A Basic Introduction to Programming in Fortran

Course notes for EP241 & EP208

Dr. Ahmet Bingül
University of Gaziantep

With contributions from:
Dr. Andrew Beddall
Dr. Bahattin Kanber

Version 2.1
Feb 2010
Computer programming is an essential part of the work of many scientists and engineers. Fortran is a powerful language for numerical programming and is easy to learn at a basic level. This guide is intended as a first introduction to Fortran 90 (compatible with Fortran 95/2003). It is primarily written as a supplement to programming courses taken by engineering faculty students, but is also suitable for students of science and mathematics. The guide is not comprehensive; after the student has familiarised her self with the topics presented in this guide she is advised to find a more detailed and comprehensive text book.

This course is for the Engineering of Physics students in the University of Gaziantep. You can find more details of this course, program sources, and other related links on the course web page at:

http://www1.gantep.edu.tr/~bingul

A local web site dedicated to Fortran can also be found at:

http://www.fortran.gantep.edu.tr/

Türkçe: Temel Yöneriyle Fortran 90 / 95 / 2003

http://www1.gantep.edu.tr/~bingul/f95

The author can be contacted by email at:

bingul(at)gantep.edu.tr
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Algorithms, Flow Charts and Problem Solving</td>
<td>6</td>
</tr>
<tr>
<td>3. Program Structure, Data Types, Arithmetic Operators</td>
<td>8</td>
</tr>
<tr>
<td>4. Intrinsic Functions, I/O, Arrays</td>
<td>13</td>
</tr>
<tr>
<td>5. Control Statements</td>
<td>17</td>
</tr>
<tr>
<td>6. Repetitive Structures (Iteration)</td>
<td>24</td>
</tr>
<tr>
<td>7. Program Flow and Tracing</td>
<td>30</td>
</tr>
<tr>
<td>8. Formatted I/O and File Processing</td>
<td>34</td>
</tr>
<tr>
<td>9. Subprograms: Programmer-defined Functions</td>
<td>38</td>
</tr>
<tr>
<td>10. Subprograms: Programmer-defined Subroutines</td>
<td>45</td>
</tr>
<tr>
<td>11. Arrays and Array Processing</td>
<td>54</td>
</tr>
<tr>
<td>12. Selected Topics</td>
<td>67</td>
</tr>
<tr>
<td>Topics Not Covered</td>
<td>72</td>
</tr>
<tr>
<td>Appendix. List of Fortran 90 Intrinsics</td>
<td>74</td>
</tr>
</tbody>
</table>
1. Introduction

1.1 This Guide

This guide is a very basic introduction to the Fortran computer programming language. The scope of the guide includes the basics of: input/output, data types and arithmetic operations, intrinsic functions, control statements and repetitive structures, program tracing, file processing, functions and subroutines, and array processing, numerical KINDs and some interesting topics. However, some more advanced topics that are not covered in this guide are listed at the end. A list of Fortran 95 intrinsics is given in the appendix.

We have tried to make this guide concise, avoiding detailed descriptions of the language and providing only a small number of example programs in each topic. By studying the example programs carefully you should be able to realise some of the features of Fortran that are otherwise unexplained in the text. We encourage the reader to pursue further studies with a more complete Fortran text book.

1.2 Computers and Programming and Fortran

A computer is an automatic device that performs calculations, making decisions, and has capacity for storing and processing vast amounts of information. A computer has two main parts:

**Hardware (=DONANIM)**

Hardware is the electronic and mechanical parts of the computer (see Figure 1.1). Hardware includes:

- **Input Units**: Keyboard, Mouse, Scanner
- **Process Units**: CPU, Central Processing Unit. This coordinates the operation of computer system and performs arithmetic logic operations. RAM, Random Access Memory HDD, Hard Disc Driver FDD, Floppy Disc Driver CD-ROM, Compact Disc – Read Only Memory
- **Output Units**: Monitor, Printer, Plotter, Scanner, Modem, Speaker
Software (YAZILIM)
The software consists of all the programs running on the computer. It includes:

*Operating System (OS)* is a program written by manufacturer (e.g. Microsoft). It interface between computer and user. All the programs run under the OS. Examples are: MS-DOS, Windows, Unix, Linux, BEOS.

*Compilers* can also be called translator. Very computer language has its own compiler. The compiler translates the statements of program written in a high level language into a low level language, the machine code. Examples are: Fortran, C, C++, Java, Pascal, Basic.

*Application Programs* are programs written by the users for their own needs. For example: Word, Excel, Logo, AutoCAD, Flash.

Science and engineering has always been closely tied to the evolution of new tools and technologies. Computer technology continues to provide powerful new tools in all areas of science and engineering. The strength of the computer lies in its ability to manipulate and store data. The speed at which computers can manipulate data, and the amount of data they can store, has increased dramatically over the years doubling about every 18 months! (Moore's law). Although the computer has already made an enormous impact on science and engineering and of course elsewhere (such as mathematics and economics) its potential is only just beginning to be tapped. A knowledge of using and programming computers is essential for scientists and engineers.

1.3 Creating and Running a Program

**Editing, Compiling, and Running**
To create and execute a program you need to invoke three environments; the first is the editor environment where you will create the program source, the second is the compilation environment where your source program will be converted into a machine language program, the third is the execution environment where your program will be run. In this guide it is assumed that you will invoke these three environments on a local Linux server in the University of Gaziantep. For this, three easy to use commands are available:
Steps of Program Development

A program consists of a set of instructions written by the programmer. Normally a high level language (such as Basic, C, or Fortran) is used to create a source code written with English-like expressions, for example:

```fortran
REAL :: A, B, C
READ *, A, B
C = A + B
PRINT *, "the sum is ", C
END
```

A compiler is then used to translate the source code into machine code (a low level language), the compiled code is called the object code. The object code may require an additional stage where it is linked with other object code that readies the program for execution. The machine code created by the linker is called the executable code or executable program. Instructions in the program are finally executed when the executable program is executed (run). During the stages of compilation, linking, and running, error messages may occur that require the programmer to make corrections to the program source (debugging). The cycle of modifying the source code, compiling, linking, and running continues until the program is complete and free of errors. This cycle is illustrated in the Figure 1.2.

![Figure 1.2: Steps of program development. Programming is often an iterative process of writing, compiling, running a program.](image)

Examples of various types of errors are given below.
Compile-time errors
These are errors that occur during compilation of the source code into object code. They are usually due to incorrect usage of the programming language, for example:

```
READ *, A, B
C = A + B
PRINT *, C
END
```

Compilation of this program results in a compile-time error something like:

```
PRINT *, C
1
Error: Unclassifiable statement at (1)
```

`PRINT` is a misspelling of the output statement `PRINT`. This error is corrected by replacing `PRINT` with `PRINT` in the source code and then recompiling the program, this process is called debugging. Object code is only created when there are no detected compile-time errors.

Compile-time warnings may also occur, these provide the programmer with advice about parts of the program that may be using non-standard syntax or that may potentially cause errors. Compile-time warnings do not prevent the creation of object code. Example:

```
REAL :: C
PRINT *, C
END
```

Compilation of this program may result in the compile-time warning something like:

```
REAL :: C
1
Warning (113): Variable 'c' at (1) is used but not set
```

An executable is created, but will give an undetermined result.

Link-time errors
These are errors that occur during the linking stage. They result when, for example, an external object required by the program cannot be found. Example:

```
PRINT *,SIN(4.3)
PRINT *,ARCSIN(.78)
END
```

Compilation of this program results in a link-time error something like:

```
PRINT *,ARCS
1
Error: Function 'arcsin' at (1) has no implicit type
```

In this case the program is compiled into object code but then fails to link the external function `ARCSIN` that does not exist in any library known to the compiler. When a link-time error occurs the executable is not created. This program may be corrected by replacing in the source code the statement `ARCSIN` with `ASIN` (the standard Fortran statement representing the inverse sine of a number) or by providing the reference subprogram. Again link-time warnings may also occur.
Run-time errors
These are errors that occur during the execution of the program (when the program is running). Such errors usually occur when the logic of the program is at fault or when an unexpected input is given (unexpected inputs or faulty logic does not necessarily result in run-time error messages, such programming errors should be detected by rigorously testing your program). When a run-time error occurs the program terminates with an appropriate error message. Example:

```
REAL :: A(5)
INTEGER :: I
DO I=1,6
   A(I)=I**2
END DO
PRINT *, A
END
```

This program compiles and links without errors, but when executed may result in the program terminating with a run-time error something like:

```
Fortran runtime error: Array element out of bounds: 6 in (1:5), dim=1
```

Run-time errors result from run-time checking that the compiler builds into the object code. Compiler options can be used to switch on and off various run-time checks, compile-time warnings, code optimisation, and various other compiler features.

1.4 Questions

[1]. What compiler options are you using when you compile your Fortran source?
[2]. How can you find out what other compiler options are available and switch them on and off?

Notes
Use this section to note down commands and procedures for editing, compiling, and running your programs on your computer platform.
2. Algorithms, Flow Charts and Problem Solving

2.1 Introduction

In this section we introduce ideas about problem solving with computers; we make use of flowcharts, algorithms, and consider the importance of defining a problem sufficiently and what assumptions we may make during the solution.

Consider the calculation of the twist factor of a yarn. Twist Factor, $T_f$, of a yarn is given by:

$$T_f = N \sqrt{\frac{m}{1000}}$$

where $N$ (turn/m) is the number of twist of a yarn per unit length and $m$ is measured in tex (a yarn count standard) that is mass in grams of a yarn whose length is 1 km. Write a Fortran program to calculate twist factor of a yarn for given $N$ and $m$.

A solution might look something like `twist.f90`, the key section is below:

```fortran
PROGRAM Twist_Factor
IMPLICIT NONE
REAL :: Tf, m
INTEGER :: N
PRINT *, "Input the value of N and m"
READ *, N, m
Tf = N * SQRT(m/1000.0)
PRINT *, "The twist factor is", Tf
END PROGRAM Twist_Factor
```

But maybe it is not as simple as this: was the problem defined clearly? what assumptions did we make in the solution, are they valid? This is discussed in detail in the lecture; some notes are given below.

2.2 Problem Solving

Problem solving with computers involves several steps:

1. Clearly define the problem.
2. Analyse the problem and formulate a method to solve it (see also “validation”).
3. Describe the solution in the form of an algorithm.
4. Draw a flowchart of the algorithm.
5. Write the computer program.
6. Compile and run the program (debugging).
7. Test the program (debugging) (see also “verification”).
8. Interpretation of results.
Verification and Validation

If the program has an important application, for example to calculate student grades or guide a rocket, then it is important to test the program to make sure it does what the programmer intends it to do and that it is actually a valid solution to the problem. The tests are commonly divided as follows:

*Verification* verify that program does what you intended it to do; steps 7(8) above attempt to do this.

*Validation* does the program actual solve the original problem i.e. is it valid? This goes back to steps 1 and 2 - if you get these steps wrong then your program is not a valid solution.

2.3 Algorithms

The algorithm gives a step-by-step description of the solution. This may be written in a non-formal language and structure. An example is given in the lecture.

2.4 Flow Charts

A flow chart gives the logical flow of the solution in a diagrammatic form, and provides a plan from which the computer program can be written. The logical flow of an algorithm can be seen by tracing through the flowchart. Some standard symbols used in the formation of flow charts are given below.

- An oval is used to indicate the beginning or end of an algorithm.

- A parallelogram indicates the input or output of information.

- A rectangle indicates a computation, with the result of the computation assigned to a variable.

- A diamond indicates a point where a decision is made.

- A hexagon indicates the beginning of the repetition structure.

- A double lined rectangle is used at a point where a subprogram is used.

- An arrow indicates the direction of flow of the algorithm. Circles with arrows connect the flowchart between pages.
3. Program Structure, Data Types, Arithmetic Operators

3.1 Introduction

In this section, you will learn the basic structure of Fortran 90, how Fortran 90 handles different data types, and study arithmetic operations in Fortran.

3.2 Fortran Program Structure

The basic program structure used in this guide is:

```fortran
PROGRAM A_Program_Name
    ! Comment explaining the purpose of the program
    IMPLICIT NONE
    REAL :: Var1, Var2  ! a declaration part...
    INTEGER :: Var3, Var4
    Var1 = 0.          ! an initialisation part ...
    Var2 = 0.
    Var3 = 0.
    Var4 = 0.
    ... some operations ...
    PRINT *, some output
END PROGRAM A_Program_Name
```

You are free to indent with spaces and add empty lines as you wish, the aim is to improve the readability of the program source.

3.3 Data Types and Constants

A key component of a program is the use of objects that store data. There are five data types: REAL, INTEGER, COMPLEX, CHARACTER, LOGICAL. Most commonly used in numerical work are type REAL and type INTEGER. In the following example program we have objects named \texttt{A}, \texttt{v}, and \texttt{Momentum} that are declared to store type real data (numbers with decimal points), and objects named \texttt{Count}, \texttt{Missed}, and \texttt{Decay}, that are declared to store type integer data, and an object named \texttt{Month} declared to store type character data. All these objects are called variables as their values can be changed (varied) during program execution.
Note that in the assignment \( V = 15.6 \times 10^3 \) the expression \( 15.6E3 \) in Fortran represents the value \( 15.6 \times 10^3 = 15600 \). The output of this program (from the \texttt{PRINT} statement) is:

\[
3870.968 \quad 13569 \quad \text{January}
\]

\textit{Named constants} are declared with the \texttt{PARAMETER} attribute. Such data is assigned a value at declaration and cannot be changed during program execution; for example:

Example execution:

\begin{verbatim}
Type the length in feet
12.0
12.000000 feet = 3.657600 metres.
\end{verbatim}

Here, identifier \texttt{FtoM} (an object that can store a real value) is declared as a constant (the \texttt{PARAMETER} attribute). The value of \texttt{FtoM} is defined in its declaration and cannot be change during the execution of the program. In this program it is not necessary to give identifier \texttt{FtoM} the \texttt{PARAMETER} attribute; but, as we do not intend the value of \texttt{FtoM} to change during program execution it is good programming practice to declare it as a constant.
3.4 Arithmetic Operations

Operators
The symbols ( ) * / + - ** are used in arithmetic operations. They represent parenthesis, multiplication, division, addition, subtraction and exponentiation, respectively.

Priority Rules
Arithmetic operations follow the normal priority; proceeding left to right, with exponentiation performed first, followed by multiplication and division, and finally addition and subtraction. Parenthesis can be used to control priority.

Mixed-mode, and integer operations
If integers and reals are mixed in arithmetic operations the result is a real. Operations involving only reals yield a type real result. Operations involving only integers yield a type integer result. Be especially careful when dividing two integers - the result is truncated to an integer; for example, $3/2 = 1$, and $1/2 = 0$. This is illustrated in the program below.

```
PROGRAM Operations
  !---------------------------------------------------------------------
  ! Program to test integer and mixed mode operations
  !---------------------------------------------------------------------
  IMPLICIT NONE
  REAL :: A, B, C
  INTEGER :: I, J, K
  A = 3.
  B = 4.
  I = 5
  J = 3
  C = A + I / J
  K = A / I + 2 * B / J
  PRINT *, C, K
END PROGRAM Operations
```

The output of this program is

```
4.000000   3
```

and is explained as follows:

Type real object $c$ is assigned the result of $A+I/J = 3.0+5/3 = 3.0+1 = 4.0$. Here, the result of the integer operation $5/3$ is an integer and so $1.66666$ is truncated to $1$; the value of $c$ is output.

Type integer object $k$ is assigned the result of $A/I+2*B/J = 3.0/5+2*4.0/3$ which, using the priority rules evaluates as $(3.0/5)+2*4.0/3$. Remember that mixed-mode arithmetic results in a type real value and so the result is $0.6+2.66666 = 3.266666$. The assignment however is to a type integer object and so the value is truncated to $3$; the value of $k$ is output.
It is advisable to avoid integer division, get into the habit of using the following forms in operations:

- Real constants should always be written with a decimal point
e.g., instead of \( \text{X=A/5} \) write \( \text{X=A/5.} \)
- Integer identifiers should be converted to real type in operations
e.g., instead of \( \text{X=A/N} \) write \( \text{X=A/REAL(N)} \) if that is what you mean.

If \( \text{A} \) is type real then both these cases are not necessary - but it is good programming practice to make a habit of using these forms (write explicitly what you mean).

**Long arithmetic expressions**

When writing long arithmetic expressions it can be useful to break them down into constituent parts. For example the expression:

\[
Z = \frac{(X^2 + 2.X + 3.)/(5.+Y)^{0.5} - ((15. - 77.*X^3)/Y^{1.5})^{0.5}}{(X^2 - 4.*X*Y - 5.*X^{(-0.8)})}
\]

can be written more clearly (and carefully) as

\[
\begin{align*}
A &= (X^2 + 2.X + 3.) / (5.+Y)^{0.5} \\
B &= (15.-77.*X^3) / Y^{1.5} \\
C &= X^2 - 4.*X*Y - 5.*X^{(-0.8)} \\
Z &= (A - B^{0.5}) / C
\end{align*}
\]

This is implemented in the program below:

```fortran
PROGRAM Equation
    IMPLICIT NONE
    REAL :: X = 0.2, Y = 1.9
    REAL :: A, B, C, Z

    A = (X**2+2.*X+3.) / (5.+Y)**0.5
    B = (15.-77.*X**3) / Y**1.5
    C = X**2 - 4.*X*Y - 5.*X**(-0.8)
    Z = ( A - B**0.5 ) / C

    PRINT *, Z
END PROGRAM Equation
```

If you don't want to separate an expression into parts you can use & operator as follows:

\[
Z = \left( \frac{(X^2 + 2.X + 3.)/(5.+Y)^{0.5} - \frac{(15. - 77.*X^3)/Y^{1.5})^{0.5}}{(X^2 - 4.*X*Y - 5.*X^{(-0.8)})} \right)
\]
3.5 Declaring and Initialising Variables

Again, it is good programming practice to get into the habit of:

- Always use \texttt{IMPLICIT NONE}. This forces you to declare all variables you use and so avoids the potential of using a misspelled identifier.
- Always initialise variables; an uninitialised variable will take, depending on the particular compiler or compiler options you are using, a value which is either zero or an unpredictable value. You can remove such uncertainties by initialising all variables you declare.

For example:

\begin{verbatim}
INTEGER :: K
REAL :: S
K = 0
S = 0.
\end{verbatim}

or

\begin{verbatim}
INTEGER :: K = 0
REAL :: S = 0.
\end{verbatim}

Note that the second form in subprograms gives the variables the \texttt{SAVE} attribute (see other texts for an explanation).
4. Intrinsic Functions, I/O, Arrays

4.1 Introduction

In this section, you will learn some Fortran intrinsic mathematical functions such as $\sin$, $\exp$, $\abs$, the basics of the input/output (I/O) and an introduction to arrays (arrays are covered in more detail in a later section).

4.2 Intrinsic Functions

These are functions that are built in to the compiler. Some are shown in the table below (the appendix at the end of this guide for a full list of Fortran 90 intrinsics):

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG(x)</td>
<td>Natural logarithm; ln(x)</td>
<td>$Y = \log(2./X)$</td>
</tr>
<tr>
<td>LOG10(x)</td>
<td>Logarithm for base 10; $\log_{10}(x)$</td>
<td>$Y = \log10(X/3.5)$</td>
</tr>
<tr>
<td>COS(x)</td>
<td>Cosine of a number in radians</td>
<td>$Y = \cos(X)$</td>
</tr>
<tr>
<td>ATAN(x)</td>
<td>Angle in radian whose tangent is x</td>
<td>$R = \tan(Y/X)$</td>
</tr>
<tr>
<td>EXP(x)</td>
<td>Natural exponent $e^x$</td>
<td>$G = \exp(-(X-M)/S)^2/2.$</td>
</tr>
<tr>
<td>SQRT(x)</td>
<td>Square-root of a real value</td>
<td>$\text{Root} = \sqrt{Y}$</td>
</tr>
<tr>
<td>INT(x)</td>
<td>Truncate to an integer</td>
<td>$K = \text{INT}(X)$</td>
</tr>
<tr>
<td>NINT(x)</td>
<td>Nearest integer of a real value</td>
<td>$K = \text{NINT}(X)$</td>
</tr>
<tr>
<td>MOD(x,y)</td>
<td>$x \mod y$</td>
<td>$\text{Remainder} = \text{MOD}(X,5)$</td>
</tr>
<tr>
<td>ABS(x)</td>
<td>Absolute value of $x$</td>
<td>$Y = \abs(X)$</td>
</tr>
</tbody>
</table>

The Gaussian probability function is defined as:

$$G(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-(x-m)^2/2\sigma^2}$$

This can be written using intrinsic functions as follows:

$$G = \exp\left(-0.5\left((X-M)/S\right)^2/2\right) / (S*\sqrt{2*3.141593})$$

The test for the number, $x$, even or odd can be made by:

$$R = \text{MOD}(X,2)$$

$\text{MOD}(X,2)$ returns an integer value which is 0 or 1.

if $R=0$ then the number, $x$, is even
otherwise $R=1$ the number is odd.
Example 4.1

In the following program the values for the position $x$, mean $m$, and standard deviation $\sigma$ are input, and value of the gaussian probability function is output.

```fortran
PROGRAM Gaussian
!---------------------------------------------------------------
! The Gaussian probability density function is symmetric about
! and maximum at $X = M$ and has a standard deviation of $S$. The
! integrated function is normalised to unity.
!---------------------------------------------------------------
IMPLICIT NONE
REAL, PARAMETER :: TwoPi = 6.283185
REAL :: X, M, S  ! inputs
REAL :: G        ! output
PRINT *, "Input the position X, mean M, and sigma S"
READ *, X, M, S
G = EXP( -0.5*((X-M)/S)**2 ) / (S*SQRT(TwoPi))
PRINT *, G
END PROGRAM Gaussian
```

Note that the symbol $\sigma$ (sigma) is not permitted in a Fortran program (only characters from the standard ASCII character set are permitted) and so this is replaced with the letter $S$ which in this case is short for $sigma$.

Example execution:

```
Input the position X, mean M, and sigma S
-0.65
1.21
2.6
0.1187972
```

4.3 Input/Output (I/O)

The idea of input and output devices is introduced very briefly. Inputs for a Fortran program are usually from a keyboard or a file. Outputs are normally to a screen or a file:

```
INPUTS
Keyboard
File

Program
READ ...
PRINT ...
WRITE ...

OUTPUTS
Screen
File
```

Two pairs of I/O statements are used, the first is for I/O involving the "standard" keyboard/screen, and the second for I/O involving files.
Keyboard/Screen I/O statements:

**READ** format specifier, input list

**PRINT** format specifier, output list

where *format specifier* specifies the format of the output.

Examples:

```
READ *, A
PRINT *, A
```

Here *A* is input and output in a "free format", i.e. the compiler decides what the format is depending on the type of data.

```
PRINT '(F6.3)', A
```

Here the format of the output is given as:

- *F* means a real value
- *6* means 6 digits (including the decimal place)
- *3* means 3 decimal places.

For example 63.78953 will be output as **63.790** (the value is rounded to the nearest decimal place).

File I/O statements:

**READ** (unit number, format specifier) input list

**WRITE** (unit number, format specifier) output list

where *unit number* specifies a number given to the file.

4.4 Introduction to Arrays

A basic introduction to arrays is given here, more details are covered in Section 10. An array is a group of variables or constants, all of the same type, which is referred to by a single name. If the following can represent a single value:

```
REAL :: Mass
```

A set of 5 values can be represented by the array

```
REAL :: Mass(5)
```

The 5 elements of the array can be assigned as follows:

```
Mass(1) = 8.471
Mass(2) = 3.683
Mass(3) = 9.107
Mass(4) = 4.739
Mass(5) = 3.918
```

or more concisely using an array constant:
Consider the following program section;

```fortran
REAL :: Mass(5)
Mass = (/ 8.471, 3.683, 9.107, 4.739, 3.918 /)
PRINT *, Mass
```

The output is:

```
8.471000  3.683000  9.107000  4.739000  3.918000
```

We can operate on individual elements, for example

```fortran
Weight = Mass(5) * 9.81
```

here, Weight is a scalar. Or we can operate on a whole array in a single statement:

```fortran
Weight = Mass * 9.81
```

Here both Weight and Mass are arrays with 5 elements (the two arrays must conform). Consider the following program section;

```fortran
REAL :: Mass(5), Weight(5)
Mass = (/ 8.471, 3.683, 9.107, 4.739, 3.918 /)
Weight = Mass * 9.81
PRINT *, Mass
PRINT *, Weight
```

The above program section is implemented in the example program below; operations involving a whole array are indicated in **bold**.

**Example 4.2**

```fortran
PROGRAM Weights
!---------------------------------------------------------------------
! Given an array of masses, this program computes
! a second array of weights using a "whole array
! assignment". The arrays must have the same
! number of elements.
!---------------------------------------------------------------------
IMPLICIT NONE
REAL, PARAMETER :: g = 9.81
REAL :: Mass(5), Weight(5)
Mass = (/ 8.471, 3.683, 9.107, 4.739, 3.918 /) ! Assign the mass values
Weight = Mass * g                                ! Compute the weights
PRINT *, Mass
PRINT *, Weight
END PROGRAM Weights
```

The output is:

```
8.471000  3.683000  9.107000  4.739000  3.918000
83.10051  36.13023  89.33968  46.48959  38.43558
```
5. Control Statements

5.1 Introduction

Control statements allow us to make decisions - the program takes one course of action or another depending on the value of a variable. Here we introduce four constructs and understand how to use them: the simple IF construct, the block IF construct, the IF-ELSE construct, IF-ELSE IF-ELSE construct and CASE construct.

5.2 Relational Operators and their Compound Forms

Control statements use relation operators; there are six relational operators as follows:

- `<` less than
- `<=` less than or equal to
- `>` greater than
- `>=` greater than or equal to
- `==` equal to. Note that this is not the same as the assignment operator `=`
- `/=` not equal to

Relational expressions can therefore be formed, for example

```
A < B
A == 5
B >= 1.
```

Compound relation expressions can be formed using the `AND`, and `OR`, (and other) operators; for example:

```
A < B .AND. C==5.
this statement is true if both A is less than B, and C is equal to 5.

A >= 0 .OR. B > 1.
this statement is true if either A is greater or equal to zero, or B is greater than one.
```

5.3 The Simple IF Construct

```
IF ( a simple or compound logical expression ) a single statement
```

For example:

```
IF ( X > 0 ) Y = SQRT(X)
```

5.4 The Block IF Construct

```
IF ( a simple or compound logical expression ) THEN
  statement 1
  statement 2
  ...
END IF
```
For example:

```
IF ( Poem == "Yes" ) THEN
    PRINT *, "A computer, to print out a fact,"
    PRINT *, "Will divide, multiply, and subtract."
    PRINT *, "But this output can be"
    PRINT *, "No more than debris,"
    PRINT *, "If the input was short of exact."
    PRINT *, "-- Gigo"
END IF
```

### 5.5 The IF-ELSE Construct

```
IF ( a simple or compound logical expression ) THEN
    statement sequence 1
ELSE
    statement sequence 2
END IF
```

For example:

```
IF (A < B) THEN
    Result = A/B
    PRINT *, "x = ", Result
ELSE
    Result = B/A
    PRINT *, "1/x = ", Result
END IF
```

### Nesting

You can nest IF ELSE construct such that:

```
IF ( a simple or compound logical expression ) THEN
    statement sequence 1
    IF ( a simple or compound logical expression ) THEN
        statement sequence 2
    ELSE
        statement sequence 3
    END IF
ELSE
    statement sequence 4
ELSE
    IF ( a simple or compound logical expression ) THEN
        statement sequence 5
    ELSE
        statement sequence 6
    END IF
END IF
```

### 5.6 IF-ELSE IF Construct

The selection structures considered thus far have involved selecting one of two alternatives. It is also possible to use the IF construct to design selection structures that contain more than two alternatives:
IF ( a simple or compound logical expression ) THEN
  statement sequence 1
ELSE IF ( a simple or compound logical expression ) THEN
  statement sequence 2
ELSE IF ( a simple or compound logical expression ) THEN
  statement sequence 3
  ...
ELSE IF ( a simple or compound logical expression ) THEN
  statement sequence n-1
ELSE
  statement sequence n
END IF

Example 5.1 consider the following piecewise function:

\[
f(x) = \begin{cases} 
-x & \text{if } x \leq 0 \\
x^2 & \text{if } 0 < x < 1 \\
1 & \text{if } x \geq 1 
\end{cases}
\]

To evaluate the function, following program can be implemented:

```fortran
PROGRAM Composite_Function
IMPLICIT NONE
REAL :: x,F

PRINT *, "Input the value of x"
READ *, x

IF (x <= 0) THEN
  F = -x
ELSE IF (x>0 .AND. x<1) THEN
  F = x**2
ELSE
  F = 1.0
END IF
PRINT *,x,F

PROGRAM Composite_Function
```

Example executions:

```
Input the value of x
-4.0
-4.000000  4.000000

Input the value of x
5
5.000000  1.000000
```
5.7 CASE Construct

In this section the CASE construct which is an alternative of IF-ELSE IF construct and useful for implementing some selection structures. A CASE construct has the following form:

```
SELECT CASE (selector) THEN
    CASE (label list 1)
        statement sequence 1
    CASE (label list 2)
        statement sequence 2
    .
    .
    CASE (label list n)
        statement sequence n
END SELECT
```

where
- `selector` is an integer, character or logical expression
- `label list i` is a list of one or more possible values of the selector and the values in this list may have any of the forms:
  - `Value` denotes a single value
  - `value1 : value2` denotes from `value1` to `value2`
  - `value1 :` denotes the set of all values greater than or equal to `value1`
  - `:` `value2` denotes the set of all values less than or equal to `value2`

For example, following CASE construct can be used to display the class name that corresponds to a numeric class code:

```
SELECT CASE(ClassCode)
    CASE(1)
        PRINT *,"Freshman"
    CASE(2)
        PRINT *,"Sophmore"
    CASE(3)
        PRINT *,"Junior"
    CASE(4)
        PRINT *,"Graduate"
    CASE DEFAULT
        PRINT *,"Illegal class code", ClassCode
END SELECT
```

Note that the use of CASE DEFAULT statement to display an error message in case the value of the selector `ClassCode` is none of 1,2,3,4 or 5. Although the CASE DEFAULT statement can be placed anywhere in the list of CASE statement.
Example 5.2 Finding a Leap Year

A leap year is a year in which one extra day (February 29) is added to the regular calendar. Most of us know that the leap years are the years that are divisible by 4. For example 1992 and 1996 are leap years. Most people, however, do not know that there is an exception to this rule: centennial years are not leap years. For example, 1800 and 1900 were not leap years. Furthermore, there is an exception to the exception: centennial years which are divisible by 400 are leap years. Thus 2000 is a leap year. The following program checks if the given year is leap or not.

```fortran
PROGRAM Leap_Year
!
!-----------------------------------------
! Finding a leap year
!-----------------------------------------
IMPLICIT NONE
INTEGER :: Y

PRINT *,"Enter a year"
READ *,Y

IF( MOD(Y,4) == 0 .AND. MOD(Y,100) /= 0 .OR.  &
    MOD(Y,400) == 0) THEN
    PRINT *,Year," is a leap year."
ELSE
    PRINT *,Year," is not a leap year."
END IF

END PROGRAM Leap_Year
```

Example 5.3: Ratio of two numbers

```fortran
PROGRAM Fractional_Ratio
!
!-------------------
! The ratio of two numbers such that it is positive and a fraction.
!-------------------
IMPLICIT NONE
REAL A, B, Ratio

PRINT *, "Input two numbers."
READ *, A, B

A=ABS(A); B=ABS(B)

IF (A < B) THEN
    Ratio = A/B
ELSE
    Ratio = B/A
END IF

PRINT *, "The ratio is ", Ratio

END PROGRAM Fractional_Ratio
```
Example 5.4 Grade calculation

```
PROGRAM Grade_Calculation
!
! Grade calculation from the weighted average of three exams. The first, second and final exam scores are weighted by 0.3, 0.3, and 0.4 respectively. The average score is converted to a grade from the grade table (right).
!
IMPLICIT NONE
REAL :: MT1, MT2, Final, Average
CHARACTER :: Grade*2

PRINT *, "Enter the three exam scores (%)"
READ *, MT1, MT2, Final
Average = 0.3*MT1 + 0.3*MT2 + 0.4*Final
PRINT '(A22,F5.1,A1)', "The weighted score is ", Average, "%"
IF (Average < 40.) Grade="FF"
IF (Average >= 40.) Grade="FD"
IF (Average >= 50.) Grade="DD"
IF (Average >= 60.) Grade="DC"
IF (Average >= 70.) Grade="CC"
IF (Average >= 75.) Grade="CB"
IF (Average >= 80.) Grade="BB"
IF (Average >= 85.) Grade="BA"
IF (Average >= 90.) Grade="AA"
PRINT *, "The grade is ", Grade
END PROGRAM Grade_Calculation
```

In this example the variable Grade maybe assigned and reassign a number of times.

Example execution:

```
Enter the three exam scores (%)
56
78
81
The weighted score is 72.6%
The grade is CC
```

Example 5.4 can also be written by using IF-ELSE IF or CASE construct. In Example 5.5 the grade calculation is done by CASE construct.
Example 5.5 Grade calculation

PROGRAM Grade_Calculation
!--------------------------------------------------
! Grade calculation from the weighted average of three exams. The first, second and final exam scores are respectively weighted by 0.3, 0.3, and 0.4 respectively. The average score is converted to a grade from the grade table (right).
!--------------------------------------------------
IMPLICIT NONE
REAL :: MT1, MT2, Final, Average
CHARACTER :: Grade*2

PRINT *, "Enter the three exam scores (%)"
READ *, MT1, MT2, Final
Average = 0.3*MT1 + 0.3*MT2 + 0.4*Final
PRINT '(A22,F5.1,A1)', "The weighted score is ", Average, "%"

SELECT CASE(NINT(Average)) ! convert Average to nearest integer
CASE(:,39);   Grade="FF"
CASE(40:49); Grade="FD"
CASE(50:59); Grade="DD"
CASE(60:69); Grade="DC"
CASE(70:74); Grade="CC"
CASE(75:79); Grade="CB"
CASE(80:84); Grade="BB"
CASE(85:89); Grade="BA"
CASE(90:);   Grade="AA"
END SELECT

PRINT *, "The grade is ", Grade

END PROGRAM Grade_Calculation
6. Repetitive Structures (Iteration)

6.1 Introduction

We can cause a program to repeat sections of statements (iterate) by using the do loop construct. There are two forms; the do loop with a counter, and the endless do loop.

6.2 The do loop with a counter

In this type of looping, the repetition is controlled by a counter. This has the general form:

\[
\text{DO } \text{counter} = \text{initial value}, \text{ limit, step size} \\
\quad \text{statement sequence} \\
\text{END DO}
\]

For example

\[
\text{DO I = 4, 12, 2} \\
\quad \text{PRINT *, I, I**2, I**3} \\
\text{END DO}
\]

gives:

\[
\begin{align*}
4 & \quad 16 & \quad 64 \\
6 & \quad 36 & \quad 216 \\
8 & \quad 64 & \quad 512 \\
10 & \quad 100 & \quad 1000 \\
12 & \quad 144 & \quad 1728
\end{align*}
\]

The counter variable I takes values starting from 4 and ending at 12 with increments of 2 (the step size) in between. The number of iterations in this loop is therefore 5. The do loop parameters counter, initial value, limit, and step size must all be type integer. To create a loop with a type real counter we can use, for example, something like the following scheme.

\[
\text{DO I = 0, 10, 2} \\
\quad R = 0.1*\text{REAL(I)} \\
\quad \text{PRINT *, R, R**2, R**3} \\
\text{END DO}
\]

Here, the real variable R is derived from the integer counter I; the result is:

\[
\begin{align*}
0.000000E+00 & \quad 0.000000E+00 & \quad 0.000000E+00 \\
0.200000 & \quad 4.000000E-02 & \quad 8.000000E-03 \\
0.400000 & \quad 0.160000 & \quad 6.400000E-02 \\
0.600000 & \quad 0.360000 & \quad 0.216000 \\
0.800000 & \quad 0.640000 & \quad 0.512000 \\
1.000000 & \quad 1.000000 & \quad 1.000000
\end{align*}
\]
6.2 General DO loops

In this type of looping, the repetition is controlled by a logical expression.

**DO-EXIT Construct**

This has the general form:

```
DO
  statement sequence 1
  IF ( a simple or compound logical expression ) EXIT
  statement sequence 2
END DO
```

The loop is repeated until the condition (a logical expression) in IF statement becomes false. If the condition is true, the loop is terminated by EXIT statement. This is useful for when we do not know how many iterations will be required.

**Example of the use of a DO-EXIT construct:**

```
DO
  PRINT *, "Input a positive number."
  READ *, A
  IF ( A >= 0. ) EXIT
  PRINT *, "That is not positive! try again."
END DO
```

This program section loops until a positive number in input.

**Example execution:**

```
Input a positive number.
-34.2
That is not positive! try again.
Input a positive number.
-1
That is not positive! try again.
Input a positive number.
3.4
the loop terminates
```

The following program section outputs the even numbers 10, 8, ..., 2 and their squares:

```
N=10
DO
  PRINT *,N,N**2
  IF(N<4) EXIT
  N=N-2
END DO
```

**Output:**

```
10     100
 8      64
 6      36
 4      16
 2       4
```
DO–CYCLE Construct
This has the general form:

```
DO  
   statement sequence 1
   IF ( a simple or compound logical expression ) CYCLE  
   statement sequence 2
   IF ( a simple or compound logical expression ) EXIT  
   statement sequence 3
END DO
```

When the CYCLE statement is executed control goes back to the top of the loop. When the EXIT statement is executed control goes to the end of the loop and the loop terminates.

Example of the use of an DO–CYCLE construct:

```
DO  
   READ *,X
   IF (X == 0) CYCLE
   F = 1.0/X
   PRINT *,X,F
   IF (X<0) EXIT
END DO
```

This program section reads a value $x$ from the keyboard and outputs a value of $x$ and $1/x$ if $x$ is not equal to zero, while $x>0$.

DO–WHILE Construct
This has the general form:

```
DO WHILE( a simple or compound logical expression )  
   ...  
   statement sequence  
   ...  
END DO
```

The logical expression is executed during the condition is true, otherwise the loop is skipped.

Example of the use of an DO–WHILE construct:

```
N=10
DO WHILE (N>=2)  
   PRINT *,N,N**2
   N=N-2
END DO
```

The program section given above will output the even numbers 10, 8, ..., 2 and their squares while $N>=2$. The results is same as the program section given in page 25.
6.4 Endless or Infinite DO loops

The logical expressions given in `DO-EXIT`, `DO-CYCLE` and `DO-WHILE` can result in an infinite (endless) loop under proper conditions. It is normal to provide the user with some way out of a loop; if you program loops infinitely then you can break out with the key sequence: Ctrl-C.

What can you say about the output of the following program sections?

```fortran
DO WHILE(2>1)
   PRINT *, "Engineering"
END DO

I=1
DO
   IF(I==0) EXIT
   PRINT *, "Engineering"
END DO

Y = 2.0
DO
   PRINT *, "Engineering"
   IF(Y<Y**2) EXIT
   Y=Y+0.0002
END DO
```

Each program sections will output the lines:

Engineering
Engineering
Engineering
Engineering
.
.
.
Example 6.1 Calculating n! (n factorial)

```fortran
PROGRAM N_Factorial
  ! Program to compute the factorial of a positive integer.
  ! It is assumed that N is positive (N >= 0)
  !------------------------------------------------------------------
  IMPLICIT NONE
  INTEGER :: I, N, Factorial
  PRINT *, "Input N"
  READ *, N
  Factorial = 1
  DO I = 2, N
    Factorial = Factorial*I
  END DO
  PRINT *, N, " factorial = ", Factorial
END PROGRAM N_Factorial
```

Example execution:

```
Input N
5
5  factorial =          120
```

Example 6.2 The mean (excluding zeros) of a list of real values.

```fortran
PROGRAM Mean
  ! A list of values is input terminated by a negative number. The mean of all non-zero entries is calculated.
  !------------------------------------------------------------------
  IMPLICIT NONE
  INTEGER :: Count
  REAL :: V, Summation
  Summation = 0.
  Count = 0
  PRINT *, "Input the values, terminating by a negative value."
  DO
    READ *, V
    IF ( V==0. ) CYCLE
    IF ( V < 0. ) EXIT
    Summation = Summation + V
    Count = Count + 1
  END DO
  PRINT *, "The sum is ", Summation
  PRINT *, "The mean is ", Summation / REAL(Count)
END PROGRAM Mean
```
Example execution:

```
18.3
43.6
23.6
89.3
78.8
0.0
45.7
0.0
34.6
-1
```
The sum is 333.9000
The mean is 47.70000

Example 3
20-by-20 table of products

```fortran
PROGRAM Table_of_Products
    !-------------------------------------------------------------
    ! This program outputs a table of products
    ! using an implied DO loop inside a DO loop.
    !-------------------------------------------------------------
    IMPLICIT NONE
    INTEGER :: I, J
    DO I = 1, 20
        PRINT '(20(1x, I3))', (I*J, J=1,20)
    END DO
END PROGRAM Table_of_Products
```

Output:

```
   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
  2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38  40
  3   6   9  12  15  18  21  24  27  30  33  36  39  42  45  48  51  54  57  60
  4   8  12  16  20  24  28  32  36  40  44  48  52  56  60  64  68  72  76  80
  5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90  95 100
  6  12  18  24  30  36  42  48  54  60  66  72  78  84  90  96 102 108 114 120
  7  14  21  28  35  42  49  56  63  70  77  84  91  98 105 112 119 126 133 140
  8  16  24  32  40  48  56  64  72  80  88  96 104 112 120 128 136 144 152 160
  9  18  27  36  45  54  63  72  81  90  99 108 117 126 135 144 153 162 171 180
 10  20  30  40  50  60  70  80  90 100 110 120 130 140 150 160 170 180 190 200
 11  22  33  44  55  66  77  88  99 110 121 132 143 154 165 176 187 198 209 220
 12  24  36  48  60  72  84  96 108 120 132 144 156 168 180 192 204 216 228 240
 13  26  39  52  65  78  91 104 117 130 143 156 169 182 195 208 221 234 247 260
 14  28  42  56  70  84  98 112 126 140 154 168 182 196 210 224 238 252 266 280
 15  30  45  60  75  90 105 120 135 150 165 180 195 210 225 240 255 270 285 300
 16  32  48  64  80  96 112 128 144 160 176 192 208 224 240 256 272 288 304 320
 17  34  51  68  85 102 119 136 153 170 187 204 221 238 255 272 289 306 323 340
 18  36  54  72  90 108 126 144 162 180 198 216 234 252 270 288 306 324 342 360
 19  38  57  76  95 114 133 152 171 190 209 228 247 266 285 304 323 342 361 380
 20  40  60  80 100 120 140 160 180 200 220 240 260 280 300 320 340 360 380 400
```
7. Program Flow and Tracing

7.1 Introduction

In this section more examples of programs using loops are given with emphasis placed on using program tracing.

7.2 The Program Trace

Flow charts help us to visualise the flow of a program, especially when the program includes control statements and loops. As well as being an aid to program design, a flowchart can also help in the debugging of a program. Another aid to debugging is the **program trace**. Here, the values that variables take are output during the program execution. This is achieved by placing output statements at appropriate points in the program.

**Example 7.1**

The output statements shown in **bold** in the following program create a program trace:

```fortran
PROGRAM Max_Int
!
! Program to find the maximum of N integer values
! A program trace is achieved by using the output
! statements indicated by "! TRACE".
!
IMPLICIT NONE
INTEGER, PARAMETER :: N = 6
INTEGER :: V(N), I, Max
PRINT *, "Input", N, " integers"
READ *, V
Max = V(1)
PRINT *, "I  N  V(I)  Max"          ! TRACE
PRINT '(4(1X,I4))', 1, N, V(1), Max ! TRACE
DO I = 2, N
   IF ( V(I) > Max ) Max = V(I)
   PRINT '(4(1X,I4))', I, N, V(I), Max ! TRACE
END DO
PRINT *, "The maximum value is ", Max
END PROGRAM Max_Int
```

The output of the program (the trace is shown in **bold**)

<table>
<thead>
<tr>
<th>K</th>
<th>V(K)</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>-56</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>4</td>
<td>89</td>
<td>89</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>89</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>89</td>
</tr>
</tbody>
</table>

The maximum value is 89

The evolution of the values can be seen for each iteration of the loop.
Example 7.2

```fortran
PROGRAM Newtons_Square_Root
!-----------------------------------------------------
! This program uses Newton's method to compute the
! square root of a positive number P. The formula:
!     Xnew = ( Xold + P / Xold ) / 2
! is iterated until the difference |Xnew - Xold| is
! * i.e. X has converged to the square root of P.
! * here "zero" means there is no difference within
!   the limited storage precision.
!-----------------------------------------------------
! A program trace is achieved by using the output
! statement indicated by "! TRACE".
!-----------------------------------------------------
IMPLICIT NONE
REAL :: P, Xold, Xnew
PRINT *, "Input a positive number"
READ *, P
Xold = P
DO
    Xnew = ( Xold + P / Xold ) / 2.
    IF ( Xnew - Xold == 0. ) EXIT
    PRINT *, P, Xnew, Xnew - Xold ! TRACE
    Xold = Xnew
END DO
PRINT *, "The square root is ", Xnew
END PROGRAM Newtons_Square_Root
```

Example executions:

```
$ run program20trace
Input a positive number
3.0 [Enter]
3.000000 2.000000 -1.000000
3.000000 1.750000 -0.250000
3.000000 1.732143 -0.01785719
3.000000 1.732051 -0.0009202957
3.000000 1.732051 0.000000
The square root is  1.732051
```

```
$ run program20trace
Input a positive number
8673.4756 [Enter]
8673.476 4337.238 -4336.238
8673.476 2169.619 -2167.619
8673.476 1086.808 -1082.811
8673.476 547.3945 -539.4138
8673.476 281.6198 -265.7747
8673.476 156.2092 -125.4106
8673.476 105.8670 -50.34219
8673.476 93.89751 -11.96944
8673.476 93.13462 -0.7628937
8673.476 93.13149 -0.003128052
8673.476 93.13149 0.000000
The square root is  93.13149
```
Example 7.3

$e^x$ is computed using the series expansion:

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + ... + x^i/i! + ...$$

It requires some thought to correctly initialise the variables and compute correctly following terms. This is a good example where a program trace can help in debugging.

```fortran
PROGRAM ExpX
!-------------------------------------------------------------
! Program to compute $e^x$ by the series expansion:
! $e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + ... + x^i/i! + ...$
! As we soon run out of range when computing i! (i=13 gives
! integer overflow) an alternative method is used to allow us
! to include more terms; we see that:
! the (i+1)th term = the (i)th term * x/i
! New terms are computed and added to the series until a term
! is less than 0.000001.
!-------------------------------------------------------------
IMPLICIT NONE
INTEGER :: I
REAL :: X, E, Term
PRINT *, "Input a number."
READ *, X
Term = 1. ! the zeroth term
E = Term
I = 0
PRINT *, I, Term, E ! TRACE
DO
  I = I + 1 ! the next term
  Term = Term * X/REAL(I)
  E = E + Term
  PRINT *, I, Term, E ! TRACE
  IF ( Term < 0.000001 ) EXIT
END DO
PRINT *, "exp(", X, ") = ", E
END PROGRAM ExpX
```

Example execution:

```
Input a number.
3
0 1.000000 1.000000
1 3.000000 4.000000
2 4.500000 8.500000
3 4.500000 13.00000
4 3.375000 16.37500
5 2.025000 18.40000
6 1.012500 19.41250
7 0.4339286 19.84643
8 0.1627232 20.00915
9 0.05424170 20.07967
10 0.001627232 20.08410
11 0.0004437906 20.08521
12 0.0001109476 20.08547
13 0.00002560330 20.08554
14 0.000005486422 20.08553
15 0.000001097284 20.08554
16 0.0000002057408 20.08554
17 3.630721E-7 20.08554
exp( 3.000000 ) = 20.08554
```
Example 7.4
The greatest common divisor of two integers is computed using Euclid’s method. A program trace shows Euclid’s algorithm in action. Again, if the program does no work correctly the program trace is a useful tool for debugging.

```
PROGRAM GCD
!-------------------------------------------------------------------
! Program to compute the greatest common divisor
! of two integers using the Euclid method:
! Given a >= b
!
! 1. Compute the remainder c of the division a/b
! 2. If c is zero then b is the gcd
! 3. If c is not zero then
!    - replace a with b
!    - replace b with c
!    - go back to step 1.
!-------------------------------------------------------------------
IMPLICIT NONE
INTEGER :: A, B, C
PRINT *, "Input two integers."
READ *, A, B
DO
   C = MOD(A,B) ! the remainder of A/B
   PRINT *, A, B, C ! TRACE
   IF (C==0) EXIT ! gcd is B
   A = B
   B = C
END DO
PRINT *, "The gcd is ", B
END PROGRAM GCD
```

Example program executions:

```
Input two integers.
21 12
  21          12           9
  12           9           3
  9           3           0
The gcd is  3

Input two integers.
364 723
  364         723         364
  723         364         359
  364         359           5
  359           5           4
  5           4           1
  4           1           0
The gcd is  1
```

If a program does not work as you intend it to, it is often useful to use a trace to help you find the error.
8. Formatted I/O and File Processing

8.1 Introduction

In this section you will learn how to use the formatted PRINT, WRITE and READ statements and study input from and output to files.

8.2 Formatted Output

We have already seen formatted output statements, for example

```fortran
DO Deg = 0, 90, 5
   Rad = REAL(Deg)*PI/180. ! convert to radians
   PRINT '(1X, I2, 2(1X, F8.6))', Deg, SIN(Rad), COS(Rad)
END DO
```

Here, 1X gives a blank space, I2 indicates that the value is a 2-digit type integer, and F8.6 indicates that the value is an 8-digit type real with 6 decimal places (the decimal point is counted as one digit). The output (first 3 lines only) of this program is:

```
0 0.000000 1.000000
5 0.087156 0.996195
10 0.173648 0.984808
```

The free-format version is less tidy and less easy to read (and compiler dependent):

```
PRINT *, Deg, SIN(Rad), COS(Rad)
```

```
0 0.000000E+00 1.000000
5 8.715575E-02 0.9961947
10 0.1736482 0.9848077
```

The list of format descriptors in Fortran is:

```
Iw Bw Ow Zw Fw.d Ew.d ESw.d ENw.d Gw.a A
"x.. x" Lw Tc nX /
```

Specifications of the width and number of decimal places can be omitted, for example: F: decimal notation, ES: scientific notation, EN: engineering notation (powers of 10³).

```
REAL :: A = 12345.67
PRINT ('( F)'), A => 12345.6699219
PRINT ('(ES)'), A => 1.2345670E+04
PRINT ('(EN)'), A => 12.3456699E+03
```

8.3 Input/Output with Files

It is often useful to input data from a file and output data to a file. This is achieved in Fortran by using the OPEN statement to open a file for read/write, the READ() statement to read data from a file, and the WRITE() statement to write data to a file.
The OPEN statement
The OPEN statement has many specifiers giving, for example, the file name, its unit number, the intended action (read or write), and so on. We look at only a basic form of the statement:

```
OPEN(UNIT=unit-number, FILE="filename", ACTION="READ or WRITE")
```

I/O statements

```
CLOSE(unit-number)
```

The READ and WRITE statements
The READ statement is used to read data from a file, the WRITE statement is used to write data to a file, they have the following basic forms:

```
READ(UNIT=unit-number, FMT="formatted-specifier") variable-list
WRITE(UNIT=unit-number, FMT="formatted-specifier") data-list
```

Example 8.1
The following program reads a list of values from a file values.dat; the I/O program statements are shown in bold.

```
PROGRAM Mean_Value

IMPLICIT NONE
INTEGER, PARAMETER :: N=8
INTEGER :: I
REAL :: Value, Total=0.

OPEN(UNIT=1, FILE="values.dat", ACTION="READ")

DO I = 1, N
  READ(UNIT=1, FMT=*) Value
  Total = Total + Value
END DO

CLOSE(1)

PRINT *, "The mean is ", Total/REAL(N)

END PROGRAM Mean_Value
```

The program output is:

```
The mean is 53.62500
```

Here, the data file values.dat is opened and given the unit number 1, this unit number is referenced instead of the name of the file in the READ and WRITE statements. Values are read from unit 1 in free format (FMT=*), and stored, one line at a time, in variable Value. Finally the file is closed. Note the optional ACTION="READ" specifier; this permits only reading from (and not writing to) the file.
Example 8.2
A data file scores.dat contains student names and three exam scores. The data is stored in the file in four columns with the format:

```
abcdefghiIIIJJJKKK
```

where abcd...gh represents a 9 character name, III, JJJ, and KKK are three 3-digit integers representing percentage scores. The content of this file is:

```
Semra     94 95 89
Mustafa   66 71 75
Ceyhun    42 37 52
Aslı      14 28 35
Leyla     78 69 81
```

In Fortran this format is represented by `(A9,3I3)`. The following program reads the student scores with the above format. Assuming that the number of records in the file is unknown, the optional `END=label` clause is used to exit the read loop when then end of the file is reached.

```
PROGRAM Student_Scores
IMPLICIT NONE
CHARACTER(LEN=9) :: Name
INTEGER :: MT1, MT2, MT3
REAL :: Total
OPEN(UNIT=2, FILE="scores.dat", ACTION="READ")
DO
    READ(UNIT=2, FMT='(A9,3I3)', END=10) Name, MT1, MT2, MT3
    Total = 0.3*MT1 + 0.3*MT2 + 0.4*MT3
    PRINT '(A9,3(1X,I3)," =>",F5.1,"%")', Name, MT1, MT2, MT3, Total
END DO
10 CONTINUE
CLOSE(2)
END PROGRAM Student_Scores
```

The program output is:

```
Semra     94 95 89 => 92.3%
Mustafa   66 71 75 => 71.1%
Ceyhun    42 37 52 => 44.5%
Aslı      14 28 35 => 26.6%
Leyla     78 69 81 => 76.5%
```

The free format specification `FMT=*` maybe used for input from files if each data is separated by one or more spaces. However, a record such as

```
A. Yilmaz 87 98100
```

will appear to a free formatted input as two character fields and two integer fields, whereas the format `(A9,3I3)` will correctly read the data as
The output of this program can be sent to a file instead of the screen by opening a second file and using the `WRITE` statement. The modifications to the above program are indicated in **bold** in the program below:

```fortran
PROGRAM Student_Scores
  IMPLICIT NONE
  CHARACTER(LEN=9) :: Name
  INTEGER :: MT1, MT2, MT3
  REAL :: Total

  OPEN(UNIT=2, FILE="scores.dat", ACTION="READ")
  OPEN(UNIT=3, FILE="scores.out", ACTION="WRITE")

  DO
    READ(UNIT=2, FMT='(A9,3I3)', END=10) Name, MT1, MT2, MT3
    Total = 0.3*MT1 + 0.3*MT2 + 0.4*MT3
    WRITE(UNIT=3, FMT='(A9,3(1X,I3)," =>",F5.1,"\%")') Name, MT1, MT2, MT3, Total
  END DO
  10 CONTINUE

  CLOSE(2)
  CLOSE(3)
END PROGRAM Student_Scores
```

Notes:

- The first file has the `ACTION="READ"` attribute and the second has the `ACTION="WRITE"` attribute; it is therefore not possible to accidentally read from or write to the wrong file.
- The second file is given a different unit number, 3. Unit numbers 5 and 6 are reserved for the keyboard and screen respectively so be careful using these numbers.

### 8.4 Non-advancing Output

A useful specifier in the `WRITE` statement is the `ADVANCE='NO'` specifier:

```fortran
WRITE (*, FMT='(A)', ADVANCE='NO') "Input a number: ">
READ *, A
```
the read prompt is positioned at the end of the text instead of on the next line, for example:

```
Input a number: 23
```
9. Subprograms: Programmer-Defined Functions

9.1 Introduction

We have seen intrinsic functions such as the SIN, ABS and SQRT functions (there are also some intrinsic subroutines). Additional functions and subroutines can be defined by the programmer, these are called programmer defined subprograms. In this section we will look at how to write programmer defined functions, and in the following section we will look at how to write programmer defined subroutines. For simplicity we will only consider internal subprograms. You can read about external subprograms and modules elsewhere; if you are writing a large program then I advise that you make use of modules.

9.2 The Concept of a Function

A function accepts some inputs and outputs a result depending on the inputs. Every function has a name and independent values of inputs. The inputs are called parameters or arguments. Figure 9.1 show a box notation of a function.

A function may have one or more inputs but has to have only one output called return value. Figure 9.2 shows the examples of one- and two-input functions:

Figure 9.1: Box notation of a function

Figure 9.2: The box notations of a one-input $\sqrt{x}$ function and a two-input $f(x, y) = x + y$ function
9.3 Programmer-defined Functions

Fortran allows user to write this type of functions. The general form of a function must be:

\[
data \text{ type } \text{ FUNCTION } \text{name}(\text{list of arguments}) \\
\quad \text{name} = \text{ an expression} \\
\quad \ldots \\
\text{END FUNCTION name}
\]

where

- \text{data type} is the type of the function (or type of the return value) such as \text{REAL}
- Function name is given by \text{name}
- \text{list of arguments} (or local variables) are inputs to the function

For example a function that returns sum of two integers can be defined as follows:

<table>
<thead>
<tr>
<th>Function declaration</th>
<th>Identity card of the function</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{INTEGER FUNCTION Add}(A,B)</td>
<td>\text{Type} \quad \text{INTEGER}</td>
</tr>
<tr>
<td>INTEGER, INTENT(IN) :: A,B</td>
<td>\text{Name} \quad \text{Add}</td>
</tr>
<tr>
<td>\text{Add} = A+B</td>
<td>\text{Input parameters} \quad A,B</td>
</tr>
<tr>
<td>END FUNCTION Add</td>
<td>\text{Return value} \quad A+B</td>
</tr>
</tbody>
</table>

9.4 Internal and External Functions

Fortran 90/95 provides two basic type of function:

Internal Functions

They are placed after the main program section between a \text{CONTAINS} statement and the \text{END PROGRAM} statement.

\[
\text{PROGRAM Main} \\
\quad \text{IMPLICIT NONE} \\
\quad \text{REAL} :: X \\
\quad \text{INTEGER} :: Y \\
\quad X = \text{Fun1}(Z) \\
\quad Y = \text{Fun2}(Z) \\
\quad . \\
\quad \text{CONTAINS} \\
\quad \text{REAL FUNCTION Fun1}(A) \\
\quad . \\
\quad \text{END FUNCTION Fun1} \\
\quad \text{INTEGER FUNCTION Fun2}(B) \\
\quad . \\
\quad \text{END FUNCTION Fun2} \\
\quad \text{END PROGRAM Main}
\]

Notes:

\text{Fun1} and \text{Fun2} are internal functions. They are used in the same way as for intrinsic functions.

The \text{IMPLICIT NONE} statement applies to both the main section and the internal functions.

Data declared in the main program section is also visible in the functions (it is \text{global}).

Data declared in a function is only visible in that function, it is \text{local} to the function and so is not seen by the rest of the program unit.

Arguments can be given the \text{INTENT(IN)} attribute to protect the variable from being changed accidentally by the function.
External Functions

They are placed after the main program section (i.e. after the `END PROGRAM` statement)

```fortran
+-----------------------------+-----------
| PROGRAM Main               |
| IMPLICIT NONE              |
| REAL  :: X, Fun1           |
| INTEGER :: Y, Fun2         |
| X = Fun1(Z)                |
| Y = Fun2(Z)                |
| END PROGRAM Main           |
| REAL FUNCTION Fun1(A)       |
| END FUNCTION Fun1          |
| INTEGER FUNCTION Fun2(B)    |
| END FUNCTION Fun2          |
+-----------------------------+-----------
```

Notes:

- `Fun1` and `Fun2` are external functions. They are used in the same way as for intrinsic functions. You have to declare functions in main program.
- The `IMPLICIT NONE` statement does not apply to both the main section and the external functions.
- Data declared in the main program section is not visible in the functions.
- Data declared in a function is only visible in that function, it is `local` to the function and so is not seen by the rest of the program unit.
- Arguments can be given the `INTENT(IN)` attribute to protect the variable from being changed accidentally by the function.

As an example the function `Add` defined at the beginning of this section can be used as follows:

```fortran
PROGRAM Summation
IMPLICIT NONE
INTEGER :: I,J,K
PRINT *,"Input two integers:"
READ *,I,J
K = Add(I,J)
PRINT *,"The sum is ",K
END PROGRAM Summation

INTEGER FUNCTION Add(A,B)
INTEGER, INTENT(IN) :: A,B
Add = A+B
END FUNCTION Add
```

The output of both program section is:

```
Input two integers:
14
22
The sum is 36
```

Note that, we will consider only internal functions in the course.
9.5 Examples of Internal Functions:

Function references and definitions are indicated in **bold** face.

**Example 9.1 Function to convert degrees to radians.**

In this example we will consider the conversion of angles among degrees and radians. The formula for conversion is defined by:

\[
\frac{D}{180} = \frac{R}{\pi}
\]

where \( D \) is the angle measured in degrees and \( R \) is in radians and the number \( \pi = 3.141592... \)

```fortran
PROGRAM Degrees2Radians
    IMPLICIT NONE
    REAL :: Degrees   ! input
    REAL :: Radians   ! output

    PRINT *, "Input the angle in degrees"
    READ *, Degrees
    Radians = Rad(Degrees)

    PRINT *, Degrees, " degrees = ", Radians, " Radians."

CONTAINS

    REAL FUNCTION Rad(A)
        REAL, INTENT(IN) :: A
        REAL, PARAMETER :: Pi = 3.141593
        Rad = A * Pi/180.
    END FUNCTION Rad

END PROGRAM Degrees2Radians
```

Example execution:

```
Input the angle in degrees
90
90.00000      degrees =    1.570796     Radians.
```

Notes:
- The function is declared as type real i.e. it returns a type real value.
- As for intrinsic functions, an internal function can take for its arguments: variables, constants, or expressions.
- The **IMPLICIT NONE** statement applies to the whole program unit (the main section and to the function sections); therefore, the argument variable \( A \) must be declared somewhere in the program unit. It this case it is declared inside the function (and so is local to the function) and is given the **INTENT(IN)** attribute, this is a safer policy.
Example 9.2 Functions to convert Celsius to Fahrenheit, and Fahrenheit to Celsius. The formula for converting temperature measured in Fahrenheit to Celsius is:

\[ C = \frac{5}{9}(F - 32) \]

where \( F \) is the Fahrenheit temperature and \( C \) is the Celsius temperature. Suppose we wish to define and use a function that performs this conversion.

```fortran
PROGRAM Temp_Conv

IMPLICIT NONE

PRINT *, Fahrenheit(50.)
PRINT *, Celsius(400.)

CONTAINS

REAL FUNCTION Fahrenheit(X)
  REAL, INTENT(IN) :: X
  Fahrenheit = X*1.8 + 32.
END FUNCTION Fahrenheit

REAL FUNCTION Celsius(X)
  REAL, INTENT(IN) :: X
  Celsius = (X-32.)/1.8
END FUNCTION Celsius

END PROGRAM Temp_Conv
```

Note that we may include more than one programmer-defined function. The output is:

122.0000
204.4445

Example 9.3 A Gaussian function.

```fortran
PROGRAM Gaussian
IMPLICIT NONE
REAL :: X, M, S

PRINT *, "Input the position X, mean M, and sigma S"
READ *, X, M, S
PRINT *, Gauss(X, M, S)

CONTAINS

REAL FUNCTION Gauss(Position, Mean, Sigma)
  REAL, INTENT(IN) :: Position, Mean, Sigma
  REAL, PARAMETER :: TwoPi = 6.283185
  Gauss = EXP(-0.5*{(Position-Mean)/Sigma)**2 ) / &
            (Sigma*SQRT(TwoPi))
END FUNCTION Gauss

END PROGRAM Gaussian
```
Example execution:

\[
\text{Input the position } X, \text{ mean } M, \text{ and sigma } S \\
1.8 \quad 1.0 \quad 0.6 \\
0.2733502
\]

Notes:

- The number and order of the actual arguments must be the same as that of the formal arguments, in this case there are three arguments representing the position, mean, and standard deviation (in that order).
- Be careful not misspell variable names inside a function, if the variable exists in the main program section then it will be valid and used without a run-time error! Similarly, all variables you use in the function should be declared in the function, in this way you will not modify a global variable by mistake.

**Example 9.4** A factorial function

```fortran
PROGRAM N_Factorial
  IMPLICIT NONE
  INTEGER :: I
  DO I = -2, 14
    PRINT *, I, Factorial(I)
  END DO
END PROGRAM N_Factorial

CONTAINS

INTEGER FUNCTION Factorial(N)
  INTEGER, INTENT(IN) :: N
  INTEGER :: I
  IF (N < 0 .OR. N > 12 ) THEN
    Factorial = 0
  ELSE
    Factorial = 1
    DO I = 2, N
      Factorial = Factorial*I
    END DO
  END IF
END FUNCTION Factorial

END PROGRAM N_Factorial
```

The output is:

\[
\begin{array}{ll}
-2 & 0 \\
-1 & 0 \\
0  & 1 \\
1  & 1 \\
2  & 2 \\
3  & 6 \\
4  & 24 \\
5  & 120 \\
6  & 720 \\
7  & 5040 \\
8  & 40320 \\
9  & 362880 \\
10 & 3628800 \\
11 & 39916800 \\
12 & 479001600 \\
13 & 0 \\
14 & 0 \\
\end{array}
\]
Notes:

- **Factorial** is an integer function, i.e. it returns an integer value.
- The argument of the function is also integer.
- Identifier `i` is used both in the main program section and in the function. It therefore must be declared also in the function (thus making it local to the function) otherwise the two data will conflict.
- If the argument `n` is negative or too large then the function does not return an incorrect result, instead it indicates that there is a problem by returning a zero value. This condition can be checked for by the programmer.

9.6 Good Programming Practice:

- Declare all function variables; this makes them local so that they do not affect variables of the same name in the main program section.
- Give all function arguments the `INTENT(IN)` attribute. If, by mistake, you try to modify the argument value inside the function then an error will occur at compilation time.
- Be careful not to misspell variable names inside a function, if the variable exists in the main program section then it will be valid and used without a run-time error! *(modules are safer in this respect).*
10. Subprograms: Programmer-defined Subroutines

10.1 Introduction

As well as programmer defined functions, a Fortran program can also contain programmer defined subroutines. Unlike functions, subroutines do not return a value. Instead, a subroutine contains a separate program section that can be called at any point in a program via the `CALL` statement. This is useful if the program section that the subroutine contains is to be executed more than once. It also helps the programmer to organise the program in a modular form.

In this guide we will only consider internal subroutines. You can read about external subroutines and modules elsewhere; if you are writing a large program then I advise that you make use of modules.

10.2 The Concept of a Subroutine

A subroutine accepts no input or one or more inputs and may output no or one or more many outputs. This is assumed to be many purpose function. Figure 10.1 show a box notation of a subroutine:

![Figure 10.1: Box notation of a subroutine](image)

The advantage of using a subroutine is, it may have more than one return value. This is not the case in a function. Figure 10.2 shows the examples of subroutines:

![Figure 10.2: The box notations of one-input and two-output subroutine \( s \), and two-input and two-output subroutine \( \text{Rect} \).](image)

9.3 Programmer-defined Subroutine
The general form of a subroutine type subprogram is:

```
SUBROUTINE name(list of arguments)
  .
END SUBROUTINE name
```

where
- Subroutine name is given by name
- list of arguments (or local variables) are inputs to the subroutine

For example, a subroutine that returns area and circumference of a rectangle with sides \(a\) and \(b\) can be defined as follows:

```fortran
SUBROUTINE Rect(A,B,Area,Circ)
  REAL, INTENT(IN) :: A,B
  REAL, INTENT(OUT) :: Area,Circ
  Area = A*B
  Circ = A+B
END SUBROUTINE Rect
```

### 10.4 Internal and External Subroutines

#### Internal Subroutine

As for internal functions, internal subroutines are placed after the main program section between a `CONTAINS` statement and the `END PROGRAM` statement.

```
+-----------------------------+
| PROGRAM Main              |
| IMPLICIT NONE             |
|   CALL Sub1(X,Y)         |
|   CALL Sub2(X,Z)         |
|                           |
| CONTAINS                  |
|   SUBROUTINE Sub1(A,B)   |
|   END SUBROUTINE Sub1    |
| SUBROUTINE Sub2(A,B)      |
|   END SUBROUTINE Sub2    |
|                           |
+-----------------------------+
```

**Notes:**

- `Sub1` and `Sub2` are external functions.
  - They are used in the same way as for intrinsic functions.
  - The `IMPLICIT NONE` statement applies to both the main section and the internal subroutines.
  - Data declared in the main program section is visible in the subroutines.
  - Arguments can be given `INTENT(IN/OUT/INOUT)` attributes to make the programmers intent clear.
External Subroutine

They are placed after the main program section (i.e. after the END PROGRAM statement)

Notes:

Sub1 and Sub2 are external functions.
They are used in the same way as for intrinsic functions.

The IMPLICIT NONE statement does not apply to both the main section and the external subroutines.

Data declared in the main program section is not visible in the functions.

Arguments can be given INTENT(IN/OUT/INOUT) attributes to make the programmers intent clear.

As an example the subroutine Rect can be implemented as follows:

Usage of an internal subroutine

```
PROGRAM Rectangle
IMPLICIT NONE
REAL :: X,Y,Alan,Cevre
PRINT *,"Input the sides:"
READ *,X,Y
CALL Rect(X,Y,Alan,Cevre)
PRINT *,"Area is ",Alan
PRINT *,"Circum. is ",Cevre
CONTAINS
SUBROUTINE Rect(A,B,Area,Circ)
REAL, INTENT(IN) :: A,B
REAL, INTENT(OUT) :: Area,Circ
Area = A*B
Circ = A+B
END SUBROUTINE Rect
END PROGRAM Rectangle
```

Usage of an external subroutine

```
PROGRAM Rectangle
IMPLICIT NONE
REAL :: X,Y,Alan,Cevre
PRINT *,"Input the sides:"
READ *,X,Y
CALL Rect(X,Y,Alan,Cevre)
PRINT *,"Area is ",Alan
PRINT *,"Circum. is ",Cevre
END PROGRAM Rectangle
```

The output of both program is:

```
Input the sides:
4.0 2.0
Area is 8.000000
Circum. is 6.000000
```

Note that, we will consider only internal functions in the course.
Data can be passed to, and return from, the subroutine via arguments. As for function arguments, arguments in subroutines can be given \texttt{INTENT} attributes; they include the \texttt{INTENT(IN)}, \texttt{INTENT(OUT)}, and \texttt{INTENT(INOUT)} attributes, examples are given below:

<table>
<thead>
<tr>
<th>CALL \texttt{Results}(Radius)</th>
<th>CALL \texttt{TwistFactor}(N, M, TF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{SUBROUTINE Results(R)}</td>
<td>\texttt{SUBROUTINE TwistFactor(N, M, TF)}</td>
</tr>
<tr>
<td>\texttt{REAL, INTENT(IN) :: R}</td>
<td>\texttt{INTEGER, INTENT(IN) :: N}</td>
</tr>
<tr>
<td>\texttt{.}</td>
<td>\texttt{REAL, INTENT(IN) :: M}</td>
</tr>
<tr>
<td>\texttt{.}</td>
<td>\texttt{REAL, INTENT(OUT) :: TF}</td>
</tr>
<tr>
<td>\texttt{END SUBROUTINE Results}</td>
<td>\texttt{END SUBROUTINE TwistFactor}</td>
</tr>
</tbody>
</table>

\textbf{10.3 Examples of Subroutines:}
Subroutine references and definitions are indicated in \textbf{bold} face.

\textbf{Example 10.1}
The following program inputs the radius of a sphere and then employs an internal subroutine to compute the sphere's surface area and volume, and output the results.

```
PROGRAM Sphere
    !------------------------------------------------------
    ! Program to compute the volume and surface area of a
    ! sphere radius R. An internal subroutine is employed.
    !------------------------------------------------------
    IMPLICIT NONE
    REAL :: Radius
    PRINT *, "Input the radius of the sphere."
    READ *, Radius
    CALL Results(Radius)
    CONTAINS
    
    SUBROUTINE Results(R)
        REAL, INTENT(IN) :: R
        REAL, PARAMETER :: Pi = 3.141593
        REAL :: Area, Volume
        Area = 4.*Pi*R**2
        Volume = 4./3.*Pi*R**3
        PRINT *, "Surface area is ", Area
```
Example execution:

```
Input the radius of the sphere.
12.6
Surface area is 1995.037
Volume is 8379.157
```

Notes:
- It is not necessary to place an `IMPLICIT NONE` statement in an internal subroutine as the statement in the main program section applies to the whole program unit.
- In this subroutine the argument has the `INTENT(IN)` attribute as it is only intended to pass into the subroutine; this is illustrated below.

Example 10.2
In the following example, we will consider the calculation of the twist factor of a yarn. Twist Factor, \( T_f \), of a yarn is given by:

\[
T_f = N \sqrt{\frac{m}{1000}}
\]

where \( N \) (turn/m) is the number of twist of a yarn per unit length and \( m \) is measured in tex (a yarn count standard) that is mass in grams of a yarn whose length is 1 km. The program first needs a value of \( m \). Then, the value of \( T_f \) is calculated for different value of \( N \) which takes values from 100 to 1000 with step 100.
A Basic Introduction to Programming in Fortran

Example execution:

Input the value of \( m \) (tex)

<table>
<thead>
<tr>
<th>( m )</th>
<th>( \text{TwistFactor} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>50.0</td>
<td>22.36068</td>
</tr>
<tr>
<td>100</td>
<td>22.36068</td>
</tr>
<tr>
<td>200</td>
<td>44.72136</td>
</tr>
<tr>
<td>300</td>
<td>67.08204</td>
</tr>
<tr>
<td>400</td>
<td>89.44272</td>
</tr>
<tr>
<td>500</td>
<td>111.8034</td>
</tr>
<tr>
<td>600</td>
<td>134.1641</td>
</tr>
<tr>
<td>700</td>
<td>156.5247</td>
</tr>
<tr>
<td>800</td>
<td>178.8854</td>
</tr>
<tr>
<td>900</td>
<td>201.2461</td>
</tr>
<tr>
<td>1000</td>
<td>223.6068</td>
</tr>
</tbody>
</table>

Notes:
- The subroutine is declared as an external subroutine
- The name of the parameters can be same as in a subroutine

Example 10.3

In the following program the length of a side of a cube is passed to a subroutine which passes back the cube's volume and surface area.

```fortran
PROGRAM Cube_Calc
    IMPLICIT NONE
    REAL :: Length, Volume, Area
    PRINT *, "Input the length of the side of the cube."
    READ *, Length
    CALL Cube(Length, Volume, Area)
    PRINT *, "The volume of the cube is ", Volume
    PRINT *, "The surface area of the cube is ", Area
    CONTAINS
        SUBROUTINE Cube(L, V, A)
            REAL, INTENT(IN) :: L
            REAL, INTENT(OUT) :: V, A
            V = L**3
            A = 6 * L**2
        END SUBROUTINE Cube
    END PROGRAM Cube_Calc
```
Example execution:

Input the length of the side of the cube.
12
The volume of the cube is 1728.000
The surface area of the cube is 864.0000

Note:
There are three arguments in the subroutine. The first argument \( L \) has the `INTENT(IN)` attribute as it passes data into the subroutine. The second and third arguments \( V \) and \( A \) have the `INTENT(OUT)` attributes as they are only intended to pass data out of the subroutine. This is illustrated below:

```fortran
CALL Cube(length, Volume, Area)
SUBROUTINE Cube(L, V, A)
  REAL, INTENT(IN) :: L
  REAL, INTENT(OUT) :: V, A
END SUBROUTINE Cube
```

Example 10.4
The following program illustrates the use of an argument with an `INTENT(INOUT)` attribute.

```fortran
PROGRAM Money_Owed
  IMPLICIT NONE
  REAL :: Owed = 1000., Payment
  DO
    PRINT *, "Input the payment"
    READ *, Payment
    CALL Payback(Owed, Payment) ! Subtract from the money owed.
    IF ( Owed == 0. ) EXIT ! Repeat until no more money is owed.
  END DO
END PROGRAM Money_Owed

SUBROUTINE Payback(Owed, Payment)
  REAL, INTENT(INOUT) :: Owed
  REAL, INTENT(IN) :: Payment
  REAL :: Overpaid = 0.
  Owed = Owed - Payment
  IF ( Owed < 0. ) THEN
    Overpaid = - Owed
    Owed = 0.
  END IF
  PRINT *, "Payment made ", Payment, ", amount owed is now ", Owed
  IF ( Overpaid /= 0. ) PRINT *, "You over paid by ", Overpaid
END SUBROUTINE Payback
```

Note that the number of and order of the arguments should be the same in the call (the actual arguments) and in the subroutine (the formal arguments).
Example execution:

```fortran
CALL Payback(Owed, Payment)
SUBROUTINE Payback(Owed, Payment)
  REAL, INTENT(INOUT) :: Owed
  REAL, INTENT(IN) :: Payment
  
END SUBROUTINE Payback
```

Input the payment
350
Payment made 350.0000, amount owed is now 650.0000
Input the payment
350
Payment made 350.0000, amount owed is now 300.0000
Input the payment
350
Payment made 350.0000, amount owed is now 0.000000
You over paid by 50.00000

Notes:
Argument `Owed` has the `INTENT(INOUT)` attribute; it passes a value into the subroutine which then passes it back via the same argument after modifying it. Argument `Payment` only passes data into the subroutine and so has the `INTENT(IN)` attribute. This is illustrated right.

10.4 Good Programming Practice:

- Declare all arguments used in the subroutine; this makes them local so that they do not affect variables of the same name in the main program section.
- Give all arguments the appropriate `INTENT(IN), INTENT(OUT)` or `INTENT(INOUT)` attribute.
- Be careful not misspell variable names inside a subroutine, if the variable exists in the main program section then it will be valid and used without a run-time error! (`modules` are safer in this respect).
11. Arrays and Array Processing

11.1 Introduction

In this section we will look more at arrays with emphasis placed on array processing, array functions, and using arrays in programmer-defined functions and subroutines.

11.2 Arrays

An array is a group of variables or constants, all of the same type, which is referred to by a single name. For example, if the following scalar variable can represent the mass of an object:

\[ \text{REAL :: Mass} \]

then the masses of a set of 5 objects can be represented by the array variable

\[ \text{REAL :: Mass(5)} \]

The 5 elements of the array can be assigned as follows:

\[
\begin{align*}
\text{Mass(1)} &= 8.471 \\
\text{Mass(2)} &= 3.683 \\
\text{Mass(3)} &= 9.107 \\
\text{Mass(4)} &= 4.739 \\
\text{Mass(5)} &= 3.918 \\
\end{align*}
\]

or more concisely using an array constant:

\[ \text{Mass} = (/ 8.471, 3.683, 9.107, 4.739, 3.918 /) \]

We can operate on individual elements, for example

\[ \text{Weight(3)} = \text{Mass(3)} \times 9.81 \]

or we can operate on a whole array in a single statement:

\[ \text{Weight} = \text{Mass} \times 9.81 \]

Here both \text{Weight} and \text{Mass} are arrays with 5 elements; the two arrays must conform (have the same size).

The above whole array assignment is equivalent to:

\[
\begin{align*}
\text{DO } I &= 1, 5 \\
\text{Weight}(I) &= \text{Mass}(I) \times 9.81 \\
\text{END DO} \\
\end{align*}
\]

Whole array assignment can be used for initialising all elements of an array with the same values:

\[
\begin{align*}
\text{A} &= 0. \\
\text{B} &= 100 \\
\end{align*}
\]

\text{A} and \text{B} are arrays.
11.3 The **WHERE** Statement and Construct

We can make the whole array assignment conditional with the **WHERE** statement. For example, we have an array $V$ of values that we want to take the square-root of, but only for positive values:

```fortran
WHERE ( V >= 0. ) V = SQRT(V)
```

This is equivalent to

```fortran
DO I = 1, N
   IF ( V(I) >= 0. ) V(I) = SQRT(V(I))
END DO
```

where $N$ is the size of the array. Or, we have an array $V$ of values that we want to take the reciprocal of, but only for non-zero values:

```fortran
WHERE (V /= 0. ) V = 1./V
```

This is equivalent to

```fortran
DO I = 1, N
   IF ( V(I) /= 0. ) V(I) = 1./V(I)
END DO
```

where $N$ is the size of the array.

The **WHERE** construct allows for a block of statements and the inclusion of an **ELSEWHERE** statement:

```fortran
WHERE ( V >= 0. )
   V = SQRT(V)
ELSEWHERE
   V = -1.
ENDWHERE
```

11.4 Array Sections

We can write the whole array assignment

```fortran
Weight = Mass * 9.81
```

in the form of an array section:

```fortran
Weight(1:N) = Mass(1:N) * 9.81
```

where $N$ is the size of the array.

Here the section $1:N$ represents the elements 1 to $N$. A part of an array (an array section) can be written, for example, as:

```fortran
Weight(2:5) = Mass(2:5) * 9.81
```
which is equivalent to

```fortran
DO I = 2, 5
  Weight(I) = Mass(I) * 9.81
END DO
```

Array sections are also useful for initialising arrays, for example:

```fortran
A(1:4) = 0.
A(5:10) = 1.
```

is equivalent to the array constant assignment:

```fortran
A = (/ 0., 0., 0., 1., 1., 1., 1., 1., 1., 1. /)
```

A third index indicates increments:

```fortran
A(2:10:2) = 5.
```

is equivalent to

```fortran
DO I = 2, 10, 2
  A(I) = 5.
END DO
```

More examples of array sections are shown below. The 10 elements of array \( A \) are represented by a series of boxes. The elements referenced by the array section are shaded.

11.5 Array Indices

In the following array declaration

```fortran
REAL :: A(9)
```

the index for the elements of the array go from 1 to 9. The index does not have to begin at 1, it can be zero or even negative; this is achieved with the "::" symbol:

```fortran
REAL :: A(0:8), B(-4:4), C(-8:0)
```

All the above arrays have 9 elements. The index of \( A \) runs from 0 to 8, the index of \( B \) runs from -4 to 4 (including zero), and the index of \( C \) runs from -8 to 0.
11.6 Assignment using Implied Loops

An array can be assigned using an implied loop, For example

\[
A = (/ (I*0.1, I=1,9) /)
\]

assigns to array \(A\) the values:

0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9

The implied loop (in **bold**) appears in an array constant.

11.7 Multi-dimensional Arrays

In the array declaration

\[
\text{REAL :: A(9)}
\]

array \(A\) has one dimension (one index); such an array is called a *vector*. An array can have more than one dimension, for example the declaration of a two-dimensional array \(B\) may be as follows:

\[
\text{REAL :: B(9,4)}
\]

A two-dimensional array has two indices and is called a *matrix*. The above vector and matrix are visualised below, vector \(A\) has 9 elements, matrix \(B\) has 36 elements arranged in 9 rows and 4 columns:

\[
\begin{array}{l}
A \\
\hline
A(2) \\
\hline
A(4:7) \\
\hline
\end{array}
\quad
\begin{array}{l}
B \\
\hline
B(3,2) \\
\hline
B(5:8,3:4) \\
\hline
\end{array}
\]

An array can have many dimensions, though it is not common to go above three-dimensions.

11.8 Array Input/Output

We will now look at input and output of arrays. Only one-dimensional arrays will be considered; for two-dimensional arrays the principle is the same except that you need to think about whether the I/O should be row-wise or column-wise.

Consider an array \(A\) declared as:

\[
\text{REAL :: A(9)}
\]

The array can be input and output as a whole array:
READ *, A
PRINT *, A

which is equivalent to the implied loops:

READ *, ( A(I), I = 1, 9 )
PRINT *, ( A(I), I = 1, 9 )

or individual elements can be referenced:

READ *, A(4)
PRINT *, A(4)

There are various methods for output of arrays; consider the array:

REAL :: A(9) = (/ (I*0.1, I = 1, 9) /)

Array A takes the values 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, and 0.9. The free format output of the whole array

PRINT *, A

which is equivalent to the implied loop:

PRINT *, (A(I), I = 1, 9)

will appear something like:

0.1000000   0.2000000   0.3000000   0.4000000   0.5000000
0.6000000   0.7000000   0.8000000   0.9000000

A formatted output, for example,

PRINT ' (9(F3.1,1X)) ', A

gives:

0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9

If the multiplier is omitted then the output will be given line by line; i.e.

PRINT ' (F3.1X) ', A

gives:

0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
This is equivalent to the `DO` loop:

```fortran
DO I = 1, 9
    PRINT '(F3.1,1X)', A(I)
END DO
```

Array sections can also be referenced, for example:

```fortran
PRINT '(9(F3.1,1X))', A(3:8)
gives:
0.3  0.4  0.5  0.6  0.7  0.8
```

Note that the multiplier in the format specifier can be 6 or greater.

### 11.9 Intrinsic Functions for Arrays

Most intrinsic functions that we use for scalars, for example `SIN`, `INT`, and `ABS`, are *elemental*; i.e. they can also apply to arrays. For example:

```fortran
REAL :: A= (/ 0.2, 0.3, 0.4, 0.5, 0.6 /)
PRINT *, SIN(A)
gives
0.1986693  0.2955202  0.3894183  0.4794255  0.5646425
```

There are, in addition to the elemental functions, intrinsic functions whose arguments are specifically arrays. Some them are listed below (see elsewhere for a more complete list).

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXVAL(A)</td>
<td>Gives the maximum value of array A</td>
</tr>
<tr>
<td>MINVAL(A)</td>
<td>Gives the minimum value of array A</td>
</tr>
<tr>
<td>MAXLOC(A)</td>
<td>Index location of maximum value of A</td>
</tr>
<tr>
<td>MINLOC(A)</td>
<td>Index location of minimum value of A</td>
</tr>
<tr>
<td>PRODUCT(A)</td>
<td>Gives product of the values in array A</td>
</tr>
<tr>
<td>SIZE(A)</td>
<td>Gives the number of values of array A</td>
</tr>
<tr>
<td>SUM(A)</td>
<td>Gives the sum of the values in array A</td>
</tr>
<tr>
<td>MATMUL(A,B)</td>
<td>Gives the cross product of arrays A and B</td>
</tr>
<tr>
<td>TRANSPOSE(A)</td>
<td>Gives the transpose of array A</td>
</tr>
</tbody>
</table>

### 11.10 Arrays as Arguments in Subprograms

When an array is passed as an argument to a subprogram the subprogram creates the array locally and then destroys it when the execution of the subprogram is complete. Such arrays are called *semi-dynamic arrays*. The declaration of semi-dynamic arrays can be of three types: `explicit-shaped`, `assumed-shaped`, and `automatic` arrays. We will only consider subprograms with `assumed-shaped arrays`; you can read about the other forms of semi-dynamic arrays elsewhere. Also read about *Dynamic (allocatable)* arrays; these are given in Section 11.11.
Example 11.1
The following program employs a function to return the range (difference between the minimum and maximum values) of an array.

```fortran
PROGRAM Range_of_Data
  !-------------------------------------------
  ! This program employs a function to return
  ! the range (difference between the minimum
  ! and maximum values) of an array.
  !-------------------------------------------
  IMPLICIT NONE

  REAL :: V(8) = (/ 16.8, 12.3, -6.2, 8.4, 
      &    31.6, 14.1, 17.3, 26.9 /)

  PRINT *, "The range is ", Range(V)

CONTAINS

  REAL FUNCTION Range(Values)
    REAL, INTENT(IN) :: Values(:)

    Range = MAXVAL(Values) - MINVAL(Values)
  END FUNCTION Range

END PROGRAM Range_of_Data
```

Output:

```
The range is    37.80000
```

Notes:

- The function creates an array `Values` (the formal argument) with the declaration `REAL, INTENT(IN) :: Values(:)`.

- The colon `":"` indicates that the size of the array should be the same as that of the actual argument `V` (the shape of the array is assumed in this declaration).

- The range of values is computed using intrinsic functions `MAXVAL` and `MINVAL`. 
Example 11.2
The program below employs a subroutine to take the square root of each element of an array, if an element value is negative then -1 is assigned.

```fortran
PROGRAM Array_Square_Root
!--------------------------------------
! This program employs a subroutine to
! take the square root of each element
! of an array, if an element value is
! negative then -1 is assigned.
!--------------------------------------
IMPLICIT NONE

REAL :: V(8) = (/ 16.8, 12.3, -6.2, 8.4, &
                 31.6, 14.1, 17.3, 26.9 /)

PRINT *, "The values are ", V
CALL SqrtN(V)
PRINT *, "Their sqrt are ", V

CONTAINS

SUBROUTINE SqrtN(Values)

    REAL, INTENT(INOUT) :: Values(:)

    WHERE ( Values >= 0 )
        Values = SQRT(Values)
    ELSEWHERE
        Values = -1.
    ENDWHERE

END SUBROUTINE

END PROGRAM Array_Square_Root
```

Output (values are rounded to 3dp):

```
The values are  16.800 12.300 -6.2000 8.4000 31.600 14.100 17.300 26.900
Their sqrt are   4.099  3.507 -1.0000 2.8983 5.6214  3.755  4.159  5.187
```

Notes:
- Again, the formal argument `Values` is created as an assumed-shaped array.
- The array is given the `INTENT(INOUT)` attribute as it is passed into the subroutine and then back out after it is modified.
- The subroutine employs the `WHERE` construct, see "The WHERE statement and construct".
Example 11.3
A list of exam scores is stored in a file called exam-scores.dat. The following program reads the scores into an array and employs a function to calculate the mean score. It is common for a few scores to be zero representing students that were absent from the exam, these zero scores are not included in the calculation of the mean.

---

<table>
<thead>
<tr>
<th>exam-scores.dat</th>
<th>PROGRAM Mean_No_Zeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>54 67 89 34 66 73 81 0 76 24 77 94 83 0 69 81</td>
<td>IMPLICIT NONE</td>
</tr>
<tr>
<td>INTEGER, PARAMETER :: Number_of_Values=16</td>
<td></td>
</tr>
<tr>
<td>REAL :: Scores(Number_of_Values)</td>
<td></td>
</tr>
<tr>
<td>OPEN(UNIT=3, FILE=&quot;exam-scores.dat&quot;, ACTION=&quot;READ&quot;)</td>
<td></td>
</tr>
<tr>
<td>READ (3, *) Scores</td>
<td></td>
</tr>
<tr>
<td>CLOSE(3)</td>
<td></td>
</tr>
<tr>
<td>PRINT *, &quot;The mean is &quot;, MeanNZ(Scores)</td>
<td></td>
</tr>
<tr>
<td>CONTAINS</td>
<td></td>
</tr>
<tr>
<td>REAL FUNCTION MeanNZ(V)</td>
<td></td>
</tr>
<tr>
<td>REAL, INTENT(IN) :: V(:)</td>
<td></td>
</tr>
<tr>
<td>REAL :: Total</td>
<td></td>
</tr>
<tr>
<td>INTEGER :: I, Count</td>
<td></td>
</tr>
<tr>
<td>Total = 0.</td>
<td></td>
</tr>
<tr>
<td>Count = 0</td>
<td></td>
</tr>
<tr>
<td>DO I = 1, SIZE(V)</td>
<td></td>
</tr>
<tr>
<td>IF ( V(I) /= 0. ) THEN</td>
<td></td>
</tr>
<tr>
<td>Total = Total + V(I)</td>
<td></td>
</tr>
<tr>
<td>Count = Count + 1</td>
<td></td>
</tr>
<tr>
<td>END IF</td>
<td></td>
</tr>
<tr>
<td>END DO</td>
<td></td>
</tr>
<tr>
<td>MeanNZ = Total/REAL(Count)</td>
<td></td>
</tr>
<tr>
<td>END FUNCTION MeanNZ</td>
<td></td>
</tr>
<tr>
<td>END PROGRAM Mean_No_Zeros</td>
<td></td>
</tr>
</tbody>
</table>

Output:
The mean is 69.14286

Notes:
- The size of the array is declared with the named constant Number_of_Values = 16, this is not very convenient as we have to recompile the program every time the number of values in the data file changes. There are various solutions to this problem (including the use of dynamic arrays) but for simplicity we will leave the program as it is.
- After opening the file, all 16 lines of data are input in the single statement READ (3,*) Scores. For this whole array assignment the number of elements in the array must equal the number of values in the file. If there are less than 16 entries in the file then the program will exit with an error something like "I/O error: input file ended", if there are more than 16 entries in the file then only the first 16 will be read.
- Again an assumed-shaped array is used in the function. However, in the DO loop we need to know the size of the array, this is obtained with the SIZE function.
- The function MeanNZ could simply be replaced with the array processing function SUM and COUNT with the following expression:

PRINT*,"The mean is ", SUM(Scores,MASK=Scores/=0.) / COUNT(Scores/=0.)
Example 11.4
The above program can be rewritten using a subroutine instead of a function; the changes are indicated in bold face.

```fortran
PROGRAM Mean_No_Zeros
IMPLICIT NONE
INTEGER, PARAMETER :: Number_of_Values=16
REAL :: Scores(Number_of_Values), Mean

OPEN(UNIT=3, FILE="exam-scores.dat", ACTION="READ")
READ (3, *) Scores
CLOSE(3)

CALL MeanNZ(Scores, Mean)
PRINT *, "The mean is ", Mean

CONTAINS

SUBROUTINE MeanNZ(V, M)
REAL, INTENT(IN) :: V(:)
REAL, INTENT(OUT) :: M
REAL :: Total
INTEGER :: I, Count
Total = 0.
Count = 0
DO I = 1, SIZE(V)
  IF ( V(I) /= 0. ) THEN
    Total = Total + V(I)
    Count = Count + 1
  END IF
END DO
M = Total/REAL(Count)
END SUBROUTINE MeanNZ

END PROGRAM Mean_No_Zeros
```

Notes:

- As for the function of Example 1, an assumed-shaped array is used to copy the array from the main program section. The SIZE function is used for the DO loop.

- The mean value is returned via the argument M instead of via a function. M is given the INTENT(OUT) attribute.
11.11 Dynamic Arrays (Allocatable Arrays)

A declaration of a compile-time array of the form:

```fortran
INTEGER, PARAMETER :: N = 10
REAL :: A(N)
```

Causes the compiler to allocate a block of memory large enough to hold 10 real values. But Fortran does not allow us to write:

```fortran
INTEGER :: N ! declare a variable
PRINT *, "How many element?" ! At run time, let the user
READ *, N ! input the size of the array
REAL :: A(N) ! and then try to allocate
! *** NOT ALLOWED ***
```

Fortran 90 does, however, provide allocatable or run-time or dynamic arrays for which memory is allocated during execution (i.e. run-time allocation). If the required size of an array is unknown at compile time then a dynamic array should be used.

A dynamic array is declared using `ALLOCATABLE` attribute as follows:

```
type, ALLOCATABLE :: array_name
```

for example:

```fortran
INTEGER, ALLOCATABLE :: A(:) ! for a vector
REAL, ALLOCATABLE :: B(:,:) ! for a matrix
```

After declaring the dynamic array, the bounds can be assigned using `ALLOCATE` statement as follows:

```fortran
ALLOCATE(array_name(lower_bound:upper_bound))
```

An example is:

```fortran
ALLOCATE(A(10)) ! 10 element vector
ALLOCATE(B(4,4)) ! 4x4 matrix
```

If you wish to check the allocation status too:

```fortran
ALLOCATE(array_name(lower_bound:upper_bound),STAT=status_variable)
```

In this form, the integer variable `status_variable` will be set to zero if allocation is successful, but will be assigned some value if there is insufficient memory.

```fortran
READ *,N ! read an integer N
ALLOCATE(A(N),STAT=AllocStat) ! try to allocate N element vector
IF(AllocStat /=0 ) THEN !--- check the memory ---
  PRINT *,"Not enough memory to allocate A"
  STOP
END IF
```
If the allocated array is no longer needed in the program, the associated memory can be freed using `DEALLOCATE` statement as follows:

```
DEALLOCATE(array_name)
```

for example:

```
DEALLOCATE(A)
```

The following example will summarise usage of the dynamic arrays:

**Example 11.5**

Write program to input \( n \) integer numbers and outputs the median of the numbers. The median is the number in the middle. In order to find the median, you have to put the values in order from lowest to highest, then find the number that is exactly in the middle.

```fortran
PROGRAM Dynamic_Array
!-----------------------------------------------------
! This program calculates median of N numbers.
! The median is the number in the middle for the
! given set of data. For example:
! Median(3,4,4,5,6,8,8,8,10) = 6.0
! Median(5,5,7,9,11,12,15,18) = (9+11)/2.0 = 10.0
!-----------------------------------------------------
IMPLICIT NONE
INTEGER, ALLOCATABLE :: A(:)
INTEGER :: N, As
REAL :: Median

PRINT *,"Input N"
READ *,N
ALLOCATE(A(N), STAT=As)
IF(As /=0 ) THEN
   PRINT *,"Not enough memory"
   STOP
END IF

PRINT *,"Input ", N , ", " integers in increasing order:"
READ *,A

IF(MOD(N,2)==1) THEN ! odd number of data
   Median = A((N+1)/2)
ELSE
   Median = (A(N/2)+A(N/2+1))/2.0
ENDIF

PRINT *,"Median of the set is ",Median

DEALLOCATE(A)
END PROGRAM Dynamic_Array
```
Example Executions:

Input N
9
Input 9 integers in increasing order:
3 4 4 5 6 8 8 8 10
Median of the set is 6.000000

Input N
8
Input 8 integers in increasing order:
5 5 7 9 11 12 15 18
Median of the set is 10.00000

Input N
6
Input 6 integers in increasing order:
80 85 90 90 90 100
Median of the set is 90.00000

Problem:

Write a Fortran program to find mean, mode and median of n integer numbers. Note that, mode is the most frequent number in a set of data. For example:

Mode of the set: 2 2 5 9 9 10 10 11 12 18 is 9. (unimodal set of data)
Mode of the set: 2 3 4 4 4 5 7 7 7 9 is 4 and 7 (bimodal set of data)
Mode of the set: 1 2 3 8 9 10 12 14 18 is ? (data has no mode)
12. Selected Topics

12.1 Numerical KINDS

The KIND type enables the user to request which intrinsic type is used based on precision and/or range. This facilitates an improved numerical environment. Programmers porting their programs to different machines must deal with differing digits of precision. Using KIND, the programmer can specify the numeric precision required.

Variables are declared with the desired precision by using the KIND attribute:

\[
\text{type(KIND = kind type value) :: variable list}
\]

For Example:

```fortran
INTEGER :: I             ! default KIND=4
INTEGER(KIND=4) :: J     ! default
INTEGER(KIND=1) :: K     ! limited precision  -127 <= K <= 127
INTEGER(KIND=2) :: L     ! limited precision  32767 <= L <= 32767
INTEGER(KIND=4) :: M     ! limited precision -1E-38 <= M <= 1E+38
INTEGER(KIND=8) :: N     ! limited precision -1E-308 <= N <= 1E+308
INTEGER(2)      :: I     ! KIND= is optional
INTEGER         :: P=1_8 ! P=1 and is of kind type 8
REAL            :: A     ! default KIND=4
REAL(KIND=4)    :: B     ! limited precision -1.0E-38 <= B <= 1.0E+38
REAL(KIND=8)    :: C     ! limited precision -1.0E-308 <= C <= 1.0E+308
```

Following program will print the largest numbers that can be stored by each kind.

```fortran
PROGRAM Numerical_Kinds

INTEGER (KIND=1) :: K1
INTEGER (KIND=2) :: K2
INTEGER (KIND=4) :: K4
INTEGER (KIND=8) :: K8
REAL    (KIND=4) :: R4
REAL    (KIND=8) :: R8

PRINT *,"Largest number for K1:" , HUGE(K1)
PRINT *,"Largest number for K2:" , HUGE(K2)
PRINT *,"Largest number for K4:" , HUGE(K4)
PRINT *,"Largest number for K8:" , HUGE(K8)
PRINT *,"Largest number for R4:" , HUGE(R4)
PRINT *,"Largest number for R8:" , HUGE(R8)

END_PROGRAM Numerical_Kinds
```

The output of the program after compiling with Intel Fortran Compiler (IFC) that we use:

- Largest number for K1: 127
- Largest number for K2: 32767
- Largest number for K4: 2147483647
- Largest number for K8: 9223372036854775807
- Largest number for R4: 3.4028235E+38
- Largest number for R8: 1.797693134862316E+308
12.2 Numerical Derivative of a Function

Let \( f(x) \) be defined (analytic) at any point \( x_0 \). The derivative of \( f(x) \) at \( x = x_0 \) is defined as:

\[
f'(x) = \frac{dy}{dx} = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}
\]

In a computer program the limit of \( h \) can not be zero because of underflow limit. But it can be selected close to zero such as \( h = 0.01 \). Thus, a computer implementation can be done as follows:

```fortran
PROGRAM Derivative
!----------------------------------------------------
! Evaluates the numerical derivative of a function
! \( F(x) \) at a given point. The function \( dF(x) \) returns
! the derivative of \( f(x) \) at point \( x_0 \).
!----------------------------------------------------
REAL :: x0
PRINT *,"Input x0"
READ *,x0
PRINT *,"F(x0) = ",F(x0)
PRINT *,"F'(x0) = ",dF(x0)

CONTAINS

REAL FUNCTION F(x) ! function definition
REAL, INTENT(IN) :: x
F = x**3 - 2*x + 5
END FUNCTION F

REAL FUNCTION dF(X) ! the derivative of the function
REAL, INTENT(IN) :: x
REAL :: h
h = 0.01

dF = (F(x+h)-F(x))/h
END FUNCTION dF

END PROGRAM Derivative

Example execution:
Input x0
2
F(x0) = 9.000000
F'(x0) = 10.06012

If we check the result:
\[ f(x) = x^3 - 2x + 5 \rightarrow f(2) = 9 \]
\[ f'(x) = 3x^2 - 2 \rightarrow f'(2) = 3(2)^2 - 2 = 10 \]
12.3 Numerical Integration of a Function

For the figure total area of the rectangles:

\[ f(x_0)\Delta x + f(x_1)\Delta x + f(x_2)\Delta x + \ldots + f(x_n)\Delta x = \sum_{k=0}^{n} f(x_k)\Delta x \]

for \( \Delta x \to 0 \) this sum will be:

\[ \lim_{\Delta x \to 0} \sum_{k=0}^{n} f(x_k)\Delta x = \int_{a}^{b} f(x)\,dx \]

Thus, the geometric meaning of the definite integral is area under the curve \( y=f(x) \). In a computer program, \( \Delta x \) cannot be zero, but can be selected close to zero such as \( \Delta x = 0.01 \).

For this case following program can be used to evaluate definite integral of \( f(x) \) between \([a,b]\).

```fortran
PROGRAM Integral

! Evaluates the numerical integration of a function
! f(x) between limits a and b by rectangles method.
! Integration is performed by the function Inegrate.
!
IMPLICIT NONE
REAL :: A,B,Result

PRINT *,"Input A and B"
READ *, A,B
Result = Integrate(A,B)

PRINT *,"The integral is:",Result

CONTAINS

REAL FUNCTION F(x)
REAL, INTENT(IN) :: x
F = x**2  ! put your function here
END FUNCTION F
```

\[ y = f(x) \]

\[ x_0, x_1, x_2, x_3, \ldots, x_n \]
REAL FUNCTION Integrate(A,B)
!-------------------------------------------------
! returns the integral of f(x) between limits
! [a,b] by rectangles method.
!-------------------------------------------------
REAL, INTENT(IN) :: A,B
INTEGER, PARAMETER :: N = 1000
INTEGER :: k
REAL :: x,dx,Sum

dx  = (B-A)/N
Sum = 0.0
x = A

DO k=1,N-1
  x = x + dx
  Sum = Sum + F(x)
END DO

Integrate = Sum*dx

END FUNCTION Integrate

END PROGRAM Integral

Example execution:

Input A and B
1 2

The integral is: 2.334912

The analytical result is:

\[ \int_{1}^{2} x^2 \, dx = 2.333333 \]
12.4 Mean and Standard Deviation

This topic is included because it is commonly used in statistical analysis, and demonstrates the power of whole array processing in Fortran. The mean value $\bar{V}$ and standard deviation $\sigma$ of $n$ values $V_i$ ($i = 1, n$) are defined below, the Fortran definitions are also shown as concise one-line expressions (here $V$ is a type real Fortran array of any size and shape).

\[
\bar{V} = \frac{1}{n} \sum_{i=1}^{n} V_i \quad \text{mean} = \frac{\text{SUM}(V)}{\text{SIZE}(V)}
\]

\[
\sigma = \sqrt{\frac{\sum_{i=1}^{n} (V_i - \bar{V})^2}{n-1}} \quad \text{sigma} = \frac{\text{SQRT}((\text{V-mean})^2)/(\text{SIZE}(V)-1))}{\text{COUNT}(\text{V}==0)}
\]

Note that the value of $n$ does not need to be known, we can use \text{SIZE()} instead. If you wish to omit from the calculation all zero values, then this can be done simply with the \text{MASK} option and the \text{COUNT} function:

\[
\text{mean} = \frac{\text{SUM}(V, \text{MASK}=V\neq0)}{\text{COUNT}(V\neq0)}
\]

\[
\text{sigma} = \frac{\text{SQRT}((\text{V-mean})^2)/(\text{SIZE}(V)-1))}{\text{COUNT}(\text{V}==0)-1})
\]

Following program is evaluates the mean and standard deviation of $n$ real values which are stored in a dynamic array.

```
PROGRAM Mean_Sd
!----------------------------------------------------------------------------------------
! Calculates mean and standard deviation ! n numbers. The values are stored in a dynamic array.
!----------------------------------------------------------------------------------------
IMPLICIT NONE
REAL,ALLOCATABLE :: X(:)
INTEGER :: N
REAL :: Mean,Sigma
PRINT *,"Input N"
READ *,N
ALLOCATE(X(N))
PRINT *,"Input N real values:"
READ *,X
Mean = SUM(X)/N
Sigma = SQRT(SUM((V-mean)**2)/(SIZE(V)-1))
PRINT *,"Mean = ",Mean
PRINT *,"Sigma= ",Sigma
DEALLOCATE(X)
END PROGRAM Mean_Sd
```

For the set: 1.1, 1.2, 1.1, 1.0, 1.5 the program will output:

Mean = 1.180000
Sigma= 0.1923538
12.5 Numerical Data Types ↔ String Conversion

Sometimes it is necessary to convert a numerical data type to a string or vice versa. Fortran provides a mechanism similar to formatted I/O statements for files, that allows you to convert numeric data from internal binary representation to 'formatted' representation.

The following examples are for INTEGER variable but of course you can use other types of variables (with proper formats):

Converting a string to an integer

```
INTEGER       :: IntVar
CHARACTER(80) :: StrVar
...
READ(UNIT=StrVar,FMT='(I5)') IntVar
```

Converting an integer to a string

```
INTEGER       :: IntVar
CHARACTER(80) :: StrVar
...
WRITE(UNIT=StrVar,FMT='(I5)') IntVar
```

The following example is the demonstration of integer to string and real to string conversion:

```
PROGRAM Conversions

INTEGER :: I = 123456
REAL    :: R = 123.456
CHARACTER(10) :: A,B

WRITE(UNIT=A, FMT='(I10)') I ! convert integer to a string
WRITE(UNIT=B, FMT='(F10.2)') R ! convert real to a string

PRINT *,"Integer I=",I
PRINT *,"String A=",A
PRINT *,"Real    R=",R
PRINT *,"String B=",B

END PROGRAM Conversions
```

Output of the program is:

```
Integer I=      123456
String  A=    123456
Real    R=   123.4560
String  B=    123.46
```

The following functions can be used to convert a string to an integer and real respectively:

```
INTEGER FUNCTION StrToInt(String) ! Converts a string to an integer
CHARACTER (*), INTENT(IN) :: String
   READ(UNIT=String,FMT='(I10)') StrToInt
END FUNCTION StrToInt

REAL FUNCTION StrToReal(String) ! Converts a string to a real
CHARACTER (*), INTENT(IN) :: String
   READ(UNIT=String,FMT='(F10.5)') StrToReal
END FUNCTION StrToReal
```
Topics Not Covered

This guide covers Fortran 90 at only a basic level and with limited depth. Intermediate and advanced topics, and extensions in Fortran 95/2003, are not covered. The following is a list of some important topics that are omitted in the guide; if you are interested in furthering your Fortran knowledge then look these up other Fortran resources.

**Pointers and linked structures** - related to memory management.
**Derived types** - combine intrinsic types into a new compound type.
**Modules (MODULE, USE)** - modular programming.
**PUBLIC, PRIVATE, SAVE, and PURE attributes** - relating to the scope of data.

There are also many more features relating to input and output, processing of multidimensional arrays, character manipulation functions and other intrinsic functions that are not covered in the guide. The programmer should also be familiar with issues such as scope, round-off errors, and numerical range.
## Appendix: List of Fortran Intrinsics

The following tables list all of the standard Fortran 95 intrinsic functions and subroutines according to their category.

Notes: values in brackets () are arguments; square brackets [] indicate optional arguments.

Single precision is assumed to be KIND=4, double precision KIND=8.

### Math Functions

See below for:
“Trigonometric and Hyperbolic Functions”, “Complex Functions”, and “Vector and Matrix Functions”.

**Notes:**
1. Most math functions are elemental, i.e. the arguments may be scaler or arrays.
2. Some math functions are defined only for a specific numerical range; exceeding a permitted range will result in a NaN or Infinite value, or a program crash.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS (X)</td>
<td>absolute value</td>
</tr>
<tr>
<td>DIM (X, Y)</td>
<td>positive difference</td>
</tr>
<tr>
<td>EXP (X)</td>
<td>$e^x$</td>
</tr>
<tr>
<td>LOG (X)</td>
<td>log$_e$x</td>
</tr>
<tr>
<td>LOG10 (X)</td>
<td>log$_{10}$x</td>
</tr>
<tr>
<td>MAX (A, B [, C,...])</td>
<td>maximum value</td>
</tr>
<tr>
<td>MIN (A, B [, C,...])</td>
<td>minimum value</td>
</tr>
<tr>
<td>MOD (A, B)</td>
<td>remainder of A/B</td>
</tr>
<tr>
<td>MODULO (A, B)</td>
<td>A modulo B</td>
</tr>
<tr>
<td>SIGN (A, B)</td>
<td>A with the sign of B</td>
</tr>
<tr>
<td>SQRT (X)</td>
<td>square-root of X</td>
</tr>
</tbody>
</table>

See also MAXVAL and MINVAL in “Array Query Functions” and PRODUCT and SUM in “Array Processing Functions”.

### Math – Trigonometric and Hyperbolic Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS (X)</td>
<td>arc-cosine of X</td>
</tr>
<tr>
<td>ASIN (X)</td>
<td>arc-sine of X</td>
</tr>
<tr>
<td>ATAN (X)</td>
<td>arc-tan of X</td>
</tr>
<tr>
<td>ATAN2 (Y, X)</td>
<td>alt. arc-tangent of X</td>
</tr>
<tr>
<td>COS (X)</td>
<td>cosine of X</td>
</tr>
<tr>
<td>COSH (X)</td>
<td>hyperbolic cosine of X</td>
</tr>
<tr>
<td>SIN (X)</td>
<td>sine of X</td>
</tr>
<tr>
<td>SINH (X)</td>
<td>hyperbolic sine of X</td>
</tr>
<tr>
<td>TAN (X)</td>
<td>tangent of X</td>
</tr>
<tr>
<td>TANH (X)</td>
<td>hyperbolic tangent of X</td>
</tr>
</tbody>
</table>

### Math – Complex Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIMAG (Z)</td>
<td>imaginary part of Z</td>
</tr>
<tr>
<td>CMPLX (X[,Y][,KIND])</td>
<td>(X + Yi)</td>
</tr>
<tr>
<td>CONJG (Z)</td>
<td>complex conjugate of Z</td>
</tr>
</tbody>
</table>

See also REAL in “Numerical Model Functions”.

### Math – Vector and Matrix Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOT_PRODUCT (V1, V2)</td>
<td>vector dot product</td>
</tr>
<tr>
<td>MATMUL (M1, M2)</td>
<td>matrix multiplication</td>
</tr>
<tr>
<td>TRANSPOSE (MATRIX)</td>
<td>matrix transpose</td>
</tr>
</tbody>
</table>
### Array Query Functions

See a text book for definitions

- `ALL (MASK [,DIM])`
- `ALLOCATED (ARRAY)`
- `ANY (MASK [,DIM])`
- `LBOUND (ARRAY [,DIM])`
- `MAXLOC (ARRAY [,DIM] [,MASK])`
- `MAXVAL (ARRAY, DIM [,MASK])`
- `MINLOC (ARRAY [,DIM] [,MASK])`
- `MINVAL (ARRAY [,DIM] [,MASK])`
- `SHAPE (SOURCE)`
- `SIZE (ARRAY [,DIM])`
- `UBOUND (ARRAY [,DIM])`

### Array Processing Functions

See a text book for definitions

- `CSHIFT (ARRAY, SHIFT [,DIM])`
- `COUNT (MASK [,DIM])`
- `EOSHIFT (ARRAY, SHIFT [,BOUNDARY] [,DIM])`
- `MERGE (A, B, MASK)`
- `PACK (ARRAY, MASK [,VECTOR])`
- `PRODUCT (ARRAY [,DIM] [,MASK])`
- `RESHAPE (SOURCE, SHAPE [,PAD] [,ORDER])`
- `SPREAD (SOURCE, DIM, N)`
- `SUM (ARRAY [,DIM] [,MASK])`
- `UNPACK (VECTOR, MASK, FIELD)`

### Character and String Functions

#### Notes:

1. Character concatenation can be achieved with the `//` operator, e.g. "forty" // "two" results in the string "fortytwo".

2. The logical operators `>=`, `>`, `<=`, and `<`, can be used to compare character strings; the processor collating sequence is used.

3. If string `A="abcdefg"`, then `A(3:5)` is the substring "cde".

- `ACHAR (I)`
- `ADJUSTL (STRING)`
- `ADJSTR (STRING)`
- `CHAR (I [,KIND])`
- `ICHAR (C)`
- `INDEX (STR1,STR2[,BACK])`
- `LEN (STRING)`
- `LEN_TRIM (STRING)`
- `LGE (STRING_A, STRING_B)`
- `LGT (STRING_A, STRING_B)`
- `LLE (STRING_A, STRING_B)`
- `LLT (STRING_A, STRING_B)`
- `REPEAT (STRING, N)`
- `SCAN (STR1,STR2[,BACK])`
- `TRIM (STRING)`
- `VERIFY (STR1,STR2[,BACK])`

ASCII character I justify string left
justify string right
processor character I ASCII position of C
processor position of C
string search
length of STRING without trailing blanks
ASCII logical A ≥ B
ASCII logical A > B
ASCII logical A ≤ B
ASCII logical A < B
repeat string N times
string search
trim trailing blanks

If `CHARACTER(25) :: Units = "centimetres and metres"` then

- `INDEX(Units,"metres") = 6` first occurrence of "metres" begins at position 6 in Units
- `INDEX(Units,"cents") = 0` there is no occurrence of "cents" in the string in Units
- `SCAN("kilo",Units) = 2` the first match from the left is the "i" of "kilo" at position 2 in the string
- `SCAN("flag",Units) = 3` the first match from the left is the "a" of "flag" at position 3 in the string
- `VERIFY("kilo",Units) = 1` character "k" at position 1 is the leftmost character that is not in Units
- `VERIFY("tennis",Units) = 0` all characters in the string "tennis" are found in Units
**Binary Bit Functions**

Argument I is type integer. Binary bit functions operate on the binary representation of the argument. If the default kind for an integer is \texttt{KIND=4}, i.e. 4 bytes, then a default integer is represented by 32 binary bits.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Argument Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTEST (I, POS)</td>
<td>Test bit position</td>
<td>I, POS</td>
</tr>
<tr>
<td>IAND (I, J)</td>
<td>Bit-by-bit logical AND</td>
<td>I, J</td>
</tr>
<tr>
<td>IBCLR (I, POS)</td>
<td>Set bit to zero</td>
<td>I, POS</td>
</tr>
<tr>
<td>IBITS (I, POS, LEN)</td>
<td>Bit substring</td>
<td>I, POS, LEN</td>
</tr>
<tr>
<td>IBSET (I, POS)</td>
<td>Set bit to one</td>
<td>I, POS</td>
</tr>
<tr>
<td>IOR (I, J)</td>
<td>Bit-by-bit inclusive-OR</td>
<td>I, J</td>
</tr>
<tr>
<td>ISHFT (I, SHIFT)</td>
<td>End-off bit shift</td>
<td>I, SHIFT</td>
</tr>
<tr>
<td>ISHFTC (I, SHIFT[,SIZE])</td>
<td>Circular bit shift</td>
<td>I, SHIFT[,SIZE]</td>
</tr>
<tr>
<td>NOT (I)</td>
<td>Bit-by-bit complement</td>
<td>I</td>
</tr>
</tbody>
</table>

See also \texttt{MVBITS} subroutine

**Rounding, Truncating, and Type Conversion Functions**

See the table below for a comparison of the rounding and truncating functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Argument Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AINT (X [,KIND])</td>
<td>Truncate to a whole real</td>
<td>X [,KIND]</td>
</tr>
<tr>
<td>ANINT (X [, KIND])</td>
<td>Round to nearest whole real</td>
<td>X [, KIND]</td>
</tr>
<tr>
<td>CEILING (X [,KIND])</td>
<td>Round up to an integer</td>
<td>X [,KIND]</td>
</tr>
<tr>
<td>FLOOR (X [,KIND])</td>
<td>Round down to an integer</td>
<td>X [,KIND]</td>
</tr>
<tr>
<td>INT (X [,KIND])</td>
<td>Truncate to an integer</td>
<td>X [,KIND]</td>
</tr>
<tr>
<td>NINT (X [,KIND])</td>
<td>Round to nearest integer</td>
<td>X [,KIND]</td>
</tr>
<tr>
<td>DPROD (X, Y)</td>
<td>Convert product to double (p)</td>
<td>X, Y</td>
</tr>
<tr>
<td>DBLE (X)</td>
<td>Convert to double precision</td>
<td>X</td>
</tr>
<tr>
<td>REAL (X [,KIND])</td>
<td>Convert to type real</td>
<td>X [,KIND]</td>
</tr>
</tbody>
</table>

Comparison of the rounding and truncating functions.

<table>
<thead>
<tr>
<th>R</th>
<th>AINT(R)</th>
<th>ANINT(R)</th>
<th>INT(R)</th>
<th>NINT(R)</th>
<th>CEILING(R)</th>
<th>FLOOR(R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.6</td>
<td>-1.0</td>
<td>-2.0</td>
<td>-1</td>
<td>-2</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>-1.5</td>
<td>-1.0</td>
<td>-2.0</td>
<td>-1</td>
<td>-2</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>-1.4</td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>-1.2</td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>-1.1</td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-0.9</td>
<td>0.0</td>
<td>-1.0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>-0.8</td>
<td>0.0</td>
<td>-1.0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>-0.6</td>
<td>0.0</td>
<td>-1.0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>-0.5</td>
<td>0.0</td>
<td>-1.0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>-0.4</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>-0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>-0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.4</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.6</td>
<td>0.0</td>
<td>1.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.8</td>
<td>0.0</td>
<td>1.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.9</td>
<td>0.0</td>
<td>1.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>1.0</td>
<td>1.0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>1.0</td>
<td>1.0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1.4</td>
<td>1.0</td>
<td>1.0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1.5</td>
<td>1.0</td>
<td>2.0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1.6</td>
<td>1.0</td>
<td>2.0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Numerical Model Functions

The term numeric model relates to the way the compiler represents numerical data in computer memory. The number of binary bits used to store a number is limited; this leads to the following model dependent limitations:

1. The numerical range of type real and type integer data is limited.
2. The precision of type real data is limited.

Numerical model functions allow the programmer to quantify these limitations and to write portable programs (giving the same results on different platforms).

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT_SIZE (I)</td>
<td>number of bits in the bit model</td>
</tr>
<tr>
<td>DIGITS (X)</td>
<td>number of significant digits</td>
</tr>
<tr>
<td>EPSILON (X)</td>
<td>almost negligible when compared to one</td>
</tr>
<tr>
<td>EXPONENT (X)</td>
<td>exponent part</td>
</tr>
<tr>
<td>FRACTION (X)</td>
<td>fractional part</td>
</tr>
<tr>
<td>HUGE (X)</td>
<td>largest number in the model</td>
</tr>
<tr>
<td>KIND (X)</td>
<td>the KIND of the value</td>
</tr>
<tr>
<td>MAXEXPONENT (X)</td>
<td>the model maximum exponent</td>
</tr>
<tr>
<td>MINEXPONENT (X)</td>
<td>the model minimum exponent</td>
</tr>
<tr>
<td>NEAREST (X, S)</td>
<td>the nearest representable value</td>
</tr>
<tr>
<td>PRECISION (X)</td>
<td>the decimal precision</td>
</tr>
<tr>
<td>RADIX (X)</td>
<td>the base of the model</td>
</tr>
<tr>
<td>RANGE (X)</td>
<td>the decimal exponent range</td>
</tr>
<tr>
<td>RRSPACING (X)</td>
<td>reciprocal of the relative spacing</td>
</tr>
<tr>
<td>SCALE (X, I)</td>
<td>exponent part change by...</td>
</tr>
<tr>
<td>SELECTED_INT_KIND (R)</td>
<td>see text book</td>
</tr>
<tr>
<td>SELECTED_REAL_KIND ([P][,R])</td>
<td>see text book</td>
</tr>
<tr>
<td>SET_EXPONENT (X, I)</td>
<td>see text book</td>
</tr>
<tr>
<td>SPACING (X)</td>
<td>spacing near the value of X</td>
</tr>
<tr>
<td>TINY (X)</td>
<td>smallest positive number</td>
</tr>
<tr>
<td>TRANSFER (SOURCE, K [,SIZE])</td>
<td>see text book</td>
</tr>
</tbody>
</table>

Other Functions

See text book.

ASSOCIATED (POINTER [,TARGET])
LOGICAL (L [,KIND])
NULL ([P])
PRESENT (A)

Subroutines

See elsewhere for details

CPU_TIME (TIME)  The processor time in seconds.
DATE_AND_TIME ([DATE [,TIME [,ZONE [,VALUES]]]])
Date and time information from the real-time clock.
MV_BITS (FROM, FROMPOS, LEN, TO, TOPOS)
A sequence of bits (bit field) is copied from one location to another
RANDOM_NUMBER (RAN) Assign the argument with numbers taken from a sequence of uniformly distributed pseudorandom numbers.
RANDOM_SEED ([SIZE [,PUT [,GET]]])
The initialisation or retrieval of pseudorandom number generator seed values.
SYSTEM_CLOCK ([COUNT [,COUNT_RATE [,COUNT_MAX]]])
Data from the processor’s real-time clock.
Fortran 2003; Access to command arguments and environment variables

Command arguments allow a program to take data from the execution command line. Similarly access to environment variables allows a program to take data from the operating system environment variables.

`COMMAND_ARGUMENT_COUNT()` is an inquiry function that returns the number of command arguments as a default integer scalar.

`CALL GET_COMMAND ([COMMAND, LENGTH, STATUS])` returns the entire command by which the program was invoked in the following `INTENT(OUT)` arguments:

- `COMMAND` (optional) is a default character scalar that is assigned the entire command.
- `LENGTH` (optional) is a default integer scalar that is assigned the significant length (number of characters) of the command.
- `STATUS` (optional) is a default integer scalar that indicates success or failure.

`CALL GET_COMMAND_ARGUMENT (NUMBER[, VALUE, LENGTH, STATUS])` returns a command argument.

- `NUMBER` is a default integer `INTENT(IN)` scalar that identifies the required command argument. Useful values are those between 0 and `COMMAND_ARGUMENT_COUNT()`.
- `VALUE` (optional) is a default character `INTENT(OUT)` scalar that is assigned the value of the command argument.
- `LENGTH` (optional) is a default integer `INTENT(OUT)` scalar that is assigned the significant length (number of characters) of the command argument.
- `STATUS` (optional) is a default integer `INTENT(OUT)` scalar that indicates success or failure.

`CALL GET_ENVIRONMENT_VARIABLE (NAME[, VALUE, LENGTH, STATUS, TRIM_NAME])` obtains the value of an environment variable.

- `NAME` is a default character `INTENT(IN)` scalar that identifies the required environment variable. The interpretation of case is processor dependent.
- `VALUE` (optional) is a default character `INTENT(OUT)` scalar that is assigned the value of the environment variable.
- `LENGTH` (optional) is a default integer `INTENT(OUT)` scalar. If the specified environment variable exists and has a value, `LENGTH` is set to the length (number of characters) of that value. Otherwise, `LENGTH` is set to 0.
- `STATUS` (optional) is a default integer `INTENT(OUT)` scalar that indicates success or failure.
- `TRIM_NAME` (optional) is a logical `INTENT(IN)` scalar that indicates whether trailing blanks in `NAME` are considered significant.

An example usage, that add two numbers input from command line, is given below:
PROGRAM Command_Line  
! Adds two integer numbers that are input from keyboard.  
! You can compile and run the program via g95 compiler such that  
! (Assuming the name of the program file add.f90)  
! $ g95 add.f90 -o add  (compile)  
! $ add number1 number2  (run)  
-------------------------------------------------------------------
IMPLICIT NONE  
CHARACTER(LEN=20) :: Command,Arg1,Arg2  
INTEGER :: N,A,B  
CALL GET_COMMAND(Command)  
CALL GET_COMMAND_ARGUMENT(1,Arg1)  
CALL GET_COMMAND_ARGUMENT(2,Arg2)  
A = StrToInt(Arg1)  
B = StrToInt(Arg2)  
PRINT *,"Sum is ",A+B  
CONTAINS  
FUNCTION StrToInt(String)  
INTEGER :: String  
READ(UNIT=String,FMT='(I10)') StrToInt  
END FUNCTION StrToInt  
END PROGRAM Command_Line

Example executions:  

Compile via g95 compiler:  

$ g95 add.f90 -o add

Test run 1 (n=3):  

$ add 7 8 9  
Missing or too few parameters.

Test run 2 (n=1):  

$ add 7  
Missing or too few parameters.

Test run 3 (n=3):  

$ add 7 8  
Sum is 15