1 .NE. nrow2 ) THEN sizecheck

      WRITE (*,1030) nrow1, ncol1, nrow2, ncol2
```

```
      1030 FORMAT (' Error--Incompatible sizes: A is ',I2, ' x ', &
                    I2,', and B is ', I2, ' x ', I2,'.')
   ELSE sizecheck

      !  Ok--Read matrices A and B.
      DO i = 1, nrow1
         READ (1,*) (a(i,j), j=1,ncol1)
      END DO

      DO i = 1, nrow2
         READ (2,*) (b(i,j), j=1,ncol2)
      END DO

      ! Multiply elements
      c = MATMUL(a,b)

      ! Write out the result.
      WRITE (*,*) 'The resulting matrix C is:'
      DO i = 1, nrow1
         WRITE (*,1050) (c(i,j), j=1,ncol2)
         1050 FORMAT (1X,8(F9.2,1X))
      END DO

      ! Deallocate arrays
      DEALLOCATE (a, b, c)

   END IF sizecheck

   ! Close files
   CLOSE (UNIT=1)
   CLOSE (UNIT=2)

ELSE openok

   ! If we get here, there was an error opening one of the files.
   ! Tell user, and quit.
   IF ( istat1 /= 0 ) THEN
      WRITE (*,1060) filename1, istat1
      1060 FORMAT (' Error opening file ',A,': IOSTAT = ',I6)
   END IF
   IF ( istat2 /= 0 ) THEN
      WRITE (*,1060) filename2, istat2
   END IF
END IF openok

END PROGRAM mat_mult
```

8-17 A program to locate the relative maxima in an array is shown below:

```
PROGRAM relative_max
!
!  Purpose:
!    To locate all of the relative maxima within an input data
!    array A.
!
!  Record of revisions:
```

```
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
REAL,ALLOCATABLE,DIMENSION(:,:) :: a   ! Array to search
CHARACTER(len=30) :: filename          ! File name
INTEGER :: i, j                        ! Index variables
INTEGER :: istat                       ! I/o status
INTEGER :: ncol                        ! No. of cols in A
INTEGER :: nrow                        ! No. of rows in A

! Get the name of the disk file containing array A.
WRITE (*,1000)
1000 FORMAT (' Enter the file name containing array a: ')
READ (*,'(A30)') filename

! Open input data files.  Status is OLD because the input data
! must already exist.
!
OPEN (UNIT=1, FILE=filename, STATUS='OLD', ACTION='READ', IOSTAT=istat)

! Was the OPEN successful?
open_ok: IF ( istat == 0 ) THEN

   ! The file was opened successfully.  Read the size of array A.
   READ (1,*) nrow, ncol

   ! Allocate A
   ALLOCATE ( A(nrow,ncol), STAT=istat )

   ! Is allocation ok?
   alloc_ok: IF ( istat /= 0 ) THEN

      ! Error.
      WRITE (*,1010) istat
      1010 FORMAT (' Error--Array allocation error:',I6)

   ELSE alloc_ok

      ! Read matrix A.
      DO i = 1, nrow
         READ (1,*) (a(i,j), j=1,ncol)
      END DO

      ! Search for relative maxima.
      DO i = 2, nrow-1
         DO j = 2, ncol-1

            IF ( ( a(i,j) > a(i-1,j-1) ) .AND. &
                 ( a(i,j) > a(i-1,j  ) ) .AND. &
                 ( a(i,j) > a(i-1,j+1) ) .AND. &
                 ( a(i,j) > a(i  ,j-1) ) .AND. &
                 ( a(i,j) > a(i  ,j+1) ) .AND. &
```

```
                 ( a(i,j) > a(i+1,j-1) ) .AND. &
                 ( a(i,j) > a(i+1,j  ) ) .AND. &
                 ( a(i,j) > a(i+1,j+1) ) ) THEN

               ! a(i,j) is a relative maximum.
               WRITE (*,1020) i, j, a(i,j)
               1020 FORMAT (' Relative maximum: a(',I3,&
                         ',',I3,') = ',F14.4)
            END IF
         END DO
      END DO

      ! Deallocate array
      DEALLOCATE (a)

   END IF alloc_ok

   ! Close file
   CLOSE (UNIT=1)

ELSE open_ok

   ! If we get here, there was an error opening the file.
   ! Tell user, and quit.
   IF ( istat /= 0 ) THEN
      WRITE (*,1030) filename, istat
      1030 FORMAT (' Error opening file ',A,': IOSTAT = ',I6)
   END IF

END IF open_ok

END PROGRAM relative_max
```

When the program is run on the array in Exercise 5-30, the results are:

```
C:\book\f95_2003\soln\ex8_17>relative_max
Enter the file name containing array a:
in8_17.dat
Relative maximum: a(  2,  4) =         5.0000
Relative maximum: a(  4,  4) =         6.0000
Relative maximum: a(  5,  2) =        -3.0000
```

8-18 A program to calculate the steady-state temperature on the plate is shown below:

```
PROGRAM calc_temperature
!
!  Purpose:
!    To calculate the steady-state temperature in a metallic
!    plate.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE
```

```
! List of variables:
INTEGER :: count = 0               ! Iteration count
REAL,DIMENSION(10,10) :: delt      ! Array of temp diffs
REAL,DIMENSION(10,10) :: old_temp  ! Array of prev temps
REAL,DIMENSION(10,10) :: temp      ! Array of temperatures
INTEGER :: i, j                    ! Index variables

! Set initial temperature conditions on plate
temp = 50.              ! Internal segments
temp(1,:) = 20.         ! Border segments
temp(10,:) = 20.        ! Border segments
temp(:,1) = 20.         ! Border segments
temp(:,10) = 20.        ! Border segments
temp(3,8) = 100.        ! Hot segment

! Now iterate temperatures.
iterate: DO
   old_temp = temp            ! Save old temperature dist
   count = count + 1          ! Bump iteration counter

   ! Iterate temperatures for all variable segments
   DO i = 2, 9
      DO j = 2, 9
         IF ( .NOT.(i==3 .AND. j==8) ) THEN  ! Not hot segment
            ! Iterate temperature on segment
            temp(i,j) = 0.25 * (temp(i+1,j) + temp(i-1,j) &
                      + temp(i,j+1) + temp(i,j-1))
         END IF
      END DO
   END DO

   ! Check for convergence
   delt = temp - old_temp
   IF ( MAXVAL(ABS(delt)) < 0.01 ) THEN

      ! We have converged.  Tell user and exit loop.
      WRITE (*,1000) count
      1000 FORMAT (' Convergence in ',I3,' iterations:')
      WRITE (*,1010) ((temp(i,j), j=1,10), i=1,10)
      1010 FORMAT (1X,10F7.2)
      EXIT iterate

   END IF
END DO iterate

END PROGRAM calc_temperature
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln\ex8_18>calc_temperature
Convergence in  48 iterations:
  20.00  20.00  20.00  20.00  20.00  20.00  20.00  20.00  20.00  20.00
  20.00  20.90  22.02  23.66  26.37  31.05  38.88  48.25  34.13  20.00
  20.00  21.58  23.50  26.26  30.75  38.96  56.21 100.00  48.25  20.00
  20.00  21.91  24.14  27.11  31.39  37.80  47.01  56.21  38.88  20.00
```

20.00	21.91	24.04	26.63	29.89	33.85	37.80	38.95	31.05	20.00
20.00	21.67	23.45	25.45	27.67	29.88	31.38	30.73	26.36	20.00
20.00	21.30	22.64	24.05	25.45	26.61	27.09	26.24	23.65	20.00
20.00	20.87	21.76	22.64	23.44	24.02	24.13	23.49	22.01	20.00
20.00	20.44	20.87	21.29	21.66	21.89	21.90	21.57	20.89	20.00
20.00	20.00	20.00	20.00	20.00	20.00	20.00	20.00	20.00	20.00

Chapter 9. Additional Features of Procedures

9-1 For explicit-shape dummy arrays, both the array and all of its bounds are passed as arguments when the subroutine is called, and the array is declared to be of shape specified by the calling arguments. When an explicit-shape dummy array is used, the procedure has complete information about the array, and all array intrinsic functions may be used with it. Bounds checkers will also work with the array. The principal disadvantage is that all of the calling bounds must be included as calling arguments to the subroutine. An example of an explicit-shape dummy array is:

```
SUBROUTINE test1 (array, l1, u1)
INTEGER,INTENT(IN) :: l1, u1
REAL,DIMENSION(l1:u1) :: array            ! Explicit-shape
...
```

Assumed-shape dummy arrays pass the same information to a procedure without having to explicitly pass all of the array boundaries. Instead, the same information is passed through an explicit interface. When an assumed-shape dummy array is used with an explicit interface, the procedure has complete information about the array, and all array intrinsic functions may be used with it. Bounds checkers will also work with the array. The principal disadvantage is that there *must* be an explicit interface to the procedure. An example of an assumed-shape dummy array is:

```
SUBROUTINE test2 (array)
REAL,DIMENSION(:) :: array                 ! Assumed-shape
...
```

Assumed-size dummy arrays do not pass the final array boundary to the procedure, either explicitly via calling arguments or implicitly via an explicit interface. *The procedure does not know the actual size and shape of the array*. Many array intrinsic functions will not work with the array, and bounds checkers will not work with the array. With such arrays, it is easy for a procedure to access elements of an array that don't really exist. **Assumed-size dummy arrays should never be used in any modern program**. An example of an assumed-size dummy array is:

```
SUBROUTINE test3 (array)
REAL,DIMENSION(*) :: array                 ! Assumed-size
...
```

9-2 Internal procedures are exactly like external procedures, with the following exceptions.

1. An internal procedure can *only* be invoked from its host procedure. No other procedure within the program can access it.

2. The name of an internal procedure may not be passed as a command line argument to another procedure.

3. An internal procedure inherits all of the data entities (parameters and variables) of its host program unit by host association.

Why use internal procedures? In some problems, there are low-level manipulations that must be performed repeatedly as a part of the solution. These low-level manipulations can be simplified by defining an internal procedure to perform them. If the low-level manipulations should be performed in only one place (the host procedure), then using internal procedures prevents accidental misuse.

9-3 According to the Fortran 95/2003 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.

9-5 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\f95_2003\soln\ex9_5>exercise9_5
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
```

9-6 A subroutine to multiply two matrices if they are of compatible sizes is shown below.

```
SUBROUTINE mat_mult ( a, idrow1, idcol1, b, idrow2, idcol2, &
                      c, idrow3, idcol3, nrow1, ncol1, nrow2, &
                      ncol2, error )
!
!  Purpose:
!    To multiply two-dimensional matrices A and B of sizes
!    (nrow1 x ncol1) and (ncol2 x ncol2) respectively, and
!    store the result in a third matrix C of size (nrow1 x ncol2).
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/12/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of calling arguments:
INTEGER,INTENT(IN) :: idrow1      ! No. of rows in matrix A
INTEGER,INTENT(IN) :: idrow2      ! No. of rows in matrix B
INTEGER,INTENT(IN) :: idrow3      ! No. of rows in matrix C
INTEGER,INTENT(IN) :: idcol1      ! No. of cols in matrix A
INTEGER,INTENT(IN) :: idcol2      ! No. of cols in matrix B
INTEGER,INTENT(IN) :: idcol3      ! No. of cols in matrix C
REAL,INTENT(IN),DIMENSION(idrow1,idcol1) ::  a  ! Matrix A
REAL,INTENT(IN),DIMENSION(idrow2,idcol2) ::  b  ! Matrix B
REAL,INTENT(OUT),DIMENSION(idrow3,idcol3) :: c  ! Matrix C
INTEGER,INTENT(IN) :: nrow1       ! No. of rows used in A
INTEGER,INTENT(IN) :: ncol1       ! No. of cols used in A
```

```
INTEGER,INTENT(IN) :: nrow2       ! No. of rows used in B
INTEGER,INTENT(IN) :: ncol2       ! No. of cols used in B
INTEGER,INTENT(OUT) :: error      ! Error flag: 0 = No error
                                  !             1 = size mismatch

! List of local variables:
INTEGER :: i, j, k                ! Loop index

! Are the sizes compatible?
IF ( ncol1 == nrow2 ) THEN

   ! Multiply elements
   DO i = 1, nrow1
      DO j = 1, ncol2
         c(i,j) = 0.
         DO k = 1, ncol1
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
         END DO
      END DO
   END DO
   error = 0

ELSE

   error = 1

END IF

END SUBROUTINE mat_mult
```

A test driver program for this subroutine is shown below.

```
PROGRAM test_mat_mult
!
!  Purpose:
!    To test subroutine mat_mult.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER, PARAMETER :: MAXSIZ = 20    ! Maximum array size

! List of variables:
REAL,DIMENSION(MAXSIZ,MAXSIZ) :: a   ! Array A
REAL,DIMENSION(MAXSIZ,MAXSIZ) :: b   ! Array B
REAL,DIMENSION(MAXSIZ,MAXSIZ) :: c   ! Array C
INTEGER :: error                     ! Error flag
CHARACTER(len=30) :: filename1       ! Name of file containing A
CHARACTER(len=30) :: filename2       ! Name of file containing B
INTEGER :: i, j                      ! Loop index
INTEGER :: istat1                    ! I/o status
```

```
INTEGER :: istat2                    ! I/o status
INTEGER :: ncol1                     ! no. of cols used in A
INTEGER :: ncol2                     ! no. of cols used in B
INTEGER :: nrow1                     ! no. of rows used in A
INTEGER :: nrow2                     ! no. of rows used in B

! Get the name of the disk file containing array A.
WRITE (*,1000)
1000 FORMAT (' Enter the file name containing array A: ')
READ (*,'(A30)') filename1

! Get the name of the disk file containing array B.
WRITE (*,1010)
1010 FORMAT (' Enter the file name containing array B: ')
READ (*,'(A30)') filename2

! Open input data files.  Status is OLD because the input data
! must already exist.
OPEN ( UNIT=1,FILE=filename1,STATUS='OLD',ACTION='READ',IOSTAT=istat1 )
OPEN ( UNIT=2,FILE=filename2,STATUS='OLD',ACTION='READ',IOSTAT=istat2 )

! Were the OPENs successful?
IF ( (istat1 == 0) .AND. (istat2 == 0) ) THEN

   ! The files were opened successfully.  Read the size of array A.
   READ (1,*) nrow1, ncol1

   ! Read the size of array B.
   READ (2,*) nrow2, ncol2

   ! If any dimension exceeds MAXSIZ, tell user and quit.
   IF ( (nrow1 > MAXSIZ) .OR. (ncol1  >  MAXSIZ) .OR. &
        (nrow2 > MAXSIZ) .OR. (ncol2  >  MAXSIZ) ) THEN

      ! Error.
      WRITE (*,1020) MAXSIZ
      1020 FORMAT (' Error--An array dimension exceeds max size:',I6)

   ELSE

      ! Read matrices A and B.
      DO i = 1, nrow1
          READ (1,*) (a(i,j), j=1,ncol1)
      END DO
      DO i = 1, nrow2
         READ (2,*) (b(i,j), j=1,ncol2)
      END DO

      ! Multiply matrices
      CALL mat_mult ( a, MAXSIZ, MAXSIZ, b, MAXSIZ, MAXSIZ, &
                      c, MAXSIZ, MAXSIZ, nrow1, ncol1, nrow2, &
                      ncol2, error )

      ! Write out the result.
      IF ( error == 0 ) THEN
         WRITE (*,*) 'The resulting matrix C = A * B is:'
```

```
         DO i = 1, nrow1
            WRITE (*,1030) (c(i,j), j=1,ncol2)
            1030 FORMAT (1X,8(F9.2,1X))
         END DO

      ELSE

         WRITE (*,1040)
         1040 FORMAT (' ERROR--Arrays are of incompatible sizes.')

      END IF

   END IF

   ! Close files
   CLOSE (UNIT=1)
   CLOSE (UNIT=2)

ELSE

   ! If we get here, there was an error opening one of the files.
   ! Tell user, and quit.
   IF ( istat1 /= 0 ) THEN
      WRITE (*,1050) filename1, istat1
      1050 FORMAT (' Error opening file ',A,': IOSTAT = ',I6)
   END IF
   IF ( istat2 /= 0 ) THEN
      WRITE (*,1050) filename2, istat2
   END IF
END IF

END PROGRAM test_mat_mult
```

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

```
C:\book\f95_2003\soln\ex9_6>test_mat_mult
Enter the file name containing array A:
in9_6.aa
Enter the file name containing array B:
in9_6.ab
The resulting matrix C = A * B is:
     6.00      7.00      6.00
     5.00      3.00     -5.00
    16.00     12.00     16.00

C:\book\f95_2003\soln\ex9_6>test_mat_mult
Enter the file name containing array A:
in9_6.ba
Enter the file name containing array B:
in9_6.bb
The resulting matrix C = A * B is:
   -11.00
     6.00
    15.00
    18.00
```

9-7 A subroutine to multiply two matrices together using assumed-shape arrays is shown below. With assumed-shape arrays, the subroutine will know the size of the arrays through the explicit interface. Note that this subroutine uses the inquiry function UBOUND to determine the sizes of the arrays when that information is necessary. The subroutine is contained in a module to create an explicit interface. Note how much simpler it is than the previous example!

```
MODULE subs
CONTAINS

SUBROUTINE mat_mult ( a, b, c, error )
!
!  Purpose:
!    To multiply two-dimensional matrices A and B, and
!    store the result in a third matrix C.  This version
!    of the subroutine uses assumed-shape arrays and an
!    explicit interface.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of calling arguments:
REAL,INTENT(IN),DIMENSION(:,:) ::  a      ! Matrix A
REAL,INTENT(IN),DIMENSION(:,:) ::  b      ! Matrix B
REAL,INTENT(OUT),DIMENSION(:,:) :: c      ! Matrix C
INTEGER,INTENT(OUT) :: error      ! Error flag: 0 = No error
                                  !             1 = size mismatch
                                  !             2 = C too small

! List of local variables:
INTEGER :: i, j, k                ! Loop index
INTEGER :: nrow1                  ! No. of rows in A
INTEGER :: ncol1                  ! No. of cols in A
INTEGER :: nrow2                  ! No. of rows in B
INTEGER :: ncol2                  ! No. of cols in B
INTEGER :: nrow3                  ! No. of rows in C
INTEGER :: ncol3                  ! No. of cols in C

! Get size of matrices
nrow1 = UBOUND(a,1)
ncol1 = UBOUND(a,2)
nrow2 = UBOUND(b,1)
ncol2 = UBOUND(b,2)
nrow3 = UBOUND(c,1)
ncol3 = UBOUND(c,2)

! Is C large enough to hold the result?
IF ( nrow1 > nrow3 .OR. ncol2 > ncol3 ) THEN
   error = 2

! Are the sizes compatible?
ELSE IF ( ncol1 /= nrow2 ) THEN
   error = 1
```

```
ELSE

   ! Multiply elements
   DO i = 1, nrow1
      DO j = 1, ncol2
         c(i,j) = 0.
         DO k = 1, ncol1
            c(i,j) = C(i,j) + a(i,k) * b(k,j)
         END DO
      END DO
   END DO
   error = 0

END IF
END SUBROUTINE mat_mult

END MODULE subs
```

A test driver program for this subroutine is shown below:

```
PROGRAM test_mat_mult
!
!  Purpose:
!    To test the version of subroutine mat_mult with an
!    explicit interface.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
USE subs
IMPLICIT NONE

! List of named constants:
INTEGER, PARAMETER :: MAXSIZ = 20    ! Maximum array size

! List of variables:
REAL,DIMENSION(MAXSIZ,MAXSIZ) :: a   ! Array A
REAL,DIMENSION(MAXSIZ,MAXSIZ) :: b   ! Array B
REAL,DIMENSION(MAXSIZ,MAXSIZ) :: c   ! Array C
INTEGER :: error                     ! Error flag
CHARACTER(len=30) :: filename1       ! Name of file containing A
CHARACTER(len=30) :: filename2       ! Name of file containing B
INTEGER :: i, j                      ! Loop index
INTEGER :: istat1                    ! I/o status
INTEGER :: istat2                    ! I/o status
INTEGER :: ncol1                     ! no. of cols used in A
INTEGER :: ncol2                     ! no. of cols used in B
INTEGER :: nrow1                     ! no. of rows used in A
INTEGER :: nrow2                     ! no. of rows used in B

! Get the name of the disk file containing array A.
WRITE (*,1000)
1000 FORMAT (' Enter the file name containing array A: ')
READ (*,'(A30)') filename1
```

```
! Get the name of the disk file containing array B.
WRITE (*,1010)
1010 FORMAT (' Enter the file name containing array B: ')
READ (*,'(A30)') filename2

! Open input data files.  Status is OLD because the input data
! must already exist.
OPEN ( UNIT=1,FILE=filename1,STATUS='OLD',ACTION='READ',IOSTAT=istat1 )
OPEN ( UNIT=2,FILE=filename2,STATUS='OLD',ACTION='READ',IOSTAT=istat2 )

! Were the OPENs successful?
IF ( (istat1 == 0) .AND. (istat2 == 0) ) THEN

   ! The files were opened successfully.  Read the size of array A.
   READ (1,*) nrow1, ncol1

   ! Read the size of array B.
   READ (2,*) nrow2, ncol2

   ! If any dimension exceeds MAXSIZ, tell user and quit.
   IF ( (nrow1 > MAXSIZ) .OR. (ncol1  >  MAXSIZ) .OR. &
        (nrow2 > MAXSIZ) .OR. (ncol2  >  MAXSIZ) ) THEN

      ! Error.
      WRITE (*,1020) MAXSIZ
      1020 FORMAT (' Error--An array dimension exceeds max size:',I6)

   ELSE

      ! Read matrices A and B.
      DO i = 1, nrow1
          READ (1,*) (a(i,j), j=1,ncol1)
      END DO
      DO i = 1, nrow2
         READ (2,*) (b(i,j), j=1,ncol2)
      END DO

      ! Multiply matrices
      CALL mat_mult ( a, b, c, error )

      ! Write out the result.
      IF ( error == 0 ) THEN
         WRITE (*,*) 'The resulting matrix C = A * B is:'
         DO i = 1, nrow1
            WRITE (*,1030) (c(i,j), j=1,ncol2)
            1030 FORMAT (1X,8(F9.2,1X))
         END DO

      ELSE

         WRITE (*,1040) error
         1040 FORMAT (' ERROR--Subroutine error', I6)

      END IF
```

```
   END IF

   ! Close files
   CLOSE (UNIT=1)
   CLOSE (UNIT=2)

ELSE

   ! If we get here, there was an error opening one of the files.
   ! Tell user, and quit.
   IF ( istat1 /= 0 ) THEN
      WRITE (*,1050) filename1, istat1
      1050 FORMAT (' Error opening file ',A,': IOSTAT = ',I6)
   END IF
   IF ( istat2 /= 0 ) THEN
      WRITE (*,1050) filename2, istat2
   END IF
END IF

END PROGRAM test_mat_mult
```

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

```
C:\book\f95_2003\soln\ex9_7>test_mat_mult
Enter the file name containing array A:
in9_7.aa
Enter the file name containing array B:
in9_7.ab
The resulting matrix C = A * B is:
     6.00      7.00      6.00
     5.00      3.00     -5.00
    16.00     12.00     16.00

C:\book\f95_2003\soln\ex9_7>test_mat_mult
Enter the file name containing array A:
in9_7.ba
Enter the file name containing array B:
in9_7.bb
The resulting matrix C = A * B is:
   -11.00
     6.00
    15.00
    18.00
```

9-8 A version of subroutine `simul` that uses assumed-shape arrays is shown below:

```
MODULE simul_module
CONTAINS

SUBROUTINE simul ( a, b, n, error )
!
!  Purpose:
!    Subroutine to solve a set of n linear equations in n
!    unknowns using Gaussian elimination and the maximum
!    pivot technique.
!
```

```
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    11/23/2006    S. J. Chapman       Original code
! 1. 05/12/2007    S. J. Chapman       Use assumed-shape arrays
IMPLICIT NONE

! Data dictionary: declare calling parameter types & definitions
REAL, INTENT(INOUT), DIMENSION(:,:) :: a
                                     ! Array of coefficients (n x n).
                                     ! (This array is destroyed
                                     ! during processing.)
REAL, INTENT(INOUT), DIMENSION(:) :: b
                                     ! Input: Right-hand side of eqns.
                                     ! Output: Solution vector.
INTEGER, INTENT(IN) :: n             ! Number of equations to solve.
INTEGER, INTENT(OUT) :: error        ! Error flag:
                                     !   0 -- No error
                                     !   1 -- Singular equations


! Data dictionary: declare constants
REAL, PARAMETER :: EPSILON = 1.0E-6  ! A "small" number for comparison
                                     ! when determining singular eqns

! Data dictionary: declare local variable types & definitions
REAL :: factor                       ! Factor to multiply eqn irow by
                                     ! before adding to eqn jrow
INTEGER :: irow                      ! Number of the equation currently
                                     ! being processed
INTEGER :: ipeak                     ! Pointer to equation containing
                                     ! maximum pivot value
INTEGER :: jrow                      ! Number of the equation compared
                                     ! to the current equation
INTEGER :: kcol                      ! Index over all columns of eqn
REAL :: temp                         ! Scratch value

! Process n times to get all equations...
mainloop: DO irow = 1, n

   ! Find peak pivot for column irow in rows irow to n
   ipeak = irow
   max_pivot: DO jrow = irow+1, n
      IF (ABS(a(jrow,irow)) > ABS(a(ipeak,irow))) THEN
         ipeak = jrow
      END IF
   END DO max_pivot

   ! Check for singular equations.
   singular: IF ( ABS(a(ipeak,irow)) < EPSILON ) THEN
      error = 1
      RETURN
   END IF singular

   ! Otherwise, if ipeak /= irow, swap equations irow & ipeak
   swap_eqn: IF ( ipeak /= irow ) THEN
```

```
      DO kcol = 1, n
         temp          = a(ipeak,kcol)
         a(ipeak,kcol) = a(irow,kcol)
         a(irow,kcol)  = temp
      END DO
      temp     = b(ipeak)
      b(ipeak) = b(irow)
      b(irow)  = temp
   END IF swap_eqn

   ! Multiply equation irow by -a(jrow,irow)/a(irow,irow),
   ! and add it to Eqn jrow (for all eqns except irow itself).
   eliminate: DO jrow = 1, n
      IF ( jrow /= irow ) THEN
         factor = -a(jrow,irow)/a(irow,irow)
         DO kcol = 1, n
            a(jrow,kcol) = a(irow,kcol)*factor + a(jrow,kcol)
         END DO
         b(jrow) = b(irow)*factor + b(jrow)
      END IF
   END DO eliminate
END DO mainloop

! End of main loop over all equations.  All off-diagonal
! terms are now zero.  To get the final answer, we must
! divide each equation by the coefficient of its on-diagonal
! term.
divide: DO irow = 1, n
   b(irow)      = b(irow) / a(irow,irow)
   a(irow,irow) = 1.
END DO divide

! Set error flag to 0 and return.
error = 0
END SUBROUTINE simul

END MODULE simul_module
```

A test driver program for this subroutine is shown below:

```
PROGRAM test_simul
!
!  Purpose:
!    To test subroutine simul, which solves a set of N linear
!    equations in N unknowns.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    11/23/2006    S. J. Chapman       Original code
! 1. 05/12/2007    S. J. Chapman       Use assumed-shape arrays
USE simul_module
IMPLICIT NONE

! Data dictionary: declare constants
INTEGER, PARAMETER :: MAX_SIZE = 10    ! Max number of eqns
```

```
! Data dictionary: declare local variable types & definitions
REAL, DIMENSION(MAX_SIZE,MAX_SIZE) :: a
                                     ! Array of coefficients (n x n).
                                     ! This array is of size ndim x
                                     ! ndim, but only n x n of the
                                     ! coefficients are being used.
                                     ! The declared dimension ndim
                                     ! must be passed to the sub, or
                                     ! it won't be able to interpret
                                     ! subscripts correctly.  (This
                                     ! array is destroyed during
                                     ! processing.)
REAL, DIMENSION(MAX_SIZE) :: b       ! Input: Right-hand side of eqns.
                                     ! Output: Solution vector.
INTEGER :: error                     ! Error flag:
                                     !   0 -- No error
                                     !   1 -- Singular equations
CHARACTER(len=20) :: file_name       ! Name of file with eqns
INTEGER :: i                         ! Loop index
INTEGER :: j                         ! Loop index
INTEGER :: n                         ! Number of simul eqns (<= MAX_SIZE)
INTEGER :: istat                     ! I/O status

! Get the name of the disk file containing the equations.
WRITE (*,"(' Enter the file name containing the eqns: ')")
READ (*,'(A20)') file_name

! Open input data file.  Status is OLD because the input data must
! already exist.
OPEN ( UNIT=1, FILE=file_name, STATUS='OLD', ACTION='READ', &
       IOSTAT=istat )

! Was the OPEN successful?
fileopen: IF ( istat == 0 ) THEN
   ! The file was opened successfully, so read the number of
   ! equations in the system.
   READ (1,*) n

   ! If the number of equations is <= MAX_SIZE, read them in
   ! and process them.
   size_ok: IF ( n <= MAX_SIZE ) THEN
      DO i = 1, n
         READ (1,*) (a(i,j), j=1,n), b(i)
      END DO

      ! Display coefficients.
      WRITE (*,"(/,1X,'Coefficients before call:')")
      DO i = 1, n
         WRITE (*,"(1X,7F11.4)") (a(i,j), j=1,n), b(i)
      END DO

      ! Solve equations.
      CALL simul (a, b, n, error )

      ! Check for error.
```

```
      error_check: IF ( error /= 0 ) THEN

         WRITE (*,1010)
         1010 FORMAT (/1X,'Zero pivot encountered!', &
                     //1X,'There is no unique solution to this system.')

      ELSE error_check

         ! No errors. Display coefficients.
         WRITE (*,"(/,1X,'Coefficients after call:')")
         DO  i = 1, n
             WRITE (*,"(1X,7F11.4)") (a(i,j), j=1,n), b(i)
         END DO

         ! Write final answer.
         WRITE (*,"(/,1X,'The solutions are:')")
         DO i = 1, n
            WRITE (*,"(3X,'X(',I2,') = ',F16.6)") i, b(i)
         END DO

      END IF error_check
   END IF size_ok
ELSE fileopen

   ! Else file open failed.  Tell user.
   WRITE (*,1020) istat
   1020 FORMAT (1X,'File open failed--status = ', I6)

END IF fileopen
END PROGRAM test_simul
```

9-9 When program `test_simul` is executed with the sample data sets, the results are:

```
C:\book\f95_2003\soln\ex9_9>test_simul
Enter the file name containing the eqns:
inputs1

Coefficients before call:
     1.0000     1.0000     1.0000     1.0000
     2.0000     1.0000     1.0000     2.0000
     1.0000     3.0000     2.0000     4.0000

Coefficients after call:
     1.0000     0.0000     0.0000     1.0000
     0.0000     1.0000     0.0000     3.0000
     0.0000     0.0000     1.0000    -3.0000

The solutions are:
  X( 1) =         1.000000
  X( 2) =         3.000000
  X( 3) =        -3.000000

C:\book\f95_2003\soln\ex9_9>test_simul
Enter the file name containing the eqns:
inputs2
```

```
Coefficients before call:
     1.0000     1.0000     1.0000     1.0000
     2.0000     6.0000     4.0000     8.0000
     1.0000     3.0000     2.0000     4.0000

Zero pivot encountered!

There is no unique solution to this system.
```

9-10 The data in a module should be declared with the `SAVE` attribute to ensure that the data remains unchanged between calls to procedures that `USE` the module. If the data is not declared with the `SAVE` attribute, then there is no guarantee that the data will be the same during different calls.

9-11 If the allocatable argument has an `INTENT(IN)` attribute, this subroutine will not work, because it tries to allocate memory with variable `a`. This is illegal with the `INTENT(IN)` attribute.

9-12 If the allocatable argument has an `INTENT(OUT)` attribute, this subroutine will work, but the results will be different than before. A subroutine argument with the `INTENT(OUT)` attribute is automatically deallocated when it enters the subroutine, so the first message displayed by the subroutine will be "In sub: the array is not allocated".

9-13 A subroutine to simulate the throw of a pair of dice is shown below. It uses an internal subroutine to calculate the results of each die.

```
SUBROUTINE throw ( die1, die2 )
!
!  Purpose:
!    To simulate the throw of a pair of dice, returning two
!    integer value between 1 and 6.  This subroutine calls
!    internal subroutine "die" to calculate the value of
!    an individual die.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of calling arguments:
INTEGER,INTENT(OUT) :: die1      ! Random value in range 1-6
INTEGER,INTENT(OUT) :: die2      ! Random value in range 1-6

! Get numbers
CALL die ( die1 )
CALL die ( die2 )

! Return to calling program.
RETURN

! Internal subroutine
CONTAINS
   SUBROUTINE die ( ival )
   !
   !  Purpose:
   !    To simulate the throw of a die, returning an integer
   !    value between 1 and 6.
```

```
   !
   IMPLICIT NONE

   ! List of calling arguments:
   INTEGER,INTENT(OUT) :: ival      ! Random value in range 1-6

   ! List of local variables:
   REAL :: value                    ! Result of call to random0

   ! Get a random number
   CALL random_number ( value )

   ! Map to the proper output range.
   IF ( value < 0.1666667 ) THEN
      ival = 1
   ELSE IF ( value < 0.3333333 ) THEN
      ival = 2
   ELSE IF ( value < 0.5 ) THEN
      ival = 3
   ELSE IF ( value < 0.6666667 ) THEN
      ival = 4
   ELSE IF ( value < 0.8333333 ) THEN
      ival = 5
   ELSE
      ival = 6
   END IF

   END SUBROUTINE die
END SUBROUTINE throw
```

A test driver program for this subroutine is shown below:

```
PROGRAM test_throw
!
!  Purpose:
!    To test subroutine throw, shich should return a pair
!    of random values between 1 and 6.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
INTEGER :: die1                 ! Value on first die
INTEGER :: die2                 ! Value on second die
INTEGER :: i                    ! Loop index

! Call program throw 10 times, and see what values are returned:
DO i = 1, 10

   CALL throw ( die1, die2 )
   WRITE (*,"(' Results: ',2I2)") die1, die2
```

```
END DO

END PROGRAM test_throw
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex9_13>test_throw

Results:  3 3
Results:  6 3
Results:  5 5
Results:  6 3
Results:  1 2
Results:  5 2
Results:  4 1
Results:  5 2
Results:  5 5
Results:  6 5
```

9-14 A set of functions to perform the sine, cosine, tangent, and their inverse functions with angles in degrees are shown below:

```
MODULE trig_deg
CONTAINS

ELEMENTAL REAL FUNCTION sind ( x )
!
!  Purpose:
!    Function to calculate sin(x), where x is in degrees.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
IMPLICIT NONE

! Data dictionary: declare constants
REAL,PARAMETER :: DEG_2_RAD = 0.01745329

! List of calling arguments:
REAL,INTENT(IN) :: x         ! Angle in degrees

! Calculate answer
sind = SIN(x * DEG_2_RAD )

END FUNCTION sind


ELEMENTAL REAL FUNCTION cosd ( x )
!
!  Purpose:
!    Function to calculate cos(x), where x is in degrees.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
```

```
!    05/12/2007    S. J. Chapman      Original code
IMPLICIT NONE

! Data dictionary: declare constants
REAL,PARAMETER :: DEG_2_RAD = 0.01745329

! List of calling arguments:
REAL,INTENT(IN) :: x         ! Angle in degrees

! Calculate answer
cosd = COS(x * DEG_2_RAD )

END FUNCTION cosd


ELEMENTAL REAL FUNCTION tand ( x )
!
!  Purpose:
!    Function to calculate tan(x), where x is in degrees.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Data dictionary: declare constants
REAL,PARAMETER :: DEG_2_RAD = 0.01745329

! List of calling arguments:
REAL,INTENT(IN) :: x         ! Angle in degrees

! Calculate answer
tand = TAN(x * DEG_2_RAD )

END FUNCTION tand


ELEMENTAL REAL FUNCTION asind ( x )
!
!  Purpose:
!    Function to calculate arcsin(x), where the result
!    is in degrees.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Data dictionary: declare constants
REAL,PARAMETER :: RAD_2_DEG = 57.295779

! List of calling arguments:
```

```
REAL,INTENT(IN) :: x         ! Sine of angle

! Calculate answer
asind = ASIN(x) * RAD_2_DEG

END FUNCTION asind


ELEMENTAL REAL FUNCTION acosd ( x )
!
!  Purpose:
!    Function to calculate arccos(x), where the result
!    is in degrees.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Data dictionary: declare constants
REAL,PARAMETER :: RAD_2_DEG = 57.295779

! List of calling arguments:
REAL,INTENT(IN) :: x         ! Cosine of angle

! Calculate answer
acosd = ACOS(x) * RAD_2_DEG

END FUNCTION acosd


ELEMENTAL REAL FUNCTION atand ( x )
!
!  Purpose:
!    Function to calculate arctan(x), where the result
!    is in degrees.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Data dictionary: declare constants
REAL,PARAMETER :: RAD_2_DEG = 57.295779

! List of calling arguments:
REAL,INTENT(IN) :: x         ! Tangent of angle

! Calculate answer
atand = ATAN(x) * RAD_2_DEG

END FUNCTION atand
```

```
END MODULE trig_deg
```

A test driver function for these functions is shown below. Note that this driver program uses the elemental property of the functions.

```
PROGRAM test_trig
!
!  Purpose:
!    To test subroutine the trig functions and inverse functions,
!    with angles are in degrees.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
USE trig_deg
IMPLICIT NONE

! List of variables:
REAL,DIMENSION(2,3) :: arr1     ! Test input array
REAL,DIMENSION(2,3) :: arr2     ! Test output array
REAL,DIMENSION(2,3) :: arr3     ! Test result of inverse
INTEGER :: i, j                 ! Loop indices

! Define input array
arr1(1,:) = (/ 10.0, 20.0, 30.0 /)
arr1(2,:) = (/ 40.0, 50.0, 60.0 /)

! Calculate sine and arcsine
arr2 = sind(arr1)
arr3 = asind(arr2)

! Write results
WRITE (*,*) 'Results of SIND test:'
WRITE (*,*) 'arr1 = '
WRITE (*,'(1X,3F11.6)') ((arr1(i,j), j = 1,3), i = 1,2)
WRITE (*,*) 'arr2 = '
WRITE (*,'(1X,3F11.6)') ((arr2(i,j), j = 1,3), i = 1,2)
WRITE (*,*) 'arr3 = '
WRITE (*,'(1X,3F11.6)') ((arr3(i,j), j = 1,3), i = 1,2)

! Calculate cosine and arccosine
arr2 = cosd(arr1)
arr3 = acosd(arr2)

! Write results
WRITE (*,*) 'Results of COSD test:'
WRITE (*,*) 'arr1 = '
WRITE (*,'(1X,3F11.6)') ((arr1(i,j), j = 1,3), i = 1,2)
WRITE (*,*) 'arr2 = '
WRITE (*,'(1X,3F11.6)') ((arr2(i,j), j = 1,3), i = 1,2)
WRITE (*,*) 'arr3 = '
WRITE (*,'(1X,3F11.6)') ((arr3(i,j), j = 1,3), i = 1,2)
```

```
! Calculate tangent and arctangent
arr2 = tand(arr1)
arr3 = atand(arr2)

! Write results
WRITE (*,*) 'Results of TAND test:'
WRITE (*,*) 'arr1 = '
WRITE (*,'(1X,3F11.6)') ((arr1(i,j), j = 1,3), i = 1,2)
WRITE (*,*) 'arr2 = '
WRITE (*,'(1X,3F11.6)') ((arr2(i,j), j = 1,3), i = 1,2)
WRITE (*,*) 'arr3 = '
WRITE (*,'(1X,3F11.6)') ((arr3(i,j), j = 1,3), i = 1,2)

END PROGRAM test_trig
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex9_14>test_trig
 Results of SIND test:
 arr1 =
   10.000000  20.000000  30.000000
   40.000000  50.000000  60.000000
 arr2 =
    0.173648   0.342020   0.500000
    0.642788   0.766044   0.866025
 arr3 =
   10.000000  20.000000  30.000000
   40.000004  50.000000  60.000000
 Results of COSD test:
 arr1 =
   10.000000  20.000000  30.000000
   40.000000  50.000000  60.000000
 arr2 =
    0.984808   0.939693   0.866025
    0.766044   0.642788   0.500000
 arr3 =
   10.000008  20.000000  30.000002
   40.000000  49.999996  60.000000
 Results of TAND test:
 arr1 =
   10.000000  20.000000  30.000000
   40.000000  50.000000  60.000000
 arr2 =
    0.176327   0.363970   0.577350
    0.839100   1.191754   1.732051
 arr3 =
   10.000000  20.000000  30.000000
   40.000000  50.000000  60.000000
```

9-15 If the program in the previous problem is modified to declare `PURE` functions, then the following compilation errors occur. This happens because the functions are declared for scalars, and are being called with arrays. Since they are no longer elemental, this produces an error (the example shown below is the error produce by the Compaq Visual Fortran compiler):

```
C:\book\f95_2003\soln>df trig_deg.f90 test_trig.f90
Compaq Visual Fortran Optimizing Compiler Version 6.6 (Update B)
```

```
Copyright 2001 Compaq Computer Corp. All rights reserved.

trig_deg.f90
test_trig.f90
test_trig.f90(26) : Error: The shape matching rules of actual arguments and dumm
y arguments have been violated.   [ARR1]
arr2 = sind(arr1)
------------^
test_trig.f90(27) : Error: The shape matching rules of actual arguments and dumm
y arguments have been violated.   [ARR2]
arr3 = asind(arr2)
-------------^
test_trig.f90(39) : Error: The shape matching rules of actual arguments and dumm
y arguments have been violated.   [ARR1]
arr2 = cosd(arr1)
------------^
test_trig.f90(40) : Error: The shape matching rules of actual arguments and dumm
y arguments have been violated.   [ARR2]
arr3 = acosd(arr2)
-------------^
test_trig.f90(52) : Error: The shape matching rules of actual arguments and dumm
y arguments have been violated.   [ARR1]
arr2 = tand(arr1)
------------^
test_trig.f90(53) : Error: The shape matching rules of actual arguments and dumm
y arguments have been violated.   [ARR2]
arr3 = atand(arr2)
-------------^
```

9-16 A subroutine to perform second-order least-squares fits is shown below. Note that this subroutine uses subroutine `simul` to solve the resulting system of simultaneous equations. It uses the form of `simul` that doesn't destroy its input arguments (from Example 9-4).

```
SUBROUTINE lsqfit_2 ( x, y, nvals, c, error )
!
!  Purpose:
!    To perform a least-squares fit of an input data set
!    to the parabola
!        y(x) = c(0) + c(1) * x + c(2) * x**2,
!    and return the resulting coefficients.  The input
!    data set consists of nvals (x,y) pairs contained in
!    arrays x and y.  The output coefficients of the
!    quadratic fit c0, c1, and c2 are placed in array c.
!
!    This subroutine contains an internal subroutine simul
!    to actually solve the system of simultaneous equations.
!    The version of simul used here does not destroy its
!    calling arguments.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE
```

```
! Data dictionary: declare dummy arugments
INTEGER, INTENT(IN) :: nvals            ! Number of input data pts
REAL, DIMENSION(nvals), INTENT(IN) :: x  ! Input x values
REAL, DIMENSION(nvals), INTENT(IN) :: y  ! Input y values
REAL, DIMENSION(0:2), INTENT(OUT) :: c   ! Coefficients of fit
INTEGER, INTENT(OUT) :: error           ! Error flag:
                                        !  0 = No error.
                                        !  1 = Singular eqns
                                        !  2 = Insufficient data

! Data dictionary: declare constants
REAL, DIMENSION(3,3) :: a       ! Coefficients of eqn to solve
REAL, DIMENSION(3) :: b         ! Right side of coefficient eqns
INTEGER :: i, j                 ! Index variable
REAL, DIMENSION(0:2) :: soln    ! Solution vector
REAL,DIMENSION(0:4) :: sum_xi   ! Sum of x**i values
REAL,DIMENSION(0:2) :: sum_xiy  ! Sum of x**i*y values

! First, check to make sure that we have enough input data.
check_data: IF ( nvals < 3 ) THEN

   ! Insufficient data.  Set error = 2, and get out.
   error = 2

ELSE check_data

   sum_xi = 0.                  ! Clear sums
   sum_xiy = 0.

   ! Build the sums required to solve the equations.
   sums: DO j = 1, nvals
      DO i = 0, 4
         sum_xi(i) = sum_xi(i) + x(j)**i
      END DO
      DO i = 0, 2
         sum_xiy(i) = sum_xiy(i) + x(j)**i * y(j)
      END DO
   END DO sums

   ! Set up the coefficients of the equations.
   a(:,1) = sum_xi(0:2)
   a(:,2) = sum_xi(1:3)
   a(:,3) = sum_xi(2:4)
   b(:) = sum_xiy(0:2)

   ! Solve for the least squares fit coefficients.  They will
   ! be returned in array b if error = 0.
   CALL simul ( a, b, soln, 3, 3, error )

   ! If error == 0, return the coefficients to the user.
   return: IF ( error == 0 ) THEN
      c = soln
   ELSE
      c = 0.
   END IF return
```

```
END IF check_data

CONTAINS    ! Internal subroutine simul

   SUBROUTINE simul ( a, b, soln, ndim, n, error )
   !
   !  Purpose:
   !    Subroutine to solve a set of N linear equations in N
   !    unknowns using Gaussian elimination and the maximum
   !    pivot technique.  This version of simul has been
   !    modified to use array sections and allocatable arrays
   !    It DOES NOT DESTROY the original input values.
   !
   !  Record of revisions:
   !      Date       Programmer          Description of change
   !      ====       ==========          =====================
   !    11/23/06    S. J. Chapman        Original code
   ! 1. 11/24/06    S. J. Chapman        Add automatic arrays
   !
   IMPLICIT NONE

   ! Data dictionary: declare calling parameter types & definitions
   INTEGER, INTENT(IN) :: ndim          ! Dimension of arrays a and b
   REAL, INTENT(IN), DIMENSION(ndim,ndim) :: a
                                       ! Array of coefficients (N x N).
                                       ! This array is of size ndim x
                                       ! ndim, but only N x N of the
                                       ! coefficients are being used.
   REAL, INTENT(IN), DIMENSION(ndim) :: b
                                       ! Input: Right-hand side of eqns.
   REAL, INTENT(OUT), DIMENSION(ndim) :: soln
                                       ! Output: Solution vector.
   INTEGER, INTENT(IN) :: n            ! Number of equations to solve.
   INTEGER, INTENT(OUT) :: error       ! Error flag:
                                       !   0 -- No error
                                       !   1 -- Singular equations

   ! Data dictionary: declare constants
   REAL, PARAMETER :: EPSILON = 1.0E-6  ! A "small" number for comparison
                                       ! when determining singular eqns

   ! Data dictionary: declare local variable types & definitions
   REAL, DIMENSION(n,n) :: a1          ! Copy of "a" which will be
                                       ! destroyed during the solution
   REAL :: factor                      ! Factor to multiply eqn irow by
                                       ! before adding to eqn jrow
   INTEGER :: irow                     ! Number of the equation currently
                                       ! being processed
   INTEGER :: ipeak                    ! Pointer to equation containing
                                       ! maximum pivot value
   INTEGER :: jrow                     ! Number of the equation compared
                                       ! to the current equation
   REAL :: temp                        ! Scratch value
   REAL, DIMENSION(n) :: temp1         ! Scratch array

   ! Make copies of arrays "a" and "b" for local use
```

```
   a1 = a(1:n,1:n)
   soln = b(1:n)

   ! Process N times to get all equations...
   mainloop: DO irow = 1, n

      ! Find peak pivot for column irow in rows irow to N
      ipeak = irow
      max_pivot: DO jrow = irow+1, n
         IF (ABS(a1(jrow,irow)) > ABS(a1(ipeak,irow))) THEN
            ipeak = jrow
         END IF
      END DO max_pivot

      ! Check for singular equations.
      singular: IF ( ABS(a1(ipeak,irow)) < EPSILON ) THEN
         error = 1
         RETURN
      END IF singular

      ! Otherwise, if ipeak /= irow, swap equations irow & ipeak
      swap_eqn: IF ( ipeak /= irow ) THEN
         temp1 = a1(ipeak,1:n)
         a1(ipeak,1:n) = a1(irow,1:n)   ! Swap rows in a
         a1(irow,1:n) = temp1
         temp = soln(ipeak)
         soln(ipeak) = soln(irow)       ! Swap rows in b
         soln(irow)  = temp
      END IF swap_eqn

      ! Multiply equation irow by -a1(jrow,irow)/a1(irow,irow),
      ! and add it to Eqn jrow (for all eqns except irow itself).
      eliminate: DO jrow = 1, n
         IF ( jrow /= irow ) THEN
            factor = -a1(jrow,irow)/a1(irow,irow)
            a1(jrow,:) = a1(irow,1:n)*factor + a1(jrow,1:n)
            soln(jrow) = soln(irow)*factor + soln(jrow)
         END IF
      END DO eliminate
   END DO mainloop

   ! End of main loop over all equations.  All off-diagonal terms
   ! are now zero.  To get the final answer, we must divide
   ! each equation by the coefficient of its on-diagonal term.
   divide: DO irow = 1, n
      soln(irow) = soln(irow) / a1(irow,irow)
      a1(irow,irow) = 1.
   END DO divide

   ! Set error flag to 0 and return.
   error = 0

   END SUBROUTINE simul

END SUBROUTINE lsqfit_2
```

A test driver program for this subroutine is shown below:

```
PROGRAM test_lsqfit_2
!
!  Purpose:
!    To test subroutine lsqfit_2, which performs a least-
!    squares fit to a parabola.  The input data for this fit
!    comes from a user-specified input data file.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Data dictionary: declare constants
INTEGER, PARAMETER :: LU = 12          ! Unit for file i/o
INTEGER, PARAMETER :: MAX_VALS = 1000  ! Maximum data pts

! Data dictionary: declare variables
REAL, DIMENSION(0:2) :: c        ! Coefficients of fit
INTEGER :: error                 ! Error flag
LOGICAL :: exceed = .FALSE.      ! Logical indicating that array
                                 ! limits are exceeded.
CHARACTER(len=20) :: filename    ! Input data file name
INTEGER :: istat                 ! Status: 0 for success
INTEGER :: nvals = 0             ! Number of vaLUes read
REAL :: t1, t2                   ! Temporary vars for read
REAL, DIMENSION(MAX_VALS) :: x   ! x vaLUes of (x,y) pairs
REAL, DIMENSION(MAX_VALS) :: y   ! y vaLUes of (x,y) pairs

! Prompt user and get the name of the input file.
WRITE (*,1000)
1000 FORMAT (1X,'This program performs a least-squares fit of an ',/, &
             1X,'input data set to a parabola.  Enter the name',/, &
             1X,'of the file containing the input (x,y) pairs: ')
READ (*,'(A)') filename

! Open the input file
OPEN (UNIT=LU, FILE=filename, STATUS='OLD', ACTION='READ', &
      IOSTAT=istat )

! Was the OPEN successful?
fileopen: IF ( istat == 0 ) THEN         ! Open successful

   ! The file was opened successfully, so read the data,
   input: DO
      READ (LU,*,IOSTAT=istat) t1, t2   ! Get vaLUes
      IF ( istat /= 0 ) EXIT            ! Exit on end of data
      nvals = nvals + 1                 ! Bump count
      size: IF ( nvals <= MAX_VALS ) THEN ! Too many vaLUes?
         x(nvals) = t1                  ! No: Save vaLUes
         y(nvals) = t2                  ! No: Save vaLUes
      ELSE
         exceed = .TRUE.                   ! Yes: Array overflow
```

```
      END IF size
   END DO input

   ! Was the array size exceeded?  If so, tell user and quit.
   toobig: IF ( exceed ) THEN
      WRITE (*,1010) nvals, MAX_VALS
      1010 FORMAT (' Max array size exceeded: ', I6, ' > ', I6 )
   ELSE

      ! Limit not exceeded: fit data to parabola.
      CALL lsq_fit_2 ( x, y, nvals, c, error )

      ! Tell user about results of fit.
      fit_error: IF ( error == 0 ) THEN
          WRITE (*, 1020 ) c, nvals
          1020 FORMAT ('0','Regression coefficients for the ', &
                       'least- squares fit parabola:', &
                     /,1X,'  c(0)  = ', F12.3, &
                     /,1X,'  c(1)  = ', F12.3, &
                     /,1X,'  c(2)  = ', F12.3, &
                     /,1X,'  nvals = ', I12 )
      ELSE
         WRITE (*,"(' Error from lsq_fit_2: ', I6 )") error
      END IF fit_error
   END IF toobig

ELSE fileopen

   ! Else file open failed.  Tell user.
   WRITE (*,1030) istat
   1030 FORMAT (1X,'File open failed--status = ', I6)

END IF fileopen

END PROGRAM test_lsqfit_2
```

If the specified data set is placed in file `ball_data` and the program is executed, the results are:

```
C:\book\f95_2003\soln\ex9_16>test_lsqfit_2
 This program performs a least-squares fit of an
 input data set to a parabola.  Enter the name
 of the file containing the input (x,y) pairs:
ball_data
0Regression coefficients for the least- squares fit parabola:
   c(0)  =       53.133
   c(1)  =       -5.618
   c(2)  =       -4.189
   nvals =           18
```

The fit has estimated that the original height of the ball was 53 m, the initial velocity of the ball was –5.6 m/s, and the acceleration was –4.189 m/s^2. The acceleration estimate is clearly poor, since the acceleration due to gravity is about –9.8 m/s^2. The errors are due to the noise on the input data set, plus the relatively short time over which measurements were made. (In general, the higher order a fit is, the more sensitive the estimates are to noise in the input data.)

9-17 A program to perform a least-squares fit to a noisy parabola is shown below. Note that this program actually adds more noise than the homework problem calls for, so that we can see the point where the fit starts to fail.

```
PROGRAM fit_parabola
!
!  Purpose:
!    To test the performance of least squares fitting
!    subroutines with noisy data sets.  This program
!    uses subroutine lsq_fit_2 to estimate the coefficients
!    of the function y(x) = x**2 - 4*x + 3 when the data
!    set is corrupted by uniform random noise with the
!    following peak amplitudes: 0, 0.1, 0.5, 1.0, 1.5, 2.0,
!    2.5, 3.0.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/12/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! Data dictionary: declare constants
INTEGER,PARAMETER :: NVALS = 51 ! No. of data values to generate

! Data dictionary: declare local variables
REAL,DIMENSION(0:2) :: coef    ! Coefficients of the fit
INTEGER :: error               ! Error flag
INTEGER :: i, j                ! Loop index
REAL,DIMENSION(8) :: noise     ! Max value of random noise
REAL :: random                 ! Random value
REAL,DIMENSION(0:NVALS-1) :: x ! Independent variable
REAL,DIMENSION(0:NVALS-1) :: y ! Dependent variable

! Set noise levels
noise = (/ 0.0, 0.1, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0 /)

! Loop over different amplitudes of added noise.
DO i = 1, 8

   ! Calculate function values for all points between
   ! 0.0 and 5.0, in 0.1 steps, and corrupt the values
   ! with uniform random noise of peak amplitude noise(i).
   DO j = 0, NVALS-1
      x(j) = REAL(j) / 10.
      y(j) = x(j)**2 - 4. * x(j) + 3.
      CALL RANDOM_NUMBER ( random )
      y(j) = y(j) + ( 2.*noise(i)*random - noise(i) )
   END DO

   ! Perform fit to data.
   CALL lsqfit_2 ( x, y, NVALS, coef, error )

   ! Tell user.
   WRITE (*,1000) noise(i), (coef(j), j=0, 2)
   1000 FORMAT (' Noise = ',F3.1,':   Coefficients = ', 3(F9.6,3X))
```

```
END DO

END PROGRAM fit_parabola
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex9_17>fit_parabola
 Noise = 0.0:   Coefficients =  2.999977   -3.999973    0.999995
 Noise = 0.1:   Coefficients =  2.991041   -3.993616    0.999350
 Noise = 0.5:   Coefficients =  3.218927   -4.166187    1.031347
 Noise = 1.0:   Coefficients =  2.998647   -4.066336    1.026366
 Noise = 1.5:   Coefficients =  3.035027   -4.096734    1.023631
 Noise = 2.0:   Coefficients =  3.493326   -4.526225    1.112118
 Noise = 2.5:   Coefficients =  3.221189   -4.562073    1.107013
 Noise = 3.0:   Coefficients =  2.488918   -2.899663    0.750668
```

The fit degrades as the noise level increases.

9-18 An n^{th} order least-squares fit subroutine is shown below.

```
SUBROUTINE lsqfit_n ( x, y, nvals, order, c, error )
!
!  Purpose:
!    To perform a least-squares fit of an input data set
!    to the polynomial
!        y(x) = c(0) + c(1)*x + c(2)*x**2 + c(3)*x**3 + ...
!    and print out the resulting coefficients.  The fit
!    can be to any polynomial of first through ninth order.
!    The input data set consists of nvals (x,y) pairs contained
!    in  arrays x and y.  The output coefficients of the
!    polynomial fit are placed in array c.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of dummy arguments:
INTEGER,INTENT(IN) :: nvals            ! Number of input values
REAL,DIMENSION(nvals),INTENT(IN) :: x  ! Array of x values
REAL,DIMENSION(nvals),INTENT(IN) :: y  ! Array of y values
INTEGER,INTENT(IN) :: order            ! Order of fit
REAL,DIMENSION(0:order),INTENT(OUT) :: c
                                       ! Coefficients c0, c1, ...
INTEGER,INTENT(OUT) :: error           ! Error flag:
                                       ! 0 - No error.
                                       ! 1 - Singular equations
                                       ! 2 - Not enough input values
                                       ! 3 - Illegal order specified

! List of local variables:
REAL,DIMENSION(0:order,0:order) :: a
                                       ! Array of coefficients of c
REAL,DIMENSION(0:order) :: b           ! Right hand side
```

```
REAL,DIMENSION(0:order) :: soln        ! Solution vector
INTEGER :: i, j                        ! Index variables
REAL,DIMENSION(0:2*order) :: sum_xn    ! Sum of all x**n values
                                       ! (n = 0,1, ..., 2*order)
REAL,DIMENSION(0:order) :: sum_xny     ! Sum of all x**n * y values
                                       ! (n = 0,1, ..., order)

! First, check to make sure that we have enough input data.
IF ( nvals < order+1 ) THEN

   ! Insufficient data.  Set error = 2, and get out.
   error = 2

ELSE IF ( order < 1 ) THEN

   ! Illegal equation order.  Set error = 3, and get out.
   error = 3

ELSE

   ! Zero the sums used to build the equations.
   sum_xn = 0.
   sum_xny = 0.

   ! Build the sums required to solve the equations.
   DO i = 1, nvals
      DO j = 0, 2*order
         sum_xn(j) = sum_xn(j) + x(i)**j
      END DO

      DO j = 0, order
         sum_xny(j) = sum_xny(j) + x(i)**j * y(i)
      END DO
   END DO

   ! Set up the coefficients of the equations.
   DO i = 0, order
      DO j = 0, order
         a(i,j) = sum_xn(i+j)
      END DO
   END DO

   DO i = 0, order
      b(i) = sum_xny(i)
   END DO

   ! Solve for the least squares fit coefficients.  They will
   ! be returned in array soln if error = 0.
   CALL simul ( a, b, soln, order+1, order+1, error )

   ! If error == 0, return the coefficients to the user.
   IF ( error == 0 ) THEN
      c = soln
   ELSE
      c = 0.0
   END IF
```

```
END IF

END SUBROUTINE lsqfit_n
```

9-19 A test program for the n^{th} order least-squares fit is shown below.

```
PROGRAM fit_curve
!
!  Purpose:
!    To test the performance of least squares fitting
!    subroutines with noisy data sets.  This program
!    uses subroutine lsq_fit_n to estimate the coefficients
!    of the function y(x) = x**5 + x**4 - 3*x**3 -4*x**2 + 2*x + 3
!    when the data set is corrupted by uniform random noise with
!    the following peak amplitudes: 0, 0.1, 0.5, 1.0.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    05/12/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of calling arguments:
INTEGER,PARAMETER :: NVALS = 51 ! No. of data values to generate

! List of variables:
REAL,DIMENSION(0:5) :: coef           ! Coefficients of the fit
INTEGER :: error                      ! Error flag
INTEGER :: i, j                       ! Loop index
REAL,DIMENSION(4) :: noise            ! Max value of random noise
REAL :: random                        ! Random value
REAL,DIMENSION(0:NVALS-1) :: x        ! Independent variable
REAL,DIMENSION(0:NVALS-1) :: y        ! Dependent variable

! Set noise levels
noise = (/ 0.0, 0.1, 0.5, 1.0 /)

! Loop over different amplitudes of added noise.
DO i = 1, 4

   ! Calculate function values for all points between
   ! 0.0 and 5.0, in 0.1 steps, and corrupt the values
   ! with uniform random noise of peak amplitude noise(i).
   DO j = 0, NVALS-1
      x(j) = REAL(j) / 10.
      y(j) = x(j)**5 + x(j)**4 - 3*x(j)**3 - 4*x(j)**2 + 2*x(j) + 3
      CALL RANDOM_NUMBER ( random )
      y(j) = y(j) + ( 2.*noise(i)*random - noise(i) )
   END DO

   ! Perform fit to data.
   CALL lsqfit_n ( x, y, NVALS, 5, coef, error )

   ! Tell user.
   WRITE (*,1000) noise(i), (coef(j), j=0, 5)
```

```
   1000 FORMAT (' Noise = ',F3.1,':   Coefs = ', 7(F8.5,1X))

END DO

END PROGRAM fit_curve
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex9_19>fit_curve
 Noise = 0.0:   Coefs =  3.05429  1.72422 -3.68661 -3.13710  1.02567  0.99827
 Noise = 0.1:   Coefs =  3.14596  1.17840 -2.96086 -3.51336  1.10945  0.99157
 Noise = 0.5:   Coefs =  3.41261  0.87353 -2.75805 -3.62438  1.14108  0.98847
 Noise = 1.0:   Coefs =  3.75150 -1.17898 -0.94079 -4.16730  1.19587  0.98798
```

The fit is best at zero noise, but not perfect. It degrades from there. The quality of this fit can be improved by using a higher-precision real data type, which we will learn about in a subsequent chapter.

9-20 A program to fit an input data set to a parabola and then to use the parabola to interpolate a data point is shown below. Since the program is intended to interpolate data, it is illegal to supply a point to interpolate that is outside the range of the input data set. This program checks for this condition, and reports it.

```
PROGRAM interpolate
!
!  Purpose:
!    To fit a noisy data set to a parabola, and then use the
!    fitted parabola to interpolate data points.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/12/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of parameters:
INTEGER, PARAMETER :: MAX_VALS = 1000  ! Maximum data pts

! List of variables:
REAL, DIMENSION(0:2) :: c         ! Coefficients of fit
INTEGER :: error                  ! Error flag
LOGICAL :: exceed = .FALSE.       ! Logical indicating that array
                                  ! limits are exceeded.
CHARACTER(len=20) :: filename     ! Input data file name
INTEGER :: istat                  ! Status: 0 for success
REAL :: max_x = -1.0E38           ! Max value
REAL :: min_x = 1.0E38            ! Min value
INTEGER :: nvals = 0              ! Number of values read
REAL :: t1, t2                    ! Temporary vars for read
REAL, DIMENSION(MAX_VALS) :: x    ! x values of (x,y) pairs
REAL :: x0                        ! Point to interpolate
REAL, DIMENSION(MAX_VALS) :: y    ! y values of (x,y) pairs
REAL :: y0                        ! Interpolated value

! Prompt user and get the name of the input file.
WRITE (*,1000)
1000 FORMAT (' This program performs a least-squares fit of an ',/, &
```

```
              ' input data set to a parabola, and uses the fitted ',/,&
              ' parabola to interpolate a point.  Enter the name',/, &
              ' of the file containing the input (x,y) pairs: ')
READ (*,'(A)') filename

WRITE (*,*) 'Enter to x position at which to interpolate the data: '
READ (*,*) x0

! Open the input file
OPEN (UNIT=8, FILE=filename, STATUS='OLD', ACTION='READ', &
      IOSTAT=istat )

! Was the OPEN successful?
fileopen: IF ( istat == 0 ) THEN         ! Open successful

   ! The file was opened successfully, so read the data,
   input: DO
      READ (8,*,IOSTAT=istat) t1, t2    ! Get values
      IF ( istat /= 0 ) EXIT            ! Exit on end of data
      nvals = nvals + 1                 ! Bump count
      size: IF ( nvals <= MAX_VALS ) THEN ! Too many values?
         x(nvals) = t1                  ! No: Save values
         y(nvals) = t2                  ! No: Save values
         min_x = MIN(t1,min_x)          ! Minimum x value
         max_x = MAX(t1,max_x)          ! Maximum x value
      ELSE
         exceed = .TRUE.                   ! Yes: Array overflow
      END IF size
   END DO input

   ! Was the array size exceeded?  If so, tell user and quit.
   toobig: IF ( exceed ) THEN
      WRITE (*,1010) nvals, MAX_VALS
      1010 FORMAT (' Max array size exceeded: ', I6, ' > ', I6 )

   ELSE toobig

      ! Limit not exceeded: fit data to parabola.
      CALL lsq_fit_2 ( x, y, nvals, c, error )

      ! Tell user about results of fit.
      fit_error: IF ( error == 0 ) THEN
          WRITE (*, 1020 ) c, nvals
          1020 FORMAT ('0','Regression coefficients for the ', &
                       'least- squares fit parabola:', &
                     /,1X,'  c(0)  = ', F12.3, &
                     /,1X,'  c(1)  = ', F12.3, &
                     /,1X,'  c(2)  = ', F12.3, &
                     /,1X,'  nvals = ', I12 )

         IF ( x0 >= min_x .AND. x0 <= max_x ) THEN

            ! Interpolate value
            y0 = c(0) + c(1) * x0 + c(2) * x0**2

            WRITE (*,1030) x0, y0
```

```
            1030 FORMAT (' The interpolated value at ',F6.2,' is ',F10.4)

         ELSE
            WRITE (*,1040) x0, min_x, max_x
            1040 FORMAT (' Value ',F6.2,' is not between ',F6.2,' and 'F6.2)
         END IF

      ELSE
         WRITE (*,"(' Error from lsq_fit_2: ', I6 )") error
      END IF fit_error
   END IF toobig

ELSE fileopen

   ! Else file open failed.  Tell user.
   WRITE (*,1050) istat
   1050 FORMAT (1X,'File open failed--status = ', I6)

END IF fileopen

END PROGRAM interpolate
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex9_20>interpolate
This program performs a least-squares fit of an
input data set to a parabola, and uses the fitted
parabola to interpolate a point.  Enter the name
of the file containing the input (x,y) pairs:
in9_20.dat
Enter to x position at which to interpolate the data:
3.5

Regression coefficients for the least- squares fit parabola:
  c(0)  =      -22.761
  c(1)  =       10.828
  c(2)  =       -1.069
  nvals =           11
The interpolated value at   3.50 is     2.0431

C:\book\f95_2003\soln\ex9_20>interpolate
This program performs a least-squares fit of an
input data set to a parabola, and uses the fitted
parabola to interpolate a point.  Enter the name
of the file containing the input (x,y) pairs:
in9_20.dat
Enter to x position at which to interpolate the data:
12.

Regression coefficients for the least- squares fit parabola:
  c(0)  =      -22.761
  c(1)  =       10.828
  c(2)  =       -1.069
  nvals =           11
Value  12.00 is not between    .00 and  10.00
```

Note that the program worked correctly for both points within the data set and points outside the data set.

9-21 A program to fit an input data set to a straight line, and then to use that line to extrapolate a data set is shown below:

```
PROGRAM extrapolate
!
!  Purpose:
!    To fit a noisy data set to a line, and then use the
!    fitted line to extrapolate data points.
!
!  Record of revisions:
!       Date      Programmer         Description of change
!       ====      ==========         =====================
!    05/12/2007   S. J. Chapman      Original code
!
IMPLICIT NONE

! List of parameters:
INTEGER, PARAMETER :: MAX_VALS = 1000  ! Maximum data pts

! List of variables:
INTEGER :: error                   ! Error flag
LOGICAL :: exceed = .FALSE.        ! Logical indicating that array
                                   ! limits are exceeded.
CHARACTER(len=20) :: filename      ! Input data file name
INTEGER :: istat                   ! Status: 0 for success
INTEGER :: nvals = 0               ! Number of values read
REAL :: slope                      ! Slope of fitted line
REAL :: t1, t2                     ! Temporary vars for read
REAL, DIMENSION(MAX_VALS) :: x     ! x values of (x,y) pairs
REAL :: x0                         ! Point to interpolate
REAL, DIMENSION(MAX_VALS) :: y     ! y values of (x,y) pairs
REAL :: y0                         ! Interpolated value
REAL :: y_int                      ! Intercept of fitted line

! Prompt user and get the name of the input file.
WRITE (*,1000)
1000 FORMAT (' This program performs a least-squares fit of an ',/, &
             ' input data set to a line, and uses the fitted ',/,&
             ' line to extrapolate a point.  Enter the name',/, &
             ' of the file containing the input (x,y) pairs: ')
READ (*,'(A)') filename

WRITE (*,*) 'Enter to x position at which to extrapolate the data: '
READ (*,*) x0

! Open the input file
OPEN (UNIT=8, FILE=filename, STATUS='OLD', ACTION='READ', &
      IOSTAT=istat )

! Was the OPEN successful?
fileopen: IF ( istat == 0 ) THEN         ! Open successful

   ! The file was opened successfully, so read the data,
   input: DO
      READ (8,*,IOSTAT=istat) t1, t2    ! Get values
```

```
      IF ( istat /= 0 ) EXIT            ! Exit on end of data
      nvals = nvals + 1                 ! Bump count
      size: IF ( nvals <= MAX_VALS ) THEN ! Too many values?
         x(nvals) = t1                  ! No: Save values
         y(nvals) = t2                  ! No: Save values
      ELSE
         exceed = .TRUE.                   ! Yes: Array overflow
      END IF size
   END DO input

   ! Was the array size exceeded?  If so, tell user and quit.
   toobig: IF ( exceed ) THEN
      WRITE (*,1010) nvals, MAX_VALS
      1010 FORMAT (' Max array size exceeded: ', I6, ' > ', I6 )
   ELSE

      ! Limit not exceeded: fit data to parabola.
      CALL lsqfit ( x, y, nvals, slope, y_int, error )

      ! Tell user about results of fit.
      fit_error: IF ( error == 0 ) THEN
          WRITE (*, 1020 ) slope, y_int, nvals
          1020 FORMAT ('0','Regression coefficients for the ', &
                       'least- squares fit parabola:', &
                     /,1X,'  slope  = ', F12.3, &
                     /,1X,'  y_int  = ', F12.3, &
                     /,1X,'  nvals  = ', I12 )

         ! Extrapolate value
         y0 = slope * x0 + y_int
         WRITE (*,1030) x0, y0
         1030 FORMAT (' The interpolated value at ',F6.2,' is ',F10.4)

      ELSE
         WRITE (*,"(' Error from lsq_fit_2: ', I6 )") error
      END IF fit_error
   END IF toobig

ELSE fileopen

   ! Else file open failed.  Tell user.
   WRITE (*,1040) istat
   1040 FORMAT (' File open failed--status = ', I6)

END IF fileopen

END PROGRAM extrapolate
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex9_21>extrapolate
This program performs a least-squares fit of an
input data set to a line, and uses the fitted
line to extrapolate a point.  Enter the name
of the file containing the input (x,y) pairs:
in9_21.dat
```

```
Enter to x position at which to extrapolate the data:
14.0

Regression coefficients for the least- squares fit parabola:
  slope  =       3.139
  y_int  =     -12.745
  nvals  =          11
The interpolated value at  14.00 is    31.2053
```

Chapter 10. More About Character Variables

10-1 The results of this code fragment will vary depending upon the collating sequence of the computer on which it is executed. If the computer uses the ASCII collating sequence, then `a < b`, and the variables will contain the following character strings:

```
a = '123456=  0123456'
b = 'ABCDEFGHIGHIMNOP'
c = 'GHIJKL1234563456'
     ----|----|----|-
         5   10   15
```

If the computer uses the EBCDIC collating sequence, then `a > b`, and the variables will contain the following character strings:

```
a = '123456=  0123456'
b = 'ABCDEFGHI123MNOP'
c = '123456GHIJKL3456'
     ----|----|----|-
         5   10   15
```

10-2 The results of this code fragment are the same regardless of collating sequence, since the results of function `LGT` are the same regardless of collating sequence. The variables will contain the following character strings:

```
a = '123456=  0123456'
b = 'ABCDEFGHIGHIMNOP'
c = 'GHIJKL1234563456'
     ----|----|----|-
         5   10   15
```

10-3 The strings in Example 10-1 are

```
Fortran
fortran
ABCD
ABC
XYZZY
9.0
A9IDL
```

Since the EBCDIC collating sequence is lower-case letters first, upper-case letters second, and numbers third, the strings will be sorted into the following order on a computer with an EBCDIC collating sequence:

```
fortran
ABC
ABCD
A9IDL
Fortran
```

```
XYZZY
9.0
```

10-4 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
   !       ====       ==========         =====================
   !    05/15/2007    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 myprocs
```

A simple test driver program is shown below.

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

```
!    05/15/2007    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 test_ucase
```

When this program is executed, the results are:

```
C:\book\f95_2003\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*
```

7-5 A routine to shift upper-case characters into lower case is shown below.

```
SUBROUTINE lcase ( string )
!
!  Purpose:
!    To shift a character string to lower case on any processor,
!    regardless of collating sequence.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/15/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare calling parameters:
CHARACTER(len=*), INTENT(INOUT) :: string

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

! Get length of string
length = LEN ( string )

! Now shift upper case letters to lower case.
DO i = 1, length
   IF ( LGE(string(i:i),'A') .AND. LLE(string(i:i),'Z') ) THEN
      string(i:i) = ACHAR ( IACHAR ( string(i:i) ) + 32 )
   END IF
END DO

END SUBROUTINE lcase
```

A simple test driver program is shown below.

```
PROGRAM test_lcase
!
!  Purpose:
```

```
!    To test subroutine lcase.
!
IMPLICIT NONE
CHARACTER(len=20) string
WRITE (*,*) 'Enter test string (up to 20 characters): '
READ (*,'(A20)') string
CALL lcase(string)
WRITE (*,*) 'The shifted string is: ', string

END PROGRAM test_lcase
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex10_5>test_lcase
 Enter test string (up to 20 characters):
This is a Test! 12#$6*
 The shifted string is: this is a test! 12#$
```

10-6 The order in which the characters strings will be sorted according to the **ASCII** collating sequence is

```
/DATA/
1DAY
2nite
?well?
AbCd
This is a test!
aBcD
quit
```

The order in which the characters strings will be sorted according to the **EBCDIC** collating sequence is

```
/DATA/
?well?
aBcD
quit
AbCd
This is a test!
1DAY
2nite
```

10-7 The contents of the variables will be as shown below. Note that the output of `b` has partially overwritten the output of `j` in this case.

```
i = 1700
j = 2400
a = 0.0
b = 0.70834
buffer = '                1700          0.7083400              0.0000'
          ----|----|----|----|----|----|----|----|----|----|----|----|
                   10        20        30        40        50        60
```

10-8 Subroutine "`caps`" locates the beginning of each word by identifying an alphanumeric character preceded by a non-alphanumeric character. It shifts each beginning character to upper case using subroutine `ucase`. All characters which are not at the beginning of a word are shifted to lower case using subroutine `lcase`. Note that it is ok to pass non-alphanumeric characters to `lcase` because the subroutine is smart enough not to change them. Characters are identified as either alphanumeric or non-alphanumeric by the logical function `alphanumeric`, which returns `.TRUE.` if a character is alphanumeric. The code for subroutine `caps` and function `alphanumeric` is shown below.

```
SUBROUTINE caps ( string )
!
!  Purpose:
!    To capitalize all of the words within a string, and force
!    all of the other letters within the word to be lower case.
!    This subroutine defines a word as any string of letters
!    and number terminated by an non-alphabetic and non-numeric
!    character.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/15/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of dummy arguments:
CHARACTER(len=*), INTENT(INOUT) :: string  ! String

! List of external functions:
LOGICAL, EXTERNAL :: alphanumeric

! List of local variables:
INTEGER :: i               ! Index variable
INTEGER :: length          ! Length of string

! Get the length of the input string.
length = LEN ( string )

! While loop looking for words.  First, check to see if
! the first character is alphanumeric, and shift it to
! upper case if it is.
!
IF ( alphanumeric(string(1:1)) ) THEN
   CALL ucase ( string(1:1) )
END IF

! Now let's check the remaining characters.  If the character
! before the current character is not alphanumeric, and the
! current character is alphanumeric, then shift the character
! to upper case.  Otherwise, shift the character to lower case.
! Note that we don't have to worry about improperly shifting
! non-alphanumeric characters, because ucase and lcase are
! smart enough ignore them.
!
DO i = 2, length
   IF ( alphanumeric(string(i:i)) .AND. &
        (.NOT. alphanumeric(string(i-1:i-1)) ) ) THEN
      CALL ucase ( string(i:i) )
   ELSE
      CALL lcase ( string(i:i) )
   END IF
END DO
```

```
END SUBROUTINE caps


FUNCTION alphanumeric ( char )
!
!  Purpose:
!    To determine whether or not a specific character is
!    alphanumeric.  If it is alphanumeric, then the function
!    will return .TRUE.; otherwise, it will return .FALSE..
!    Note that this function is designed to work correctly
!    on either ASCII or EBCDIC machines.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/15/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of dummy arguments:
CHARACTER(len=1),INTENT(IN) :: char   ! Input character
LOGICAL :: alphanumeric               ! Function result

IF ( ( LGE(char,'a') .AND. LLE(char,'z') ) .OR. &
     ( LGE(char,'A') .AND. LLE(char,'Z') ) .OR. &
     ( LGE(char,'0') .AND. LLE(char,'9') ) ) THEN
   alphanumeric = .TRUE.
ELSE
   alphanumeric = .FALSE.
END IF

END FUNCTION alphanumeric
```

A test driver program for subroutine caps is shown below.

```
PROGRAM test_caps
!
!  Purpose:
!    To test subroutine caps.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/15/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of local variables:
CHARACTER(len=40) :: a          ! Test character variable
CHARACTER(len=40) :: b          ! Test character variable
CHARACTER(len=40) :: c          ! Test character variable

! Initialize strings
a = 'this is a test--does it work?'
b = 'this iS the 2nd test!'
```

```
c = '123 WHAT NOW?!?  xxxoooxxx.'

! Write out strings.
WRITE (*,'(/,1X,A)') 'Before:'
WRITE (*,*) 'a = ', a
WRITE (*,*) 'b = ', b
WRITE (*,*) 'c = ', c

! Capitalize strings.
CALL CAPS ( a )
CALL CAPS ( b )
CALL CAPS ( c )

! Write out strings.
WRITE (*,'(/,1X,A)') 'After:'
WRITE (*,*) 'a = ', a
WRITE (*,*) 'b = ', b
WRITE (*,*) 'c = ', c

END PROGRAM test_caps
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex10_8>test_caps

Before:
a = this is a test--does it work?
b = this iS the 2nd test!
c = 123 WHAT NOW?!?  xxxoooxxx.

After:
a = This Is A Test--Does It Work?
b = This Is The 2nd Test!
c = 123 What Now?!?  Xxxoooxxx.
```

10-9 Function "caps" is shown below. Note that it is included in a module to create an explicit interface.

```
MODULE myprocs
CONTAINS
   FUNCTION caps ( string )
   !
   !  Purpose:
   !    To capitalize all of the words within a string, and force
   !    all of the other letters within the word to be lower case.
   !    This subroutine defines a word as any string of letters
   !    and number terminated by an non-alphabetic and non-numeric
   !    character.
   !
   !  Record of revisions:
   !       Date       Programmer         Description of change
   !       ====       ==========         =====================
   !    05/15/2007    S. J. Chapman      Original code
   !
   IMPLICIT NONE

   ! List of dummy arguments:
```

```
CHARACTER(len=*), INTENT(IN) :: string  ! String
CHARACTER(len=LEN(string)) :: caps      ! Capitalized version

! List of local variables:
INTEGER :: i               ! Index variable
INTEGER :: length          ! Length of string
CHARACTER(len=LEN(string)) :: str      ! Copy of input string

! Make local copy so as not to modify input string.
str = string

! Get the length of the string.
length = LEN ( str )

! While loop looking for words.  First, check to see if
! the first character is alphanumeric, and shift it to
! upper case if it is.
!
IF ( alphanumeric(str(1:1)) ) THEN
   CALL ucase ( str(1:1) )
END IF

! Now let's check the remaining characters.  If the character
! before the current character is not alphanumeric, and the
! current character is alphanumeric, then shift the character
! to upper case.  Otherwise, shift the character to lower case.
! Note that we don't have to worry about improperly shifting
! non-alphanumeric characters, because ucase and lcase are
! smart enough ignore them.
DO i = 2, length
   IF ( alphanumeric(str(i:i)) .AND. &
        (.NOT. alphanumeric(str(i-1:i-1)) ) ) THEN
      CALL ucase ( str(i:i) )
   ELSE
      CALL lcase ( str(i:i) )
   END IF
END DO

! Set result.
caps = str

END FUNCTION caps


FUNCTION alphanumeric ( char )
!
!  Purpose:
!    To determine whether or not a specific character is
!    alphanumeric.  If it is alphanumeric, then the function
!    will return .TRUE.; otherwise, it will return .FALSE..
!    Note that this function is designed to work correctly
!    on either ASCII or EBCDIC machines.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
```

```
   !    05/15/2007    S. J. Chapman      Original code
   !
   IMPLICIT NONE

   ! List of dummy arguments:
   CHARACTER(len=1),INTENT(IN) :: char   ! Input character
   LOGICAL :: alphanumeric               ! Function result

   IF ( ( LGE(char,'a') .AND. LLE(char,'z') ) .OR. &
        ( LGE(char,'A') .AND. LLE(char,'Z') ) .OR. &
        ( LGE(char,'0') .AND. LLE(char,'9') ) ) THEN
      alphanumeric = .TRUE.
   ELSE
      alphanumeric = .FALSE.
   END IF

   END FUNCTION alphanumeric

END MODULE myprocs
```

10-10 A function to calculate the number of characters actually used within a character string is shown below:

```
FUNCTION len_used ( string )
!
!  Purpose:
!    To determine the number of characters used within
!    a character string.  This number is the number of
!    characters between the first and last nonblank
!    characters in the variable.  It returns 0 for
!    completely blank strings.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/15/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of dummy arguments:
CHARACTER(len=*),INTENT(IN) :: string  ! Input string
INTEGER :: len_used             ! Length used in string

! List of local variables:
INTEGER :: ibeg                 ! Position of first non-blank char
INTEGER :: iend                 ! Position of last non-blank char
INTEGER :: length               ! Length of character 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
! a 0 into len_used.  Otherwise, find the last nonblank
! character, and calculate len_used.
IF ( ibeg > length ) THEN
   len_used = 0
ELSE
   ! Find last nonblank character.
   iend = length + 1
   DO
      iend = iend - 1
      IF ( string(iend:iend) /= ' ' ) EXIT
   END DO

   ! Calculate len_used.
   len_used = iend - ibeg + 1
END IF

END FUNCTION len_used
```

A test driver program for function `len_used` is shown below.

```
PROGRAM test_len_used
!
!  Purpose:
!    To test function len_used.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/15/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! External function:
INTEGER, EXTERNAL :: len_used     ! Length used

! List of local variables:
CHARACTER(len=30),DIMENSION(3) :: a  ! Test strings
INTEGER :: i                         ! Index variable

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

! Write lengths of strings.
DO i = 1, 3
   WRITE (*,'(/1X,A,I1,2A)') 'a(',i,') = ', a(i)
   WRITE (*,'(1X,A,I3)')     'LEN()      = ', LEN(a(i))
   WRITE (*,'(1X,A,I3)')     'LEN_TRIM() = ', LEN_TRIM(a(i))
   WRITE (*,'(1X,A,I3)')     'len_used() = ', len_used(a(i))
END DO
```

```
END PROGRAM test_len_used
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln\ex10_10>test_len_used

a(1) = How many characters are used?
LEN()      =  30
LEN_TRIM() =  29
len_used() =  29

a(2) =    ...and how about this one?
LEN()      =  30
LEN_TRIM() =  29
len_used() =  26

a(3) =    !   !
LEN()      =  30
LEN_TRIM() =   8
len_used() =   5
```

10-11 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
!       ====       ==========         =====================
!    05/15/2007    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
!       ====       ==========         =====================
!    05/15/2007    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:\book\f95_2003\soln\ex10_11>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
```

10-12 A subroutine to parse lines from an input parameter file is shown below.

```
SUBROUTINE parse ( string, start, stop, dt, plot )
!
!  Purpose:
!    To check values from an input parameter file and update the
!    corresponding data values.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/15/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of dummy arguments:
CHARACTER(len=*), INTENT(IN) :: string  ! Input string
REAL,INTENT(OUT) :: start               ! Starting time
REAL,INTENT(OUT) :: stop                ! Ending time
REAL,INTENT(OUT) :: dt                  ! Delta time
LOGICAL,INTENT(OUT) :: plot             ! Plot on/off flag

! Lost of local variables:
INTEGER :: length                  ! Length of string
INTEGER :: i_eq                    ! Position of equal sign

! Get the length of the input string.
length = LEN ( string )

! Shift string to upper case.
CALL ucase ( string )

! Check to see if this line contains any known keyword,
! and process the value associated with the keyword.
IF ( INDEX(string,'START') /= 0 ) THEN

   ! This is a start card.  Skip '=' sign and get value.
   ! If there is no = sign, tell user of invalid format
   ! and skip further processing.  Otherwise, get the value.
   i_eq = INDEX(string,'=')
```

```
   IF ( i_eq == 0 ) THEN
      WRITE (*,1000) string
      1000 FORMAT (' Invalid card format: ',A)
   ELSE
      READ (string(i_eq+1:length),'(F30.0)') start
   END IF

ELSE IF ( INDEX(string,'STOP') /= 0 ) THEN

   ! This is a stop card.  Skip '=' sign and get value.
   ! If there is no = sign, tell user of invalid format
   ! and skip further processing.  Otherwise, get the value.
   i_eq = INDEX(string,'=')
   IF ( i_eq == 0 ) THEN
      WRITE (*,1000) string
   ELSE
      READ (string(i_eq+1:length),'(F30.0)') stop
   END IF

ELSE IF ( INDEX(string,'DT') /= 0 ) THEN

   ! This is a dt card.  Skip '=' sign and get value.
   ! If there is no = sign, tell user of invalid format
   ! and skip further processing.  Otherwise, get the value.
   i_eq = INDEX(string,'=')
   IF ( i_eq == 0 ) THEN
      WRITE (*,1000) string
   ELSE
      READ (string(i_eq+1:length),'(F30.0)') dt
   END IF

ELSE IF ( INDEX(string,'PLOT') /= 0 ) THEN

   ! This is a plot card.  Legal values for plot are
   ! ON or OFF only.
   IF ( INDEX(string,'ON') .NE. 0 ) THEN
      plot = .TRUE.
   ELSE IF ( INDEX(string,'OFF') .NE. 0 ) THEN
      plot = .FALSE.
   ELSE
      WRITE (*,1010) string
      1010 FORMAT (' Invalid plot card: ',A,/, &
                   ' Legal values are ON or OFF.')
   END IF

ELSE

   ! This is an unrecognized card.  Tell user.
   WRITE (*,1020) string
   1020 FORMAT (' Unknown card: ',A)

END IF

END SUBROUTINE parse
```

A test driver program for this subroutine is shown below.

```fortran
PROGRAM test_parse
!
!  Purpose:
!    To test reading values from an input parameter file.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/15/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
REAL :: dt = 0.1                 ! Delta time
CHARACTER(len=30) :: filename    ! Input file name
INTEGER :: ierror                ! Error flag
LOGICAL :: plot = .FALSE.        ! Plot on/off flag
REAL :: start = 0.0              ! Starting time
REAL :: stop = 1.0               ! Ending time
CHARACTER(len=80) :: string      ! Input string

! Prompt user and get the name of the input file.
WRITE (*,1000)
1000 FORMAT (' Enter name of input parameter file: ' )
READ (*,'(A)') filename

! Open the input file
OPEN (UNIT=10,FILE=filename,STATUS='OLD',ACTION='READ',IOSTAT=ierror)

! Check to see of the OPEN failed.
IF ( ierror > 0 ) THEN
   WRITE (*,1020) filename, ierror
   1020 FORMAT (' ERROR: Open error on file ',A,': IOSTAT = ',I6)
ELSE

   DO
      ! Read the data value from the input file.
      READ (10,'(A)',IOSTAT=ierror) string

      ! Check for end of data
      IF ( ierror /= 0 ) EXIT

      ! Process string
      CALL parse ( string, start, stop, dt, plot )

   END DO

   ! Write out resulting parameters.
   WRITE (*,1030) start
   1030 FORMAT (/' start = ', F10.4)
   WRITE (*,1040) stop
   1040 FORMAT ( ' stop  = ', F10.4)
   WRITE (*,1050) dt
   1050 FORMAT ( ' dt    = ', F10.4)
   WRITE (*,1060) plot
```

```
   1060 FORMAT ( ' plot  = ', L10)

   ! Close input file, and quit.
   CLOSE (UNIT=10)

END IF

END PROGRAM test_parse
```

This subroutine will be tested with the following two parameter files. The first file has all valid values, and the second file has some invalid values.

in10_12.1:

```
DT = .2
Stop = 17.1
Plot ON
START = -.2
```

in10_12.2:

```
dt = .2
PLOT TRUE
Start = -17
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln\ex10_12>test_parse
Enter name of input parameter file:
in10_12.1

start =     -.2000
stop  =    17.1000
dt    =      .2000
plot  =          T

C:\book\f95_2003\soln\ex10_12>test_parse
Enter name of input parameter file:
in10_12.2
Invalid plot card: PLOT TRUE

Legal values are ON or OFF.

start =   -17.0000
stop  =     1.0000
dt    =      .2000
plot  =          F
```

In the second example, STOP and PLOT were left with their default values, since no valid input cards were found for them.

10-13 To calculate and plot the histogram of a data set, a subroutine must first accumulate the statistics about how many data values fall within each bin. Then, it must plot the number of values appearing in each of the bins on a common scale. The steps required to create and plot the histogram are:

1. Use the maximum bin value, minimum bin value, and number of bins provided by the user to calculate the range of data values to be summed into each bin.

2. Search through the input data set and determine how many data samples fall within each bin. Also, include one bin for all samples below user-specified minimum value, and one bin for all samples above the user-specified maximum value. These bins will keep track of data falling outside the range specified by the user.
3. Find the peak number of samples within any bin so that the plot can be scaled properly.
4. Plot the data using a technique similar to the line-printer plot developed in this chapter.

A subroutine to collect statistical data and plot a histogram is shown below.

```
SUBROUTINE hist ( values, nvals, nbins, minbin, maxbin, &
                  unit, error )
!
!  Purpose:
!    Subroutine to plot a histogram of an input data set contained
!    in array "values".  This program divides the data up into a
!    user-specified number of bins (up to maxstat), and counts up
!    the number of occurrences falling in each bin.  It then plots
!    a histogram of the data.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/15/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of dummy arguments:
INTEGER,INTENT(IN) :: nvals                ! No. of input values
REAL,DIMENSION(nvals),INTENT(IN) :: values  ! Input data array
INTEGER,INTENT(IN) :: nbins                ! No. of bins in histogram
INTEGER,INTENT(IN) :: minbin               ! Value of smallest bin
INTEGER,INTENT(IN) :: maxbin               ! Value of largest bin
INTEGER,INTENT(IN) :: unit                 ! Plot i/o unit
INTEGER,INTENT(OUT) :: error               ! Error flag:
                                           !  0 - Successful completion
                                           !  1 - Too few bins (<=1)
                                           !  2 - minbin & maxbin must differ

!  List of local variables:
CHARACTER(len=60) :: ast            ! Array of asterisks
REAL :: bin                         ! Label of current bin
REAL :: delta_bin                   ! Width between bin centers
INTEGER :: i                        ! Loop index
INTEGER :: index                    ! Index of bin for current sample
INTEGER :: ipeak                    ! Index of bin with largest count
INTEGER :: level                    ! No. of asterisks for current bin
CHARACTER(len=60) :: scl            ! Scale on border of histogram
CHARACTER(len=79) :: line           ! Line to print out
INTEGER,DIMENSION(-1:nbins+1) :: stat ! Array of bins

! Data used to draw borders and plot columns of histogram.
ast = '******************************&
      &******************************'
scl = '+-----------------------------+&
      &-----------------------------+'
```

```
! Check for errors.
!
errchk: IF ( nbins <= 1 ) THEN
   ! Too few bins requested.
   error = 1

ELSE IF ( ABS(maxbin-minbin) < 1.0E-12 ) THEN
   ! Error: maxbin must not equal minbin.
   error = 2

ELSE
   ! OK.  Process data.
   error = 0

   ! Clear statistics array where the number of samples that
   ! fall into each bin will be accumulated.
   stat = 0.

   ! Set the width of each bin based on the number of bins and
   ! the range they are to cover.
   delta_bin = ( maxbin - minbin ) / REAL(nbins)

   ! Accumulate statistics.  Determine the bin into which each
   ! sample falls.  Include bins for samples ABOVE and BELOW the
   ! specified range, so that we don't lose track of them.
   stats: DO i = 1, nvals

      ! Get bin number of this sample.
      index = NINT ( ( values(i) - minbin ) / delta_bin )

      ! Limit samples outside of desired range so that they fall
      ! into the single bin above or below the specified range.
      index = MAX ( index,      -1 )
      index = MIN ( index, nbins+1 )

      ! Add sample to bin.
      stat(index) = stat(index) + 1

   END DO stats

   ! Now we have accumulated the statistics.  Find the
   ! number of counts in the peak bin so that we can
   ! scale the plots.
   ipeak = 0
   findpeak: DO i = -1, nbins+1
      IF ( stat(i) > ipeak ) THEN
         ipeak = stat(i)
      END IF
   END DO findpeak

   ! Print the amplitude scale.
   WRITE (unit,'(18X,26X,I6,24X,I6)') ipeak/2, ipeak
   WRITE (unit,'(19X,A)') scl

   ! Plot the histogram.
   plot: DO i = -1, nbins+1
```

```
      ! Clear line.
      line = ' '

      ! Set level for this line of the histogram.
      level = REAL(stat(i)) / REAL(ipeak) * 60.
      IF ( level > 0 ) THEN
         line(20:79) = ast(1:level)
      ELSE
         line(20:79) = ' '
      END IF

      ! Set label for this bin.
      bin   = REAL(i) * delta_bin + minbin
      WRITE (line(5:17),'(ES13.6)') bin

      ! Set signs, as appropriate.
      IF ( i == -1 ) THEN
         line(3:4) = '<='
      ELSE IF ( i == nbins+1 ) THEN
         line(3:4) = '>='
      END IF

      ! Output complete line.
      WRITE (unit,'(A)') line
   END DO plot

   ! The histogram is complete.  Print amplitude scale again.
   WRITE (unit,'(19X,A)') scl
   WRITE (unit,'(18X,26X,I6,24X,I6)') ipeak/2, ipeak

   ! Write total samples in histogram.
   WRITE (unit,1030) nvals
   1030 FORMAT (//,18X,'Number of samples = ',I6)

END IF errchk

END SUBROUTINE parse
```

This subroutine is tested in the next exercise.

10-14 A program to generate an array of 20,000 random numbers in the range [0,1), and calculate and plot the histogram of the numbers is shown below.

```
PROGRAM test_hist
!
!  Purpose:
!    To test subroutine hist by calculating a histogram of 20,000
!    values produced by subroutine random0.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/15/2007    S. J. Chapman       Original code
!
IMPLICIT NONE
```

```
! List of named constants:
INTEGER,PARAMETER :: UNIT = 6           ! Plot i/o unit
INTEGER,PARAMETER :: MAXBIN = 0.0       ! Maximum bin for histogram
INTEGER,PARAMETER :: MINBIN = 1.0       ! Minimum bin for histogram
INTEGER,PARAMETER :: NBINS = 20         ! Number of bins in histogram
INTEGER,PARAMETER :: NVALS = 20000      ! Number of values in histogram

! List of variables:
INTEGER :: error                        ! Error flag
INTEGER :: i                            ! Index variable
REAL,DIMENSION(nvals) :: values         ! Random values

! Calculate random values.
DO i = 1, nvals
   CALL random0 ( values(i) )
END DO

! Plot histogram.
CALL hist ( values, NVALS, NBINS, MINBIN, MAXBIN, UNIT, error )

END PROGRAM test_hist
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex10_14>test_hist
                                              554                         1109
                  +----------------------------+-----------------------------+
 <= 1.050000E+00
    1.000000E+00  ***************************
    9.500000E-01  *****************************************************
    9.000000E-01  ****************************************************
    8.500000E-01  ************************************************************
    8.000000E-01  *****************************************************
    7.500000E-01  ****************************************************
    7.000000E-01  *****************************************************
    6.500000E-01  ****************************************************
    6.000000E-01  *****************************************************
    5.500000E-01  *******************************************************
    5.000000E-01  ********************************************************
    4.500000E-01  ******************************************************
    4.000000E-01  *****************************************************
    3.500000E-01  ****************************************************
    3.000000E-01  ***************************************************
    2.500000E-01  *****************************************************
    2.000000E-01  ***********************************************************
    1.500000E-01  *******************************************************
    9.999999E-02  ****************************************************
    4.999999E-02  ****************************************************
   -1.490116E-08  **************************
 >=-5.000002E-02
                  +----------------------------+-----------------------------+
                                              554                         1109

                 Number of samples =  20000
```

The distribution of numbers produced by the uniform random number generator `random0` was nearly uniform over the 20 bins. However, the two edges require some explanation. Let's take a look at the last bin. Note that the labels on each bin are values at the *center* of the bin. For example, the last bin, which is labeled 1.00, is really 1.00±0.025. Since the random numbers fall within the range [0.0,1.0), only half of the last bin falls within the range of the random number generator. Therefore, it has half as many points. The same thing applies to the first bin.

10-15 A program that will open a user-specified disk file containing the source code for a Fortran program, and copy the source code from the input file to a user-specified output file, stripping out any comment lines during the copying process is shown below.

```
PROGRAM fcopy
!
!  Purpose:
!    To read Fortran source code from an input file and copy it to
!    an output file, stripping out any comment lines in program.
!
!  Record of revisions:
!      Date       Programmer         Description of change
!      ====       ==========         =====================
!    05/15/2007   S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
LOGICAL :: char_context = .FALSE.    ! Character context flag
CHARACTER(len=36) :: filename1       ! Input file name
CHARACTER(len=36) :: filename2       ! Output file name
INTEGER :: i                         ! Loop index
INTEGER :: istat                     ! I/o status
INTEGER :: istat1                    ! File 1 open status
INTEGER :: istat2                    ! File 2 open status
CHARACTER(len=132) :: line           ! Data line

! Get the name of the file containing the input data.
WRITE (*,*) 'FCOPY -- Source file copy program.'
WRITE (*,*) 'Enter the input file name: '
READ (*,'(A36)') filename1

!Get the name of the file to write the output data to.
WRITE (*,*) 'Enter the output file name: '
READ (*,'(A36)') filename2

! Open input data file.  Status is OLD because the input data
! must already exist.
OPEN (UNIT=8,FILE=filename1,STATUS='OLD',ACTION='READ',IOSTAT=istat1)

! Open output data file.  Status is NEW so that we don't overwrite
! existing data.
OPEN (UNIT=9,FILE=filename2,STATUS='NEW',ACTION='WRITE',IOSTAT=istat2)

! Was the OPEN successful?
openok: IF ( ( istat1 == 0 ) .AND. ( istat2 == 0 ) ) THEN

   ! The files were opened successfully, so read the data from
   ! the input file and put it into the output file if it is
   ! not a comment line.
```

```
   loop: DO
      ! Get new line
      READ (8,'(A)',IOSTAT=istat) line

      ! Exit on error
      IF ( istat /= 0 ) EXIT

      ! Cycle if first character is a "!", since the entire line
      ! will be a comment.
      IF ( line(1:1) == '!' ) CYCLE

      ! Otherwise, search for a comment embedded in the line. A
      ! comment is any exclamation point not in a character
      ! context.  Search from the beginning to the end of the line.
      line_check: DO i = 1, LEN_TRIM(line)

         IF ( line(i:i) == "'" .OR. line(i:i) == '"' ) THEN

            ! Start or end of comment: toggle context switch
            char_context = .NOT. char_context

         ELSE IF ( line(i:i) == "!" ) THEN

            ! If this is in a character context, ignore it.
            ! Otherwise, skip the rest of this line.
            IF ( .NOT. char_context ) THEN
               WRITE (9,'(A)') line(1:i-1)
               EXIT line_check
            END IF

         ELSE IF ( i == LEN_TRIM(line) ) THEN

            WRITE (9,'(A)') line(1:i)

         END IF

      END DO line_check

   END DO loop

ELSE openok

   ! Handle file open errors.
   IF ( istat1 /= 0 ) THEN
      WRITE (*,1000) istat1
      1000 FORMAT (' Open error on input file: istat = ', I6)
   END IF

   IF ( istat2 /= 0 ) THEN
      WRITE (*,1010) istat2
      1010 FORMAT (' Open error on output file: istat = ', I6)
   END IF

END IF openok

END PROGRAM fcopy
```

A program written to test the stripping function is shown below:

```
PROGRAM test_stripping
!
! File to test stripping.
!
IMPLICIT NONE  ! Comment at end of line.

CHARACTER(len=5) :: string = '12!12' ! Exclamation point on char context

string = '12!&
          &!5'    ! Exclamation points in char context across lines.

WRITE (*,*) string

END PROGRAM test_stripping
```

A program written to test the stripping function is shown below:

```
C:\book\f95_2003\soln\ex10_15>fcopy
FCOPY -- Source file copy program.
Enter the input file name:
test_stripping.f90
Enter the output file name:
temp.f90
```

After the program is executed, the contents of file TEMP.F90 are as shown below. The comments have indeed been stripped from the source code. Notice that the program correctly ignored all exclamation points in a character context, even those in strings extending across multiple lines.

```
PROGRAM test_stripping
IMPLICIT NONE
CHARACTER(len=5) :: string = '12!12'
string = '12!&
          &!5'
WRITE (*,*) string
END PROGRAM test_stripping
```

This program only works correctly if the source code is correct. It can be confused if the source code contains errors such as extra out-of-place quotation marks.

Chapter 11. Additional Intrinsic Data Types

11-1 "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.

11-2 The answer to these questions is processor dependent. The instructor must supply this information for the particular processor used by the students.

11-3 Calculations with double precision real numbers have more significant digits than calculations with single precision real numbers, and the range of double precision numbers is greater than the range of single precision numbers. Therefore, double precision numbers should be used in calculations requiring either great precision or large ranges. The disadvantages of double precision real numbers compared to single precision real numbers are that calculations involving double precision numbers are slower than calculations involving single precision numbers, and that double precision numbers take up more memory than single precision numbers.

The first disadvantage listed above does not apply on computers using Intel or AMD processors, since they do a floating point calculations at greater than double precision at all times.

11-4 An ill-conditioned system of equations is one whose solution is very sensitive to small changes in coefficients. It is hard to find the solution to an ill-conditioned set of equations because small round-off errors in floating-point calculations accumulate to cause serious errors in the final answer. Double precision arithmetic helps in the solution of ill-conditioned systems by reducing the amount of round-off error during the solution process.

11-5 *(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

```
  1.111111111111111E+015    2.222222E+15
```

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

11-6 The subroutine shown below evaluates the derivative of a double precision function *f(x)* at position $x = x_0$, where the function *f(x)* is passed to the subroutine as a command line argument.

```
SUBROUTINE dderiv ( F, x0, dx, df_dx, error )
!
!  Purpose:
!    To take the derivative of function f(x) at point x0
!    using 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
!      ====       ==========          =====================
!    05/16/2007    S. J. Chapman      Original code
!
IMPLICIT NONE
```

```
! Declare named constants:
INTEGER, PARAMETER :: DBL = SELECTED_REAL_KIND(p=12)

! List of dummy arguments:
REAL(KIND=DBL),EXTERNAL :: f          ! Function to differentiate
REAL(KIND=DBL),INTENT(IN) :: x0       ! Location to take derivative
REAL(KIND=DBL),INTENT(IN) :: dx       ! Desired step size
REAL(KIND=DBL),INTENT(OUT) :: df_dx   ! Derivative
INTEGER,INTENT(OUT) :: error          ! Error flag: 0 = no error

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

   error = 1         ! dx must be > 0.

ELSE

   ! Calculate derivative using the specified dx.
   df_dx = (f(x0+dx) - f(x0) ) / dx
   error = 0

END IF

END SUBROUTINE dderiv
```

To test the subroutine, we will write a test driver program to evaluate the derivative of the function $f(x) = 10 \sin 20x$ at position $x = 0$. The test driver program is:

```
PROGRAM test_dderiv
!
!  Purpose:
!    To test subroutine dderiv by evaluating the derivative of
!    the function f(x) = 10. * SIN ( 20. * x) at x = 0.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/16/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare named constants:
INTEGER, PARAMETER :: DBL = SELECTED_REAL_KIND(p=12)

! External functions:
REAL(KIND=DBL),EXTERNAL :: fun   ! Function to take derivative of

! List of variables:
REAL(KIND=DBL) :: df_dx          ! Derivative
REAL(KIND=DBL) :: dx = 0.0001    ! Step size
INTEGER :: error                 ! Error flag

! Calculate derivative.
CALL dderiv ( fun, 0.0_DBL, dx, df_dx, error )
```

```
! Tell user.
WRITE (*,'(A,F16.7)') ' The derivative is = ', df_dx

END PROGRAM test_dderiv

FUNCTION fun(x)
!
!  Purpose:
!    To evaluate the function f(x) = 10. * SIN ( 20. * x)
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/16/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! Declare named constants:
INTEGER, PARAMETER :: DBL = SELECTED_REAL_KIND(p=12)

! List of dummy arguments:
REAL(KIND=DBL) :: fun             ! Function result
REAL(KIND=DBL),INTENT(IN) :: x    ! Location to evaluate fun

! Evaluate function
fun = 10. * SIN ( 20. * x )

END FUNCTION fun
```

When the test driver program is executed, the results are

```
C:\book\f95_2003\soln\ex11_6>test_dderiv
The derivative is =       199.9998667
```

If we calculate the derivative analytically, the result is $\frac{d}{dx}\left(10\sin 20x\right) = 200\cos 20x = 200$ for $x = 0$, which agrees well with the results of the program.

11-7 The test driver program shown below reads a set of equations from disk, and solves it in both single precision and double precision, comparing the resulting accuracy and speed. It is a modification of program `test_dsimul` in Figure 11-7. In the code shown below, the program solves each system of equations 10000 times to get an accurate time estimate. (The number of repetitions is controlled by parameter `n_loops`.)

```
PROGRAM time_simul
!
!  Purpose:
!    To time subroutines simul and dsimul, which solve a set of
!    N linear equations in N unknowns in single and double
!    precision respectively.  The results of the two solutions
!    are displayed together with their errors and their timings
!    in a summary table.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    11/27/06    S. J. Chapman        Original code
```

```
! 1. 05/16/07    S. J. Chapman        Modified from test_dsimul
!                                     (from Chapter 11 Fig 11-7)
!
IMPLICIT NONE

! Declare parameters
INTEGER, PARAMETER :: SGL = SELECTED_REAL_KIND(p=6)   ! Single
INTEGER, PARAMETER :: DBL = SELECTED_REAL_KIND(p=13)  ! Double
INTEGER, PARAMETER :: N_LOOPS = 20000 ! No. of times to solve eqns

! List of local variables
REAL(KIND=SGL), ALLOCATABLE, DIMENSION(:,:) :: a
                                 ! Single-precision coefficients
REAL(KIND=SGL), ALLOCATABLE, DIMENSION(:) :: b
                                 ! Single-precision constant values
REAL(KIND=SGL), ALLOCATABLE, DIMENSION(:) :: soln
                                 ! Single-precision solution
REAL(KIND=SGL), ALLOCATABLE, DIMENSION(:) :: serror
                                 ! Array of single-precision errors
REAL(KIND=SGL) :: serror_max     ! Max single precision error
REAL(KIND=SGL) :: sp_time        ! Single-precision solution time
REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:,:) :: da
                                 ! Double-precision coefficients
REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:) :: db
                                 ! Double-precision constant values
REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:) :: dsoln
                                 ! Double-precision solution
REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:) :: derror
                                 ! Array of double-precision errors
REAL(KIND=DBL) :: derror_max     ! Max double precision error
REAL(KIND=SGL) :: dp_time        ! Double-precision solution time
INTEGER :: error_flag            ! Error flag from subroutines
INTEGER :: i, j                  ! Loop index
INTEGER :: istat                 ! I/O status
INTEGER :: n                     ! Size of system of eqns to solve
CHARACTER(len=20) :: filename    ! Input data file name

! Get the name of the disk file containing the equations.
WRITE (*,*) 'Enter the file name containing the eqns: '
READ (*,'(A20)') filename

! Open input data file.  Status is OLD because the input data must
! already exist.
OPEN ( UNIT=1, FILE=filename, STATUS='OLD', ACTION='READ', &
     IOSTAT=istat )

! Was the OPEN successful?
open_ok: IF ( istat == 0 ) THEN

   ! The file was opened successfully, so read the number of
   ! equations in the system.
   READ (1,*) n

   ! Allocate memory for that number of equations
   ALLOCATE ( a(n,n), b(n), soln(n), serror(n), &
              da(n,n), db(n), dsoln(n), derror(n), STAT=istat )
```

```
! If the memory is available, read in equations and
! process them.
solve: IF ( istat == 0 ) THEN

   DO i = 1, n
      READ (1,*) (da(i,j), j=1,n), db(i)
   END DO

   ! Copy the coefficients in single precision for the
   ! single precision solution.
   a = da
   b = db

   ! Display coefficients.
   WRITE (*,1010)
   1010 FORMAT (/,1X,'Coefficients:')
   DO i = 1, n
      WRITE (*,'(1X,7F11.4)') (a(i,j), j=1,n), b(i)
   END DO

   ! Set up loop to solve single-precision equations.
   ! First, reset timer.
   CALL set_timer

   ! Solve single precision equations.
   DO i = 1, N_LOOPS
      CALL simul (a,  b,  soln,  n, n, error_flag )
   END DO

   ! Check time and get average.
   CALL elapsed_time ( sp_time )
   sp_time = sp_time / REAL(N_LOOPS)


   ! Set up loop to solve double-precision equations.
   ! First, reset timer.
   CALL set_timer

   ! Solve double precision equations.
   DO i = 1, N_LOOPS
      CALL dsimul (da, db, dsoln, n, n, error_flag )
   END DO

   ! Check time and get average.
   CALL elapsed_time ( dp_time )
   dp_time = dp_time / REAL(N_LOOPS)

   ! Check for error.
   error_check: IF ( error_flag /= 0 ) THEN
      WRITE (*,1020)
      1020 FORMAT (/1X,'Zero pivot encountered!', &
           //1X,'There is no unique solution to this system.')

   ELSE error_check
```

```
         ! No errors.  Check for roundoff by substituting into
         ! the original equations, and calculate the differences.
         serror_max = 0.
         derror_max = 0._DBL
         serror = 0.
         derror = 0._DBL
         DO i = 1, n
            serror(i) = SUM ( a(i,:)  * soln(:)  ) - b(i)
            derror(i) = SUM ( da(i,:) * dsoln(:) ) - db(i)
         END DO
         serror_max = MAXVAL ( ABS ( serror ) )
         derror_max = MAXVAL ( ABS ( derror ) )

         ! Tell user about it.
         WRITE (*,1030)
         1030 FORMAT (/1X,'  i     SP x(i)        DP x(i)      ', &
              '       SP Err         DP Err  ')
         WRITE (*,1040)
         1040 FORMAT ( 1X,' ===   =========      =========     ', &
              '      ========       ======== ')
         DO i = 1, n
            WRITE (*,1050) i, soln(i), dsoln(i), serror(i), derror(i)
            1050 FORMAT (1X, I3, 2X, 2G15.6, 2F15.8)
         END DO

         ! Write maximum errors.
         WRITE (*,1060) serror_max, derror_max, sp_time, dp_time
         1060 FORMAT (/,' Max single-precision error:',F15.8, &
                     /,' Max double-precision error:',F15.8, &
                     /,' Single-precision time:     ',F15.8, &
                     /,' Double-precision time:     ',F15.8)

      END IF error_check
   END IF solve

   ! Deallocate dynamic memory
   DEALLOCATE ( a, b, soln, serror, da, db, dsoln, derror )

ELSE open_ok
   ! Else file open failed.  Tell user.
   WRITE (*,1070) istat
   1070 FORMAT (1X,'File open failed--status = ', I6)
END IF open_ok

END PROGRAM time_simul
```

When this code is executed, the results are:

```
C:\book\f95_2003\soln\ex11_7>time_simul
Enter the file name containing the eqns:
sys10

 Coefficients:
    -2.0000     5.0000     1.0000     3.0000     4.0000    -1.0000     2.0000
    -1.0000    -5.0000    -2.0000    -5.0000
     6.0000     4.0000    -1.0000     6.0000    -4.0000    -5.0000     3.0000
```

```
     -1.0000     4.0000     3.0000    -6.0000
     -6.0000    -5.0000    -2.0000    -2.0000    -3.0000     6.0000     4.0000
      2.0000    -6.0000     4.0000    -7.0000
      2.0000     4.0000     4.0000     4.0000     5.0000    -4.0000     0.0000
      0.0000    -4.0000     6.0000     0.0000
     -4.0000    -1.0000     3.0000    -3.0000    -4.0000    -4.0000    -4.0000
      4.0000     3.0000    -3.0000     5.0000
      4.0000     3.0000     5.0000     1.0000     1.0000     1.0000     0.0000
      3.0000     3.0000     6.0000    -8.0000
      1.0000     2.0000    -2.0000     0.0000     3.0000    -5.0000     5.0000
      0.0000     1.0000    -4.0000     1.0000
     -3.0000    -4.0000     2.0000    -1.0000    -2.0000     5.0000    -1.0000
     -1.0000    -4.0000     1.0000    -4.0000
      5.0000     5.0000    -2.0000    -5.0000     1.0000    -4.0000    -1.0000
      0.0000    -2.0000    -3.0000    -7.0000
     -5.0000    -2.0000    -5.0000     2.0000     1.0000    -3.0000     4.0000
     -1.0000    -4.0000     4.0000     6.0000

   i     SP x(i)        DP x(i)           SP Err        DP Err
  ===   =========      =========         ========      ========
   1     0.956894E-01   0.956888E-01    -0.00000064    0.00000000
   2     -2.01321       -2.01321         0.00000063    0.00000000
   3     -1.19987       -1.19987         0.00000283    0.00000000
   4      1.45360        1.45360        -0.00000012    0.00000000
   5      1.42511        1.42511         0.00000004    0.00000000
   6    -0.830689      -0.830689        -0.00000136    0.00000000
   7     -1.60190       -1.60190        -0.00000073    0.00000000
   8      1.33138        1.33138         0.00000063    0.00000000
   9     0.173693       0.173694        -0.00000095    0.00000000
  10    -0.484499      -0.484498         0.00000048    0.00000000

Max single-precision error:     0.00000283
 Max double-precision error:     0.00000000
 Single-precision time:          0.00000545
 Double-precision time:          0.00000550
```

Note that the times are almost the same for the single precision and double precision solutions. This is true because the problem was run on an Pentium-based PC. Intel floating-point processors calculate all numbers to 80-bit accuracy regardless of precision, so the time required for single and double precision numbers is almost the same. *This will not be true on most other computers*. On other machines, there will be a significant speed penalty for double precision calculations.

11-8 A program to determine the kinds of integers available on a given processor is shown below:

```
PROGRAM find_int_kinds
!
!  Purpose:
!    To determine the valid integer kinds on a particular
!    computer.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/16/2007    S. J. Chapman       Original code
!
IMPLICIT NONE
```

```
! Declare variables
INTEGER :: kind                ! Kind number
INTEGER :: range = 0           ! Range (power of 10)

! Find all kinds until the functionr returns a -1.
DO
   range = range + 1               ! Increase range
   kind = SELECTED_INT_KIND(range) ! Get kind number

   ! Get info about this kind number if it is new.
   WRITE (*,'(A,I2,A,I2)') ' range = 10**', range,' kind = ', kind

   IF ( kind < 0 ) EXIT
END DO

END PROGRAM find_int_kinds
```

When the program is executed on a PC with the Compaq Visual Fortran 6.6 compiler, the results are as shown. Different results may be expected with other computers and compilers.

```
C:\book\f95_2003\soln\ex11_8>find_int_kinds
range = 10** 1 kind =  1
range = 10** 2 kind =  1
range = 10** 3 kind =  2
range = 10** 4 kind =  2
range = 10** 5 kind =  4
range = 10** 6 kind =  4
range = 10** 7 kind =  4
range = 10** 8 kind =  4
range = 10** 9 kind =  4
range = 10**10 kind = -1
```

11-9 Subroutine `csimul` is shown below:

```
SUBROUTINE csimul ( a, b, soln, ndim, n, error )
!
!  Purpose:
!    Subroutine to solve a set of N complex linear equations
!    in N unknowns using Gaussian elimination and the maximum
!    pivot technique.  This version of simul has been
!    modified to use array sections and automatic arrays.
!    It DOES NOT DESTROY the original input values.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    11/23/06    S. J. Chapman        Original code
! 1. 11/24/06    S. J. Chapman        Add automatic arrays
! 2. 11/27/06    S. J. Chapman        Double precision
! 3. 05/17/07    S. J. Chapman        Single prec. complex
!
IMPLICIT NONE

! Declare parameters
INTEGER, PARAMETER :: SGL = SELECTED_REAL_KIND(p=6)
```

```
! Declare calling arguments:
INTEGER, INTENT(IN) :: ndim          ! Dimension of arrays a and b
COMPLEX(KIND=SGL), INTENT(IN), DIMENSION(ndim,ndim) :: a
                                     ! Array of coefficients (N x N).
                                     ! This array is of size ndim x
                                     ! ndim, but only N x N of the
                                     ! coefficients are being used.
COMPLEX(KIND=SGL), INTENT(IN), DIMENSION(ndim) :: b
                                     ! Input: Right-hand side of eqns.
COMPLEX(KIND=SGL), INTENT(OUT), DIMENSION(ndim) :: soln
                                     ! Output: Solution vector.
INTEGER, INTENT(IN) :: n             ! Number of equations to solve.
INTEGER, INTENT(OUT) :: error        ! Error flag:
                                     !   0 -- No error
                                     !   1 -- Singular equations

! Declare local parameters
REAL(KIND=SGL), PARAMETER :: EPSILON = 1.0E-5
                                     ! A "small" number for comparison
                                     ! when determining singular eqns

! Declare local variables:
COMPLEX(KIND=SGL), DIMENSION(n,n) :: a1 ! Copy of "a" which will be
                                     ! destroyed during the solution
COMPLEX(KIND=SGL) :: factor          ! Factor to multiply eqn irow by
                                     ! before adding to eqn jrow
INTEGER :: irow                      ! Number of the equation currently
                                     ! currently being processed
INTEGER :: ipeak                     ! Pointer to equation containing
                                     ! maximum pivot value
INTEGER :: jrow                      ! Number of the equation compared
                                     ! to the current equation
COMPLEX(KIND=SGL) :: temp            ! Scratch value
COMPLEX(KIND=SGL),DIMENSION(n) :: temp1 ! Scratch array

! Make copies of arrays "a" and "b" for local use
a1 = a(1:n,1:n)
soln = b(1:n)

! Process N times to get all equations...
mainloop: DO irow = 1, n

   ! Find peak pivot for column irow in rows irow to N
   ipeak = irow
   max_pivot: DO jrow = irow+1, n
      IF (ABS(a1(jrow,irow)) > ABS(a1(ipeak,irow))) THEN
         ipeak = jrow
      END IF
   END DO max_pivot

   ! Check for singular equations.
   singular: IF ( ABS(a1(ipeak,irow)) < EPSILON ) THEN
      error = 1
      RETURN
   END IF singular
```

```
   ! Otherwise, if ipeak /= irow, swap equations irow & ipeak
   swap_eqn: IF ( ipeak /= irow ) THEN
      temp1 = a1(ipeak,1:n)
      a1(ipeak,1:n) = a1(irow,1:n)   ! Swap rows in a
      a1(irow,1:n) = temp1
      temp = soln(ipeak)
      soln(ipeak) = soln(irow)       ! Swap rows in b
      soln(irow)  = temp
   END IF swap_eqn

   ! Multiply equation irow by -a1(jrow,irow)/a1(irow,irow),
   ! and add it to Eqn jrow (for all eqns except irow itself).
   eliminate: DO jrow = 1, n
      IF ( jrow /= irow ) THEN
         factor = -a1(jrow,irow)/a1(irow,irow)
         a1(jrow,1:n) = a1(irow,1:n)*factor + a1(jrow,1:n)
         soln(jrow) = soln(irow)*factor + soln(jrow)
      END IF
   END DO eliminate
END DO mainloop

! End of main loop over all equations.  All off-diagonal
! terms are now zero.  To get the final answer, we must
! divide each equation by the coefficient of its on-diagonal
! term.
divide: DO irow = 1, n
   soln(irow) = soln(irow) / a1(irow,irow)
END DO divide

! Set error flag to 0 and return.
error = 0

END SUBROUTINE csimul
```

A test driver routine for this subroutine is shown below:

```
PROGRAM test_csimul
!
!  Purpose:
!    To test subroutine csimul, which solves a set of N complex
!    linear equations in N unknowns.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/17/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare parameters
INTEGER, PARAMETER :: SGL = SELECTED_REAL_KIND(p=6)   ! Single

! List of local variables
COMPLEX(KIND=SGL), ALLOCATABLE, DIMENSION(:,:) :: a
                                 ! Single-precision coefficients
```

```
COMPLEX(KIND=SGL), ALLOCATABLE, DIMENSION(:) :: b
                                  ! Single-precision constant values
COMPLEX(KIND=SGL), ALLOCATABLE, DIMENSION(:) :: soln
                                  ! Single-precision solution
INTEGER :: error_flag             ! Error flag from subroutines
INTEGER :: i, j                   ! Loop index
INTEGER :: istat                  ! I/O status
INTEGER :: n                      ! Size of system of eqns to solve
CHARACTER(len=20) :: filename     ! Input data file name

! Get the name of the disk file containing the equations.
WRITE (*,*) 'Enter the file name containing the eqns: '
READ (*,'(A20)') filename

! Open input data file.  Status is OLD because the input data must
! already exist.
OPEN ( UNIT=1, FILE=filename, STATUS='OLD', ACTION='READ', &
     IOSTAT=istat )

! Was the OPEN successful?
open_ok: IF ( istat == 0 ) THEN

   ! The file was opened successfully, so read the number of
   ! equations in the system.
   READ (1,*) n

   ! Allocate memory for that number of equations
   ALLOCATE ( a(n,n), b(n), soln(n), STAT=istat )

   ! If the memory is available, read in equations and
   ! process them.
   solve: IF ( istat == 0 ) THEN

      DO i = 1, n
         READ (1,*) (a(i,j), j=1,n), b(i)
      END DO

      ! Display coefficients.
      WRITE (*,1010)
      1010 FORMAT (/,1X,'Coefficients:')
      DO i = 1, n
         WRITE (*,1020) (a(i,j), j=1,n), b(i)
         1020 FORMAT (1X,4(:,'(',F7.3,',',F7.3,')',1X))
      END DO

      ! Solve equations.
      CALL csimul  (a, b, soln, n, n, error_flag )

      ! Check for error.
      error_check: IF ( error_flag /= 0 ) THEN
         WRITE (*,1030)
         1030 FORMAT (/1X,'Zero pivot encountered!', &
              //1X,'There is no unique solution to this system.')

      ELSE error_check
```

```
         WRITE (*,1040)
         1040 FORMAT (/,' The solutions are:')
         DO i = 1, n
            WRITE (*,1050) i, soln(i)
            1050 FORMAT (3X,'X(',I2,') = ','(',F12.6,',',F12.6,')')
         END DO

      END IF error_check
   END IF solve

   ! Deallocate dynamic memory
   DEALLOCATE ( a, b, soln )

ELSE open_ok
   ! Else file open failed.  Tell user.
   WRITE (*,1070) istat
   1070 FORMAT (' File open failed--status = ', I6)
END IF open_ok

END PROGRAM test_csimul
```

When this routine is tested with the specified system of equations, the results are:

```
C:\book\f95_2003\soln\ex11_9>test_csimul
Enter the file name containing the eqns:
sysc3

Coefficients:
( -2.000,  5.000) (  1.000,  3.000) (  4.000, -1.000) (  7.000,  5.000)
(  2.000, -1.000) ( -5.000, -2.000) (  6.000,  4.000) (-10.000, -8.000)
( -1.000,  6.000) ( -4.000, -5.000) (  3.000, -1.000) ( -3.000, -3.000)

The solutions are:
  X( 1) = (     .155630,   -1.352914)
  X( 2) = (    1.342764,    -.668149)
  X( 3) = (    -.550470,    -.598647)
```

11-10 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
!       ====       ==========          =====================
!    05/17/2007    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 complex_2_amp_phase
```

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
!       ====       ==========         =====================
!    05/17/2007    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 test
```

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

```
C:\book\f95_2003\soln\ex11_10>test
Enter a complex number:
(1,0)
Amplitude =     1.0000
Phase     =      .0000

C:\book\f95_2003\soln\ex11_10>test
Enter a complex number:
(0,1)
Amplitude =     1.0000
Phase     =     1.5708
```

```
C:\book\f95_2003\soln\ex11_10>test
Enter a complex number:
(-1,0)
Amplitude =     1.0000
Phase     =     3.1416

C:\book\f95_2003\soln\ex11_10>test
Enter a complex number:
(0,-1)
Amplitude =     1.0000
Phase     =    -1.5708
```

11-11 A function to calculate Euler's equation is shown below:

```
FUNCTION euler ( theta )
!
!  Purpose:
!    Function to calculate Euler's equation.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/17/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
REAL,INTENT(IN) :: theta            ! Angle in radians
COMPLEX :: euler                    ! Function result

! Calculate Euler's equation.
euler = CMPLX ( COS(theta), SIN(theta) )

END FUNCTION euler
```

A test driver program is shown below:

```
PROGRAM test_euler
!
!  Purpose:
!    To test function euler, which uses Euler's equation to
!    calculate e**(i*theta).
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/17/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of external functions:
COMPLEX,EXTERNAL :: euler             ! Euler's equation

! List of variables:
REAL :: theta                         ! Angle in radians
```

```
! Get input values.
WRITE (*,'(A)') ' Enter phase in radians: '
READ (*,*) theta

! Tell user.
WRITE (*,*) "The exponent by Euler's Equation is", euler(theta)
WRITE (*,*) 'The exponent by CEXP is            ', CEXP (CMPLX(0.,theta))

END PROGRAM test_euler
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex11_11>test_euler
Enter phase in radians:
0
The exponent by Euler's Equation is         (1.000000,0.000000E+00)
The exponent by CEXP is                     (1.000000,0.000000E+00)

C:\book\f95_2003\soln\ex11_11>test_euler
Enter phase in radians:
1.570796
The exponent by Euler's Equation is         (3.139165E-07,1.000000)
The exponent by CEXP is                     (3.139165E-07,1.000000)

C:\book\f95_2003\soln\ex11_11>test_euler
Enter phase in radians:
3.141593
The exponent by Euler's Equation is       (-1.000000,-3.258414E-07)
The exponent by CEXP is                   (-1.000000,-3.258414E-07)
```

Chapter 12. Additional Data Types

12-1 In Example 12-1, "APO" was placed ahead of "Anywhere" because the sorting was done according to the ASCII collating sequence, and in that sequence upper case letters are lower than lower case letters. To fix this problem, we must rewrite the example to always do comparisons in upper case. An easy way to do so is to take advantage of subroutine `ucase`.

```
MODULE types
!
!  Purpose:
!    To define the derived data type used for the customer
!    database.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    12/04/06    S. J. Chapman        Original code
!
IMPLICIT NONE

! Declare type personal_info
TYPE :: personal_info
   CHARACTER(len=12) :: first        ! First name
   CHARACTER         :: mi           ! Middle Initial
   CHARACTER(len=12) :: last         ! Last name
   CHARACTER(len=26) :: street       ! Street Address
   CHARACTER(len=12) :: city         ! City
   CHARACTER(len=2)  :: state        ! State
   INTEGER           :: zip          ! Zip code
END TYPE personal_info

END MODULE types


PROGRAM customer_database
!
!  Purpose:
!    To read in a character input data set, sort it into ascending
!    order using the selection sort algorithm, and to write the
!    sorted data to the standard output device.  This program calls
!    subroutine "sort_database" to do the actual sorting.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    12/04/06    S. J. Chapman        Original code
!    05/18/07    S. J. Chapman        Modified to do char comparisons
!                                     in upper case
```

```
!
USE types                       ! Declare the module types
IMPLICIT NONE

! Data dictionary: declare constants
INTEGER, PARAMETER :: MAX_SIZE = 100  ! Max addresses in database

! Data dictionary: declare external functions
LOGICAL, EXTERNAL :: lt_last     ! Comparison fn for last names
LOGICAL, EXTERNAL :: lt_city     ! Comparison fn for cities
LOGICAL, EXTERNAL :: lt_zip      ! Comparison fn for zip codes

! Data dictionary: declare variable types & definitions
TYPE(personal_info), DIMENSION(MAX_SIZE) :: customers
                                 ! Data array to sort
INTEGER :: choice                ! Choice of how to sort database
LOGICAL :: exceed = .FALSE.      ! Logical indicating that array
                                 !   limits are exceeded.
CHARACTER(len=20) :: filename    ! Input data file name
INTEGER :: i                     ! Loop index
INTEGER :: nvals = 0             ! Number of data values to sort
INTEGER :: status                ! I/O status: 0 for success
TYPE(personal_info) :: temp      ! Temporary variable for reading

! Get the name of the file containing the input data.
WRITE (*,*) 'Enter the file name with customer database: '
READ (*,'(A20)') filename

! Open input data file.  Status is OLD because the input data must
! already exist.
OPEN ( UNIT=9, FILE=filename, STATUS='OLD', IOSTAT=status )

! Was the OPEN successful?
fileopen: IF ( status == 0 ) THEN       ! Open successful

   ! The file was opened successfully, so read the customer
   ! database from it.
   DO
      READ (9, 1010, IOSTAT=status) temp   ! Get value
      1010 FORMAT (A12,1X,A1,1X,A12,1X,A26,1X,A12,1X,A2,1X,I5)
      IF ( status /= 0 ) EXIT             ! Exit on end of data
      nvals = nvals + 1                   ! Bump count
      size: IF ( nvals <= MAX_SIZE ) THEN  ! Too many values?
         customers(nvals) = temp          ! No: Save value in array
      ELSE
         exceed = .TRUE.                  ! Yes: Array overflow
      END IF size
   END DO

   ! Was the array size exceeded?  If so, tell user and quit.
   toobig: IF ( exceed ) THEN
      WRITE (*,1020) nvals, MAX_SIZE
      1020 FORMAT (' Maximum array size exceeded: ', I6, ' > ', I6 )
   ELSE

      ! Limit not exceeded: find out how to sort data.
```

```
      WRITE (*,1030)
      1030 FORMAT (1X,'Enter way to sort database:',/, &
                   1X,'  1 -- By last name ',/, &
                   1X,'  2 -- By city ',/, &
                   1X,'  3 -- By zip code ')
      READ (*,*) choice

      ! Sort database
      SELECT CASE ( choice)
      CASE (1)
         CALL sort_database (customers, nvals, lt_last )
      CASE (2)
         CALL sort_database (customers, nvals, lt_city )
      CASE (3)
         CALL sort_database (customers, nvals, lt_zip )
      CASE DEFAULT
         WRITE (*,*) 'Invalid choice entered!'
      END SELECT

      ! Now write out the sorted data.
      WRITE (*,'(A)') ' The sorted database values are: '
      WRITE (*,1040) ( customers(i), i = 1, nvals )
      1040 FORMAT (1X,A12,1X,A1,1X,A12,1X,A26,1X,A12,1X,A2,1X,I5)

   END IF toobig

ELSE fileopen

   ! Status /= 0, so an open error occurred.
   WRITE (*,'(A,I6)') ' File open error: IOSTAT = ', status

END IF fileopen

END PROGRAM customer_database


SUBROUTINE sort_database (array, n, lt_fun )
!
!  Purpose:
!    To sort array "array" into ascending order using a selection
!    sort, where "array" is an array of the derived data type
!    "personal_info".  The sort is based on the the external
!    comparison function "lt_fun", which will differ depending on
!    which component of the derived type array is used for
!    comparison.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    12/04/06    S. J. Chapman        Original code
!
USE types                        ! Declare the module types
IMPLICIT NONE

! Data dictionary: declare calling parameter types & definitions
INTEGER, INTENT(IN) :: n                   ! Number of values
```

```
TYPE(personal_info), DIMENSION(n), INTENT(INOUT) :: array
                                           ! Array to be sorted
LOGICAL, EXTERNAL :: lt_fun                ! Comparison function

! Data dictionary: declare local variable types & definitions
INTEGER :: i                    ! Loop index
INTEGER :: iptr                 ! Pointer to smallest value
INTEGER :: j                    ! Loop index
TYPE(personal_info) :: temp     ! Temp variable for swaps

! Sort the array
outer: DO i = 1, n-1

   ! Find the minimum value in array(i) through array(n)
   iptr = i
   inner: DO j = i+1, n
      minval: IF ( lt_fun(array(J),array(iptr)) ) THEN
         iptr = j
      END IF minval
   END DO inner

   ! iptr now points to the minimum value, so swap array(iptr)
   ! with array(i) if i /= iptr.
   swap: IF ( i /= iptr ) THEN
      temp        = array(i)
      array(i)    = array(iptr)
      array(iptr) = temp
   END IF swap

END DO outer
END SUBROUTINE sort_database


LOGICAL FUNCTION lt_last (a, b)
!
!  Purpose:
!    To compare variables "a" and "b" and determine which
!    has the smaller last name (lower alphabetical order).
!
USE types                     ! Declare the module types
IMPLICIT NONE

! Declare calling arguments
TYPE (personal_info), INTENT(IN) :: a, b

! Declare local variables:
CHARACTER(len=LEN(a%last)) :: last1
CHARACTER(len=LEN(b%last)) :: last2

! Get upper-case versions:
last1 = a%last
last2 = b%last
CALL ucase(last1)
CALL ucase(last2)

! Make comparison.
```

```
lt_last = LLT ( last1, last2 )

END FUNCTION lt_last


LOGICAL FUNCTION lt_city (a, b)
!
!  Purpose:
!    To compare variables "a" and "b" and determine which
!    has the smaller city (lower alphabetical order).
!
USE types                    ! Declare the module types
IMPLICIT NONE

! Declare calling arguments
TYPE (personal_info), INTENT(IN) :: a, b

! Declare local variables:
CHARACTER(len=LEN(a%city)) :: city1
CHARACTER(len=LEN(b%city)) :: city2

! Get upper-case versions:
city1 = a%city
city2 = b%city
CALL ucase(city1)
CALL ucase(city2)

! Make comparison.
lt_city = LLT ( city1, city2 )

END FUNCTION lt_city


LOGICAL FUNCTION lt_zip (a, b)
!
!  Purpose:
!    To compare variables "a" and "b" and determine which
!    has the smaller zip code (lower numerical value).
!
USE types                    ! Declare the module types
IMPLICIT NONE

! Data dictionary: declare calling parameter types & definitions
TYPE (personal_info), INTENT(IN) :: a, b

! Make comparison.
lt_zip = a%zip < b%zip

END FUNCTION lt_zip
```

When this version of the program is tested, the results are:

```
C:\book\f95_2003\soln\ex12_1>customer_database
Enter the file name with customer database:
database
Enter way to sort database:
```

```
  1 -- By last name
  2 -- By city
  3 -- By zip code
1
The sorted database values are:
Jane         X Doe          12 Lakeside Drive           Glenview     IL 60025
Andrew       D Jackson      Jackson Square              New Orleans  LA 70003
Colin        A Jeffries     11 Main Street              Chicago      IL 60003
James        R Johnson      Rt. 5 Box 207C              West Monroe  LA 71291
John         Q Public       123 Sesame Street           Anywhere     NY 10035
Joseph       P Ziskend      P. O. Box 433               APO          AP 96555


C:\book\f95_2003\soln\ex12_1>customer_database
Enter the file name with customer database:
database
Enter way to sort database:
  1 -- By last name
  2 -- By city
  3 -- By zip code
2
The sorted database values are:
John         Q Public       123 Sesame Street           Anywhere     NY 10035
Joseph       P Ziskend      P. O. Box 433               APO          AP 96555
Colin        A Jeffries     11 Main Street              Chicago      IL 60003
Jane         X Doe          12 Lakeside Drive           Glenview     IL 60025
Andrew       D Jackson      Jackson Square              New Orleans  LA 70003
James        R Johnson      Rt. 5 Box 207C              West Monroe  LA 71291


C:\book\f95_2003\soln\ex12_1>customer_database
Enter the file name with customer database:
database
Enter way to sort database:
  1 -- By last name
  2 -- By city
  3 -- By zip code
3
The sorted database values are:
John         Q Public       123 Sesame Street           Anywhere     NY 10035
Colin        A Jeffries     11 Main Street              Chicago      IL 60003
Jane         X Doe          12 Lakeside Drive           Glenview     IL 60025
Andrew       D Jackson      Jackson Square              New Orleans  LA 70003
James        R Johnson      Rt. 5 Box 207C              West Monroe  LA 71291
Joseph       P Ziskend      P. O. Box 433               APO          AP 96555
```

12-2 A module that declares a type "polar" and defines two functions to convert between complex numbers and polar number is shown below:

```
MODULE polar_math
!
!  Purpose:
!    To define the derived data type "polar" plus two functions
!    that use it.
!
!  Record of revisions:
```

```
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare type "polar"
TYPE :: polar
   REAL :: z                      ! magnitude
   REAL :: phase                  ! Angle in degrees
END TYPE polar

! Declare named constants:
REAL,PARAMETER :: DEG_2_RAD = .017453293  ! Degrees to radians
REAL,PARAMETER :: RAD_2_DEG = 57.2957795  ! Radians to degrees

CONTAINS

FUNCTION complex_2_polar(c)
!
!  Purpose:
!    To convert a complex number to type "polar".
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
COMPLEX,INTENT(IN) :: c             ! Complex number
TYPE (polar) :: complex_2_polar     ! Result in polar form

! Get magnitude and angle
complex_2_polar%z     = ABS ( c )
complex_2_polar%phase = ATAN2( AIMAG(c), REAL(c) ) * RAD_2_DEG

END FUNCTION complex_2_polar

FUNCTION polar_2_complex(polar_value)
!
!  Purpose:
!    To convert a "polar" number to complex.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
TYPE (polar),INTENT(IN) :: polar_value ! Polar number
COMPLEX :: polar_2_complex             ! Result in complex form
```

```
! Declare local variables:
REAL :: re                          ! Real component
REAL :: im                          ! Imaginary component

! Get real and imaginary parts
re = polar_value%z * COS ( polar_value%phase * DEG_2_RAD )
im = polar_value%z * SIN ( polar_value%phase * DEG_2_RAD )
polar_2_complex = CMPLX ( re, im )

END FUNCTION polar_2_complex

END MODULE polar_math

PROGRAM test_polar
USE polar_math

COMPLEX :: c
TYPE (polar) :: p

c = (1.,1.)
p = complex_2_polar(c)

WRITE (*,*) 'c = ', c
WRITE (*,*) 'complex_2_polar(c) = ', complex_2_polar(c)
WRITE (*,*) 'polar_2_complex(p) = ', polar_2_complex(p)

END PROGRAM test_polar
```

12-3 Function `polar_times_polar` is shown below. Note that it is in a module to provide an explicit interface.

```
MODULE polar_math
!
!  Purpose:
!    To define the derived data type "polar" plus two functions
!    that use it.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare type "polar"
TYPE :: polar
   REAL :: z                     ! magnitude
   REAL :: phase                 ! Angle in degrees
END TYPE polar

! Declare named constants:
REAL,PARAMETER :: DEG_2_RAD = .017453293  ! Degrees to radians
REAL,PARAMETER :: RAD_2_DEG = 57.2957795  ! Radians to degrees

CONTAINS

FUNCTION polar_times_polar(polar1, polar2)
```

```
!
!  Purpose:
!    To multiply two polar numbers and produce a polar result.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/18/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! Declare dummy arguments:
TYPE (polar),INTENT(IN) :: polar1      ! Polar value 1
TYPE (polar),INTENT(IN) :: polar2      ! Polar value 2
TYPE (polar) :: polar_times_polar      ! Function result

! Calculate result
polar_times_polar%z = polar1%z * polar2%z
polar_times_polar%phase = polar1%phase + polar2%phase

! Now limit phase to valid range: -180 < phase <= 180.
DO
   IF ( polar_times_polar%phase > -180. ) EXIT
   polar_times_polar%phase = polar_times_polar%phase + 360.
END DO

DO
   IF ( polar_times_polar%phase <= 180. ) EXIT
   polar_times_polar%phase = polar_times_polar%phase - 360.
END DO

END FUNCTION polar_times_polar

END MODULE polar_math
```

A test driver program is shown below.

```
PROGRAM test_polar_times_polar
USE polar_math
IMPLICIT NONE

TYPE (polar) :: p1, p2, p_result

WRITE (*,*) 'Enter first polar number (mag,angle): '
READ (*,*) p1%z, p1%phase
WRITE (*,*) 'Enter second polar number (mag,angle): '
READ (*,*) p2%z, p2%phase
p_result = polar_times_polar(p1,p2)
WRITE (*,*) 'The result is: ', p_result

END PROGRAM test_polar_times_polar
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln\ex12_3>polar_times_polar
Enter first polar number (mag,angle):
```

```
3. 270.
Enter second polar number (mag,angle):
2. 90.
The result is:        6.000000    0.000000E+00

C:\book\f95_2003\soln\ex12_3>polar_times_polar
Enter first polar number (mag,angle):
.5 -70.
Enter second polar number (mag,angle):
6 90.
The result is:        3.000000       20.000000

C:\book\f95_2003\soln\ex12_3>polar_times_polar
Enter first polar number (mag,angle):
12. -170.
Enter second polar number (mag,angle):
0.4 -20.
The result is:        4.800000      170.000000
```

These answers are correct, as we can show by simple hand calculations.

12-4 Function `polar_div_polar` is shown below. Note that it is in a module to provide an explicit interface.

```
MODULE polar_math
!
!  Purpose:
!    To define the derived data type "polar" plus two functions
!    that use it.
!
!  Record of revisions:
!      Date       Programmer         Description of change
!      ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare type "polar"
TYPE :: polar
   REAL :: z                      ! magnitude
   REAL :: phase                  ! Angle in degrees
END TYPE polar

! Declare named constants:
REAL,PARAMETER :: DEG_2_RAD = .017453293  ! Degrees to radians
REAL,PARAMETER :: RAD_2_DEG = 57.2957795  ! Radians to degrees

CONTAINS

FUNCTION polar_div_polar(polar1, polar2)
!
!  Purpose:
!    To divide two polar numbers and produce a polar result.
!
!  Record of revisions:
!      Date       Programmer         Description of change
!      ====       ==========         =====================
```

```
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
TYPE (polar),INTENT(IN) :: polar1      ! Polar value 1
TYPE (polar),INTENT(IN) :: polar2      ! Polar value 2
TYPE (polar) :: polar_div_polar        ! Function result

! Calculate result
polar_div_polar%z = polar1%z / polar2%z
polar_div_polar%phase = polar1%phase - polar2%phase

! Now limit%phase to valid range: -180 <%phase <= 180.
DO
   IF ( polar_div_polar%phase > -180. ) EXIT
   polar_div_polar%phase = polar_div_polar%phase + 360.
END DO

DO
   IF ( polar_div_polar%phase <= 180. ) EXIT
   polar_div_polar%phase = polar_div_polar%phase - 360.
END DO

END FUNCTION polar_div_polar

END MODULE polar_math
```

A test driver program is shown below.

```
PROGRAM test_polar_div_polar
USE polar_math
IMPLICIT NONE

TYPE (polar) :: p1, p2, p_result

WRITE (*,*) 'Enter first polar number (mag,angle): '
READ (*,*) p1%z, p1%phase
WRITE (*,*) 'Enter second polar number (mag,angle): '
READ (*,*) p2%z, p2%phase
p_result = polar_div_polar(p1,p2)
WRITE (*,*) 'The result is: ', p_result

END PROGRAM test_polar_div_polar
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln\ex12_4>polar_div_polar
Enter first polar number (mag,angle):
10   30
Enter second polar number (mag,angle):
2   180
The result is:        5.000000     -150.000000

C:\book\f95_2003\soln\ex12_4>polar_div_polar
Enter first polar number (mag,angle):
```

```
10  -30
Enter second polar number (mag,angle):
2  180
The result is:        5.000000      150.000000
```

These answers are correct, as we can show by simple hand calculations.

12-5 A version of the polar data type with bound procedures is shown below. Note that functions `to_complex`, `times`, and `div` are bound, but function `to_polar` is not, because it does not make sense in this context (there is no polar number to bind that function to).

```
MODULE polar_math
!
!  Purpose:
!    To define the derived data type "polar" plus four functions
!    that use it.  The functions (except for complex_2_polar)
!    are Fortran 2003 bound procedures.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare type "polar"
TYPE :: polar
   REAL :: z                       ! magnitude
   REAL :: phase                   ! Angle in degrees
CONTAINS
   PROCEDURE,PASS :: to_complex
   PROCEDURE,PASS :: times
   PROCEDURE,PASS :: div
END TYPE polar

! Declare named constants:
REAL,PARAMETER :: DEG_2_RAD = .017453293  ! Degrees to radians
REAL,PARAMETER :: RAD_2_DEG = 57.2957795  ! Radians to degrees

CONTAINS


FUNCTION to_polar(c)
!
!  Purpose:
!    To convert a complex number to type "polar".
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
COMPLEX,INTENT(IN) :: c            ! Complex number
```

```
TYPE (polar) :: to_polar            ! Result in polar form

! Get magnitude and angle
to_polar%z     = ABS ( c )
to_polar%phase = ATAN2( AIMAG(c), REAL(c) ) * RAD_2_DEG

END FUNCTION to_polar


FUNCTION to_complex(this)
!
!  Purpose:
!    To convert the polar number "this" to complex.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
CLASS(polar),INTENT(IN) :: this        ! Polar number
COMPLEX :: to_complex                  ! Result in complex form

! Declare local variables:
REAL :: re                          ! Real component
REAL :: im                          ! Imaginary component

! Get real and imaginary parts
re = this%z * COS ( this%phase * DEG_2_RAD )
im = this%z * SIN ( this%phase * DEG_2_RAD )
to_complex = CMPLX ( re, im )

END FUNCTION to_complex


FUNCTION times(this, polar2)
!
!  Purpose:
!    To multiply two polar numbers "this * polar2" and
!    produce a polar result.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
CLASS(polar),INTENT(IN) :: this        ! Polar value 1
TYPE (polar),INTENT(IN) :: polar2      ! Polar value 2
TYPE (polar) :: times                  ! Function result

! Calculate result
```

```
times%z = this%z * polar2%z
times%phase = this%phase + polar2%phase

! Now limit phase to valid range: -180 < phase <= 180.
DO WHILE ( times%phase <= -180. )
   times%phase = times%phase + 360.
END DO

DO WHILE ( times%phase > 180. )
   times%phase = times%phase - 360.
END DO

END FUNCTION times


FUNCTION div(this, polar2)
!
!  Purpose:
!    To divide two polar numbers "this / polar2" and produce
!    a polar result.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

CLASS(polar),INTENT(IN) :: this        ! Polar value 1
TYPE (polar),INTENT(IN) :: polar2      ! Polar value 2
TYPE (polar) :: div                    ! Function result

! Calculate result
div%z = this%z / polar2%z
div%phase = this%phase - polar2%phase

! Now limit phase to valid range: -180 < phase <= 180.
DO WHILE ( div%phase <= -180. )
   div%phase = div%phase + 360.
END DO

DO WHILE ( div%phase > 180. )
   div%phase = div%phase - 360.
END DO

END FUNCTION div


END MODULE polar_math
```

A test driver program is shown below.

```
PROGRAM test_polar
USE polar_math

! Declare variables
```

```
COMPLEX :: c1, c2
TYPE (polar) :: p1, p2, p3, p4

! Get input data
WRITE (*,*) 'Enter first complex number:'
READ(*,*) c1
WRITE (*,*) 'Enter second complex number:'
READ(*,*) c2

! Convert these numbers to polar form
p1 = to_polar(c1)
p2 = to_polar(c2)

! Multiply these numbers
p3 = p1%times(p2)
WRITE (*,*) 'p3  = ', p3, ': complex = ', p3%to_complex()

! Divide these numbers
p4 = p1%div(p2)
WRITE (*,*) 'p4  = ', p4, ': complex = ', p4%to_complex()

END PROGRAM test_polar
```

When the program is executed, the results are:

```
C:\book\f95_2003\soln\ex12_5>test_polar
Enter first complex number:
(1.,1.)
 Enter second complex number:
(1.,1.)
 p3  =    1.9999999  90.0000000 : complex =  (2.4335927E-08,1.9999999)
 p4  =    1.0000000   0.0000000 : complex =  (1.0000000,0.0000000)
```

The two input numbers are both $1+j1=\sqrt{2}\angle 45°$. Therefore the results of these operations are

$$p3 = p1 \times p2 = \left(\sqrt{2}\angle 45°\right)\left(\sqrt{2}\angle 45°\right) = 2\angle 90° = j2$$

$$p4 = \frac{p1}{p2} = \frac{\left(\sqrt{2}\angle 45°\right)}{\left(\sqrt{2}\angle 45°\right)} = 1$$

The answers from the program match the answers calculated by hand.

12-6 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
```

12-7 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
!      ====       ==========          =====================
!    05/18/2007    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
!      ====       ==========          =====================
!    05/18/2007    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
!       ====       ==========          =====================
!    05/18/2007    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 test_distance
```

When this program is executed, the results are.

```
C:\book\f95_2003\soln\ex12_7>test_distance
Enter first point:
0 0
Enter second point:
3 4
The result is:        5.000000

C:\book\f95_2003\soln\ex12_7>test_distance
Enter first point:
1 -1
Enter second point:
1 1
The result is:        2.000000
```

12-8 A function to calculate the slope and intercept of a line from 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
!       ====       ==========          =====================
!    05/18/2007    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 calc_line (p1, p2)
!
!  Purpose:
!    To calculate the slope and intercept of the line determined
!    by the two points p1 and p2.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/18/2007    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
TYPE (line) :: calc_line             ! Resulting_line

! Calculate slope
IF ( p1%x /= p2%x ) THEN
   calc_line%m = ( p2%y - p1%y ) / ( p2%x - p1%x )
   calc_line%b = p1%y - calc_line%m * p1%x
ELSE
   calc_line%m = 0.
   calc_line%b = 0.
END IF

END FUNCTION calc_line

END MODULE geometry
```

A test driver program for this function is:

```
PROGRAM test_calc_line
!
!  Purpose:
!    To test function calc_line%
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/18/2007    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: ', calc_line(p1,p2)

END PROGRAM test_calc_line
```

When this program is executed, the results are.

```
C:\book\f95_2003\soln>test_calc_line
Enter first point:
0 0
Enter second point:
3 4
The result is:        1.333333    0.000000E+00

C:\book\f95_2003\soln\ex12_8>test_calc_line
Enter first point:
0 6
Enter second point:
6 0
The result is:       -1.000000        6.000000

C:\book\f95_2003\soln\ex12_8>test_calc_line
Enter first point:
3 4
Enter second point:
3 4
The result is:    0.000000E+00    0.000000E+00
```

This function appears to be working correctly.

12-9 The conversion between polar and rectangular coordinates in this exercise is especially tricky. On a compass, 0º is due North, and angles increase in a clockwise direction. In addition, angles are measured in degrees instead of radians. As a result, the rectangular-polar conversions differ from those in an ordinary cartesian plane. The correct equations are:

$$x = r \sin\theta$$

$$x = r \cos\theta$$

$$r = \sqrt{x^2 + y^2}$$

$$\theta = 90° - \mathrm{ATAN2}(y, x)$$

A program to implement the radar tracker is shown below. Note that this program prints out both estimate position and velocity data.

```
MODULE track_data
!
!  Purpose:
!    To define track file data structures.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Create a derived data type for the raw (range, angle, time)
! target observations.
TYPE :: polar_obs             ! Polar observation
   REAL :: time               ! Time of state vector (s)
   REAL :: r                  ! Range of measurement (m)
   REAL :: theta              ! Compass angle of measurement (deg)
END TYPE polar_obs

! Create a derived dat type for the converted (x, y, time)
! target observations.
TYPE :: rect_obs              ! Rectangular observation
   REAL :: time               ! Time of state vector (s)
   REAL :: x                  ! X (East-West) measurement (m)
   REAL :: y                  ! Y (North-South) measurement (m)
END TYPE rect_obs

! Create a derived data type for the smoothed target track file.
TYPE :: track_file
   REAL :: time               ! Time of state vector (s)
   REAL :: x                  ! Smoothed X (E-W) position (m)
   REAL :: y                  ! Smoothed Y (N-S) position (m)
   REAL :: vx                 ! Estimated X velocity (m/s)
   REAL :: vy                 ! Estimated Y velocity (m/s)
END TYPE track_file

END MODULE track_data


PROGRAM tracker
!
! Purpose:
!   To implement a radar tracker.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
USE track_data
IMPLICIT NONE

! Declare named constants:
INTEGER,PARAMETER :: MAXSIZ = 10000  ! Up to 10000 observations
```

```
! Declare variables:
CHARACTER(len=30) :: filename        ! Input data file name
INTEGER :: i                         ! Loop index
INTEGER :: istat                     ! I/o status
INTEGER :: n_obs = 0                 ! Number of observations
REAL :: maxplt                       ! Dummy argument
REAL :: minplt                       ! Dummy argument
TYPE (polar_obs) :: ob               ! Target observation (polar)
TYPE (rect_obs) :: rect              ! Target observation (rect)
TYPE (track_file) :: track           ! Smoothed target track

REAL,DIMENSION(MAXSIZ) :: time = 0.   ! Array of times (s)
REAL,DIMENSION(MAXSIZ) :: x_obs = 0.  ! Measured x component (m)
REAL,DIMENSION(MAXSIZ) :: x_pred = 0. ! Predicted x component (m)
REAL,DIMENSION(MAXSIZ) :: x_track = 0.! Tracker x component (m)
REAL,DIMENSION(MAXSIZ) :: x_vel = 0.  ! Tracker x velocity (m/s)
REAL,DIMENSION(MAXSIZ) :: y_obs = 0.  ! Measured y component (m)
REAL,DIMENSION(MAXSIZ) :: y_pred = 0. ! Predicted y component (m)
REAL,DIMENSION(MAXSIZ) :: y_track = 0.! Tracker y component (m)
REAL,DIMENSION(MAXSIZ) :: y_vel = 0.  ! Tracker y velocity (m/s)

! Get input file name:
WRITE (*,*) 'This program implements an alpha-beta radar tracker.'
WRITE (*,*) 'Enter file name for input data: '
READ (*,'(A)') filename

! Open input file.
OPEN(UNIT=8,FILE=filename,STATUS='OLD',ACTION='READ',IOSTAT=istat)
open_ok: IF ( istat == 0 ) THEN

   ! Read first line of data and initialize track file.
   READ (8,*,IOSTAT=istat) ob%time, ob%r, ob%theta
   n_obs = n_obs + 1

   ! If read is successful, initialize tracker.
   read1_ok: IF ( istat == 0 ) THEN

      ! Convert measurement to rectangular form
      rect%time = ob%time
      CALL polar_2_rect ( ob%r, ob%theta, rect%x, rect%y )

      ! Initialize tracker.
      track%time = rect%time          ! Initial time
      track%x = rect%x                ! Initial x pos.
      track%y = rect%y                ! Initial y pos.
      track%vx = 0.                   ! Initial velocity guess = 0.
      track%vy = 0.                   ! Initial velocity guess = 0.

      ! Save (x,y) results
      time(n_obs) = track%time
      x_obs(n_obs) = rect%x
      x_track(n_obs) = track%x
      x_vel(n_obs) = track%vx
      y_obs(n_obs) = rect%y
      y_track(n_obs) = track%y
```

```
      y_vel(n_obs) = track%vy

      ! Now process all measurements as long as data is available.
      loop: DO

         ! Get measurement
         READ (8,*,IOSTAT=istat) ob%time, ob%r, ob%theta
         IF ( istat /= 0 ) EXIT
         n_obs = n_obs + 1

         ! Convert to rectangular form
         rect%time = ob%time
         CALL polar_2_rect ( ob%r, ob%theta, rect%x, rect%y )

         ! Filter data
         CALL filter ( rect, track, x_pred(n_obs), y_pred(n_obs) )

         ! Save (x,y) results
         time(n_obs) = track%time
         x_obs(n_obs) = rect%x
         x_track(n_obs) = track%x
         x_vel(n_obs) = track%vx
         y_obs(n_obs) = rect%y
         y_track(n_obs) = track%y
         y_vel(n_obs) = track%vy


      END DO loop

   END IF read1_ok

   ! Now print out the position information.
   WRITE (*,1000)
   1000 FORMAT (/T6,'Time',T15,'X_obs',T26,'X_pred',T36,'X_track', &
                        T48,'Y_obs',T59,'Y_pred',T69,'Y_track')
   DO i = 1, n_obs
      WRITE (*,1010) time(i), x_obs(i), x_pred(i), x_track(i), &
                     y_obs(i), y_pred(i), y_track(i)
      1010 FORMAT (3X,F6.1,6(2X,F9.1))
   END DO

   ! Now print out the velocity information.
   WRITE (*,1020)
   1020 FORMAT (/T6,'Time',T15,'X_vel',T26,'Y_vel')
   DO i = 1, n_obs
      WRITE (*,1030) time(i), x_vel(i), y_vel(i)
      1030 FORMAT (3X,F6.1,2(2X,F9.1))
   END DO

ELSE open_ok

   WRITE (*,'(A,I6)') ' File open failed: IOSTAT = ', istat

END IF open_ok

END PROGRAM tracker
```

```
SUBROUTINE polar_2_rect ( r, theta, x, y )
!
! Purpose:
!   To convert polar (range,comapss angle) measurements into
!   rectangluar map (E,N) measurements.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
REAL,INTENT(IN) :: r             ! Range (m)
REAL,INTENT(IN) :: theta         ! Compass angle (degrees)
REAL,INTENT(OUT) :: x            ! x-component (m)
REAL,INTENT(OUT) :: y            ! y-component (m)

! Declare named constants:
REAL,PARAMETER :: DEG_2_RAD = .017453293  ! Degrees to radians

! Note that the input measurements are in range in meters and
! compass angle in degrees clockwise relative to North.  The x-y
! coordinate system is laid out with x to the East and y to the North,
! so the conversion between (r,theta) and (x,y) is as shown:
x = r * SIN ( theta * DEG_2_RAD )
y = r * COS ( theta * DEG_2_RAD )

END SUBROUTINE polar_2_rect


SUBROUTINE rect_2_polar ( x, y, r, theta )
!
! Purpose:
!   To convert rectangluar map (E,N) measurements into polar
!  (range,comapss angle) measurements.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
REAL,INTENT(IN) :: x             ! x-component (m)
REAL,INTENT(IN) :: y             ! y-component (m)
REAL,INTENT(OUT) :: r            ! Range (m)
REAL,INTENT(OUT) :: theta        ! Compass angle (degrees)

! Declare named constants:
REAL,PARAMETER :: RAD_2_DEG = 57.2957795  ! Radians to degrees
```

```
! Note that the x-y coordinate system is laid out with x to the East
! and y to the North, while the polar measurements are range in
! meters and compass angle in degrees clockwise relative to North.
! Therefore, the conversion between (x,y) and (r,theta) is as shown:
r = SQRT( x**2 + y**2 )
theta = 90. - RAD_2_DEG * ATAN2( y, x )

END SUBROUTINE rect_2_polar


SUBROUTINE filter ( rect, track, x_pred, y_pred )
!
! Purpose:
!   To apply the alpha-beta filter to a new measurement, and
!   update the estimated position and velocity of a target.
!   The predicted target position x_pred and y_pred is also
!   returned so that I can be printed out.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
USE track_data
IMPLICIT NONE

TYPE (rect_obs),INTENT(IN) :: rect       ! Rectangular measurement
TYPE (track_file),INTENT(INOUT) :: track ! Smoothed track
REAL,INTENT(INOUT) :: x_pred             ! Predicted X position
REAL,INTENT(INOUT) :: y_pred             ! Predicted Y position

! List of named constants:
REAL,PARAMETER :: alpha = 0.55           ! Filter weights
REAL,PARAMETER :: beta = 0.38            ! Filter weights

! List of local variables:
REAL :: dt                 ! Delta time since last measurement

! Calculate time difference in seconds since last measurement:
dt = rect%time - track%time

! Update track file time
track%time = rect%time

! Predict target position to the new time using Eqns 8-25:
x_pred = track%x + track%vx * dt       ! X prediction
y_pred = track%y + track%vy * dt       ! Y prediction

! Update position with new measurement using Eqns 8-23:
track%x = x_pred + alpha * ( rect%x - x_pred )
track%y = y_pred + alpha * ( rect%y - y_pred )

! Update velocity with new measurement using Eqns 8-24:
track%vx = track%vx + ( beta / dt ) * ( rect%x - x_pred )
track%vy = track%vy + ( beta / dt ) * ( rect%y - y_pred )
```

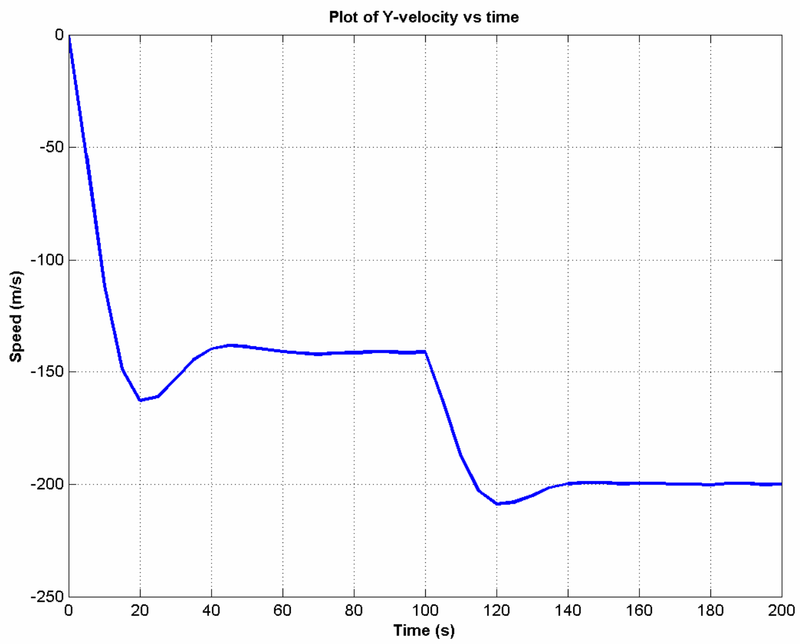
```
END SUBROUTINE filter
```

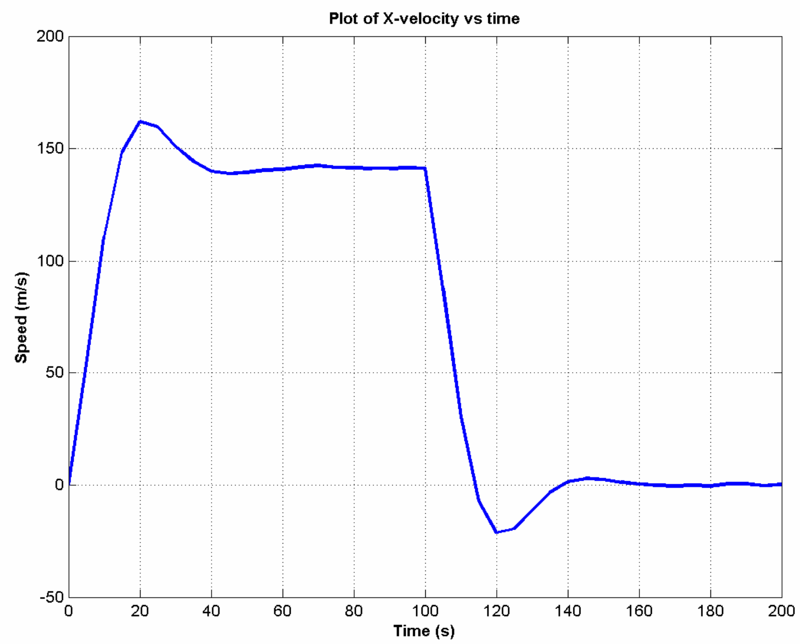
When this program is executed with the noise-free data set in file **track1.dat**, the results are:

```
C:\book\f95_2003\soln\ex12_9>tracker
This program implements an alpha-beta radar tracker.
Enter file name for input data:
track1.dat

     Time     X_obs      X_pred     X_track     Y_obs      Y_pred     Y_track
      0.0   -4987.5        0.0    -4987.5    20003.6        0.0    20003.6
      5.0   -4277.9    -4987.5    -4597.2    19296.5    20003.6    19614.7
     10.0   -3579.4    -4327.6    -3916.1    18587.5    19346.0    18928.8
     15.0   -2864.1    -3362.1    -3088.2    17881.1    18371.9    18101.9
     20.0   -2169.4    -2345.0    -2248.4    17172.5    17358.5    17256.2
     25.0   -1469.4    -1438.5    -1455.5    16464.6    16442.1    16454.4
     30.0    -770.7     -657.3     -719.7    15757.2    15648.9    15708.4
     35.0     -52.5       35.5      -12.9    15050.9    14944.0    15002.8
     40.0     651.4      708.7      677.2    14344.2    14279.0    14314.9
     45.0    1361.2     1377.1     1368.3    13637.2    13615.9    13627.6
     50.0    2071.1     2062.2     2067.1    12930.2    12936.7    12933.1
     55.0    2776.9     2764.3     2771.2    12222.5    12239.7    12230.3
     60.0    3477.3     3473.2     3475.5    11517.5    11530.4    11523.3
     65.0    4192.3     4179.1     4186.4    10808.4    10818.5    10813.0
     70.0    4904.0     4895.0     4899.9    10099.3    10104.3    10101.6
     75.0    5601.7     5612.0     5606.3     9397.0     9391.1     9394.3
     80.0    6312.2     6314.5     6313.2     8688.0     8686.1     8687.2
     85.0    7014.5     7020.5     7017.2     7984.4     7979.6     7982.3
     90.0    7722.4     7722.2     7722.3     7277.2     7276.6     7276.9
     95.0    8433.7     8427.4     8430.9     6565.5     6571.5     6568.2
    100.0    9136.3     9138.4     9137.2     5865.4     5860.4     5863.2
    105.0    9140.3     9843.9     9456.9     4860.0     5157.3     4993.8
    110.0    9140.0     9896.2     9480.3     3860.9     4174.9     4002.2
    115.0    9138.8     9632.2     9360.8     2863.9     3064.0     2954.0
    120.0    9139.8     9325.3     9223.2     1859.5     1939.7     1895.6
    125.0    9139.3     9117.2     9129.3      863.9      850.9      858.1
    130.0    9139.9     9031.6     9091.2     -143.6     -181.7     -160.7
    135.0    9140.4     9034.6     9092.8    -1138.5    -1186.0    -1159.9
    140.0    9139.0     9076.4     9110.8    -2143.5    -2167.1    -2154.1
    145.0    9137.5     9118.2     9128.8    -3146.3    -3152.4    -3149.0
    150.0    9137.8     9143.6     9140.4    -4145.1    -4145.0    -4145.0
    155.0    9135.5     9152.9     9143.4    -5147.6    -5141.0    -5144.7
    160.0    9139.3     9149.3     9143.8    -6141.3    -6143.1    -6142.1
    165.0    9139.3     9145.9     9142.3    -7140.4    -7139.9    -7140.2
    170.0    9138.1     9141.9     9139.8    -8141.8    -8138.2    -8140.2
    175.0    9140.1     9138.0     9139.1    -9140.1    -9139.5    -9139.8
    180.0    9134.3     9138.1     9136.0   -10144.7   -10139.4   -10142.3
    185.0    9146.5     9133.5     9140.6   -11135.1   -11143.9   -11139.0
    190.0    9145.2     9143.1     9144.2   -12136.1   -12137.3   -12136.6
    195.0    9134.8     9147.5     9140.5   -13143.3   -13134.4   -13139.3
    200.0    9145.4     9139.0     9142.5   -14136.7   -14140.5   -14138.4

    Time     X_vel      Y_vel
     0.0       0.0        0.0
     5.0      53.9      -53.7
    10.0     110.8     -111.4
```

15.0	148.6	-148.7
20.0	162.0	-162.8
25.0	159.6	-161.1
30.0	151.0	-152.9
35.0	144.3	-144.8
40.0	140.0	-139.8
45.0	138.8	-138.2
50.0	139.4	-138.7
55.0	140.4	-140.0
60.0	140.7	-141.0
65.0	141.7	-141.7
70.0	142.4	-142.1
75.0	141.6	-141.7
80.0	141.5	-141.5
85.0	141.0	-141.1
90.0	141.0	-141.1
95.0	141.5	-141.5
100.0	141.3	-141.2
105.0	87.9	-163.8
110.0	30.4	-187.6
115.0	-7.1	-202.8
120.0	-21.2	-208.9
125.0	-19.5	-208.0
130.0	-11.3	-205.1
135.0	-3.3	-201.4
140.0	1.5	-199.7
145.0	2.9	-199.2
150.0	2.5	-199.2
155.0	1.2	-199.7
160.0	0.4	-199.6
165.0	-0.1	-199.6
170.0	-0.4	-199.9
175.0	-0.2	-199.9
180.0	-0.5	-200.3
185.0	0.5	-199.6
190.0	0.6	-199.6
195.0	-0.3	-200.2
200.0	0.2	-199.9

Plots of the predicted x- and y-velocity vs time are shown below:





When this program is executed with the noist data set in file **track2.dat**, the results are as shown below.

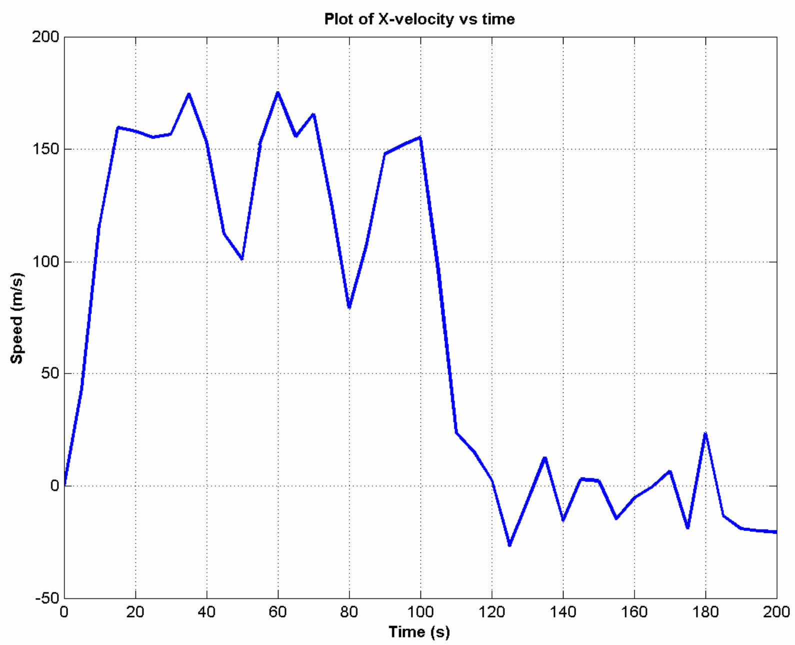
```
C:\book\f95_2003\soln\ex12_9>tracker
This program implements an alpha-beta radar tracker.
Enter file name for input data:
track2.dat

    Time      X_obs      X_pred     X_track     Y_obs      Y_pred     Y_track
      .0    -5004.6          .0    -5004.6    20529.8          .0    20529.8
     5.0    -4434.8     -5004.6    -4691.2    18907.9     20529.8    19637.7
    10.0    -3509.1     -4474.7    -3943.6    18750.5     19021.4    18872.4
```

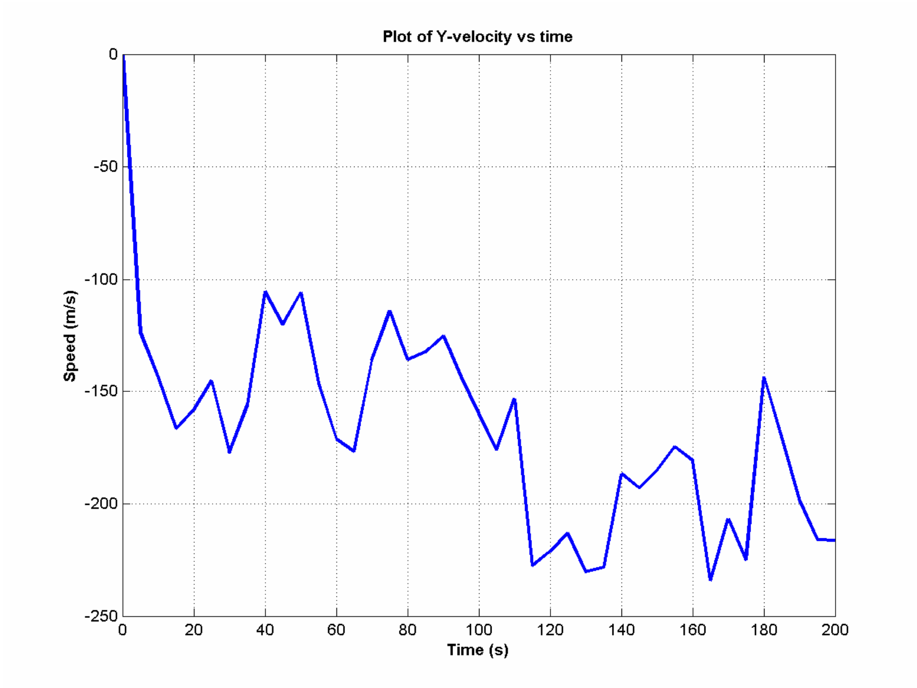
15.0	-2795.8	-3360.1	-3049.7	17853.4	18153.1	17988.3
20.0	-2273.4	-2251.8	-2263.7	17268.0	17155.1	17217.2
25.0	-1510.4	-1473.9	-1494.0	16596.4	16426.9	16520.1
30.0	-698.0	-718.1	-707.0	15370.2	15794.3	15561.0
35.0	313.2	76.5	206.7	14953.7	14674.0	14827.8
40.0	797.0	1080.2	924.4	14716.4	14047.1	14415.2
45.0	1150.0	1690.3	1393.1	13694.8	13888.9	13782.1
50.0	1807.6	1953.7	1873.3	13369.4	13182.0	13285.0
55.0	3047.7	2378.4	2746.5	12223.8	12756.1	12463.3
60.0	3815.0	3505.9	3675.9	11401.7	11732.1	11550.4
65.0	4291.5	4552.7	4409.0	10621.8	10693.6	10654.1
70.0	5319.9	5186.6	5259.9	10307.1	9770.1	10065.4
75.0	5564.6	6088.1	5800.2	9677.2	9385.4	9545.9
80.0	5816.6	6429.5	6092.4	8688.8	8976.8	8818.4
85.0	6866.9	6488.8	6696.8	8183.7	8139.8	8163.9
90.0	7759.9	7236.8	7524.5	7599.0	7502.0	7555.4
95.0	8317.1	8263.3	8292.9	6687.1	6930.4	6796.6
100.0	9097.5	9052.2	9077.1	5862.9	6079.1	5960.2
105.0	9077.5	9853.5	9426.7	4949.2	5160.6	5044.3
110.0	8953.8	9908.3	9383.3	4464.2	4164.4	4329.3
115.0	9389.3	9502.2	9440.1	2586.3	3563.3	3025.9
120.0	9348.7	9516.1	9424.0	1970.1	1888.6	1933.4
125.0	9049.8	9436.4	9223.8	935.2	827.1	886.6
130.0	9346.1	9089.2	9230.5	-408.1	-178.7	-304.8
135.0	9457.5	9193.6	9338.7	-1430.3	-1457.3	-1442.4
140.0	9030.5	9402.1	9197.7	-2035.1	-2584.6	-2282.4
145.0	9362.4	9119.9	9253.3	-3297.0	-3215.7	-3260.5
150.0	9260.6	9267.6	9263.8	-4123.1	-4224.7	-4168.8
155.0	9051.3	9275.4	9152.1	-4955.4	-5094.5	-5018.0
160.0	9199.4	9078.6	9145.0	-5974.1	-5890.8	-5936.6
165.0	9183.9	9117.4	9154.0	-7543.8	-6841.1	-7227.6
170.0	9245.2	9151.7	9203.1	-8036.7	-8399.1	-8199.8
175.0	8897.1	9236.3	9049.7	-9474.4	-9233.6	-9366.0
180.0	9516.9	8954.0	9263.6	-9417.8	-10491.3	-9900.9
185.0	8894.8	9381.8	9113.9	-10984.2	-10618.2	-10819.5
190.0	8972.1	9047.1	9005.9	-12037.1	-11675.9	-11874.5
195.0	8898.3	8910.5	8903.8	-13093.5	-12868.2	-12992.1
200.0	8796.7	8803.9	8799.9	-14077.6	-14071.4	-14074.8

Time	X_vel	Y_vel
0.0	0.0	0.0
5.0	43.3	-123.3
10.0	116.7	-143.9
15.0	159.6	-166.6
20.0	157.9	-158.1
25.0	155.2	-145.2
30.0	156.7	-177.4
35.0	174.7	-156.1
40.0	153.2	-105.3
45.0	112.1	-120.0
50.0	101.0	-105.8
55.0	151.9	-146.2
60.0	175.4	-171.4
65.0	155.5	-176.8
70.0	165.6	-136.0

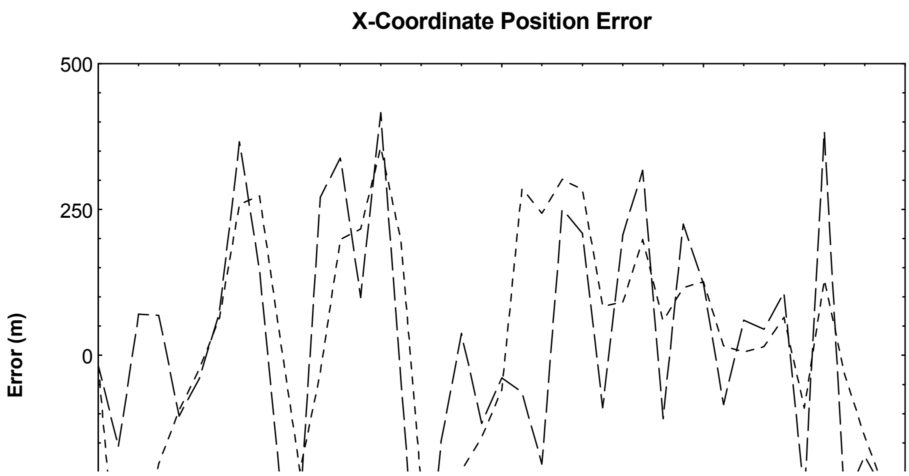
75.0	125.9	-113.8
80.0	79.3	-135.7
85.0	108.0	-132.4
90.0	147.8	-125.0
95.0	151.9	-143.5
100.0	155.3	-159.9
105.0	96.3	-176.0
110.0	23.8	-153.2
115.0	15.2	-227.5
120.0	2.5	-221.3
125.0	-26.9	-213.0
130.0	-7.4	-230.5
135.0	12.7	-228.4
140.0	-15.6	-186.7
145.0	2.9	-192.9
150.0	2.3	-185.1
155.0	-14.7	-174.6
160.0	-5.5	-180.9
165.0	-0.5	-234.3
170.0	6.6	-206.8
175.0	-19.1	-225.1
180.0	23.6	-143.5
185.0	-13.4	-171.3
190.0	-19.1	-198.7
195.0	-20.0	-215.9
200.0	-20.5	-216.3

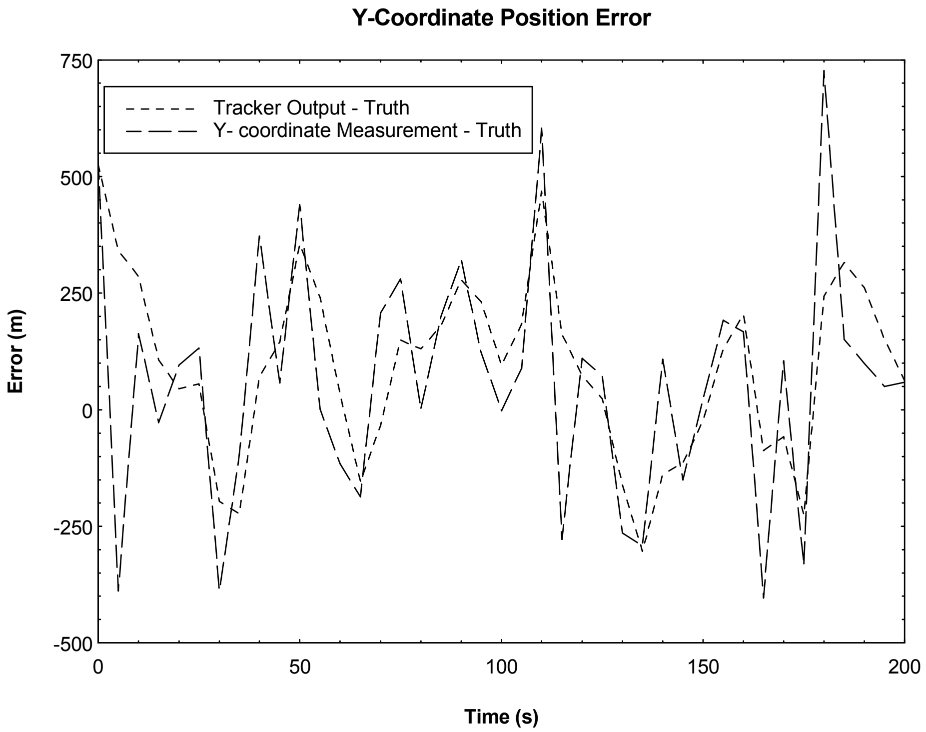
Plots of the predicted *x*- and *y*-velocity vs time are shown below. As you can see, the velocity estimate is much coarser if the input data is noisy.





The effect of the tracker is to smooth the estimated position of the target, eliminating some of the effects of measurement noise. This is shown by the following two figures, which compares the difference between the measured and true posistions to the difference between the tracker and true positions. As you can see, the tracker smooths out some of the wilder excursions.




Y-Coordinate Position Error
Tracker Output - Truth
Y- coordinate Measurement - Truth
750
500
250
0
-250
-500
Error (m)
0
50
100
150
200
Time (s)

Chapter 13. Advanced Features of Procedures and Modules

13-1 A version of function `lt_city` that contains an internal function to shift the strings to uppercase temporarily for comparison is shown below.

```
LOGICAL FUNCTION lt_city (a, b)
!
!  Purpose:
!    To compare variables "a" and "b" and determine which
!    has the smaller city (lower alphabetical order).
!
USE types                    ! Declare the module types
IMPLICIT NONE

! Data dictionary: declare calling parameter types & definitions
TYPE (personal_info), INTENT(IN) :: a, b

! Make comparison.
lt_city = compare(a%city, b%city)

CONTAINS

   LOGICAL FUNCTION compare ( str1, str2 )
   !
   !  Purpose:
   !    To shift a character string to upper case on any processor,
   !    regardless of collating sequence.
   !
   !  Record of revisions:
   !      Date       Programmer          Description of change
   !      ====       ==========          =====================
   !    05/09/2007   S. J. Chapman       Original code
   !
   IMPLICIT NONE

   ! Declare calling parameters:
   CHARACTER(len=*), INTENT(IN) :: str1
   CHARACTER(len=*), INTENT(IN) :: str2

   ! Declare local variables:
   CHARACTER(len=LEN(str1)) :: s1   ! Local variables for uppercase
   CHARACTER(len=LEN(str2)) :: s2   ! Local variables for uppercase
   INTEGER :: i                     ! Loop index
   INTEGER :: length                ! Length of input string

   ! Get length of string
   s1 = str1
   length = LEN ( s1 )
```

```
   ! Now shift lower case letters to upper case.
   DO i = 1, length
      IF ( LGE(s1(i:i),'a') .AND. LLE(s1(i:i),'z') ) THEN
         s1(i:i) = ACHAR ( IACHAR ( s1(i:i) ) - 32 )
      END IF
   END DO

   ! Get length of string
   s2 = str2
   length = LEN ( s2 )

   ! Now shift lower case letters to upper case.
   DO i = 1, length
      IF ( LGE(s2(i:i),'a') .AND. LLE(s2(i:i),'z') ) THEN
         s2(i:i) = ACHAR ( IACHAR ( s2(i:i) ) - 32 )
      END IF
   END DO

   ! Do comparison
   compare = s1 < s2

   END FUNCTION compare

END FUNCTION lt_city
```

13-2 A program to test subroutine `factorial` and function `fact` is shown below:

```
PROGRAM test_factorial
!
!  Purpose:
!    To test subroutine "factorial" and function "fact".
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!   05/18/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of external functions:
INTEGER,EXTERNAL :: fact       ! Factiroal function

! List of variables:
INTEGER :: n                   ! Value to calculate
INTEGER :: result              ! Result

! Get the value to calculate the factorial of.
WRITE (*,*) 'Enter a non-negative integer: '
READ (*,*) n

! Calculate factorial with subroutine
CALL factorial ( n, result )

! Write results for both procedures:
WRITE (*,'(1X,I3,A,I6)') n, '! = ', result
```

```
WRITE (*,'(1X,I3,A,I6)') n, '! = ', fact(n)

END PROGRAM test_factorial
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex13_2>test_factorial
 Enter a non-negative integer:
7
   7! =   5040
   7! =   5040
```

13-3 A program to test subroutine `extremes` is shown below. Note that this program tests both keyword arguments and optional arguments:

```
MODULE procs

CONTAINS
   SUBROUTINE extremes(a, n, maxval, pos_maxval, minval, pos_minval)
   !
   !  Purpose:
   !    To find the maximum and minimum values in an array, and
   !    the location of those values in the array.  This subroutine
   !    returns its output values in optional arguments.
   !
   !  Record of revisions:
   !      Date       Programmer          Description of change
   !      ====       ==========          =====================
   !   05/18/2007    S. J. Chapman       Original code
   !
   IMPLICIT NONE

   ! Data dictionary: declare calling parameter types & definitions
   INTEGER, INTENT(IN) :: n                     ! # vals in array a
   REAL, INTENT(IN), DIMENSION(n) :: a          ! Input data.
   REAL, INTENT(OUT), OPTIONAL :: maxval        ! Maximum value.
   INTEGER, INTENT(OUT), OPTIONAL :: pos_maxval ! Pos of maxval
   REAL, INTENT(OUT), OPTIONAL :: minval        ! Minimum value.
   INTEGER, INTENT(OUT), OPTIONAL :: pos_minval ! Pos of minval

   ! Data dictionary: declare local variable types & definitions
   INTEGER :: i                            ! Index
   REAL :: real_max                        ! Max value
   INTEGER :: pos_max                      ! Pos of max value
   REAL :: real_min                        ! Min value
   INTEGER :: pos_min                      ! Pos of min value

   ! Initialize the values to first value in array.
   real_max = a(1)
   pos_max = 1
   real_min = a(1)
   pos_min = 1

   ! Find the extreme values in a(2) through a(n).
   DO i = 2, n
      max: IF ( a(i) > real_max ) THEN
```

```
         real_max = a(i)
         pos_max = i
      END IF max
      min: IF ( a(i) < real_min ) THEN
         real_min = a(i)
         pos_min = i
      END IF min
   END DO

   ! Report the results
   IF ( PRESENT(maxval) ) THEN
      maxval = real_max
   END IF
   IF ( PRESENT(pos_maxval) ) THEN
      pos_maxval = pos_max
   END IF
   IF ( PRESENT(minval) ) THEN
      minval = real_min
   END IF
   IF ( PRESENT(pos_minval) ) THEN
      pos_minval = pos_min
   END IF

   END SUBROUTINE extremes
END MODULE procs

PROGRAM test_extremes
!
!  Purpose:
!    To read in a real input data set, and use it to test subroutine
!    "extremes".  The optional arguments feature of the subroutine
!    will be tested by calling the subroutine 3 times with different
!    combinations of arguments.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!   05/18/2007    S. J. Chapman       Original code
!
USE procs
IMPLICIT NONE

! List of parameters:
INTEGER, PARAMETER :: MAX_SIZE = 1000

! List of variables:
REAL, DIMENSION(MAX_SIZE) :: a   ! Data array to sort
LOGICAL :: exceed = .FALSE.      ! Logical indicating that array
                                 ! limits are exceeded.
CHARACTER(len=20) :: filename    ! Input data file name
REAL :: large                    ! Largest value in a
INTEGER :: large_pos             ! Pos of largest value in a
INTEGER :: nvals = 0             ! Number of data values in a
REAL :: small                    ! Smallest value in a
INTEGER :: small_pos             ! Pos of smallest value in a
INTEGER :: status                ! I/O status: 0 for success
```

```
REAL :: temp                     ! Temporary variable

! Get the name of the file containing the input data.
WRITE (*,*) 'Enter the file name with input data set: '
READ (*,'(A20)') filename

! Open input data file.  Status is OLD because the input data must
! already exist.
OPEN ( UNIT=2, FILE=filename, STATUS='OLD', IOSTAT=status )

! Was the OPEN successful?
fileopen: IF ( status == 0 ) THEN       ! Open successful

   ! The file was opened successfully, so read the data to sort
   ! from it, sort the data, and write out the results.
   ! First read in data.
   DO
      READ (2, *, IOSTAT=status) temp      ! Get value
      IF ( status /= 0 ) EXIT              ! Exit on end of data
      nvals = nvals + 1                    ! Bump count
      size: IF ( nvals <= MAX_SIZE ) THEN  ! Too many values?
         a(nvals) = temp                   ! No: Save value in array
      ELSE
         exceed = .TRUE.                   ! Yes: Array overflow
      END IF size
   END DO

   ! Was the array size exceeded?  If so, tell user and quit.
   toobig: IF ( exceed ) THEN
      WRITE (*,1000) nvals, MAX_SIZE
      1000 FORMAT (' Maximum array size exceeded: ', I6, ' > ', I6 )
   ELSE

      ! Limit not exceeded.  Find extremes specifying all arguments
      ! in order.  Tell user.
      CALL extremes (a, nvals, large, large_pos, small, small_pos)
      WRITE (*,1020) 'All arguments in order:           ', &
                     large, large_pos, small, small_pos
      1020 FORMAT (1X,A,2(2X,F6.2,2X,I4))

      ! Find extremes specifying all arguments in arbitrary
      ! order.  Tell user.
      CALL extremes (a, nvals, MAXVAL=large, MINVAL=small, &
                     POS_MAXVAL=large_pos, POS_MINVAL=small_pos)
      WRITE (*,1020) 'All arguments in arbitrary order: ', &
                     large, large_pos, small, small_pos

      ! Find extremes specifying only max and min values.
      CALL extremes (a, nvals, MAXVAL=large, MINVAL=small)
      WRITE (*,1030) 'Large and small only:             ', &
                     large, small
      1030 FORMAT (1X,A,2(2X,F6.2,6X))

   END IF toobig

ELSE fileopen
```

```
   ! If we get here, the file open failed.  Tell user.
   WRITE (*,'(A,I6)') ' File open failed: status = ', status

END IF fileopen

END PROGRAM test_extremes
```

A data set used to exercise this program is found in file `in13_3.dat`:

```
1.
34.
-21.1
0.2
-.04
0.
5.
2.2
-17.2
-11.1
4.4
0.12
```

When this program is executed using the data in file `in13_3.dat`, the results are:

```
C:\book\f95_2003\soln\ex13_3>test_extremes
Enter the file name with input data set:
in13_3.dat
All arguments in order:              34.00     2  -21.10     3
All arguments in arbitrary order:    34.00     2  -21.10     3
Large and small only:                34.00        -21.10
```

13-4 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\f95_2003\soln\ex13_4>exercise13_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
```

13-5 The program is valid. Variable `b` in the subroutine is inherited by host association, so its value is 4.0. The value if `output` is 0.75.

13-6 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 95/2003 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.

13-7 A scoping unit is the portion of a Fortran program over which a local object is defined. The scoping units in a Fortran 95/2003 program are:

1. A **main program**, **internal** or **external procedure**, or **module**, excluding any derived type definitions or procedures contained within it.

2. A **derived type definition**.

3. An **interface definition**.

13-8 A keyword argument is an argument of the form

```
keyword = actual_argument
```

where `keyword` is the name of the dummy argument which is being associated with the actual argument. If the procedure invocation uses keyword arguments, then the calling arguments can be arranged in any order, because the keywords allow the compiler to sort out which actual argument goes with which dummy argument. Keyword arguments can only be used if a procedure has an explicit interface.

13-9 *(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)`. *(b)* This statement is illegal. Dummy argument `b` has `INTENT(OUT)`, but the corresponding actual argument is a constant. *(c)* This statement is illegal. Dummy argument `d` is not optional, and is missing in the `CALL` statement. *(d)* This statement is legal. The two optional arguments are missing, and the non-optional argument following the first missing argument uses a keyword. However, *p* and *r* should be initialized before the `CALL` statement, since they correspond to dummy arguments with `INTENT(IN)`. *(e)* This statement is illegal. Dummy argument `b` is a non-keyword argument after a keyword argument, which is not allowed. *(f)* This statement is legal. It uses keyword arguments and specifies the arguments in arbitrary order. However, *p*, *r*, *s*, and *t* should be initialized before the `CALL` statement, since they correspond to dummy arguments with `INTENT(IN)`.

13-10 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.

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

13-11 An explicit interface for subroutine `simul` from Example 9-1 is given below:

```
INTERFACE
   SUBROUTINE simul ( a, b, ndim, n, error )
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: ndim
   REAL, INTENT(INOUT), DIMENSION(ndim,ndim) :: a
   REAL, INTENT(INOUT), DIMENSION(ndim) :: b
   INTEGER, INTENT(IN) :: n
   INTEGER, INTENT(OUT) :: error
   END SUBROUTINE simul
END INTERFACE
```

13-12 A generic procedure is a procedure that is deigned 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 95/2003 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 either a complete description of the input and output arguments of the procedure, or 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.

13-13 To define a generic bound procedure, add a `GENERIC` statement to the derived data type. The `GENERIC` statement declares the name of the generic function, plus the names of all the specific function associated with it.

```
TYPE :: my_type
   component 1
   component 2
   ...
CONTAINS
   EXTERNAL generic_proc => specific_proc_1, specific_proc_2
END INTERFACE
```

13-14 An interface block for generic subroutine `simul` is shown below:

```
INTERFACE simul

   SUBROUTINE simul2 ( a, b, soln, ndim, n, error )
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: ndim
   REAL, INTENT(IN), DIMENSION(ndim,ndim) :: a
   REAL, INTENT(IN), DIMENSION(ndim) :: b
   REAL, INTENT(OUT), DIMENSION(ndim) :: soln
   INTEGER, INTENT(IN) :: n
   INTEGER, INTENT(OUT) :: error
   END SUBROUTINE simul2

   SUBROUTINE dsimul ( a, b, soln, ndim, n, error )
   IMPLICIT NONE
```

```
   INTEGER, PARAMETER :: dbl = SELECTED_REAL_KIND(p=13)
   INTEGER, INTENT(IN) :: ndim
   REAL(KIND=dbl), INTENT(IN), DIMENSION(ndim,ndim) :: a
   REAL(KIND=dbl), INTENT(IN), DIMENSION(ndim) :: b
   REAL(KIND=dbl), INTENT(OUT), DIMENSION(ndim) :: soln
   INTEGER, INTENT(IN) :: n
   INTEGER, INTENT(OUT) :: error
   END SUBROUTINE dsimul

   SUBROUTINE csimul ( a, b, soln, ndim, n, error )
   IMPLICIT NONE
   INTEGER, PARAMETER :: sgl = SELECTED_REAL_KIND(p=6)
   INTEGER, INTENT(IN) :: ndim
   COMPLEX(KIND=sgl), INTENT(IN), DIMENSION(ndim,ndim) :: a
   COMPLEX(KIND=sgl), INTENT(IN), DIMENSION(ndim) :: b
   COMPLEX(KIND=sgl), INTENT(OUT), DIMENSION(ndim) :: soln
   INTEGER, INTENT(IN) :: n
   INTEGER, INTENT(OUT) :: error
   END SUBROUTINE csimul

END INTERFACE
```

13-15 *(a)* This generic interface block is illegal, since the two subroutines cannot be distinguished by the type and sequence of their *non-optional* arguments. *(b)* This generic interface block is legal.

13-16 A new operator can be defined using an **interface operator block**. The name of the new operator can be any sequence of up to 31 characters, surrounded by periods. The actions to be performed by the new operator are specified by writing one or more functions describing the relationships between the operands and the function's resulting value. If the operator is a unary operator, then the corresponding functions must have only one argument. If the operator is a binary operator, then the corresponding functions should have two arguments. The first argument will correspond to the operand on the left-hand side of the operator, and the second argument will correspond to the operand on the right-hand side of the operator.

Once the function(s) to implement the operator are written, the operator is declared in an interface operator block of the form

```
INTERFACE OPERATOR (operator_symbol)
    MODULE PROCEDURE function_1
    ...
END INTERFACE
```

where `operator_symbol` is the symbol of the new operator, and the defining functions are specified in the interface body.

13-17 An existing intrinsic operator (+, -, *, /, **, *etc.*) can be extended to work with derived data types using an interface operator block. If the meaning of an intrinsic operator is being extended, then the following three constraints must be satisfied:

1. It is not possible to change the meaning of an intrinsic operator for pre-defined intrinsic data types. It is only possible to *extend* the meaning of the operator by defining the actions to perform when the operator is applied to derived data types, or combinations of derived data types and intrinsic data types.
2. The number of arguments in a function must be consistent with the normal use of the operator. For example, multiplication (*) is a binary operator, so any function extending its meaning must have two arguments.
3. If a relational operator is extended, then the same extension applies regardless of which way the operator is written. For example, if the relational operator “greater than” is given an additional meaning, then the extension applies whether “greater than” is written as > or `.GT.`

13-18 The assignment operator (=) can be extended using an **interface assignment block** of the form:

```
INTERFACE ASSIGNMENT (=)
    MODULE PROCEDURE subroutine_1
    ...
END INTERFACE
```

For an assignment operator, the interface body must refer to a subroutine with two arguments. The first argument is the output of the assignment statement, and must have `INTENT(OUT)`. The second dummy argument is the input to the assignment statement, and must have `INTENT(IN)`. The first argument corresponds to the left hand side of the assignment statement, and the second argument corresponds to the right hand side of the assignment statement.

More than one subroutine can be associated with the assignment symbol, but the subroutines must be distinguishable from one another by having different types of dummy arguments. When the compiler encounters the assignment symbol in a program, it invokes the subroutine whose dummy arguments match the types of the values on either side of the equal sign. If no associated subroutine has dummy arguments that match the values, then a compilation error results.

13-19 A module that declares the `polar` data type and allows the assignment of polar numbers to complex numbers and vice versa is shown below:

```
MODULE polar_math
!
!  Purpose:
!    To define the derived data type "polar" plus the
!    mathematical opertors that use it.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/18/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! Declare type "polar"
TYPE :: polar
   REAL :: z                       ! magnitude
   REAL :: phase                   ! Angle in degrees
END TYPE polar

! Declare named constants:
REAL,PARAMETER :: DEG_2_RAD = .017453293  ! Degrees to radians
REAL,PARAMETER :: RAD_2_DEG = 57.2957795  ! Radians to degrees

! Declare operations
INTERFACE ASSIGNMENT ( = )
   MODULE PROCEDURE complex_2_polar
   MODULE PROCEDURE polar_2_complex
END INTERFACE

CONTAINS

SUBROUTINE complex_2_polar(p, c)
!
!  Purpose:
!    To convert a complex number to type "polar".
```

```
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
TYPE(polar),INTENT(OUT) :: p        ! Result in polar form
COMPLEX,INTENT(IN) :: c             ! Complex number

! Get magnitude and angle
p%z     = ABS ( c )
p%phase = ATAN2( AIMAG(c), REAL(c) ) * RAD_2_DEG

END SUBROUTINE complex_2_polar

SUBROUTINE polar_2_complex(c, p)
!
!  Purpose:
!    To convert a "polar" number to complex.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!     01/19/97    S. J. Chapman        Original code
!
IMPLICIT NONE

! Declare dummy arguments:
COMPLEX, INTENT(OUT) :: c            ! Result in complex form
TYPE (polar),INTENT(IN) :: p         ! Polar number

! Declare local variables:
REAL :: re                          ! Real component
REAL :: im                          ! Imaginary component

! Get real and imaginary parts
re = p%z * COS ( p%phase * DEG_2_RAD )
im = p%z * SIN ( p%phase * DEG_2_RAD )
c = CMPLX ( re, im )

END SUBROUTINE polar_2_complex

END MODULE polar_math
```

This module will be tested in an expanded form in Exercise 13-18.

13-20 A module that implements the polar data type, assignments between polar and complex data, and polar multiplication and division is shown below:

```
MODULE polar_math
!
!  Purpose:
!    To define the derived data type "polar" plus the
```

```
!    mathematical opertors that use it.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare type "polar"
TYPE :: polar
   REAL :: z                       ! magnitude
   REAL :: phase                   ! Angle in degrees
END TYPE polar

! Declare named constants:
REAL,PARAMETER :: DEG_2_RAD = .017453293  ! Degrees to radians
REAL,PARAMETER :: RAD_2_DEG = 57.2957795  ! Radians to degrees

! Declare operations
INTERFACE ASSIGNMENT (=)
   MODULE PROCEDURE complex_2_polar
   MODULE PROCEDURE polar_2_complex
END INTERFACE

INTERFACE OPERATOR (*)
   MODULE PROCEDURE polar_times_polar
END INTERFACE

INTERFACE OPERATOR (/)
   MODULE PROCEDURE polar_div_polar
END INTERFACE

CONTAINS

SUBROUTINE complex_2_polar(p, c)
!
!  Purpose:
!    To convert a complex number to type "polar".
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
TYPE(polar),INTENT(OUT) :: p        ! Result in polar form
COMPLEX,INTENT(IN) :: c             ! Complex number

! Get magnitude and angle
p%z     = ABS ( c )
p%phase = ATAN2( AIMAG(c), REAL(c) ) * RAD_2_DEG

END SUBROUTINE complex_2_polar
```

```
SUBROUTINE polar_2_complex(c, p)
!
!  Purpose:
!    To convert a "polar" number to complex.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
COMPLEX, INTENT(OUT) :: c            ! Result in complex form
TYPE (polar),INTENT(IN) :: p         ! Polar number

! Declare local variables:
REAL :: re                          ! Real component
REAL :: im                          ! Imaginary component

! Get real and imaginary parts
re = p%z * COS ( p%phase * DEG_2_RAD )
im = p%z * SIN ( p%phase * DEG_2_RAD )
c = CMPLX ( re, im )

END SUBROUTINE polar_2_complex

FUNCTION polar_times_polar(polar1, polar2)
!
!  Purpose:
!    To multiply two polar numbers and produce a polar result.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare dummy arguments:
TYPE (polar),INTENT(IN) :: polar1      ! Polar value 1
TYPE (polar),INTENT(IN) :: polar2      ! Polar value 2
TYPE (polar) :: polar_times_polar      ! Function result

! Calculate result
polar_times_polar%z = polar1%z * polar2%z
polar_times_polar%phase = polar1%phase + polar2%phase

! Now limit phase to valid range: -180 < phase <= 180.
DO
   IF ( polar_times_polar%phase > -180. ) EXIT
   polar_times_polar%phase = polar_times_polar%phase + 360.
END DO

DO
```

```
   IF ( polar_times_polar%phase <= 180. ) EXIT
   polar_times_polar%phase = polar_times_polar%phase - 360.
END DO

END FUNCTION polar_times_polar

FUNCTION polar_div_polar(polar1, polar2)
!
!  Purpose:
!    To divide two polar numbers and produce a polar result.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/18/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! Declare dummy arguments:
TYPE (polar),INTENT(IN) :: polar1      ! Polar value 1
TYPE (polar),INTENT(IN) :: polar2      ! Polar value 2
TYPE (polar) :: polar_div_polar        ! Function result

! Calculate result
polar_div_polar%z = polar1%z / polar2%z
polar_div_polar%phase = polar1%phase - polar2%phase

! Now limit%phase to valid range: -180 <%phase <= 180.
DO
   IF ( polar_div_polar%phase > -180. ) EXIT
   polar_div_polar%phase = polar_div_polar%phase + 360.
END DO

DO
   IF ( polar_div_polar%phase <= 180. ) EXIT
   polar_div_polar%phase = polar_div_polar%phase - 360.
END DO

END FUNCTION polar_div_polar

END MODULE polar_math

PROGRAM test_polar_math
!
!  Purpose:
!    To test the polar mathematics functions.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/18/2007    S. J. Chapman       Original code
!
USE polar_math
IMPLICIT NONE

! Declare local variavles
```

```
COMPLEX :: c1, c2, c3                  ! Complex values
TYPE (polar) :: p1, p2, p3             ! Polar values

! Get input data
WRITE (*,*) 'Enter first polar number (mag,angle): '
READ (*,*) p1%z, p1%phase
WRITE (*,*) 'Enter second polar number (mag,angle): '
READ (*,*) p2%z, p2%phase

! Convert to complex
c1 = p1
c2 = p2

! Calculate product using complex math.
c3 = c1 * c2
p3 = c3
WRITE (*,*) 'Product using complex math:  ', p3

! Calculate product using polar math.
p3 = p1 * p2
WRITE (*,*) 'Product using polar math:    ', p3

! Calculate dividend using complex math.
c3 = c1 / c2
p3 = c3
WRITE (*,*) 'Dividend using complex math: ', p3

! Calculate dividend using polar math.
p3 = p1 / p2
WRITE (*,*) 'Dividend using polar math:   ', p3

END PROGRAM test_polar_math
```

A test driver program is shown below:

```
PROGRAM test_polar_math
!
!  Purpose:
!    To test the polar mathematics functions.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/18/2007    S. J. Chapman       Original code
!
USE polar_math
IMPLICIT NONE

! Declare local variavles
COMPLEX :: c1, c2, c3                  ! Complex values
TYPE (polar) :: p1, p2, p3             ! Polar values

! Get input data
WRITE (*,*) 'Enter first polar number (mag,angle): '
READ (*,*) p1%z, p1%phase
WRITE (*,*) 'Enter second polar number (mag,angle): '
```

```
READ (*,*) p2%z, p2%phase

! Convert to complex
c1 = p1
c2 = p2

! Calculate product using complex math.
c3 = c1 * c2
p3 = c3
WRITE (*,*) 'Product using complex math:  ', p3

! Calculate product using polar math.
p3 = p1 * p2
WRITE (*,*) 'Product using polar math:    ', p3

! Calculate dividend using complex math.
c3 = c1 / c2
p3 = c3
WRITE (*,*) 'Dividend using complex math: ', p3

! Calculate dividend using polar math.
p3 = p1 / p2
WRITE (*,*) 'Dividend using polar math:   ', p3

END PROGRAM test_polar_math
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex13_20>test_polar_math

Enter first polar number (mag,angle):
3,36.87
Enter second polar number (mag,angle):
4,53.12
Product using complex math:      12.0000        89.9900
Product using polar math:        12.0000        89.9900
Dividend using complex math:    0.750000       -16.2500
Dividend using polar math:      0.750000       -16.2500
```

13-21 Assess 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 that are to be used in the program unit containing the USE statement.

13-22 *(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.

13-23 The modified module is shown below, with the access control attributes highlighted.

```
MODULE polar_math
!
```

```
!  Purpose:
!    To define the derived data type "polar" plus the
!    mathematical opertors that use it.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/18/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! Restrict access
PRIVATE
PUBLIC polar, ASSIGNMENT(=), OPERATOR(*), OPERATOR(/)

! Declare type "polar"
TYPE :: polar
   REAL :: z                        ! magnitude
   REAL :: phase                    ! Angle in degrees
END TYPE polar

! Declare named constants:
REAL,PARAMETER :: deg_2_rad = .017453293  ! Degrees to radians
REAL,PARAMETER :: rad_2_deg = 57.2957795  ! Radians to degrees

...                                      (Rest of module is unchanged)
```

13-24 *(a)* Named constants "PI" and "TWOPI", and data type "name" will be available. In addition, variable "name1" will be available, but it will be renamed to "sample_name" in the program. *(b)* Only named constant "PI" will be available.

Chapter 14. Advanced I/O Concepts

14-1 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 that is a multiple of 3 and a mantissa between 1.0 and 999.9999. The differences 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 test
```

When this simple program is executed, the results are:

```
C:\book\f95_2003\soln\ex14_1>test
  1.234567E+04
 12.345670E+03
```

14-2 The B, O, and Z format descriptors may be used to display either real or integer data. They display the data in binary, octal, and hexadecimal format respectively.

14-3 The form of the G format descriptor that will display 7 significant digits is G13.7. It is 13 characters wide. (Note that a G12.7 descriptor will work if the number to be displayed is positive, but a G13.7 descriptor is the smallest width that will work for any number, positive or negative.)

14-4 When these numbers are printed with the I8 and I8.8 format descriptors, the results are as shown below. The I8.8 descriptor for the number -128 is all asterisks, since it takes nine spaces to display 8 digits plus a negative sign.

```
    1024    00001024
    -128    ********
   30000    00030000
```

14-5 The program shown below displays the three numbers in the binary, octal, and hexadecimal formats:

```
PROGRAM test
INTEGER :: i = 1024, j = -128, k = 30000
WRITE (*,'(1X,I6,2X,B32,2X,O11,2X,Z8)') i, i, i, i
WRITE (*,'(1X,I6,2X,B32,2X,O11,2X,Z8)') j, j, j, j
WRITE (*,'(1X,I6,2X,B32,2X,O11,2X,Z8)') k, k, k, k
END PROGRAM tests
```

When it is executed, the results are:

```
C:\book\f95_2003\soln\ex14_5>test
  1024                       10000000000         2000       400
  -128  11111111111111111111111110000000  37777777600  FFFFFF80
 30000                    111010100110000        72460      7530
```

The very large binary, octal, and hexadecimal values for the negative number are due to the two's complement representation used for numbers on this computer.

14-6 A possible program to generate and display the random numbers is shown below:

```
PROGRAM test_g_desc
!
!  Purpose:
!    To generate 9 random numbers in the range [-100000,
!    100000), and display them using the G11.5 and SP
!    format descriptors.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/20/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! Declare variables.
INTEGER :: i                   ! Loop index
REAL,DIMENSION(9) :: value     ! Random values

! Get the numbers
DO i = 1, 9
   CALL random0 ( value(i) )
   value(I) = 200000. * value(i) - 100000.
END DO

! Display the numbers.
WRITE (*,1000) value
1000 FORMAT (' Value = ',/,(5X,G11.5))

END PROGRAM test_g_desc
```

When this program is executed, a typical result is

```
Value =
     42249.
    -55516.
     406.09
    -59957.
    -69841.
     59966.
     26841.
     22038.
     16402.
```

14-7 The appropriate format descriptor is:

```
! Display the numbers.
WRITE (*,1020) (i, value(i), i=1,9)
1020 FORMAT (' VALUE(',I1,') = ',F10.2,:,'   VALUE(',I1,') = ',F10.2)
```

The output from this descriptor is

```
        10        20        30        40        50        60
----|----|----|----|----|----|----|----|----|----|----|----|
VALUE(1) =   42248.77   VALUE(2) =  -55515.56
VALUE(3) =     406.09   VALUE(4) =  -59957.16
VALUE(5) =  -69841.44   VALUE(6) =   59966.09
VALUE(7) =   26841.50   VALUE(8) =   22038.44
VALUE(9) =   16402.40
----|----|----|----|----|----|----|----|----|----|----|----|
        10        20        30        40        50        60
```

Note that the colon descriptor prevented the last `VALUE(` from being printed.

14-8 *(a)* -.6388E+11 *(b)* -638.8 *(c)* -.6388 *(d)* 2346. *(e)* Nine spaces followed by a T *(f)* The string `'   String!'`. Note the three spaces at the front of the printed output.

14-9 When the first 4 values are displayed with the `EN15.6` format descriptor, the results are:

```
 -63.876500E+09
-638.765000E+00
-638.765000E-03
   2.345600E+03
```

14-10 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 `NAMELIST` 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.

14-11 These statements will output a namelist containing the values of the array in column-major order. The exact form of the namelist will vary slightly from compiler to compiler. For example, the Microsoft Fortran Powerstation 4.0 compiler produces the output:

```
&IO
ARRAY =     0.000000E+00    0.000000E+00    0.000000E+00        10.000000
0.000000        30.000000       20.000000       40.000000       60.000000
/
```

while the Compaq Visual Fortran Compiler 6.6C produces the output:

```
 &IO
 ARRAY   = 3*0.0000000E+00  ,    10.00000     ,    20.00000     ,    30.00000     ,
20.00000     ,    40.00000     ,    60.00000
 /
```

and the Lahey Fortran 90 Compiler produces the output:

```
 &IO ARRAY=0.000000,0.000000,0.000000,10.0000,20.0000,30.0000,20.0000,
40.0000,60.0000 /
```

14-12 When these statements are executed, `a(1,1)`, `a(3,1)`, and `a(1,3)` will be updated. The value for `a(2,2)` will be ignored, since it is after the first slash.

```
C:\book\f95_2003\soln\ex14_12>test_read_namelist

 &IO A=-100.000,0.000000,6.00000,10.0000,20.0000,30.0000,-6.00000,
40.0000,60.0000 /
```

14-13 The `TRn` format descriptor moves *n* characters to the right in the i/o buffer without disturbing the contents of those characters, while the `nX` format descriptor moves *n* characters to the right in the i/o buffer, writing blanks in those characters.

14-14 *(a)* The output is:

```
Value =   356.2480  0.36E+03 356.25    356.248     3.562E+02
```

(b) The `DO` loop initializes the values of array `i` to 1, 4, 9, 16, and 25. The list-directed read then reads new values into i(1) and i(3), skipping i(2) because of the two commas. The read then terminates at the slash, so the resulting values in the array are:

```
          -101              4              17
            16             25
----|----|----|----|----|----|----|----|----|
    5   10   15   20   25   30   35   40   45
```

14-15 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.

14-16 *(a)* The status of the file is `'UNKNOWN'`. It is a formatted file opened for direct access. It is opened for both reading and writing. The length of each record is 80 characters. 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 there is an error in the open process, the program containing this statement will continue, with `istat` set to an appropriate error code.

(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 dies 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.

(c) The status of the file is `'OLD'`. It is a formatted file opened for sequential access, and the location of the file pointer is at the end of the file, just before the end-of-file marker. It is opened for both reading and writing, with a

variable record length. List-directed character strings will be written to the file delimited by quotes ("). If the file is not found, the open will fail with a value set into `istat`. If there is an error in the open process, the program containing this statement will continue, with `istat` set to an appropriate error code.

(d) The status of the file is `'SCRATCH'`. It is a formatted file opened for sequential access, and the location of the file pointer is at the beginning of the empty file. 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. The file will be created when it is opened, and deleted when it is closed. If there is an error in the open process, the program containing this statement will continue, with `istat` set to an appropriate error code.

14-17 Positive values returned by the `IOSTAT=` clause in a `READ` statement mean that a read error occurred. A negative one (-1) returned by the `IOSTAT=` clause in a `READ` statement mean that the end of file has been reached. A negative two (-2) returned by the `IOSTAT=` clause in a non-advancing `READ` statement mean that the end of record has been reached. A zero value means that the read was successful.

14-18 The program shown below copies data from an input file to an output file, stripping off any trailing blanks in the process. It opens the input file with `STATUS='OLD'`, since the input data must already exist. It then calls function `OPEN2` to open the output file. The function opens the output file with `STATUS='NEW'`. If the file already exists, it asks the user whether or not to overwrite it. If the answer is yes, then the function opens the file with `STATUS='REPLACE'`.

```
PROGRAM blank_remove
!
!  Purpose:
!    To read Fortran source code from an input file and copy it to
!    an output file, stripping out trailing blanks.  This program
!    uses STATUS='OLD' on the input file to make sure that the
!    input file already exists, and STATUS='NEW' on the output file
!    to make sure that the output file is new.  If the output file
!    already exists, it prompts the user to see if it should be
!    overwritten.
!
!  Record of revisions:
!      Date       Programmer         Description of change
!      ====       ==========         =====================
!    05/20/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of variables:
CHARACTER(len=30) :: filename1     ! Input file name
CHARACTER(len=30) :: filename2     ! Output file name
INTEGER :: istat                   ! I/o status
INTEGER :: istat1                  ! Open 1 status
INTEGER :: istat2                  ! Open 2 status
CHARACTER(len=132) :: line         ! Source line
CHARACTER(len=1) :: yn             ! Yes/No character

! Get the name of the file containing the input data.
WRITE (*,*) 'blank_remove: Copy removing trailing blanks'
WRITE (*,*) 'Enter the input file name: '
READ (*,'(A20)') filename1

! Get the name of the file to write the output data to.
WRITE (*,*) 'Enter the output file name: '
READ (*,'(A20)') filename2
```

```
! Open the input file.  Status is OLD because the input data
! must already exist.
OPEN (UNIT=8,FILE=filename1,STATUS='OLD',ACTION='READ',IOSTAT=istat1)

! Does the input file exist?  If so, open the output file.
open1_ok:  IF ( istat1 == 0 ) THEN

   ! Open output file
   OPEN (UNIT=9,FILE=filename2,STATUS='NEW',ACTION='WRITE', &
         IOSTAT=istat2)

   ! Was the open ok?  If not, check with user about what to do.
   open2_ok: IF ( istat2 /= 0 ) THEN

      WRITE (*,1010) filename2(1:LEN_TRIM(filename2))
      1010 FORMAT (' File ',A, ' exists.  Overwrite it? (Y/N)')
      READ (*,'(A)') yn
      IF ( yn == 'Y' .OR. yn == 'y' ) THEN

         OPEN (UNIT=9,FILE=filename2,STATUS='REPLACE', &
              ACTION='WRITE',IOSTAT=istat2)

      END IF
   END IF open2_ok

   ! Is open ok after all?
   file2_open: IF ( istat2 == 0 ) THEN

      ! Copy data from input file to output file.
      DO
         READ (8,'(A)',IOSTAT=istat ) line
         IF ( istat /= 0 ) EXIT
         WRITE (9,'(A)',IOSTAT=istat) line(1:LEN_TRIM(line))
      END DO

      ! All done.  Close output file.
      CLOSE (UNIT=9,STATUS='KEEP')

      ! Do we want to keep the input file?
      WRITE (*,*) 'Delete input file? (Y/N)'
      READ (*,'(A)') yn
      IF ( ( yn == 'Y' ) .OR. ( yn == 'y' ) ) THEN
           CLOSE ( UNIT=8, STATUS='DELETE')
      ELSE
           CLOSE ( UNIT=8, STATUS='KEEP')
      END IF

   ELSE file2_open

      ! File 2 open failed.
      WRITE (*,1020) istat2
      1020 FORMAT (' Open error on output file: ISTAT = ', I6)

   END IF file2_open
```

```
ELSE open1_ok

   ! File 1 open failed.
   WRITE (*,1030) istat1
   1030 FORMAT (' Open error on input file: ISTAT = ', I6)

END IF open1_ok

END PROGRAM blank_remove
```

14-19 *(a)* These statements are valid. They check on the status of file INPUT. The output of this program is:

```
File status:  Exists = T Opened = T Named = T Access = SEQUENTIAL
Format = FORMATTED  Action = READWRITE
Delims = NONE
```

(b) These statements are invalid. You must include a record length clause when opening a direct access file.

14-20 A program to copy a file while reversing the order of the lines is shown below. This program counts the number of lines in the input file, and then reads backwards through the file, copying each line from the end of the input file into the beginning of the output file. Note that the BACKSPACE statement is used *twice* each time a new line is read. The first BACKSPACE statement returns the file pointer to the line that was just read, while the second BACKSPACE statement sets the pointer to the before that one.

```
PROGRAM reverse
!
!  Purpose:
!    To read Fortran source code from an input file and copy it to
!    an output file in reversed order.  This program uses
!    STATUS='OLD' on the input file to make sure that the
!    input file already exists, and STATUS='NEW' on the output file
!    to make sure that the output file is new.  If the output file
!    already exists, it prompts the user to see if it should be
!    overwritten.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/20/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! List of variables:
CHARACTER(len=30) :: filename1     ! Input file name
CHARACTER(len=30) :: filename2     ! Output file name
INTEGER :: i                       ! Loop index
INTEGER :: istat                   ! READ i/o status
INTEGER :: istat1                  ! OPEN 1 status
INTEGER :: istat2                  ! OPEN 2 status
INTEGER :: istat3                  ! BACKSPACE i/o status
INTEGER :: nlines = 0              ! # of lines in input file
CHARACTER(len=132) :: line         ! Source line
CHARACTER(len=1) :: yn             ! Yes/No character

! Get the name of the file containing the input data.
WRITE (*,*) 'blank_remove: Copy removing trailing blanks'
```

```
WRITE (*,*) 'Enter the input file name: '
READ (*,'(A20)') filename1

! Get the name of the file to write the output data to.
WRITE (*,*) 'Enter the output file name: '
READ (*,'(A20)') filename2

! Open the input file with file pointer at end of file.
! Status is OLD because the input data must already exist.
OPEN (UNIT=8,FILE=filename1,STATUS='OLD',ACTION='READ', &
      POSITION='REWIND',IOSTAT=istat1)

! Does the input file exist?  If so, open the output file.
open1_ok:  IF ( istat1 == 0 ) THEN

   ! Open output file
   OPEN (UNIT=9,FILE=filename2,STATUS='NEW',ACTION='WRITE', &
         IOSTAT=istat2)

   ! Was the open ok?  If not, check with user about what to do.
   open2_ok: IF ( istat2 /= 0 ) THEN

      WRITE (*,1010) filename2(1:LEN_TRIM(filename2))
      1010 FORMAT (' File ',A, ' exists.  Overwrite it? (Y/N)')
      READ (*,'(A)') yn
      IF ( yn == 'Y' .OR. yn == 'y' ) THEN

         OPEN (UNIT=9,FILE=filename2,STATUS='REPLACE', &
              ACTION='WRITE',IOSTAT=istat2)

      END IF
   END IF open2_ok

   ! Is open ok after all?
   file2_open: IF ( istat2 == 0 ) THEN

      ! If the opens were successful, advance to the end of the
      ! input file, and back up to the last record in the file.
      ! (Note the two backspaces.  One gets us to the EOF marker,
      !  and one gets us back to the last line in the file.)
      DO
         READ (8,'(A)',IOSTAT=istat) line
         IF ( istat /= 0 ) EXIT
         nlines = nlines + 1
      END DO

      ! Now we know how many lines there are.  Read the data from
      ! the input file in reverse order, writing it to the output
      ! file.  The two backspaces put the file pointer in front of
      ! the record before the previously-read record.
      DO i = 1, nlines
         BACKSPACE (8, IOSTAT=istat3)
         BACKSPACE (8, IOSTAT=istat3)
         READ(8,'(A)',IOSTAT=istat) line
         IF ( istat /= 0 ) EXIT
         WRITE (9,'(A)',IOSTAT=istat) line(1:LEN_TRIM(line))
```

```
      END DO

      ! All done.  Close input and output file.
      CLOSE (UNIT=8,STATUS='KEEP')
      CLOSE (UNIT=9,STATUS='KEEP')

   ELSE file2_open

      ! File 2 open failed.
      WRITE (*,1020) istat2
      1020 FORMAT (' Open error on output file: ISTAT = ', I6)

   END IF file2_open

ELSE open1_ok

   ! File 1 open failed.
   WRITE (*,1030) istat1
   1030 FORMAT (' Open error on input file: ISTAT = ', I6)

END IF open1_ok

END PROGRAM reverse
```

14-21 **Note: The results of this exercise are operating system and compiler dependent. You may get different answers than the ones given here.**

The following program opens two files, one formatted and one unformatted, and writes 10000 values to each file. The actual `WRITE` statements are inside a `DO` loop so that they can be repeated as many times as necessary on your computer to get reliable timings.

```
PROGRAM time_it
!
!  Purpose:
!    To write files containing 1000 elements in both formatted
!    and unformatted format, comparing the resulting file sizes
!    and execution speeds.
!
!  Record of revisions:
!      Date       Programmer         Description of change
!      ====       ==========         =====================
!    05/20/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER,PARAMETER :: N_LOOPS = 50    ! Number of times to write data
INTEGER,PARAMETER :: NVALS = 10000   ! Number of values to write

! List of variables:

CHARACTER(len=12) :: filename1 = 'FORMAT.DAT'   ! Formatted file
CHARACTER(len=12) :: filename2 = 'UNFORMAT.DAT' ! Unformatted file
INTEGER :: i, j, k                    ! Loop indexes
INTEGER :: istat1                     ! File 1 open status
INTEGER :: istat2                     ! File 2 open status
```

```
REAL :: time_formatted = 0.          ! Time for formatted file
REAL :: time_unformatted = 0.        ! Time for unformatted file
REAL,DIMENSION(NVALS) :: values      ! Output data array

! Generate raw data in the range [-1.0E6,1.0E6).
CALL RANDOM_NUMBER ( values )
values = 2000000. * values - 1000000.

! Reset the timer before writing to formatted file.
CALL set_timer

! Open the formatted file.
OPEN (UNIT=8,FILE=filename1,STATUS='REPLACE',FORM='FORMATTED', &
      ACTION='WRITE',IOSTAT=istat1)

! Write the data to the formatted file, timing it as we do so.
! N_LOOPS can be adjusted to give valid timings on any system,
! regardless of its speed.
DO k = 1, N_LOOPS
   DO i = 1, NVALS, 10
      WRITE (8,'(1X,10ES14.7)') (values(j), j = i, i+9)
   END DO
END DO

! Close formatted file.
CLOSE (UNIT=8)

! Get elapsed time for formatted write.
CALL elapsed_time ( time_formatted )
time_formatted = time_formatted / REAL(N_LOOPS)

! Reset the timer before writing to unformatted file.
CALL set_timer

! Open the unformatted file.  Note that sequential access is
! specified.
OPEN (UNIT=8,FILE=filename2,STATUS='REPLACE',ACCESS='SEQUENTIAL',&
      ACTION='WRITE',FORM='UNFORMATTED',IOSTAT=istat2)

! Write the data to the formatted file, timing it as we do so.
! N_LOOPS can be adjusted to give valid timings on any system,
! regardless of its speed.
!
DO k = 1, N_LOOPS
   DO i = 1, NVALS, 10
      WRITE (8) (values(j), j = i, i+9)
   END DO
END DO

! Close unformatted file.
CLOSE (UNIT=8)

! Get elapsed time for formatted write.
CALL elapsed_time ( time_unformatted )
time_unformatted = time_unformatted / REAL (N_LOOPS)
```

```
! Tell user.
WRITE (*,1000) 'Formatted file time   = ', time_formatted, ' sec.'
WRITE (*,1000) 'Unformatted file time = ', time_unformatted, ' sec.'
1000 FORMAT (1X,A,F10.3,A)

END PROGRAM time_it
```

The following questions are answered for one particular 1.8-GHz Coure 2 Duo computer and the Compaq Visual Fortran Compiler 6.6C. You will see different results with different processors and compilers, but the basic pattern should be the same. The formatted file created by the program occupies 7,150,000 bytes, while the unformatted file occupies 2,400,000 bytes. The unformatted file is more than three times smaller than the formatted file. When the program is executed with the Compaq Visual Fortran Compiler, the results are:

```
C:\book\f95_2003\soln\ex14_21>time_it
 Formatted file time   =      0.015 sec.
 Unformatted file time =      0.005 sec.
```

Thus the unformatted `WRITE` was much faster and produced a file the was smaller by a factor of 3.

14-22 __Note__: **The results of this exercise are operating system and compiler dependent. You may get different answers than the ones given here.**

A program to write a 1000-element data set to a formatted sequential access file, a formatted direct access file, and an unformatted direct access file is shown below. After writing the data, the program reads 100 elements back in, in the order 1, 1000, 2, 999, 3, 998, etc., and times how long it takes to recover the data from each type of file. Note that the direct access files are *much* faster than the sequential access files, so the recovery is repeated many times for them in order to come up with a valid time. You should adjust the number of repetitions in the program to the speed of your computer.

```
PROGRAM compare_files
!
!  Purpose:
!    To write formatted sequential access, formatted direct access,
!    and unformatted direct access files containing 1000 records
!    to the disk, and then retrieve 100 of them in the order 1,
!    1000, 2, 999, 3, 998, etc.  The program will measure the
!    time required to retrieve the records from each file.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/20/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER,PARAMETER :: N_LOOPS = 2000  ! No of times to read dir access
INTEGER,PARAMETER :: N_LOOP_SEQ = 5  ! No of times to read seq access
INTEGER,PARAMETER :: MAXVAL = 100    ! Number of values to read
INTEGER,PARAMETER :: NVALS = 1000    ! Number of values to write

! List of variables:
CHARACTER(len=12) :: filename1 = 'FMTSEQ.DAT'   ! Formatted seq file
CHARACTER(len=12) :: filename2 = 'FMTDIR.DAT'   ! Formatted dir file
CHARACTER(len=12) :: filename3 = 'UNFDIR.DAT' ! Unformatted dir file
INTEGER :: i, j, k                     ! Loop indexes
```

```
INTEGER :: irec                        ! Record pointer
INTEGER :: istat1                      ! File 1 open status
INTEGER :: istat2                      ! File 2 open status
INTEGER :: istat3                      ! File 3 open status
REAL,DIMENSION(MAXVAL) :: out1         ! Values from file 1
REAL,DIMENSION(MAXVAL) :: out2         ! Values from file 2
REAL,DIMENSION(MAXVAL) :: out3         ! Values from file 3
INTEGER :: reclen                      ! Record length for unf dir access
REAL :: time_fmt_seq = 0.              ! Time for formatted sequential file
REAL :: time_fmt_dir = 0.              ! Time for formatted direct file
REAL :: time_unf_dir = 0.              ! Time for unformatted direct file
REAL,DIMENSION(NVALS) :: values        ! Output data array

! Generate raw data in the range [-1.0E5,1.0E5).
CALL RANDOM_NUMBER ( values )
values = 200000. * values - 100000.

! Open the formatted sequential access file.
OPEN (UNIT=8,FILE=filename1,STATUS='REPLACE',FORM='FORMATTED', &
      IOSTAT=istat1)

! Write the data to the formatted sequential file.
DO i = 1, NVALS
   WRITE (8,'(1X,ES14.7)') values(i)
END DO

! Open the formatted direct access file.  Record lengths are in
! bytes, so RECL=14.
OPEN (UNIT=9,FILE=filename2,STATUS='REPLACE',FORM='FORMATTED', &
      ACCESS='DIRECT',RECL=14,IOSTAT=istat2)

! Write the data to the formatted direct access file.
DO i=1, NVALS
   WRITE (9,'(E14.7)',REC=i) values(i)
END DO

! Open the unformatted direct access file. Since lengths are in
! processor-dependent units, we must for use INQUIRE to get the
! proper record size.
INQUIRE (IOLENGTH=reclen) values(1)
OPEN (UNIT=10,FILE=filename3,STATUS='REPLACE',FORM='UNFORMATTED', &
      ACCESS='DIRECT',RECL=reclen,IOSTAT=istat3)

! Write the data to the unformatted direct access file.
DO i=1, NVALS
   WRITE (10,REC=i) values(i)
END DO

! Now recover the records from the formatted sequential access
! file.  It is very hard to maneuver through a sequential access
! file.  The best that we can do going forward is to read and
! discard all intervening lines.  The best that we can do going
! backward is to rewind to the front of the file and go forward
! to the desired record.  (Using repeated BACKSPACE commands is
! even slower.)  This process will be slow, so it will only be
! repeated a few times for timing purposes.  First, reset timer.
```

```
CALL set_timer

time_loop_1: DO k = 1, N_LOOP_SEQ
   j = 0
   read_fmt_seq: DO irec = 1, 50

      ! Move from start of file to position "irec"
      REWIND (UNIT=8)
      DO i = 1, irec-1                  ! Skip irec-1 records
         READ (8,'(1X)')
      END DO
      j = j + 1                         ! Bump pointer
      READ (8,'(1X,E14.7)') out1(j)     ! Read record irec

      ! Now get record NVALS-irec.  Advance to the record before
      ! it, and then read the record.
      DO i = irec+1, NVALS-irec
         READ (8,'(1X)')                ! Skip records
      END DO
      j = j + 1                         ! Bump pointer
      READ (8,'(1X,E14.7)') out1(j)     ! Read record irec

   END DO read_fmt_seq
END DO time_loop_1

! Get elapsed time to read the file.
CALL elapsed_time ( time_fmt_seq )
time_fmt_seq = time_fmt_seq / N_LOOP_SEQ

! Now recover the records from the formatted direct access
! file.  It is easy to read these records--just specify the
! record we want to get next.  Because this read is so
! fast, we will repeat it many time to come up with a valid
! time.  First, reset timer.
CALL set_timer

time_loop_2: DO k = 1, N_LOOPS
   j = 0
   read_fmt_dir: DO irec = 1, 50

      ! Read record irec.
      j = j + 1
      READ (9,'(E14.7)',REC=irec) out2(j)

      ! Read record NVALS-irec.
      j = j + 1
      READ (9,'(E14.7)',REC=NVALS+1-irec) out2(j)

   END DO read_fmt_dir
END DO time_loop_2

! Get elapsed time to read the file.
CALL elapsed_time ( time_fmt_dir )
time_fmt_dir = time_fmt_dir / N_LOOPS

! Now recover the records from the unformatted direct access
```

```
! file.  It is easy to read these records--just specify the
! record we want to get next.  Because this read is so
! fast, we will repeat it many time to come up with a valid
! time.  First, reset timer.
CALL set_timer

time_loop_3: DO k = 1, N_LOOPS
   j = 0
   read_unf_dir: DO irec = 1, 50

      ! Read record irec.
      j = j + 1
      READ (10,REC=irec) out3(j)

      ! Read record NVALS-irec.
      j = j + 1
      READ (10,REC=NVALS+1-irec) out3(j)

   END DO read_unf_dir
END DO time_loop_3

! Get elapsed time to read the file.
CALL elapsed_time ( time_unf_dir )
time_unf_dir = time_unf_dir / N_LOOPS

! Write out the records to demonstrate that we have recovered
! the same data from all three files.
WRITE (*,*) 'The data recovered from the three files were: '
j = 0
out: DO irec = 1, 50
   j = j + 1
   WRITE (*,1000) irec, out1(j), out2(j), out3(j)
   j = j + 1
   WRITE (*,1000) NVALS+1-irec, out1(j), out2(j), out3(j)
   1000 FORMAT (5X,I5,2X,ES14.7,2X,ES14.7,2X,ES14.7)
END DO out

! Display timing info.
WRITE (*,1010) 'Formatted sequential file time      = ', &
               time_fmt_seq, ' sec.'
WRITE (*,1010) 'Formatted direct access file time   = ', &
               time_fmt_dir, ' sec.'
WRITE (*,1010) 'Unformatted direct access file time = ', &
               time_unf_dir, ' sec.'
1010 FORMAT (1X,A,F10.6,A)

END PROGRAM compare_files
```

The following questions are answered for one particular 1.8-GHz Core 2 Duo computer and the Compaq Visual Fortran 6.6C Compiler. You will see different results with different processors and compilers, but the basic pattern should be the same. The formatted sequential access file occupied 17000 bytes, the formatted direct access file occupied 14000 bytes, and the unformatted direct access file occupied 4000 bytes. When the program is executed with the Compaq Visual Fortran 6.6C Compiler, the results are:

```
 Formatted sequential file time      =   0.015600 sec.
 Formatted direct access file time   =   0.000422 sec.
```

```
 Unformatted direct access file time =   0.000250 sec.
```

Note that the unformatted direct access method was the most efficient method for accessing randomly-sorted data. This is usually true on most processors.

Chapter 15. Pointers and Dynamic Data Structures

15-1 An ordinary variable contains a value, while a pointer variable contains the *address* of a target variable, which contains a value.

15-2 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`.

15-3 This code is incorrect. Variables `x1` and `x2` are not declared to be targets, so pointer cannot point at them. Even if they were declared as targets the code would still be wrong, since it attempts to point an integer pointer at a real target, and vice versa.

15-4 When a pointer is first declared, its status is undefined. When it is associated with a target, its status is associated, and when the association is broken, its status is disassociated. The association status of a pointer can be checked using the `ASSOCIATED()` intrinsic function.

15-5 These statements are correct. They declare two pointers `p1` and `p2` and two targets `x1` and `x2`, and then associate pointer `p1` with target `x1`. When the WRITE statement executes, `ASSOCIATED(p1)` is true, `ASSOCIATED(p2)` is false, and `ASSOCIATED(p1,x2)` is false since `p1` is not associated with `x2`.

15-6 The `NULL()` function in Fortran 95 is a function to nullify a pointer. It has an advantage over the `NULLIFY` statement in that it can be used in a type declaration statement to initialize the pointer at the same time that it is declared.

15-7 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)
```

15-8 This program creates a 51-element array `info`, and then points `ptr1` at every fifth element in the array. Pointer `ptr2` then points to every second element of the array pointed to by `ptr1`, and `ptr3` points to the third through fifth elements pointed to by `ptr2`. When this program is executed, the results are:

```
ptr1 =  -52.5 -42.0 -31.5 -21.0 -10.5   0.0  10.5  21.0  31.5  42.0  52.5
ptr2 =  -52.5 -31.5 -10.5  10.5  31.5  52.5
ptr3 =  -10.5  10.5  31.5
ave of ptr3 =   10.5
```

15-9 Dynamic memory may be allocated with pointers using the `ALLOCATE` statement, and may be deallocated using the `DEALLOCATE` statement. When the `ALLOCATE` statement is executed with a pointer, a new unnamed dynamic variable or array is created, and the pointer is associated with it. Pointers are more flexible than allocatable arrays, since the same pointer can be used repeatedly to allocate dynamic memory.

15-10 A memory leak is a situation in which an unnamed variable or array is created using a pointer `ALLOCATE` statement, and then the association between the unnamed object and any pointer in the program is broken. Once the pointer association is lost, there is no way to use or deallocate the memory object, so that memory is "lost" for the remainder of the program's execution. This problem can be avoided by always keeping a pointer associated with any dynamically-allocated memory object, and by deallocating the memory object when it is no longer needed.

15-11 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 my_sub

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 sum_values
```

When this program is compiled and executed with the Compaq Visual Fortran compiler, the results are:

```
C:\book\f95_2003\soln\ex15_11>df sum_values.f90
Compaq Visual Fortran Optimizing Compiler Version 6.6 (Update B)
Copyright 2001 Compaq Computer Corp. All rights reserved.

sum_values.f90
Microsoft (R) Incremental Linker Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/subsystem:console
/entry:mainCRTStartup
/ignore:505
/debugtype:cv
/debug:minimal
```

```
/pdb:none
C:\DOCUME~1\SCHAPM~1.000\LOCALS~1\Temp\obj42.tmp
dfor.lib
libc.lib
dfconsol.lib
dfport.lib
kernel32.lib
/out:sum_values.exe

C:\book\f95_2003\soln\ex15_11>sum_values

 Enter values to add:
4
  The sum is    4.000000
2
  The sum is    6.000000
5
  The sum is    11.00000
7
  The sum is    18.00000
4
  The sum is    22.00000
^D
```

15-12 This program is incorrect. It allocates an array on pointer `ptr1`, and then associates the pointer `ptr2` with the array as well. Next, it prints out the array using both pointers to illustrate that they are pointing to the same location. Then the program deallocates the memory using `ptr1`, which automatically disassociates the pointer, but `ptr2` is left pointing *to the location in memory where the variable used to be*. It then allocates another array on pointer `ptr1`, and attempts to write out the memory associated with both pointers. When `ptr2` is written, the results are invalid, since the pointer no longer points to a valid dynamic array. The results will depend on how dynamic memory is allocated and re-used by a particular processor. For the Compaq Visual Fortran Compiler, the results are:

```
C:\book\f95_2003\soln\ex15_12>ex11_12
 ptr1 =   1  2  3  4  5  6  7  8  9 10
 ptr2 =   1  2  3  4  5  6  7  8  9 10
 ptr1 =  -2  0  2
 ptr2 =  -2  0  2  4***  0******  9 10
```

For the Lahey Fortran 90 Compiler, the results are:

```
C:\book\f95_2003\soln\ex15_12>ex11_12

ptr1 =   1  2  3  4  5  6  7  8  9 10
ptr2 =   1  2  3  4  5  6  7  8  9 10
ptr1 =  -2  0  2
ptr2 =  -2  0  2  4  5  6  7  8  9 10
```

15-13 A program to perform an insertion sort on character variables in a case-insensitive manner using the ASCII collating sequence is shown below. Major changes from the integer insertion sort program are shown in bold face. Note that we are reusing function `ucase()` from Exercise 7-4 to shift the strings to upper case for comparison purposes.

```
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
   !       ====       ==========         =====================
   !     12/23/06     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 myprocs

PROGRAM insertion_sort
!
!  Purpose:
!    To read a series of character strings from an input data
!    file and sort them into ascending order on a case-insensitive
!    basis using the ASCII collating sequence and an insertion sort.
!    After the values are sorted, they will be written back to the
!    standard output device.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!     12/23/06     S. J. Chapman      Original code
! 1.  05/22/07     S. J. Chapman      Modified from integer sort
!
USE myprocs
IMPLICIT NONE

! Derived data type to store character values in
TYPE :: chr_value
   CHARACTER(len=40) :: value
```

```
   TYPE (chr_value), POINTER :: next_value
END TYPE

! List of variables:
TYPE (chr_value), POINTER :: head   ! Pointer to head of list
CHARACTER(len=30) :: filename       ! Input data file name
INTEGER :: istat                    ! Status: 0 for success
INTEGER :: nvals = 0                ! Number of data read
TYPE (chr_value), POINTER :: ptr    ! Ptr to new value
TYPE (chr_value), POINTER :: ptr1   ! Temp ptr for search
TYPE (chr_value), POINTER :: ptr2   ! Temp ptr for search
TYPE (chr_value), POINTER :: tail   ! Pointer to tail of list
CHARACTER(len=40) :: temp           ! Temporary variable

! Get the name of the file containing the input data.
WRITE (*,*) 'Enter the file name with the data to be sorted: '
READ (*,'(A30)') filename

! Open input data file.
OPEN ( UNIT=9, FILE=filename, STATUS='OLD', ACTION='READ', &
       IOSTAT=istat )

! Was the OPEN successful?
fileopen: IF ( istat == 0 ) THEN         ! Open successful

   ! The file was opened successfully, so read the data value
   ! to sort, allocate a variable for it, and locate the proper
   ! point to insert the new value into the list.
   input: DO
      READ (9,'(A)', IOSTAT=istat) temp   ! Get value
      IF ( istat /= 0 ) EXIT input        ! Exit on end of data
      nvals = nvals + 1                   ! Bump count

      ALLOCATE (ptr,STAT=istat)           ! Allocate space
      ptr%value = temp                    ! Store string

      ! Now find out where to put it in the list.
      new: IF (.NOT. ASSOCIATED(head)) THEN ! No values in list
         head => ptr                      ! Place at front
         tail => head                     ! Tail pts to new value
         NULLIFY (ptr%next_value)         ! Nullify next ptr
      ELSE
         ! Values already in list.  Check for location.
         front: IF ( ucase(ptr%value) < ucase(head%value) ) THEN
            ! Add at front of list
            ptr%next_value => head
            head => ptr
         ELSE IF ( ucase(ptr%value) >= ucase(tail%value) ) THEN
            ! Add at end of list
            tail%next_value => ptr
            tail => ptr
            NULLIFY ( tail%next_value )
         ELSE
            ! Find place to add value
            ptr1 => head
            ptr2 => ptr1%next_value
```

```
            search: DO
               IF ( (ucase(ptr%value) >= ucase(ptr1%value)) .AND. &
                    (ucase(ptr%value) < ucase(ptr2%value)) ) THEN
                  ! Insert value here
                  ptr%next_value => ptr2
                  ptr1%next_value => ptr
                  EXIT search
               END IF
               ptr1 => ptr2
               ptr2 => ptr2%next_value
            END DO search
         END IF front
      END IF new
   END DO input

   ! Now, write out the data.
   ptr => head
   output: DO
      IF ( .NOT. ASSOCIATED(ptr) ) EXIT   ! Pointer valid?
      WRITE (*,'(1X,A)') ptr%value        ! Yes: Write value
      ptr => ptr%next_value               ! Get next pointer
   END DO output

ELSE fileopen

   ! Else file open failed.  Tell user.
   WRITE (*,'(1X,A,I6)') 'File open failed--status = ', istat

END IF fileopen

END PROGRAM insertion_sort
```

An appropriate data set to test this program is contained in file `in15_13.dat`:

```
"This is a test"
123
string
sTr1
Str2
HELP
```

When this program is compiled and executed with the above data set, the results are:

```
C:\book\f95_2003\soln\ex15_13>insertion_sor

Enter the file name with the data to be sorted:
in15_13.dat

"This is a test"
123
HELP
sTr1
Str2
string
```

The program is sorting the strings according to the ASCII collating sequence in a case-independent manner.

15-14 *(a)* A subroutine to sort the real data using an insertion sort with a linked list is shown below:

```
SUBROUTINE sort_linked_list (values, nvals)
!
!  Purpose:
!    To sort a series of real values from an input array
!    and sort them using an insertion sort.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/18/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of dummy arguments:
INTEGER,INTENT(IN) :: nvals          ! Number of values to sort
REAL,INTENT(INOUT),DIMENSION(nvals) :: values ! List of values

! Derived data type to store integer values in
TYPE :: real_value
   REAL :: value
   TYPE (real_value), POINTER :: next_value
END TYPE

! List of variables:
TYPE (real_value), POINTER :: head  ! Pointer to head of list
INTEGER :: i                        ! Loop index
INTEGER :: istat                    ! Status: 0 for success
TYPE (real_value), POINTER :: ptr   ! Ptr to new value
TYPE (real_value), POINTER :: ptr1  ! Temp ptr for search
TYPE (real_value), POINTER :: ptr2  ! Temp ptr for search
TYPE (real_value), POINTER :: tail  ! Pointer to tail of list

! Get each data value to sort, allocate a variable for it, and
! locate the proper point to insert the new value into the list.
input: DO i = 1, nvals

   ALLOCATE (ptr,STAT=istat)           ! Allocate space
   ptr%value = values(i)               ! Store number

   ! Now find out where to put it in the list.
   new: IF (.NOT. ASSOCIATED(head)) THEN ! No values in list
      head => ptr                      ! Place at front
      tail => head                     ! Tail pts to new value
      NULLIFY (ptr%next_value)         ! Nullify next ptr
   ELSE
      ! Values already in list.  Check for location.
      front: IF ( ptr%value < head%value ) THEN
         ! Add at front of list
         ptr%next_value => head
         head => ptr
      ELSE IF ( ptr%value >= tail%value ) THEN
         ! Add at end of list
         tail%next_value => ptr
         tail => ptr
```

```
         NULLIFY ( tail%next_value )
      ELSE
         ! Find place to add value
         ptr1 => head
         ptr2 => ptr1%next_value
         search: DO
            IF ( (ptr%value >= ptr1%value) .AND. &
                 (ptr%value < ptr2%value) ) THEN
               ! Insert value here
               ptr%next_value => ptr2
               ptr1%next_value => ptr
               EXIT search
            END IF
            ptr1 => ptr2
            ptr2 => ptr2%next_value
         END DO search
      END IF front
   END IF new
END DO input

! Now, output the sorted data.
ptr => head
i = 0
output: DO
   IF ( .NOT. ASSOCIATED(ptr) ) EXIT   ! Pointer valid?
   i = i + 1                           ! Yes: Write value
   values(i) = ptr%value
   ptr => ptr%next_value               ! Get next pointer
END DO output

END SUBROUTINE sort_linked_list
```

(b) A subroutine to sort the real data using an insertion sort with a binary tree structure is shown below:

```
MODULE btree
!
!  Purpose:
!    To define the derived data type used as a node in the
!    binary tree, and to define the operations >, <. and ==
!    for this data type.  This module also contains the
!    subroutines to add a node to the tree, write out the
!    values in the tree, and find a value in the tree.
!
!  Record of revisions:
!      Date      Programmer         Description of change
!      ====      ==========         =====================
!    12/24/06    S. J. Chapman      Original code
! 1. 05/18/07    S. J. Chapman      Original code
!
IMPLICIT NONE

! Restrict access to module contents.
PRIVATE
PUBLIC :: node, OPERATOR(>), OPERATOR(<), OPERATOR(==)
PUBLIC :: add_node, write_node
```

```
! Declare type for a node of the binary tree.
TYPE :: node
   REAL :: value
   TYPE (node), POINTER :: before
   TYPE (node), POINTER :: after
END TYPE

INTERFACE OPERATOR (>)
   MODULE PROCEDURE greater_than
END INTERFACE

INTERFACE OPERATOR (<)
   MODULE PROCEDURE less_than
END INTERFACE

INTERFACE OPERATOR (==)
   MODULE PROCEDURE equal_to
END INTERFACE

CONTAINS
   RECURSIVE SUBROUTINE add_node (ptr, new_node)
   !
   !  Purpose:
   !    To add a new node to the binary tree structure.
   !
   TYPE (node), POINTER :: ptr  ! Pointer to current pos. in tree
   TYPE (node), POINTER :: new_node ! Pointer to new node

   IF ( .NOT. ASSOCIATED(ptr) ) THEN
      ! There is no tree yet.  Add the node right here.
      ptr => new_node
   ELSE IF ( new_node < ptr ) THEN
      IF ( ASSOCIATED(ptr%before) ) THEN
         CALL add_node ( ptr%before, new_node )
      ELSE
         ptr%before => new_node
      END IF
   ELSE
      IF ( ASSOCIATED(ptr%after) ) THEN
         CALL add_node ( ptr%after, new_node )
      ELSE
         ptr%after => new_node
      END IF
   END IF
   END SUBROUTINE add_node

   RECURSIVE SUBROUTINE write_node (ptr, values, nvals, icount)
   !
   !  Purpose:
   !    To write out the contents of the binary tree
   !    structure in order.
   !
   TYPE (node), POINTER :: ptr  ! Pointer to current pos. in tree
   INTEGER,INTENT(IN) :: nvals  ! Sise of array
   REAL,INTENT(INOUT),DIMENSION(nvals) :: values ! Array
   INTEGER,INTENT(INOUT) :: icount ! Current position in array
```

```
! Write contents of previous node.
IF ( ASSOCIATED(ptr%before) ) THEN
   CALL write_node ( ptr%before, values, nvals, icount )
END IF

! Output contents of current node.
icount = icount + 1
values(icount) = ptr%value

! Write contents of next node.
IF ( ASSOCIATED(ptr%after) ) THEN
   CALL write_node ( ptr%after, values, nvals, icount )
END IF
END SUBROUTINE write_node

LOGICAL FUNCTION greater_than (op1, op2)
!
!  Purpose:
!    To test to see if operand 1 is > operand 2
!    in alphabetical order.
!
TYPE (node), INTENT(IN) :: op1, op2

IF (op1%value > op2%value) THEN
   greater_than = .TRUE.
ELSE
   greater_than = .FALSE.
END IF
END FUNCTION greater_than

LOGICAL FUNCTION less_than (op1, op2)
!
!  Purpose:
!    To test to see if operand 1 is < operand 2
!    in alphabetical order.
!
TYPE (node), INTENT(IN) :: op1, op2

IF (op1%value < op2%value) THEN
   less_than = .TRUE.
ELSE
   less_than = .FALSE.
END IF
END FUNCTION less_than

 LOGICAL FUNCTION equal_to (op1, op2)
!
!  Purpose:
!    To test to see if operand 1 is equal to operand 2
!    alphabetically.
!
TYPE (node), INTENT(IN) :: op1, op2

IF ( op1%value == op2%value ) THEN
   equal_to = .TRUE.
```

```
   ELSE
      equal_to = .FALSE.
   END IF
   END FUNCTION equal_to

END MODULE btree

SUBROUTINE sort_binary_tree (values, nvals)
!
!  Purpose:
!    To sort a series of real values from an input array
!    and sort them using an insertion sort.
!
!  Record of revisions:
!      Date      Programmer         Description of change
!      ====      ==========         =====================
!    05/18/07    S. J. Chapman      Original code
!
USE btree
IMPLICIT NONE

! List of dummy arguments:
INTEGER,INTENT(IN) :: nvals           ! Number of values to sort
REAL,INTENT(INOUT),DIMENSION(nvals) :: values ! List of values

! List of variables:
INTEGER :: i                          ! Loop index
INTEGER :: istat                      ! Status: 0 for success
TYPE (node), POINTER :: root          ! Pointer to root node
TYPE (node), POINTER :: temp          ! Temp pointer to node

! Nullify new pointers
NULLIFY ( root, temp )

! Allocate space for each node, read the data into that node,
! and insert it into the binary tree.
DO i = 1, nvals
   ALLOCATE (temp,STAT=istat)          ! Allocate node
   NULLIFY ( temp%before, temp%after)  ! Nullify pointers
   temp%value = values(i)
   CALL add_node(root, temp)           ! Add to binary tree
END DO

! Now, output the sorted data.
i = 0
CALL write_node(root, values, nvals, i)

END SUBROUTINE sort_binary_tree
```

(c) A program to sort 50,000 real values using both the linked list and the binary tree sorting subroutines is shown below:

```
PROGRAM test_sorts
!
!  Purpose:
!    To create a an array of 50,000 random real numbers, and to
```

```
!    sort that array with both the linked list and the binary
!    tree.  The two methods will be timed.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/23/2007    S. J. Chapman      Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER,PARAMETER :: NVALS = 50000   ! Number of values to write

! List of variables:
REAL,DIMENSION(NVALS) :: array       ! Array to sort
REAL,DIMENSION(NVALS) :: array_saved ! Saved copy of array
REAL :: time_linked_list = 0.        ! Time for formatted file
REAL :: time_binary_tree = 0.        ! Time for unformatted file

! Create array of random values
CALL RANDOM_NUMBER ( array )
array = 20000. * array - 10000.      ! Scale to [-10000, 10000]
array_saved = array                  ! Save copy

! Reset the timer before sorting.
CALL set_timer

! Sort using linked list
CALL sort_linked_list ( array, NVALS )

! Get elapsed time for linked list.
CALL elapsed_time ( time_linked_list )

! To show that the subroutine is working, write out first
! 20 sorted values.
WRITE (*,'(1X,A)') 'First 20 sorted values from liked list: '
WRITE (*,*) array(1:20)

! Get new copy of array.
array = array_saved

! Reset the timer before sorting.
CALL set_timer

! Sort using linked list
CALL sort_binary_tree ( array, NVALS )

! Get elapsed time for linked list.
CALL elapsed_time ( time_binary_tree )

! To show that the subroutine is working, write out first
! 20 sorted values.
WRITE (*,'(/1X,A)') 'First 20 sorted values from binary tree: '
WRITE (*,*) array(1:20)

! Tell user.
```

```
WRITE (*,1000) 'Linked list time = ', time_linked_list, ' sec.'
WRITE (*,1010) 'Binary tree time = ', time_binary_tree, ' sec.'
1000 FORMAT (/1X,A,F10.3,A)
1010 FORMAT (1X,A,F10.3,A)

END PROGRAM test_sorts
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex15_14>test_sorts
 First 20 sorted values from liked list:
  -9999.992      -9999.865      -9999.798      -9999.577      -9999.354
  -9998.749      -9998.473      -9997.710      -9997.572      -9997.428
  -9996.795      -9996.067      -9995.680      -9995.493      -9995.312
  -9994.740      -9993.296      -9993.115      -9992.635      -9992.563

 First 20 sorted values from binary tree:
  -9999.992      -9999.865      -9999.798      -9999.577      -9999.354
  -9998.749      -9998.473      -9997.710      -9997.572      -9997.428
  -9996.795      -9996.067      -9995.680      -9995.493      -9995.312
  -9994.740      -9993.296      -9993.115      -9992.635      -9992.563

 Linked list time =      8.843 sec.
 Binary tree time =      0.047 sec.
```

The binary tree sort is ***much*** faster than the linked list sort! This result illustrates the power of the binary tree to efficiently access large quantities of data. Note that the times shown were for a 1.8 GHz Core 2 Duo processor. If you have a computer that is slower, you may need to reduce the size of the array to be sorted so that the program will finish in a reasonable time.

15-15 An array of pointers can be defined by creating a derived data type whose only element is a pointer to an array, and then creating an array of that derived data type.

15-16 This program declares an array consisting of four variables of a derived data type, where each derived data type contains a pointer to an array. Each pointer is associated with a target that is a rank-1 array, and then the arrays are manipulated. The first `WRITE` statement writes out the sum of the second element of the array pointed to by `p(1)` plus the fourth element of the array pointed to by `p(4)` plus the third element of the array pointed to by `p(3)`, which is 2 + 13 + 9 = 24. The next `WRITE` statement prints out the arrays pointed to by each element of `p`. When the program is executed, the results are:

```
C:\book\f95_2003\soln\ex15_16>ex15_16
  24.0

   1.0   2.0   3.0   4.0
   5.0   6.0
   7.0   8.0   9.0
  10.0  11.0  12.0  13.0  14.0
```

15-17 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
   !       ====       ==========         =====================
   !    05/24/2007    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 subs
```

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
!       ====       ==========         =====================
!    05/24/2007    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 test_maxval
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex15_17>test_maxval
The max value is:  87.00
```

15-18 This problem is slightly different than the previous one, in that a pointer to the array is passed to the function, not the array itself. The resulting code is shown below:

```
MODULE subs
CONTAINS

   FUNCTION maxval (array) RESULT (ptr_maxval)
   !
   !  Purpose:
   !    To return a pointer to the maximum value in a
   !    rank one array.  A pointer to the arrys is
   !    passed to the function.
   !
   !  Record of revisions:
   !       Date       Programmer         Description of change
   !       ====       ==========         =====================
   !    05/24/2007    S. J. Chapman      Original code
   !
   IMPLICIT NONE

   ! Declare calling arguments:
   REAL,DIMENSION(:),POINTER :: array   ! Ptr to 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 subs
```

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
```

```
!       ====       ==========         ====================
!    05/24/2007    S. J. Chapman      Original code
!
USE subs
IMPLICIT NONE

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

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

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

END PROGRAM test_maxval
```

When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex15_18>test_maxval
The max value is:  87.00
```

15-19 A subroutine to calculate a least squares to a linked list of input data values is shown below.

```
MODULE subs
IMPLICIT NONE

! Derived data type to store real values in
TYPE :: real_pair
   REAL :: x
   REAL :: y
   TYPE (real_pair), POINTER :: p
END TYPE

CONTAINS
   SUBROUTINE lsqfit_list ( head, nvals, slope, y_int, error )
   !
   !  Purpose:
   !    To perform a least-squares fit of an input data set
   !    to the line Y(X) = slope * x + y_int and return the
   !    resulting coefficients.
   !
   !  Record of revisions:
   !       Date       Programmer         Description of change
   !       ====       ==========         ====================
   !    05/24/2007    S. J. Chapman      Original code
   !
   IMPLICIT NONE

   ! List of calling arguments:
   INTEGER,INTENT(IN) :: nvals       ! No. of values
   TYPE(real_pair),POINTER :: head   ! List of (x,y) pairs
   REAL,INTENT(OUT) :: slope         ! Slope of fitted line
```

```
   REAL,INTENT(OUT) :: y_int         ! y-axis intercept of line
   INTEGER,INTENT(OUT) :: error      ! Error flag: 0 = no error
                                     !  1 = not enough input values
   ! List of local variables:
   INTEGER :: i                ! Index variable
   TYPE(real_pair),POINTER :: ptr ! Pointer to (x,y) pair
   REAL :: sum_x               ! The sum of all input x values
   REAL :: sum_x2              ! The sum of all input x values squared
   REAL :: sum_xy              ! The sum of all input x*y values
   REAL :: sum_y               ! The sum of all input y values
   REAL :: xbar                ! The average x value
   REAL :: ybar                ! The average y value

   ! First, check to make sure that we have enough input data.
   IF ( nvals < 2 ) THEN

      ! Insufficient data.  Set error = 1, and get out.
      error = 1

   ELSE

      ! Reset error flag.
      error = 0

      ! Zero the sums used to build the equations.
      sum_x  = 0.
      sum_x2 = 0.
      sum_xy = 0.
      sum_y  = 0.

      ! Build the sums required to solve the equations.
      ptr => head
      DO i = 1, nvals
         sum_x  = sum_x + ptr%x
         sum_y  = sum_y + ptr%y
         sum_x2 = sum_x2 + ptr%x**2
         sum_xy = sum_xy + ptr%x * ptr%y
         ptr => ptr%p
      END DO

      ! Now calculate the slope and intercept.
      xbar  = sum_x / REAL(nvals)
      ybar  = sum_y / REAL(nvals)
      slope = (sum_xy - sum_x * ybar) / ( sum_x2 - sum_x * xbar)
      y_int  = ybar - slope * xbar

   END IF

   END SUBROUTINE lsqfit_list
END MODULE subs
```

A test driver program for this subroutine is shown below. Note that this program has an advantage over the other least squares fit programs that we have examined, in that there is no arbitrary upper limit to the amount of data that the program can accept, and we do not have to know in advance how many values there will be.

```
PROGRAM test_lsqfit_list
```

```
!
!  Purpose:
!    To read a series of real values from an input data file
!    and store them in a linked list.  After the list is read,
!    it is passed to subroutine lsqfit_list for a least-squares
!    fit calculation.
!
!  Record of revisions:
!       Date       Programmer         Description of change
!       ====       ==========         =====================
!    05/24/2007    S. J. Chapman      Original code
!
USE subs
IMPLICIT NONE

! List of variables:
INTEGER :: error                    ! Error flag
TYPE (real_pair), POINTER :: head  ! Pointer to head of list
CHARACTER(len=30) :: filename      ! Input data file name
INTEGER :: nvals = 0               ! Number of data read
REAL :: slope                      ! Slope of line
TYPE (real_pair), POINTER :: tail  ! Pointer to tail of list
INTEGER :: istat                   ! Status: 0 for success
REAL :: x                          ! Temporary variable
REAL :: y                          ! Temporary variable
REAL :: y_int                      ! Y-axis intercept of line

! Get the name of the file containing the input data.
WRITE (*,*) 'Enter the input file name: '
READ (*,'(A30)') filename

! Open input data file.
OPEN ( UNIT=9, FILE=filename, STATUS='OLD', ACTION='READ', &
       IOSTAT=istat )

! Was the OPEN successful?
fileopen: IF ( istat == 0 ) THEN         ! Open successful

   ! The file was opened successfully, so read the data from
   ! it, and store it in the linked list.
   input: DO
      READ (9, *, IOSTAT=istat) x, y      ! Get value
      IF ( istat /= 0 ) EXIT              ! Exit on end of data
      nvals = nvals + 1                   ! Bump count

      IF (.NOT. ASSOCIATED(head)) THEN    ! No values in list
         ALLOCATE (head,STAT=istat)       ! Allocate new value
         tail => head                     ! Tail pts to new value
         NULLIFY (tail%p)                 ! Nullify p in new value
         tail%x = x                       ! Store number
         tail%y = y                       ! Store number
      ELSE                                ! Values already in list
         ALLOCATE (tail%p,STAT=istat)     ! Allocate new value
         tail => tail%p                   ! Tail pts to new value
         NULLIFY (tail%p)                 ! Nullify p in new value
         tail%x = x                       ! Store number
```

```
         tail%y = y                       ! Store number
      END IF
   END DO input

   ! Now call subroutine lsqfit_list.
   CALL lsqfit_list ( head, nvals, slope, y_int, error )

   ! Tell user about fit.
   WRITE (*, 1000 ) slope, y_int, nvals
   1000 FORMAT ('Regression coefficients for the least-squares line:',&
              /,'   Slope (m)     = ', F12.3,&
              /,'   Intercept (b) = ', F12.3,&
              /,'   No of points  = ', I12 )

ELSE fileopen

   ! Else file open failed.  Tell user.
   WRITE (*,'(1X,A,I6)') 'File open failed--status = ', istat

END IF fileopen

END PROGRAM test_lsqfit_list
```

This subroutine can be tested with the same data set as the least-squares-fit subroutine in Exercise 6-28, and it will produce the same answers:

```
C:\book\f95_2003\soln\ex15_19>test_lsqfit_list
 Enter the input file name:
in15_19.dat
Regression coefficients for the least-squares line:
   Slope (m)     =        1.844
   Intercept (b) =        0.191
   No of points  =           20
```

15-20 A program that creates a doubly-linked list is shown below:

```
PROGRAM doubly_linked_list
!
!  Purpose:
!    To read in a series of real values from an input data file
!    and store them in a doubly linked list.  After the list
!    is read, it will be written back in both forward and reverse
!    order to the standard output device.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!    05/24/2007    S. J. Chapman       Original code
!
IMPLICIT NONE

! Derived data type to store real values in
TYPE :: real_value
   REAL :: value
   TYPE (real_value), POINTER :: before
   TYPE (real_value), POINTER :: after
```

```
END TYPE

! List of variables:
TYPE (real_value), POINTER :: head  ! Pointer to head of list
CHARACTER(len=20) :: filename       ! Input data file name
INTEGER :: nvals = 0                ! Number of data read
TYPE (real_value), POINTER :: ptr   ! Temporary pointer
TYPE (real_value), POINTER :: tail  ! Pointer to tail of list
INTEGER :: istat                    ! Status: 0 for success
REAL :: temp                        ! Temporary variable

! Get the name of the file containing the input data.
WRITE (*,*) 'Enter the file name with the data to be read: '
READ (*,'(A20)') filename

! Open input data file.
OPEN ( UNIT=9, FILE=filename, STATUS='OLD', ACTION='READ', &
       IOSTAT=istat )

! Was the OPEN successful?
fileopen: IF ( istat == 0 ) THEN         ! Open successful

   ! The file was opened successfully, so read the data from
   ! it, and store it in the linked list.
   input: DO
      READ (9, *, IOSTAT=istat) temp      ! Get value
      IF ( istat /= 0 ) EXIT              ! Exit on end of data
      nvals = nvals + 1                   ! Bump count

      IF (.NOT. ASSOCIATED(head)) THEN    ! No values in list
         ALLOCATE (head,STAT=istat)       ! Allocate new value
         tail => head                     ! Tail pts to new value
         NULLIFY (tail%before,tail%after) ! Nullify ptrs in new value
         tail%value = temp                ! Store number
      ELSE                                ! Values already in list
         ALLOCATE (tail%after,STAT=istat) ! Allocate new value
         ptr => tail                      ! Temp ptr to prev link
         tail => tail%after               ! Tail pts to new value
         tail%before => ptr               ! Set new before pointer
         NULLIFY (tail%after)             ! Nullify p in new value
         tail%value = temp                ! Store number
      END IF
   END DO input

   ! Now, write out the in the forward data.
   WRITE (*,'(/1X,A)') 'Data in the forward direction:'
   ptr => head
   fwd: DO
      IF ( .NOT. ASSOCIATED(ptr) ) EXIT   ! Pointer valid?
      WRITE (*,'(1X,F10.4)') ptr%value    ! Yes: Write value
      ptr => ptr%after                    ! Get next pointer
   END DO fwd

   ! Now, write out the in the reverse data.
   WRITE (*,'(/1X,A)') 'Data in the reverse direction:'
   ptr => tail
```

```
   rev: DO
      IF ( .NOT. ASSOCIATED(ptr) ) EXIT   ! Pointer valid?
      WRITE (*,'(1X,F10.4)') ptr%value    ! Yes: Write value
      ptr => ptr%before                   ! Get next pointer
   END DO rev

ELSE fileopen

   ! Else file open failed.  Tell user.
   WRITE (*,'(1X,A,I6)') 'File open failed--status = ', istat

END IF fileopen

END PROGRAM doubly_linked_list
```

A data set consisting of 20 random values between -110 and 100 stored in file `in15-20.dat`. When this program is executed, the results are:

```
C:\book\f95_2003\soln\ex15_20>doubly_linked_list

 Enter the file name with the data to be read:
in15_20.dat

 Data in the forward direction:
  -100.0000
   -82.9900
    20.2700
    78.3200
    93.5900
   -62.0600
     3.0000
   -20.4000
   -47.4200
    48.7000
   -82.0900
    12.0800
    16.4500
    61.9100
    18.3800
     2.3400
    75.3300
    99.0200
    45.2400
    93.3200

 Data in the reverse direction:
    93.3200
    45.2400
    99.0200
    75.3300
     2.3400
    18.3800
    61.9100
    16.4500
    12.0800
   -82.0900
```

```
    48.7000
   -47.4200
   -20.4000
     3.0000
   -62.0600
    93.5900
    78.3200
    20.2700
   -82.9900
  -100.0000
```

15-21 A program to perform an insertion sort into a doubly-linked list is shown below:

```
PROGRAM insertion_sort
!
!  Purpose:
!    To read a series of integer values from an input data file
!    and sort them using an insertion sort.  After the values
!    are sorted, they will be written back to the standard
!    output device.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!     12/23/06     S. J. Chapman       Original code
! 1.  05/24/07     S. J. Chapman       Doubly linked list
!
IMPLICIT NONE

! Derived data type to store integer values in
TYPE :: real_value
   REAL :: value
   TYPE (real_value), POINTER :: prev_value
   TYPE (real_value), POINTER :: next_value
END TYPE

! List of variables:
TYPE (real_value), POINTER :: head  ! Pointer to head of list
CHARACTER(len=30) :: filename       ! Input data file name
INTEGER :: istat                    ! Status: 0 for success
INTEGER :: nvals = 0                ! Number of data read
TYPE (real_value), POINTER :: ptr   ! Ptr to new value
TYPE (real_value), POINTER :: ptr1  ! Temp ptr for search
TYPE (real_value), POINTER :: ptr2  ! Temp ptr for search
TYPE (real_value), POINTER :: tail  ! Pointer to tail of list
REAL :: temp                        ! Temporary variable

! Get the name of the file containing the input data.
WRITE (*,*) 'Enter the file name with the data to be sorted: '
READ (*,'(A30)') filename

! Open input data file.
OPEN ( UNIT=9, FILE=filename, STATUS='OLD', ACTION='READ', &
       IOSTAT=istat )

! Was the OPEN successful?
```

```
fileopen: IF ( istat == 0 ) THEN        ! Open successful

   ! The file was opened successfully, so read the data value
   ! to sort, allocate a variable for it, and locate the proper
   ! point to insert the new value into the list.
   input: DO
      READ (9, *, IOSTAT=istat) temp      ! Get value
      IF ( istat /= 0 ) EXIT input        ! Exit on end of data
      nvals = nvals + 1                   ! Bump count

      ALLOCATE (ptr,STAT=istat)           ! Allocate space
      ptr%value = temp                    ! Store number

      ! Now find out where to put it in the list.
      new: IF (.NOT. ASSOCIATED(head)) THEN ! No values in list
         head => ptr                      ! Place at front
         tail => head                     ! Tail pts to new value
         NULLIFY (ptr%prev_value)         ! Nullify previous ptr
         NULLIFY (ptr%next_value)         ! Nullify next ptr
      ELSE
         ! Values already in list.  Check for location.
         front: IF ( ptr%value < head%value ) THEN
            ! Add at front of list
            NULLIFY ( ptr%prev_value )
            ptr%next_value => head
            head => ptr
         ELSE IF ( ptr%value >= tail%value ) THEN
            ! Add at end of list
            tail%next_value => ptr
            ptr%prev_value => tail
            tail => ptr
            NULLIFY ( tail%next_value )
         ELSE
            ! Find place to add value
            ptr1 => head
            ptr2 => ptr1%next_value
            search: DO
               IF ( (ptr%value >= ptr1%value) .AND. &
                    (ptr%value < ptr2%value) ) THEN
                  ! Insert value here
                  ptr%prev_value => ptr1
                  ptr%next_value => ptr2
                  ptr1%next_value => ptr
                  ptr2%prev_value => ptr
                  EXIT search
               END IF
               ptr1 => ptr2
               ptr2 => ptr2%next_value
            END DO search
         END IF front
      END IF new
   END DO input

   ! Now, write out the data in ascending order.
   WRITE (*,'(1X,A)') 'List in ascending order:'
   ptr => head
```

```
   ascend: DO
      IF ( .NOT. ASSOCIATED(ptr) ) EXIT   ! Pointer valid?
      WRITE (*,'(1X,F10.3)') ptr%value    ! Yes: Write value
      ptr => ptr%next_value               ! Get next pointer
   END DO ascend

   ! Now, write out the data in ascending order.
   WRITE (*,'(/1X,A)') 'List in descending order:'
   ptr => tail
   descend: DO
      IF ( .NOT. ASSOCIATED(ptr) ) EXIT   ! Pointer valid?
      WRITE (*,'(1X,F10.3)') ptr%value    ! Yes: Write value
      ptr => ptr%prev_value               ! Get prev pointer
   END DO descend

ELSE fileopen

   ! Else file open failed.  Tell user.
   WRITE (*,'(1X,A,I6)') 'File open failed--status = ', istat

END IF fileopen

END PROGRAM insertion_sort
```

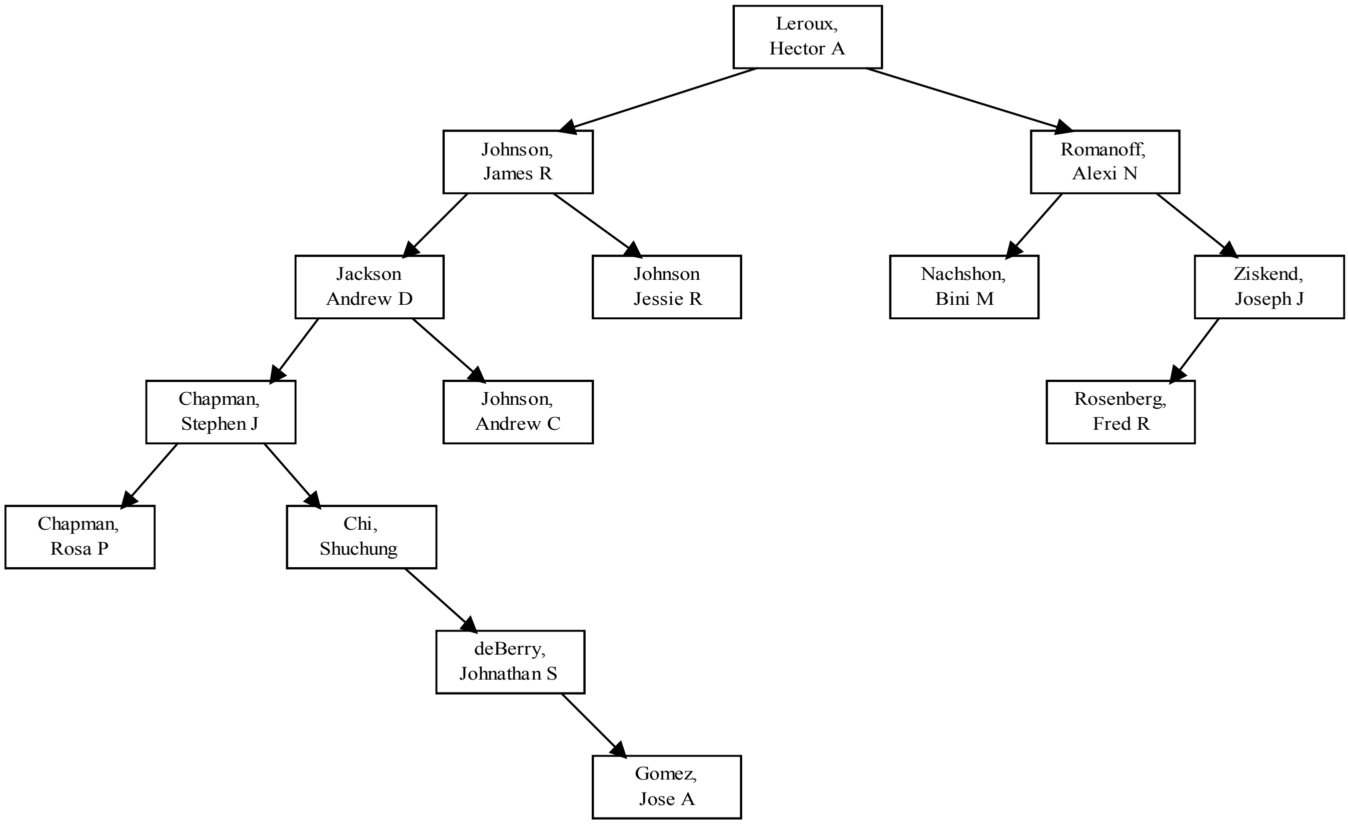
When this program is executed with 50 random values between -1000 and 1000, the results are:

```
C:\book\f95_2003\soln\ex15_21>insertion_sort
Enter the file name with the data to be sorted:
in15_21.dat
List in ascending order:
  -999.960
  -856.680
  -829.940
  -820.900
  -812.740
  -670.570
  -620.620
  -616.280
  -552.780
  -474.190
  -411.950
  -405.800
  -383.090
  -209.140
  -203.980
  -170.710
  -147.900
  -135.480
   -15.240
    23.430
    29.790
    29.950
    88.550
   120.780
   164.460
   183.840
```

```
   184.810
   199.100
   202.710
   218.970
   306.000
   378.280
   452.420
   487.020
   545.690
   560.730
   579.570
   619.130
   689.850
   715.970
   753.270
   783.220
   799.000
   803.070
   813.690
   872.490
   923.070
   933.220
   935.910
   990.170

List in descending order:
   990.170
   935.910
   933.220
   923.070
   872.490
   813.690
   803.070
   799.000
   783.220
   753.270
   715.970
   689.850
   619.130
   579.570
   560.730
   545.690
   487.020
   452.420
   378.280
   306.000
   218.970
   202.710
   199.100
   184.810
   183.840
   164.460
   120.780
    88.550
    29.950
    29.790
```

```
 23.430
 -15.240
-135.480
-147.900
-170.710
-203.980
-209.140
-383.090
-405.800
-411.950
-474.190
-552.780
-616.280
-620.620
-670.570
-812.740
-820.900
-829.940
-856.680
-999.960
```

15-22 The binary tree is shown below. It is irregular, and it gets as deep as 7 layers at one point.



Chapter 16. Object Oriented Programming in Fortran

NOTE: At the current state of Fortran 2003 compiler development in May 2007, the examples in this chapter are not compiling properly. I will be releasing solutions to the problems in this chapter as soon as the next generation of compilers is released.

16-1 :

Appendix A. Library Procedure Descriptions

A small library that contains a number of useful procedures is available for use with this book. The `BOOKLIB` library contains procedures that are useful as classroom exercises, and also serves as an example of good programming style.

The procedures in `BOOKLIB` are indexed by name and by function in the table shown below.

Table A-1: Procedures included in library `BOOKLIB`

Name	Function	Page
`cross_prod`	Calculate the cross product of two 3-element vectors.	359
`deriv`	Calculate derivative of a user-supplied function.	360
`dft`	Calculate Discrete Fourier Transform from its definition.	361
`fft`	Calculate Fast Discrete Fourier Transform.	362
`heapsort`	Sort an array into ascending order using the heapsort algorithm.	363
`heapsort_2`	Sort an array into ascending order while carrying along a second array, using the heapsort algorithm.	364
`histogram`	Print a histogram of an input data set on a line printer.	365
`idft`	Calculate inverse Discrete Fourier Transform from its definition.	367
`ifft`	Calculate inverse Fast Discrete Fourier Transform.	368
`integ`	Integrate a user-supplied function $f(x)$ between points x_1 and x_2 using rectangles of width Δx.	369
`integ_d`	Integrate a discrete function specified by a series of (x,y) values between points x_1 and x_2, where x_1 and x_2 both lie within the range of input values in the (x,y) pairs.	370
`interp`	Linearly interpolate the value y_0 at position x_0, given a set of (x,y) measurements organized in increasing order of x.	371
`lcase`	Shift a character string to lower case.	372
`lsq_fit`	Perform a least-squares fit of an input data set to the nth order polynomial.	373
`mat_inv`	Invert an `N x N` matrix using Gaussian elimination and the maximum pivot technique.	374
`nxtmul`	Calculate the next power of a base above a specific number.	375
`plot`	Print a line printer plot a set of a set of data points.	376
`plotxy`	Print a line printer cross-plot a set of (x,y) data points.	378
`random_u`	Uniform distribution random number generator (function).	380
`random_n`	Normal distribution random number generator (function).	380
`random_r`	Rayleigh distribution random number generator (function).	380
`simul`	Solve a system of simultaneous equations.	381
`sinc`	Calculate the sinc function: $\mathrm{sinc}(x) = \sin(x) / x$.	382
`spline_fit`	Calculate the set of cubic spline polynomials that fit an input data set.	383
`spline_int`	Interpolate a point using the set of cubic spline polynomials generated by subroutine `spline_fit`.	385
`ssort`	Sort an array into ascending order using the selection sort algorithm.	386
`statistics`	Calculate the average, standard deviation, and mean of a data set.	387
`ucase`	Shift a character string to upper case.	372

Many of the procedures in these libraries are generic procedures, which work with multiple types of input data. The types of data supported by each procedure is shown in parentheses after the procedure name. The **keywords** associated with dummy procedure arguments are shown in `CAPITAL LETTERS` in the calling sequences, and the keywords for optional arguments are shown in *`ITALICS`*.

In the following procedure descriptions, data types are given by the following abbreviations:

Abbreviation	**Type**

`R`	Single Prec. Real
`D`	Double Prec. Real
`C`	Single Prec. Complex
`D`	Double Prec. Complex
`I`	Integer
`L`	Logical
`Char`	Character

`cross_prod` (Single/Double Precision Real)

Purpose: To calculate the cross product of two three-element real vectors.

Usage:
```
USE booklib
vector = cross_prod ( VA, VB )
```

Arguments:

Name	Type	Dim	I/O	Description
VA	R/D	3	I	First vector.
VB	R/D	3	I	Second vector.
cross_prod	R/D	3	O	Cross product of VA and VB.

Algorithm:

This function calculates the cross product of two vectors according to the equations:

```
cross_prod(1) = v1(2) * v2(3) - v2(2) * v1(3)
cross_prod(2) = v1(3) * v2(1) - v2(3) * v1(1)
cross_prod(3) = v1(1) * v2(2) - v2(1) * v1(2)
```

Example:

This example calculates cross product of two vectors va = [1. 0. 1.] and vb = [-1 1 -1].

```
USE booklib
IMPLICIT NONE
REAL, DIMENSION(3) :: va = (/  1., 0.,  1. /)
REAL, DIMENSION(3) :: vb = (/ -1., 1., -1. /)
WRITE (*,'(A,3(2X,F10.4))') ' The cross product is ', &
      cross_prod (va, vb)
END PROGRAM
```

Result:

```
The cross product is     -1.0000       .0000      1.0000
```

`deriv` (Single/Double Precision Real)

Purpose: To calculate the derivative of a function *f(x)* at point x_0 using step size Δx. If $\Delta x = 0.0$, then take the derivative with as much accuracy as possible. This subroutine expects the function *f(x)* to be passed as a calling argument.

Usage:
```
USE booklib
CALL deriv ( F, X0, DX, DFDX, ERROR )
```

Arguments:

Name	Type	Dim	I/O	Description
F	R/D Func.		I	Function to take derivative of
X0	R/D		I	Point at which to take derivative
DX	R/D		I/O	Step size to use when taking derivative (≥ 0.0). If DX = 0.0, then the routine calculates an optimal step size, and returns that step size in this variable.
DFDX	R/D		O	The derivative *df(x)/dx*
ERROR	I		O	Error flag: 0 = No error 1 = DX < 0.

Algorithm:

This subroutine calculates the derivative using the central difference method:

$$\frac{d}{dx} f(x) \approx \frac{f(x + \Delta x / 2) - f(x - \Delta x / 2)}{\Delta x}$$

The subroutine uses the user-specified Δx if it is > 0. Otherwise, it tries values of $\Delta x = 0.1, 0.01$, *etc.* until roundoff errors start to dominate in the solution. If Δx is zero, then the actual Δx used to calculate the derivative is returned in variable DX.

Example:

This example calculates derivative of function sin(x) at $x_0 = 1.0$ using the default step size.

```
USE booklib
INTRINSIC SIN
INTEGER :: error
REAL :: dfdx, dx = 0.
CALL deriv ( SIN, 1.0, dx, dfdx, error )
WRITE (*,1000) dfdx
1000 FORMAT (' The derivative of SIN(X) at X0 = 1.0 is:  ', F10.6)
WRITE (*,1010) COS(1.0)
1010 FORMAT (' The theoretical value is:                 ', F10.6)
WRITE (*,1020) dx
1020 FORMAT (' The step size used is:                    ', F10.6)
```

Result:

```
The derivative of SIN(X) at X0 = 1.0 is:     .540316
The theoretical value is:                    .540302
The step size used is:                       .001000
```

`dft` (Single / Double Precision Complex)

Purpose: To perform a discrete Fourier transform on complex array `ARRAY_IN` with the result returned in array `ARRAY_OUT`. This routine calculates the DFT directly from its definition.

Usage:
```
USE booklib
CALL dft ( ARRAY_IN, ARRAY_OUT, N )
```

Arguments:

Name	Type	Dim	I/O	Description
ARRAY_IN	C/Z	N	I	Time series to analyze
ARRAY_OUT	Same as above	N	O	Frequency spectrum of data set
N	I		I	Number of values in array

Algorithm:

This subroutine calculates the DFT directly from its definition. It is very slow compared to subroutine `fft` for large arrays of data. Unlike subroutine `fft`, it does not require that the number of input points be a power of 2. If there are N input values, t_k is the kth value in the input time sequence, and F_n is the nth component in the output frequency spectrum, then

$$F_n = \sum_{k=0}^{N-1} t_k \, e^{-2\pi ikn/N}$$

WARNING: This routine is very slow for large array sizes. It is included in the library to support homework problems only. For real work, use subroutine `fft` instead.

Example:

This example calculates the frequency spectrum of a 16-point complex data set consisting of all (1.0,0.0). Because this data set is constant, the peak of the frequency spectrum of the data should be 0 Hz (DC).

```
USE booklib
COMPLEX, DIMENSION(16) :: array_in(16) = (/ ((1.,0.), i=1,16) /)
COMPLEX, DIMENSION(16) :: array_out(16)
CALL dft ( array_in, array_out, 16 )
WRITE (*,1000) (i,array_out(i), i=1,16)
1000 FORMAT (' array_out(',I2,') = (',F10.4,',',F10.4,')')
```

Result:

```
array_out( 1) = (   16.0000,    0.0000)
array_out( 2) = (    0.0000,    0.0000)
array_out( 3) = (    0.0000,    0.0000)
array_out( 4) = (    0.0000,    0.0000)
array_out( 5) = (    0.0000,    0.0000)
array_out( 6) = (    0.0000,    0.0000)
    ...
array_out(15) = (    0.0000,    0.0000)
array_out(16) = (    0.0000,    0.0000)
```

fft (Single / Double Precision Complex)

Purpose: To perform a fast discrete Fourier transform on complex array `ARRAY_IN`. The resulting frequency spectrum is returned in array `ARRAY_OUT`. The size of the data set in array `ARRAY_IN` must be a power of 2 (32, 64, 128, etc.).

Usage:
```
USE booklib
CALL fft ( ARRAY_IN, ARRAY_OUT, N, ERROR )
```

Arguments:

Name	Type	Dim	I/O	Description
ARRAY_IN	C/Z	N	I	Time series to analyze
ARRAY_OUT	Same as above	N	O	Frequency spectrum of data set
N	I		I	Number of data points (must be a power of 2)
ERROR	I		O	Error flag: 0 = No error 1 = N not a power of 2

Algorithm:

This subroutine employs a Radix 2, in-place, decimation in frequency algorithm. To avoid destroying the input data set, it copies the contents of `ARRAY_IN` to `ARRAY_OUT` before performing the FFT. For details, see Oppenheim and Shaffer, *Digital Signal Processing*, Prentice-Hall, 1975. If there are N input values, t_k is the kth value in the input time sequence, and F_n is the nth component in the output frequency spectrum, then

$$F_n = \sum_{k=0}^{N-1} t_k \, e^{-2\pi ikn/N}$$

Example:

This example calculates the frequency spectrum of a 16-point complex data set consisting of all (1.0,0.0). Because this data set is constant, the peak of the frequency spectrum of the data should be 0 Hz (DC).

```
USE booklib
INTEGER :: error
COMPLEX, DIMENSION(16) :: array_in(16) = (/ ((1.,0.), i=1,16) /)
COMPLEX, DIMENSION(16) :: array_out(16)
CALL fft ( array_in, array_out, 16, error )
WRITE (*,1000) (i,array_out(i), i=1,16)
1000 FORMAT (' array_out(',I2,') = (',F10.4,',',F10.4,')')
```

Result:

```
array_out( 1) = (   16.0000,    0.0000)
array_out( 2) = (    0.0000,    0.0000)
array_out( 3) = (    0.0000,    0.0000)
array_out( 4) = (    0.0000,    0.0000)
array_out( 5) = (    0.0000,    0.0000)
array_out( 6) = (    0.0000,    0.0000)
    ...
array_out(15) = (    0.0000,    0.0000)
array_out(16) = (    0.0000,    0.0000)
```

`heapsort` (Integer/Single Prec. Real/Double Prec. Real/Character)

Purpose: To sort an array into ascending order using the heapsort algorithm.

Usage:
```
USE booklib
CALL heapsort ( ARRAY, N, ERROR )
```

Arguments:

Name	Type	Dim	I/O	Description
ARRAY	I/R/D/C	N	I/O	Array to sort
N	I		I	Number of elements in array
ERROR	I		O	Error flag: 0 = No error 1 = N <= 0

Algorithm:

These subroutines sort arrays into ascending order using the heapsort algorithm. This algorithm is much more efficient than the selection sort algorithm. It should be used instead of the selection sort whenever large arrays are to be sorted.

Example:

This example declares an integer array `iarray` and initializes it with 15 values. It uses subroutine `heapsort` to sort the array into ascending order.

```
USE booklib
INTEGER, DIMENSION(15) :: iarray = &
      (/ -100,    0,  -20,    1,  -20, &
           90, -123,  602,    5,   17, &
           91,   -4,    0,   37,  -11 /)
INTEGER :: n = 15, error
WRITE (*,*) ' iarray before sorting: '
WRITE (*,'(3X,5I6)') iarray
CALL heapsort ( iarray, n, error )
WRITE (*,*) ' iarray after sorting: '
WRITE (*,'(3X,5I6)') iarray
```

Result:

```
 iarray before sorting:
    -100     0   -20     1   -20
      90  -123   602     5    17
      91    -4     0    37   -11
 iarray after sorting:
    -123  -100   -20   -20   -11
      -4     0     0     1     5
      17    37    90    91   602
```

heapsort_2 (Integer/Single Prec. Real/Double Prec. Real/Character)

Purpose: To sort an array into ascending order while carrying along a second array, using the heapsort algorithm.

Usage:
```
USE booklib
CALL heapsort_2 ( ARRAY, ARRAY_2, N, ERROR )
```

Arguments:

Name	Type	Dim	I/O	Description
ARRAY	I/R/D/C	N	I/O	Array to sort
ARRAY_2	Same as above	N	I/O	Array to carry along
N	I		I	Number of elements in array
ERROR	I		O	Error flag: 0 = No error 1 = N <= 0

Algorithm:

This subroutine sorts arrays into ascending order using the heapsort algorithm, and carries along a second array. For example, if ARRAY(i) is moved to the top of array ARRAY, then ARRAY_2(i) is moved to the top of array ARRAY_2. This subroutine permits a user to sort the contents of one array according to the values in another one.

Example:

This example declares two integer arrays iarray and ipoint. It initializes iarray with 15 arbitrary values, and ipoint with the numbers 1 through 15. After sorting the arrays with subroutine heapsort_2, the values in ipoint are pointers to the original locations of the values in iarray.

```
USE booklib
INTEGER, DIMENSION(15) :: iarray = &
      (/ -100,    0,  -20,    1,  -20, &
           90, -123,  602,    5,   17, &
           91,   -4,    0,   37,  -11 /)
INTEGER, DIMENSION(15) :: ipoint = &
      (/  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 /)
INTEGER :: n = 15, error
CALL heapsort_2 ( iarray, ipoint, n, error )
WRITE (*,*) ' Sorted array and original locations: '
WRITE (*,1000) (iarray(i), ipoint(i), I = 1, 15)
1000 FORMAT (3X,2I6,8X,2I6,8X,2I6)
```

Result:

```
 Sorted array and original locations:
    -123     7          -100     1          -20     3
     -20     5           -11    15           -4    12
       0    13             0     2            1     4
       5     9            17    10           37    14
      90     6            91    11          602     8
```

`histogram` (Single Precision Real)

Purpose: Subroutine to print a histogram of an input data set on a line printer.

Usage:
```
USE booklib
CALL histogram ( DATA1, NPTS, LU, ERROR, NBINS, MINBIN, MAXBIN )
```

Arguments:

Name	Type	Dim	I/O	Description
`DATA1`	R	`NPTS`	I	Data set to analyze
`NPTS`	I		I	Number of points in input data set
`LU`	I		I	I/o unit to print histogram on.
`ERROR`	I		O	Error flag: 0 = No error 1 = Too few bins requested (<1) 2 = `MAXBIN` = `MINBIN`. These values must differ.
`NBINS`	I		I	Number of bins to accumulate statistics in. Optional argument. If present, `NBINS` must be greater than 1. If absent, it defaults to 20.
`MINBIN`	R		I	Value of the smallest bin in the histogram. Optional argument. The default value is the smallest number in the data set.
`MAXBIN`	R		I	Value of the largest bin in the histogram. Optional argument. The default value is the largest number in the data set.

Algorithm:

This subroutine calculates the range of values associated with each bin, and then accumulates statistics on the input data set. It then prints the resulting histogram on the device specified by i/o unit `LU`.

Example:

This example uses function `random_n` to generate 10000 random numbers with a normal distribution, and then plots a histogram of the data using subroutine `histogram`. Note that the values of *`NBINS`*, *`MINBIN`*, and *`MAXBIN`* are defaulted.

```
USE booklib
INTEGER :: error
REAL, DIMENSION(10000) :: data1
DO i = 1, 10000
   data1(i) = random_n()
END DO
CALL histogram(data1, 10000, 6, error )
```

Result:

```
                                             584                         1169
                 +-------------+--------------+--------------+--------------+
<=-3.300000E+00
  -3.000000E+00
  -2.700000E+00  *
  -2.400000E+00  ***
  -2.100000E+00  ******
  -1.800000E+00  ************
  -1.500000E+00  *********************
  -1.200000E+00  *****************************
  -8.999999E-01  ******************************************
  -5.999999E-01  ****************************************************
  -2.999999E-01  ********************************************************
   1.192093E-07  ************************************************************
   3.000001E-01  *********************************************************
   6.000001E-01  **************************************************
   9.000002E-01  ***************************************
   1.200000E+00  *******************************
   1.500000E+00  ********************
   1.800000E+00  ************
   2.100000E+00  *******
   2.400000E+00  ***
   2.700000E+00  **
   3.000000E+00
>= 3.300000E+00
                 +-------------+--------------+--------------+--------------+
                                             584                         1169

                Number of samples =  10000
```

`idft` (Single / Double Precision Complex)

Purpose: To perform an inverse discrete Fourier transform on complex array `ARRAY_IN` with the result returned in array `ARRAY_OUT`. This subroutine calculates the inverse DFT directly from its definition.

Usage:
```
USE booklib
CALL idft ( ARRAY_IN, ARRAY_OUT, N )
```

Arguments:

Name	Type	Dim	I/O	Description
ARRAY_IN	C/Z	N	I	Frequency spectrum of data set
ARRAY_OUT	Same as above	N	O	Resulting time series
N	I		I	Number of values in array

Algorithm:

This subroutine calculates the inverse DFT directly from its definition. It is very slow compared to subroutine `ifft` for large arrays of data. Unlike subroutine `ifft`, it does not require that the number of input points be a power of 2. If there are N input values, t_k is the kth value in the input time sequence, and F_n is the nth component in the output frequency spectrum, then

$$F_n = \frac{1}{N}\sum_{k=0}^{N-1} t_k \; e^{2\pi ikn/N}$$

WARNING: This routine is very slow for large array sizes. It is included in the library to support homework problems only. For real work, use subroutine `ifft` instead.

Example:

This example shows that `idft` is the inverse of `dft`. Here, we take both the DFT and the inverse DFT of a data set, and wind up with the data we started with.

```
USE booklib
COMPLEX, DIMENSION(16) :: c_in, c_inter, c_out
c_in = (/ (0.,0.), (1.,0.),  (2.,0.),  (1.,0.),  &
          (0.,0.), (-1.,0.), (-2.,0.), (-1.,0.), &
          (0.,0.), (1.,0.),  (2.,0.),  (1.,0.),  &
          (0.,0.), (-1.,0.), (-2.,0.), (-1.,0.) /)
CALL dft ( c_in, c_inter, 16 )
CALL idft ( c_inter, c_out, 16 )
WRITE (*,1000) (c_out(i), i=1, 16)
1000 FORMAT ('  c_out = ',/,4('  (',F4.1,',',F4.1,')'))
```

Result:

```
 c_out =
 ( 0.0, 0.0)  ( 1.0, 0.0)  ( 2.0, 0.0)  ( 1.0, 0.0)
 ( 0.0, 0.0)  (-1.0, 0.0)  (-2.0, 0.0)  (-1.0, 0.0)
 ( 0.0, 0.0)  ( 1.0, 0.0)  ( 2.0, 0.0)  ( 1.0, 0.0)
 ( 0.0, 0.0)  (-1.0, 0.0)  (-2.0, 0.0)  (-1.0, 0.0)
```

`ifft` (Single / Double Precision Complex)

Purpose: To perform a fast inverse discrete Fourier transform on the frequency spectrum in complex array `ARRAY_IN`. The resulting time series is returned in array `ARRAY_OUT`. The size of the data set in array `ARRAY_IN` must be a power of 2 (32, 64, 128, etc.).

Usage:
```
USE booklib
CALL ifft ( ARRAY_IN, ARRAY_OUT, N, ERROR )
```

Arguments:

Name	Type	Dim	I/O	Description
ARRAY_IN	C/Z	N	I	Frequency spectrum of data set
ARRAY_OUT	Same as above	N	O	Resulting time series
N	I		I	Number of data points (must be a power of 2)
ERROR	I		O	Error flag: 0 = No error 1 = N not a power of 2

Algorithm:

This subroutine employs a Radix 2, in-place, decimation in frequency algorithm. To avoid destroying the input data set, it copies the contents of `ARRAY_IN` to `ARRAY_OUT` before performing the inverse FFT. For details, see Oppenheim and Shaffer, *Digital Signal Processing*, Prentice-Hall, 1975. If there are N input values, t_k is the kth value in the input time sequence, and F_n is the nth component in the output frequency spectrum, then

$$t_k = \frac{1}{N}\sum_{n=0}^{N-1} F_n \, e^{2\pi ikn/N}$$

Example:

This example shows that `ifft` is the inverse of `fft`. Here, we take both the FFT and the inverse FFT of a data set, and wind up with the data we started with.

```
USE booklib
INTEGER :: error
COMPLEX, DIMENSION(16) :: c_in, c_inter, c_out
c_in = (/ (0.,0.), (1.,0.),  (2.,0.),  (1.,0.),  &
          (0.,0.), (-1.,0.), (-2.,0.), (-1.,0.), &
          (0.,0.), (1.,0.),  (2.,0.),  (1.,0.),  &
          (0.,0.), (-1.,0.), (-2.,0.), (-1.,0.) /)
CALL fft ( c_in, c_inter, 16, error )
CALL ifft ( c_inter, c_out, 16, error )
WRITE (*,1000) (c_out(i), i=1, 16)
1000 FORMAT ('  c_out = ',/,4('  (',F4.1,',',F4.1,')'))
```

Result:

```
 c_out =
 ( 0.0, 0.0)  ( 1.0, 0.0)  ( 2.0, 0.0)  ( 1.0, 0.0)
 ( 0.0, 0.0)  (-1.0, 0.0)  (-2.0, 0.0)  (-1.0, 0.0)
 ( 0.0, 0.0)  ( 1.0, 0.0)  ( 2.0, 0.0)  ( 1.0, 0.0)
 ( 0.0, 0.0)  (-1.0, 0.0)  (-2.0, 0.0)  (-1.0, 0.0)
```

`integ` (Single / Double Precision Real)

Purpose: To integrate a function *f(x)* between points x_1 and x_2 using rectangles of width Δx. This subroutine expects the function *f(x)* to be passed as a calling argument.

Usage:

```
USE booklib
CALL integ ( F, X1, X2, DX, AREA, ERROR )
```

Arguments:

Name	Type	Dim	I/O	Description
`F`	`R/D FUN`		`I`	Name of function to integrate
`X1`	`Same as above`		`I`	Starting point for integration
`X2`	`Same`		`I`	Ending point for integration
`DX`	`Same`		`I`	Step size for integration
`AREA`	`Same`		`O`	Integrated value
`ERROR`	`I`		`O`	Error flag: 0 = No error 1 = `X1 > X2`

Algorithm:

This subroutine calculates the area under the curve *f(x)* by dividing the distance between x_1 and x_2 into steps of size Δx and calculating the area under the curve for each step. The area calculation is done by approximating the area under the curve as a rectangle whose height is the value of *f(x)* at the center of the step interval.

Example:

This example uses `integ` to integrate the intrinsic function sin(x) from 0 to π. (The theoretical area of this integral is 2.0.)

```
USE booklib
INTRINSIC SIN
INTEGER :: error
REAL :: x1 = 0., x2 = 3.141592, dx = 0.05, area
CALL integ ( SIN, x1, x2, dx, area, error )
WRITE (*,1000) area
1000 FORMAT (' The area under curve SIN(x) from 0. to PI is: ', F10.6)
```

Result:

```
The area under curve SIN(x) from 0. to PI is:   2.000208
```

integ_d (Single / Double Precision Real)

Purpose: To integrate a discrete function specified by a series of *(x,y)* values between points x_1 and x_2, where x_1 and x_2 both lie within the range of input values in the *(x,y)* pairs. The *(x,y)* pairs must be passed to this subroutine in increasing order of *x*.

Usage:
```
USE booklib
CALL integ_d ( X, Y, NPTS, X1, X2, AREA, ERROR )
```

Arguments:

Name	Type	Dim	I/O	Description
X	R/D	NPTS	I	Values of independent variable *x*
Y	Same as X	NPTS	I	Values of dependent variable *y*
NPTS	I		I	Number of *(x,y)* values passed to the subroutine
X1	Same as X		I	Starting point for integration
X2	Same as X		I	Ending point for integration
AREA	Same		O	Integrated value
ERROR	I		O	Error flag: 0 = No error 1 = X1 > X2 2 = X1 < X(1) 3 = X2 > X(NPTS)

Algorithm:

This subroutine calculates the area under a curve specified by a series of discrete *(x,y)* points by calculating the area under the trapezoids formed by adjacent pairs of *(x,y)* values.

Example:

This example uses `integ_d` to integrate the intrinsic function sin(*x*) from 0 to π. Note that sin(x) is specified by *(x,y)* values in a pair of arrays. (The theoretical area of this integral is 2.0.)

```
USE booklib
INTEGER :: i, npts = 101, error
REAL, DIMENSION(101) :: x, y
REAL :: x1 = 0., x2 = 3.141592, dx, area
dx = ( x2 - x1 ) / REAL(npts - 1)
DO i = 1, npts
   x(i) = dx * REAL(i-1)
   y(i) = SIN(x(i))
END DO
CALL integ_d ( x, y, npts, x1, x2, area, error )
WRITE (*,1000) area
1000 FORMAT (' The area under curve SIN(x) from 0. to PI is: ', F10.6)
```

Result:

```
The area under curve SIN(x) from 0. to PI is:   1.999836
```

`interp` (Single / Double Precision Real)

Purpose: To linearly interpolate the value y_o at position x_o, given a set of *(x,y)* measurements organized in increasing order of *x*.

Usage:
```
USE booklib
CALL interp ( X, Y, NPTS, X0, Y0, ERROR )
```

Arguments:

Name	Type	Dim	I/O	Description
X	R/D	NPTS	I	Values of independent variable *x*
Y	Same as X	NPTS	I	Values of dependent variable *y*
NPTS	I		I	Number of *(x,y)* measurements
X0	Same as X		I	Point at which to interpolate Y0
Y0	Same as X		O	Interpolated value at point X0
AREA	Same		O	Integrated value
ERROR	I		O	Error flag: 0 = No error -1 = X0 < X(1) 1 = X0 > X(NPTS)

Algorithm:
```
Find points X(I) and X(I+1) that straddle X0
slope ← ( Y(I+1)-Y(I) ) / ( X(I+1)-X(I) )
Y0 ← slope * ( X0 - X(I) ) + Y(I)
```

This routine requires that X0 fall between two points in array X. If X0 is outside the range of the points in X, the subroutine returns an error. (Also see subroutines `spline_fit` and `spline_int`.)

Example:

This example interpolates the value at X0 = 5.2.

```
USE booklib
INTEGER :: npts = 4, error
REAL, DIMENSION(4) :: x = (/ 3., 4., 5., 6. /)
REAL, DIMENSION(4) :: y = (/ 2.0, 0.9, 0.0, -0.9 /)
REAL :: x0 = 5.2, y0
CALL interp ( x, y, npts, x0, y0, error )
WRITE (*,1000) ' x0 = ', x0, '  y0 = ', y0
1000 FORMAT (1X,A,F8.3,A,F8.3)
```

Result:

```
 X0 =    5.200  Y0 =    -.180
```

lcase/ucase (Character)

Purpose: Subroutine to shift a character string to lower/upper case.

Usage:
```
USE booklib
CALL lcase ( STRING )
CALL ucase ( STRING )
```

Arguments:

Name	Type	Dim	I/O	Description
STRING	CHAR		I/O	Input: Input character string Output: Lower/upper case character string

Algorithm:

Subroutine `lcase` shifts all upper case letters in an input character string to lower case, and leaves all other letters unchanged. Subroutine `ucase` shifts all lower case letters in an input character string to upper case, and leaves all other letters unchanged. They work for both ASCII and EBCDIC collating sequences.

Example:

```
USE booklib
CHARACTER(len=30) :: string = 'This is a Test: 12345%!?.'
WRITE (*,'(A,A)') ' Before LCASE: ', string
CALL lcase ( string )
WRITE (*,'(A,A)') ' After LCASE:  ', string
CALL ucase ( string )
WRITE (*,'(A,A)') ' After UCASE:  ', string
```

Result:

```
Before LCASE: This is a Test: 12345%!?.
After LCASE:  this is a test: 12345%!?.
After UCASE:  THIS IS A TEST: 12345%!?.
```

`lsq_fit` (Single/Double Precision Real)

Purpose: Subroutine to perform a least-squares fit of an input data set to the *n*th order polynomial: $y(x) = c_o + c_1 x + c_2 x^2 + ... + c_n x^n$.

Usage:
```
USE booklib
CALL lsq_fit ( X, Y, NVALS, ORDER, C, ERROR )
```

Arguments:

Name	Type	Dim	I/O	Description
X	R/D	NVALS	I	Values of independent variable *x*
Y	Same as X	NVALS	I	Values of dependent variable *y*
NVALS	I		I	Number of *(x,y)* measurements
ORDER	I		I	Order (highest power) of polynomial to fit
C	Same as X	0:ORDER	O	Coefficients of least squares fit
ERROR	I		O	Error flag: 0 = No error 1 = Singular equations 2 = Not enough input values 3 = Illegal polynomial order specified

Algorithm:

Subroutine `lsq_fit` performs a least squares fit of an input data set consisting of *(x,y)* pairs of data points to an *n*th order polynomial. The algorithm implemented is described in Exercise 12-6.

Example:

```
! This code fits a 3rd order polynomial to 6 input data points.
! The data points were produced by the eqn:
!        y(x) = 1. - x + x**2 - x**3
USE booklib
REAL, DIMENSION(6) :: x = (/ 0., 1., 2., 3., 4., 5. /)
REAL, DIMENSION(6) :: y = (/ 1., 0., -5., -20., -51., -104. /)
REAL, DIMENSION(0:3) :: c
INTEGER :: nvals = 6, order = 3, error
CALL lsq_fit ( x, y, nvals, order, c, error )
WRITE (*,'(A,4(F10.5,1X))') ' The coefficients are:  ', c
```

Result:

```
The coefficients are:     0.99999   -0.99999    1.00000   -1.00000
```

mat_inv (single prec. real/double prec. real/single complex/double complex)

Purpose: To invert an N x N matrix using Gauss-Jordan elimination and the maximum pivot technique.

Usage:
```
USE booklib
CALL mat_inv ( A, B, NDIM, N, ERROR )
```

Arguments:

Name	Type	Dim	I/O	Description
A	R/D/C/Z	ndim x ndim	I	Matrix to invert (May be any kind of real or complex.)
B	Same as A	ndim x ndim	O	Inverse matrix a^{-1} (Same kind as a.)
NDIM	I		I	Declared size of matrices.
N	I		I	No. of rows and columns actually used in a
ERROR	I		O	Error flag: 0 = No error 1 = No inverse found (pivot too small)

Algorithm:

This subroutine uses Gauss-Jordan elimination and the maximum pivot technique to construct the inverse of an n x n matrix. It initializes matrix B to the identity matrix, and then performs Gaussian elimination on a copy of matrix A, applying exactly the same operations to matrix B that were applied to matrix A. When the operation is over and the copy of a contains the identity matrix, B will contain matrix A^{-1}. These matrix inversion routines suffer from the same conditioning problems as Gaussian elimination routines, so the double precision version will be required for large and/or ill-conditioned matrices.

Example:

This example declares two 10 x 10 arrays a and b, initializes array a with a 2 x 2 matrix, and inverts the matrix using subroutine mat_inv.

```
USE booklib
IMPLICIT NONE
INTEGER, PARAMETER :: ndim = 10
REAL, DIMENSION(ndim,ndim) :: a, b
INTEGER :: error, i, j, n = 2
a(1,1) = 1.; a(2,1) = 2.; a(1,2) = 3.; a(2,2) = 4.
CALL mat_inv (a, b, ndim, n, error )
WRITE (*,1000) ((b(i,j), j=1, n), i=1, n)
1000 FORMAT (1X,'b = ',/,(4X,F10.4,4X,F10.4))
```

Result:

```
b =
      -2.0000        1.5000
       1.0000        -.5000
```

`nxtmul` (Integer)

Purpose: Subroutine to calculate the smallest exponent `EXP` that satisfies the expression `VALUE` <= `MUL` (= `BASE**EXP`). This calculation is useful for sizing FFT's, etc.

Usage:

```
USE booklib
CALL NXTMUL ( VALUE, BASE, EXP, MUL )
```

Arguments:

Name	Type	Dim	I/O	Description
`VALUE`	I		I	First matrix to multiply
`BASE`	I		I	Base value for the exponent
`EXP`	I		O	Smallest exponent satisfying the inequality given above
`MUL`	I		O	The next power of `BASE` that is greater than `VALUE`: `MUL` = `BASE**EXP`

Algorithm:

This subroutine calculates successive powers of the base number `BASE` until one of the exceeds the value `VALUE`. When that happens, the subroutine returns both the exponent `EXP` and the base raised to the exponent `MUL`. The subroutine is useful for calculating the next power of two when working with FFTs. For example, the call

```
CALL nxtmul ( 48, 2, EXP, MUL )
```

would return with `EXP` = 6 and `MUL` = 64, since 2**6 = 64, which is greater than 48.

Example:

This example calculates the next power of two greater than the number 997:

```
USE booklib
INTEGER :: exponent, mult
CALL nxtmul ( 997, 2, EXP=exponent, MUL=mult )
WRITE (*,1000) exponent, mult
1000 FORMAT (' Exponent = ', I6, '   Multiple = ', I6 )
```

Result:

```
Exponent =     10   Multiple =   1024
```

plot (Single Prec. Real/Double Prec. Real)

Purpose: Subroutine to print a line printer plot of a set of data points.

Usage:
```
USE booklib
CALL plot ( DATA1, NPTS, LU, MINAMP, MAXAMP )
```

Arguments:

Name	Type	Dim	I/O	Description
DATA1	R/D	NPTS	I	Data set to plot
NPTS	I		I	Number of points in input data set
LU	I		I	I/o unit to print plot on.
MINAMP	Same as DATA1		I	Smallest value to plot. Optional argument. The default value is the smallest number in the data set.
MAXAMP	Same as DATA1		I	Largest value to plot. Optional argument. The default value is the largest number in the data set.

Algorithm:

This subroutine makes a line printer plot of an input data set on the device specified by i/o unit LU.

Example:

This example plots the function sin(x) for 0 to 10 in steps of 0.5. Note that the values of *MINAMP* and *MAXAMP* are defaulted.

```
USE booklib
REAL, DIMENSION(20) :: data1
DO i = 1, 20
   data1(i) = SIN(REAL(i)/2.)
END DO
CALL plot (data1, 20, 6 )
```

Result:

```
        -0.9775                                                  0.9975
         +-------------------------------+-------------------------------+
 0.4794  |                               |              *                |
 0.8415  |                               |                          *    |
 0.9975  |                               |                               *
 0.9093  |                               |                            *  |
 0.5985  |                               |                  *            |
 0.1411  |                               |   *                           |
-0.3508  |                   *           |                               |
-0.7568  |      *                        |                               |
-0.9775  *                               |                               |
-0.9589  |*                              |                               |
-0.7055  |        *                      |                               |
-0.2794  |                      *        |                               |
 0.2151  |                               |      *                        |
 0.6570  |                               |                    *          |
 0.9380  |                               |                             * |
 0.9894  |                               |                               *
 0.7985  |                               |                         *     |
 0.4121  |                               |            *                  |
-0.0752  |                            *  |                               |
-0.5440  |             *                 |                               |
         +-------------------------------+-------------------------------+
        -0.9775                                                  0.9975

    Number of Points =            20
```

`plotxy` (Single Prec. Real/Double Prec. Real)

Purpose: Subroutine to print a line printer cross-plot a set of (*x*,*y*) data points.

Usage:
```
USE booklib
CALL plotxy ( X, Y, NPTS, LU, MINX, MAXX, MINY, MAXY, &
              NBINX, NBINY )
```

Arguments:

Name	Type	Dim	I/O	Description
X	R/D	NPTS	I	X values of points to plot.
Y	Same as X	NPTS	I	Y values of points to plot.
NPTS	I		I	Number of points in input data set
LU	I		I	I/o unit to print plot on.
MINX	Same as X		I	Smallest X value to plot. Optional argument. The default value is the smallest number in the data set.
MAXX	Same as X		I	Largest X value to plot. Optional argument. The default value is the largest number in the data set.
MINY	Same as X		I	Smallest Y value to plot. Optional argument. The default value is the smallest number in the data set.
MAXY	Same as X		I	Largest Y value to plot. Optional argument. The default value is the largest number in the data set.
NBINX	I		I	Number of *x* bins to plot. Optional argument. This value is in the range $1 \leq$ *NBINX* ≤ 65, and the default is 65.
NBINY	I		I	Number of *y* bins to plot. Optional argument. This value is in the range $1 \leq$ *NBINY* ≤ 65, and the default is 65.

Algorithm:

This subroutine makes a line printer plot of an input data set consisting of NPTS pairs of *(x,y)* values on the device specified by i/o unit LU.

Example:

This example plots the *(x,y)* pairs formed by the functions $x(t) = \sin(t)$ and $y(t) = \sin(2t)$ for $t = 0$ to 2π in steps of $\pi/20$. Note that the values of *MINX*, *MAXX*, *MINY*, and *MAXY* are defaulted.

```
USE booklib
REAL, PARAMETER :: pi = 3.141593
REAL, DIMENSION(40) :: x, y
INTEGER  :: npts = 40
DO i = 1, NPTS
   x(i) = SIN(REAL(i)*(pi/20.))
   y(i) = SIN(2.*REAL(i)*(pi/20.))
END DO
CALL plotxy ( x, y, npts, 6, NBINX = 41, NBINY = 41 )
```

Result:

```
         -1.0000              0              1.0000
          +-------------------+-------------------+
 -1.0000  |             *     *     *             |
  -.9500  |       *           |           *       |
  -.9000  |   *               |               *   |
  -.8500  |                   |                   |
  -.8000  |*                  |                  *|
  -.7500  |                   |                   |
  -.7000  *                   |                   *
  -.6500  |                   |                   |
  -.6000  |*                  |                  *|
  -.5500  |                   |                   |
  -.5000  |                   |                   |
  -.4500  |   *               |               *   |
  -.4000  |                   |                   |
  -.3500  |                   |                   |
  -.3000  |       *           |           *       |
  -.2500  |                   |                   |
  -.2000  |                   |                   |
  -.1500  |             *     |     *             |
  -.1000  |                   |                   |
  -.0500  |                   |                   |
   .0000  |                   *                   |
   .0500  |                   |                   |
   .1000  |                   |                   |
   .1500  |             *     |     *             |
   .2000  |                   |                   |
   .2500  |                   |                   |
   .3000  |       *           |           *       |
   .3500  |                   |                   |
   .4000  |                   |                   |
   .4500  |   *               |               *   |
   .5000  |                   |                   |
   .5500  |                   |                   |
   .6000  |*                  |                  *|
   .6500  |                   |                   |
   .7000  *                   |                   *
   .7500  |                   |                   |
   .8000  |*                  |                  *|
   .8500  |                   |                   |
   .9000  |   *               |               *   |
   .9500  |       *           |           *       |
  1.0000  |             *     *     *             |
          +-------------------+-------------------+
         -1.0000              0              1.0000

     Number of Points =           40
```

random_u (Single prec. real function)
random_n (Single prec. real function)
random_r (Single prec. real function)

Purpose: These procedures generate a sequence of pseudorandom numbers. Function random_u generates numbers uniformly distributed in the range [0,1). Function random_n generates a normal or Gaussian distribution with zero mean and a standard deviation of 1.0. Function random_r generates a Rayleigh distribution with a mean of 1.25 and a standard deviation equal to $\sqrt{\frac{4}{\pi}-1}$ times the mean.

Usage:
```
USE booklib
value = random_u()
value = random_n()
value = random_r()
```

Arguments:

Name	Type	Dim	I/O	Description
none				

Algorithm:

Functions random_u, random_n, and random_r generate pseudorandom number sequences of the specified distributions. These functions are convenient if a random number is needed as a part of a larger calculation.

Example:

This example uses function random_u to generate 20 random numbers between 0 and 1.

```
USE booklib
WRITE (*,*) 'Uniform random number sequence:'
DO i = 1, 4
   WRITE (*,'(1X,5F10.6)') ( random_u(), j=1,5 )
END DO
```

Result:

```
Uniform random number sequence:
  0.434307  0.454378  0.914314  0.482636  0.719896
  0.731396  0.896297  0.470225  0.098918  0.326672
  0.703519  0.228887  0.617632  0.079673  0.671855
  0.257803  0.743262  0.741481  0.954373  0.729823
```

`simul` (single prec. real/double prec. real/single complex/double complex)

Purpose: To solve a system of N simultaneous equations in N unknowns of the form AX = B, where A is an N x N matrix, and B is an N-dimensional column vector.

Usage:
```
USE booklib
CALL simul ( A, B, soln, NDIM, N, ERROR )
```

Arguments:

Name	Type	Dim	I/O	Description
A	R/D/C/Z	ndim x ndim	I	Coefficients of X
B	Same as A	ndim	I	Vector of constant terms
soln	Same as A	ndim	O	Solution to the system of equations
NDIM	I		I	Declared size of arrays.
N	I		I	Order of the system of equations.
ERROR	I		O	Error flag: 0 = No error 1 = Singular equations

Algorithm:

This subroutine uses the Gauss-Jordan method with maximum pivots for finding the solution to a system of simultaneous equations.

Example:

This example calculates the solution to a 2 x 2 set of equations, and prints the results. The arrays are declared large enough for a 4 x 4 set of equations, but only a part of each array is used in this problem.

```
USE booklib
INTEGER :: error
REAL, DIMENSION(4,4) :: a
REAL, DIMENSION(4) :: b = (/ 5., 2., 0., 0. /), soln
a(1,1) = 1.; a(1,2) = 4.; a(2,1) = 2.; a(2,2) = -3.
CALL simul ( a, b, soln, 4, 2, error )
WRITE (*,1000) soln(1), soln(2)
1000 FORMAT (' The solution is X(1) = ', F10.4, ' and X(2) = ', F10.4)
```

Result:

```
The solution is X(1) =     2.0909 and X(2) =      .7273
```

sinc (Single Prec. Real / Double Prec. Real function)

Purpose: To calculate the sinc function: $\mathrm{sinc}(x) = \sin(x) / x$.

Usage:
```
USE booklib
result = sinc(X)
```

Arguments:

Name	Type	Dim	I/O	Description
X	R/D		I	Value for which to calculate the sinc function

Algorithm:

This function calculates the function $\mathrm{sinc}(x) = \dfrac{\sin(x)}{x}$, with special handling of the computation near $x = 0$. The result is of the same kind as the input argument (single or double precision real).

Example:

This example calculates sinc(x) for an arbitrary value of x, and prints the results.

```
REAL ::x
WRITE (*,*) 'Enter value of X: '
READ (*,*) x
WRITE (*,1000) x, sinc(x)
1000 FORMAT (1X,' sinc(',F10.4,') = ', F10.4)
```

Result:

```
Enter value of X:
1.0
 SINC(    1.0000) =      .8415
```

spline_fit (single prec. real/double prec. real)

Purpose: To perform a cubic spline fit to an input data set.

Usage: `USE booklib`
`CALL spline_fit ( X, Y, N, YPP, ERROR, YP1, YPN )`

Arguments:

Name	Type	Dim	I/O	Description
X	R/D	N	I	X coefficients of data to fit. X *values must be monotonically increasing*.
Y	Same as A	N	I	Y coefficients of data to fit
N	I		I	Number of data points
YPP	Same as A	N	O	Second derivatives of curves at each point
ERROR	I		O	Error flag: 0 = No error 1 = Insufficient data
YP1	Same as A		I	First derivative at the beginning of the data set (optional argument)
YPN	Same as A		I	First derivative at the end of the data set (optional argument)

Algorithm:

This subroutine calculates a set of polynomials that fit an input data set of *(x,y)* points with a curve which is smooth in the first derivative and continuous in the second derivative, both within an interval and at its boundaries. There are two possible boundary conditions at the at the edges of the data set. If the values of first derivatives are specified at those points, then the equations will be constrained to have those values at the boundaries. If not, then the subroutine will solve for the *natural cubic spline*, which satisfies the condition that the second derivatives at the beginning and end of the data set are 0.

Example:

This example calculates the spline fit coefficients for ten points from the function $f(x) = \sin x$, and compares the resulting derivatives values calculated analytically.

```
USE booklib
INTEGER, PARAMETER :: n = 10
REAL, PARAMETER :: pi = 3.141593
REAL,DIMENSION(n) :: x, y, ypp
INTEGER :: error
REAL :: yp1, ypn
DO i = 1, n                          ! Generate input pts
   x(i) = REAL(i-1) * pi / REAL(n-1)
   y(i) = sin(x(i))
END DO
yp1 = cos(x(1))                      ! Set deriv at start
ypn = cos(x(n))                      ! Set deriv at end
call spline_fit(x,y,n,ypp,error,yp1,ypn)  ! Fit curves

WRITE (*,'(18X,A,T35,A)') 'Spline','Actual'
WRITE (*,'(T6,A,T17,A,T33,A)') 'angle','2nd deriv','2nd deriv'
DO i= 1, n
   WRITE (*,'(1X,F8.2,2F16.6)') x(i),ypp(i),-sin(x(i))
END DO
```

Result:

```
              Spline          Actual
angle        2nd deriv       2nd deriv
0.00         -0.000831        0.000000
0.35         -0.345284       -0.342020
0.70         -0.649401       -0.642788
1.05         -0.874837       -0.866026
1.40         -0.994851       -0.984808
1.75         -0.994850       -0.984808
2.09         -0.874840       -0.866025
2.44         -0.649396       -0.642787
2.79         -0.345289       -0.342020
3.14         -0.000825        0.000000
```

`spline_int` (single prec. real/double prec. real)

Purpose: To interpolate points using the cubic spline fit polynomials calculated by subroutine `spline_fit`.

Usage:
```
USE booklib
CALL spline_int (X, Y, N, YPP, X0, Y0, ERROR)
```

Arguments:

Name	Type	Dim	I/O	Description
X	R/D	N	I	X coefficients of data to fit. X *values must be monotonically increasing.*
Y	Same as A	N	I	Y coefficients of data to fit
N	I		I	Number of data points
YPP	Same as A	N	I	Second derivatives of curve at each point, calculated by subroutine `spline_fit`
X0	Same as A		I	Point at which to interpolate value
Y0	Same as A		O	Interpolated value
ERROR	I		O	Error flag: 0 - No error 1 - $\Delta x = 0$

Algorithm:

This subroutine calculates an interpolated value Y0 at position X0 using the cubic spline coefficients calculated by subroutine `spline_fit`. It first determines the pair of points which X0 lies between, and then uses the cubic equation for that particular interval to estimate Y0. (Also see procedure `interp`.)

Example:

This example interpolates a value at $x_0 = \pi/2$ from the function $f(x) = \sin x$, using cubic spline coefficients calculated from subroutine `spline_fit`.

```
USE booklib
INTEGER, PARAMETER :: n = 10
REAL, PARAMETER :: pi = 3.141593
REAL,DIMENSION(n) :: x, y, ypp
REAL :: yp1, ypn, y0
INTEGER :: error
DO i = 1, n                         ! Generate input pts
   x(i) = REAL(i-1) * pi / REAL(n-1)
   y(i) = sin(x(i))
END DO
yp1 = cos(x(1))                     ! Set deriv at start
ypn = cos(x(n))                     ! Set deriv at end
call spline_fit(x,y,n,ypp,error,yp1,ypn)  ! Fit curves
call spline_int(x,y,n,ypp,pi/2,y0,error)

WRITE (*,'(1X,2(A,F16.6))') 'Actual = ', sin(pi/2.), &
                          '  Interp = ', y0
END PROGRAM
```

Result:

```
Actual =         1.000000  Interp =         0.999960
```

`ssort` (Integer/Single Prec. Real/Double Prec. Real/Character)

Purpose: To sort an array into ascending order using the selection sort algorithm.

Usage:
```
USE booklib
CALL ssort ( ARRAY, N )
```

Arguments:

Name	Type	Dim	I/O	Description
ARRAY	I/R/D/C	N	I/O	Array to sort
N	I		I	Number of elements in array

Algorithm:

This subroutine sorts arrays into ascending order using the selection sort algorithm. This algorithm is very inefficient for large data sets, and is included here only form comparison to the better heapsort algorithm. Use the heapsort algorithm instead of this one.

Example:

This example declares an integer array `iarray` and initializes it with 15 values. It uses subroutine `ssort` to sort the array into ascending order.

```
USE booklib
INTEGER, DIMENSION(15) :: iarray = &
      (/ -100,    0,  -20,    1,  -20, &
           90, -123,  602,    5,   17, &
           91,   -4,    0,   37,  -11 /)
INTEGER :: n = 15
WRITE (*,*) ' iarray before sorting: '
WRITE (*,'(3X,5I6)') iarray
CALL ssort ( iarray, n )
WRITE (*,*) ' iarray after sorting: '
WRITE (*,'(3X,5I6)') iarray
```

Result:

```
 iarray before sorting:
    -100     0   -20     1   -20
      90  -123   602     5    17
      91    -4     0    37   -11
 iarray after sorting:
    -123  -100   -20   -20   -11
      -4     0     0     1     5
      17    37    90    91   602
```

`statistics` (Single Prec. Real/Double Prec. Real)

Purpose: To calculate the user-requested statistical values associated with a data set.

Usage:
```
USE booklib
CALL statistics ( A, N, ERROR, AVE, STD_DEV, MEDIAN )
```

Arguments:

Name	Type	Dim	I/O	Description
A	R	N	I	Data set to analyze
N	I		I	Number of data points
ERROR	I		O	Error flag: 0 = No error 1 = SD invalid (N = 1) 2 = AVE and SD invalid (N < 1)
AVE	R		O	Average of data set (optional argument)
STD_DEV	R		O	Standard deviation of data set (optional argument)
MEDIAN	R		O	Standard deviation of data set (optional argument)

Algorithm:

This subroutine calculates the average, and standard deviation, and median of an input data set, of the corresponding optional arguments are included in the subroutine call. The formulas use are:

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$\sigma = \sqrt{\frac{N\sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2}{N(N-1)}}$$

and

median = middle value of sorted data set

Example:

This example calculates average and median of a small data set.

```
USE booklib
REAL :: ave, med
INTEGER :: error
REAL, DIMENSION(7) :: a = (/ 1., 4., 1., -4., 0., 2., 6. /)
CALL statistics ( a, 7, error, AVE=ave, MEDIAN=med )
WRITE (*,1000) ave, med
1000 FORMAT (' AVE = ', F10.4,'    median = ',F10.4)
```

Result:

```
AVE =     1.4286    median =     1.0000
```