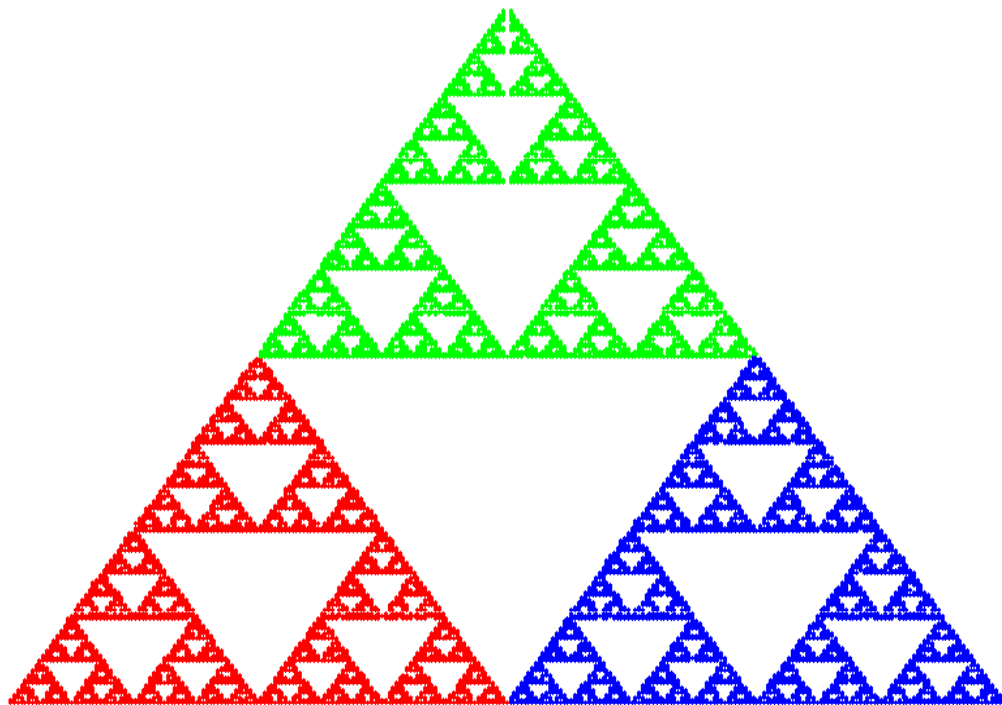


# **Introduction to Programming using Fortran 95/2003/2008**



**Ed Jorgensen**

**March 2018  
Version 3.0.51**

## Cover Diagram

The cover image is the plotted output from the *chaos game* program from chapter 11. The image was plotted with GNUplot.

## Copyright

Ed Jorgensen 2013, 2014, 2015, 2016, 2017, 2018



You are free:

- to Share — to copy, distribute and transmit the work
- to Remix — to adapt the work

Under the following conditions:

- Attribution. You must attribute the work to “Introduction to Programming using Fortran 95/2003/2008” (but not in any way that suggests that the author endorses you or your use of the work).
- Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to

- <http://creativecommons.org/licenses/by-sa/3.0/>

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.

# Table of Contents

---

<b>1</b>	<b>Introduction.....</b>	<b>1</b>
1.1	Why Learn Programming.....	1
1.2	Fortran.....	1
1.3	Complete Fortran 95/2003/2008 Documentation.....	1
1.4	What Is A Program.....	2
1.5	Operating System.....	2
<b>2</b>	<b>Computer Organization.....</b>	<b>3</b>
2.1	Architecture Overview.....	3
2.2	Compiler.....	4
2.3	Information Representation.....	4
2.3.1	Decimal Numbers.....	4
2.3.2	Binary Numbers.....	5
2.3.3	Character Representation.....	5
2.4	Exercises.....	5
2.4.1	Quiz Questions.....	5
<b>3</b>	<b>Getting Started.....</b>	<b>7</b>
3.1	Required Skills.....	7
3.2	Program Formats.....	7
3.2.1	Program Statement.....	7
3.2.2	Comments.....	8
3.2.3	Simple Output.....	8
3.2.4	Example – First Program.....	8
3.3	Text Editor.....	8
3.4	Compiling.....	9
3.4.1	Advanced Compiler Options.....	9
3.5	Executing.....	9
3.6	Exercises.....	10
3.6.1	Quiz Questions.....	10
3.6.2	Suggested Projects.....	11
<b>4</b>	<b>Fortran 95/2003/2008 – Basic Elements.....</b>	<b>13</b>
4.1	Variables.....	13
4.1.1	Variable Names.....	13
4.1.2	Keywords.....	14
4.2	Data Types.....	14
4.2.1	Integer.....	14
4.2.2	Real.....	15
4.2.3	Complex.....	15
4.2.4	Character.....	15
4.2.5	Logical.....	15
4.2.6	Historical Data Typing.....	15
4.3	Declarations.....	16

4.3.1	Declaring Variables.....	16
4.3.2	Variable Ranges.....	16
4.3.3	Type Checking.....	16
4.3.4	Initialization.....	17
4.3.5	Constants.....	17
4.4	Comments.....	17
4.5	Continuation Lines.....	18
4.5.1	Example.....	18
4.6	Declarations, Extended Size Variables.....	18
4.6.1	Integers.....	19
4.6.2	Real.....	19
4.7	Exercises.....	19
4.7.1	Quiz Questions.....	19
4.7.2	Suggested Projects.....	20
<b>5</b>	<b>Expressions.....</b>	<b>21</b>
5.1	Literals.....	21
5.1.1	Integer Literals.....	21
5.1.2	Real Literals.....	21
5.1.2.1	E-Notation.....	21
5.1.3	Complex Literals.....	22
5.1.4	Character Literals.....	22
5.1.5	Logical Constants.....	23
5.2	Arithmetic Operations.....	23
5.2.1	Assignment.....	23
5.2.2	Addition.....	23
5.2.3	Subtraction.....	24
5.2.4	Multiplication.....	24
5.2.5	Division.....	24
5.2.6	Exponentiation.....	25
5.3	Order of Operations.....	25
5.4	Intrinsic Functions.....	26
5.4.1	Mathematical Intrinsic Functions.....	26
5.4.2	Conversion Functions.....	26
5.4.3	Summary.....	27
5.5	Mixed Mode.....	27
5.6	Examples.....	28
5.7	Exercises.....	28
5.7.1	Quiz Questions.....	28
5.7.2	Suggested Projects.....	29
<b>6</b>	<b>Simple Input and Output.....</b>	<b>31</b>
6.1	Output – Write.....	31
6.1.1	Output – Print.....	32
6.2	Input – Read.....	32
6.3	Example.....	33
6.4	Exercises.....	34

6.4.1	Quiz Questions.....	34
6.4.2	Suggested Projects.....	34
<b>7</b>	<b>Program Development.....</b>	<b>37</b>
7.1	Understand the Problem.....	37
7.2	Create the Algorithm.....	38
7.3	Implement the Program.....	38
7.4	Test/Debug the Program.....	39
7.4.1	Error Terminology.....	40
7.4.1.1	Compiler Error.....	40
7.4.1.2	Run-time Error.....	40
7.4.1.3	Logic Error.....	41
7.5	Exercises.....	42
7.5.1	Quiz Questions.....	42
7.5.2	Suggested Projects.....	42
<b>8</b>	<b>Selection Statements.....</b>	<b>43</b>
8.1	Conditional Expressions.....	43
8.2	Logical Operators.....	44
8.3	IF Statements.....	44
8.3.1	IF THEN Statement.....	45
8.3.1.1	IF THEN Statement, Simple Form.....	45
8.3.2	IF THEN ELSE Statement.....	45
8.3.3	IF THEN ELSE IF Statement.....	46
8.4	Example One.....	47
8.4.1	Understand the Problem.....	47
8.4.2	Create the Algorithm.....	48
8.4.3	Implement the Program.....	49
8.4.4	Test/Debug the Program.....	50
8.5	SELECT CASE Statement.....	50
8.6	Example Two.....	53
8.6.1	Understand the Problem.....	53
8.6.2	Create the Algorithm.....	53
8.6.3	Implement the Program.....	53
8.6.4	Test/Debug the Program.....	54
8.7	Exercises.....	55
8.7.1	Quiz Questions.....	55
8.7.2	Suggested Projects.....	56
<b>9</b>	<b>Looping.....</b>	<b>59</b>
9.1	Counter Controlled Looping.....	59
9.2	EXIT and CYCLE Statements.....	61
9.3	Counter Controlled Example.....	61
9.3.1	Understand the Problem.....	62
9.3.2	Create the Algorithm.....	62
9.3.3	Implement the Program.....	62
9.3.4	Test/Debug the Program.....	63

9.4	Conditional Controlled Looping.....	63
9.5	Conditionally Controlled Loop Example.....	65
9.5.1	Understand the Problem.....	65
9.5.2	Create the Algorithm.....	65
9.5.3	Implement the Program.....	66
9.5.4	Test/Debug the Program.....	67
9.6	Exercises.....	67
9.6.1	Quiz Questions.....	67
9.6.2	Suggested Projects.....	69
<b>10</b>	<b>Formatted Input/Output.....</b>	<b>71</b>
10.1	Format.....	71
10.2	Format Specifiers.....	71
10.3	Integer Format Specifier.....	72
10.4	Real Format Specifier.....	73
10.5	Horizontal Positioning Specifiers.....	73
10.6	Logical Format Specifier.....	74
10.7	Character Format Specifier.....	74
10.8	Advance Clause.....	75
10.9	Example.....	76
10.9.1	Understand the Problem.....	76
10.9.2	Create the Algorithm.....	76
10.9.3	Implement the Program.....	77
10.9.4	Test/Debug the Program.....	79
10.10	Exercises.....	79
10.10.1	Quiz Questions.....	79
10.10.2	Suggested Projects.....	80
<b>11</b>	<b>Characters and Strings.....</b>	<b>83</b>
11.1	Character and String Constants.....	83
11.2	Character Variable Declaration.....	84
11.3	Character Variable Initialization.....	84
11.4	Character Constants.....	84
11.5	Character Assignment.....	85
11.6	Character Operators.....	85
11.7	Character Substrings.....	85
11.8	Character Comparisons.....	86
11.9	Intrinsic Character Operations.....	86
11.10	Example.....	87
11.10.1	Understand the Problem.....	87
11.10.2	Create the Algorithm.....	87
11.10.3	Implement the Program.....	88
11.10.4	Test/Debug the Program.....	89
11.11	Exercises.....	89
11.11.1	Quiz Questions.....	89
11.11.2	Suggested Projects.....	90

<b>12</b>	<b>File Operations.....</b>	<b>91</b>
12.1	File Open.....	91
12.2	File Write.....	92
12.3	Stop Statement.....	92
12.4	File Read.....	93
12.5	Rewind.....	93
12.6	Backspace.....	93
12.7	Close File.....	94
12.8	Example.....	94
12.8.1	Understand the Problem.....	94
12.8.2	Create the Algorithm.....	94
12.8.3	Implement the Program.....	95
12.8.4	Test/Debug the Program.....	96
12.9	Exercises.....	96
12.9.1	Quiz Questions.....	97
12.9.2	Suggested Projects.....	97
<b>13</b>	<b>Single Dimension Arrays.....</b>	<b>99</b>
13.1	Array Declaration.....	100
13.1.1	Static Declaration.....	100
13.1.2	Static Array Declaration.....	100
13.1.3	Dynamic Array Declaration.....	101
13.1.3.1	Dynamic Array Allocation.....	101
13.2	Accessing Array Elements.....	102
13.2.1	Array Bounds.....	102
13.3	Implied Do-Loop.....	103
13.4	Intrinsic Functions.....	103
13.5	Initializing Arrays.....	104
13.6	Example.....	104
13.6.1	Understand the Problem.....	104
13.6.2	Create the Algorithm.....	105
13.6.3	Implement the Program.....	106
13.6.4	Test/Debug the Program.....	108
13.7	Arrays of Strings.....	109
13.8	Exercises.....	109
13.8.1	Quiz Questions.....	109
13.8.2	Suggested Projects.....	110
<b>14</b>	<b>Multidimensional Arrays.....</b>	<b>113</b>
14.1	Array Declaration.....	113
14.1.1	Static Declaration.....	114
14.1.2	Dynamic Declaration.....	114
14.1.3	Dynamic Array Allocation.....	114
14.2	Accessing Array Elements.....	115
14.3	Example.....	116
14.3.1	Understand the Problem.....	116
14.3.2	Create the Algorithm.....	117

14.3.3	Implement the Program.....	118
14.3.4	Test/Debug the Program.....	119
14.4	Exercises.....	119
14.4.1	Quiz Questions.....	119
14.4.2	Suggested Projects.....	120
<b>15</b>	<b>Subprograms.....</b>	<b>123</b>
15.1	Subprogram Types.....	123
15.2	Program Layout.....	123
15.2.1	Internal Routines.....	124
15.2.2	External Routines.....	124
15.3	Arguments.....	124
15.3.1	Argument Intent.....	124
15.4	Variable Scope.....	124
15.5	Using Functions and Subroutines.....	124
15.5.1	Argument Passing.....	125
15.6	Functions.....	125
15.6.1	Intrinsic Functions.....	126
15.6.2	User-Defined Functions.....	126
15.6.2.1	Side Effects.....	126
15.7	Subroutines.....	127
15.8	Example.....	128
15.8.1	Understand the Problem.....	129
15.8.2	Create the Algorithm.....	130
15.8.3	Implement the Program.....	131
15.8.4	Test/Debug the Program.....	132
15.9	Exercises.....	132
15.9.1	Quiz Questions.....	132
15.9.2	Suggested Projects.....	134
<b>16</b>	<b>Derived Data Types.....</b>	<b>137</b>
16.1	Definition.....	137
16.2	Declaration.....	138
16.3	Accessing Components.....	138
16.4	Example One.....	139
16.4.1	Understand the Problem.....	139
16.4.2	Create the Algorithm.....	140
16.4.3	Implement the Program.....	140
16.4.4	Test/Debug the Program.....	142
16.5	Arrays of Derived Data.....	143
16.6	Example Two.....	144
16.6.1	Understand the Problem.....	144
16.6.2	Create the Algorithm.....	144
16.6.3	Implement the Program.....	145
16.6.4	Test/Debug the Program.....	148
16.7	Exercises.....	148
16.7.1	Quiz Questions.....	148



16.7.2	Suggested Projects.....	149
<b>17</b>	<b>Modules.....</b>	<b>151</b>
17.1	Module Declaration.....	151
17.2	Use Statement.....	152
17.3	Updated Compilation Commands.....	152
17.4	Module Example Program.....	153
17.4.1	Understand the Problem.....	153
17.4.2	Create the Algorithm.....	153
17.4.3	Implement the Program.....	154
17.4.3.1	Main Program.....	154
17.4.3.2	Module Routines.....	155
17.4.4	Compile the Program.....	156
17.4.5	Test/Debug the Program.....	156
17.5	Exercises.....	156
17.5.1	Quiz Questions.....	156
17.5.2	Suggested Projects.....	157
<b>18</b>	<b>Recursion.....</b>	<b>159</b>
18.1	Recursive Subroutines.....	159
18.2	Recursive Print Binary Example.....	160
18.2.1	Understand the Problem.....	160
18.2.2	Create the Algorithm.....	160
18.2.3	Implement the Program.....	161
18.2.4	Test/Debug the Program.....	162
18.3	Recursive Functions.....	162
18.4	Recursive Factorial Example.....	162
18.4.1	Understand the Problem.....	162
18.4.2	Create the Algorithm.....	163
18.4.3	Implement the Program.....	163
18.4.4	Test/Debug the Program.....	164
18.5	Recursive Factorial Function Call Tree.....	165
18.6	Exercises.....	166
18.6.1	Quiz Questions.....	166
18.6.2	Suggested Projects.....	166
<b>19</b>	<b>Character String / Numeric Conversions.....</b>	<b>169</b>
19.1	Character String to Numeric Conversion.....	169
19.2	Numeric to Character String Conversion.....	171
19.3	Exercises.....	172
19.3.1	Quiz Questions.....	172
19.3.2	Suggested Projects.....	172
<b>20</b>	<b>System Services.....</b>	<b>175</b>
20.1	Date and Time.....	175
20.1.1	Date and Time Options.....	175
20.1.2	Date and Time Example Program.....	176

20.2	Command Line Arguments.....	178
20.2.1	Argument Count.....	179
20.2.2	Get Arguments.....	179
20.2.3	Command Line Arguments, Example Program.....	180
20.3	Exercises.....	182
20.3.1	Quiz Questions.....	183
20.3.2	Suggested Projects.....	183
<b>21</b>	<b>Appendix A – ASCII Table.....</b>	<b>185</b>
<b>22</b>	<b>Appendix B – Start-Up Instructions.....</b>	<b>187</b>
22.1	Working Files.....	187
22.2	Obtaining The Compiler.....	187
22.2.1	Windows Download and Install.....	187
22.2.2	MacOS Download and Install.....	188
22.3	Windows Compilation.....	188
22.3.1	Command Prompt Window.....	188
22.3.2	Device and Directory.....	189
22.3.3	Compiler Installation Verification.....	189
22.3.4	Compilation.....	189
22.3.5	Executing.....	190
22.3.6	Example.....	190
22.4	MacOS Compilation.....	190
22.4.1	Terminal Window.....	191
22.4.2	Device and Directory.....	191
22.4.3	Compiler Installation Verification.....	192
22.4.4	Compilation.....	192
22.4.5	Executing.....	193
22.4.6	Example.....	193
<b>23</b>	<b>Appendix C – Random Number Generation.....</b>	<b>195</b>
23.1	Initialization.....	195
23.2	Generating Random Number.....	195
23.3	Example.....	196
23.4	Example.....	197
<b>24</b>	<b>Appendix D – Intrinsic Functions.....</b>	<b>199</b>
24.1	Conversion Functions.....	199
24.2	Integer Functions.....	199
24.3	Real Functions.....	200
24.4	Character Functions.....	200
24.5	Complex Functions.....	201
24.6	Array Functions.....	201
24.7	System Information Functions.....	202
<b>25</b>	<b>Appendix E – Visualization with GNUplot.....</b>	<b>203</b>
25.1	Obtaining GNUplot.....	203

25.2	Formatting Plot Files.....	203
25.2.1	Header.....	204
25.2.2	Footer.....	204
25.3	Plotting Files.....	204
25.4	Example.....	204
25.4.1	Plot Program.....	205
25.4.2	Plot File.....	206
25.4.3	Plot Output.....	206
<b>26</b>	<b>Appendix F – Quiz Question Answers.....</b>	<b>207</b>
26.1	Quiz Question Answers, Chapter 1.....	207
26.2	Quiz Question Answers, Chapter 2.....	207
26.3	Quiz Question Answers, Chapter 3.....	207
26.4	Quiz Question Answers, Chapter 4.....	208
26.5	Quiz Question Answers, Chapter 5.....	208
26.6	Quiz Question Answers, Chapter 6.....	209
26.7	Quiz Question Answers, Chapter 7.....	209
26.8	Quiz Question Answers, Chapter 8.....	209
26.9	Quiz Question Answers, Chapter 9.....	210
26.10	Quiz Question Answers, Chapter 10.....	211
26.11	Quiz Question Answers, Chapter 11.....	212
26.12	Quiz Question Answers, Chapter 12.....	212
26.13	Quiz Question Answers, Chapter 13.....	212
26.14	Quiz Question Answers, Chapter 14.....	213
26.15	Quiz Question Answers, Chapter 15.....	214
26.16	Quiz Question Answers, Chapter 16.....	215
26.17	Quiz Question Answers, Chapter 17.....	216
26.18	Quiz Question Answers, Chapter 18.....	216
26.19	Quiz Question Answers, Chapter 19.....	217
26.20	Quiz Question Answers, Chapter 20.....	218
<b>27</b>	<b>Appendix G – Fortran 95/2003/2008 Keywords.....</b>	<b>219</b>
	<b>Index.....</b>	<b>223</b>

## Illustration Index

Illustration 1: Computer Architecture.....	3
Illustration 2: Fortran 95/2003/2008 Compile Process.....	4
Illustration 3: Factorial Recursion Tree.....	165



# 1 Introduction

---

Computers are everywhere in our daily lives. Between the desktop, laptop, phone, bank, and vehicle, it is difficult to completely get away from computers. It only makes sense to learn a little about how a computer really works.

This text provides an introduction to programming and problem solving using the Fortran 95/2003/2008 programming language. This introduction is geared for non-computer science majors. The primary focus is on an introduction to problem solving and algorithm development. As such, many details of the Fortran 95/2003/2008 language are omitted.

## 1.1 Why Learn Programming

For science and technical majors, computers are used extensively in all aspects of every discipline. Learning the basics of how computers work and how programs are created is useful and directly applicable.

Programming a computer is basically applied problem solving. You are given a problem, the problem is analyzed, a solution is developed, and then that solution is implemented and tested. Enhanced problem solving skills can be applied to any endeavor. These basic skills, once developed, can be applied to other programming languages, MATLAB, or even spreadsheet macros.

Unfortunately, learning programming and how a computer really works may ruin some B movies.

## 1.2 Fortran

Fortran is a programming language often used by the scientific community. Its name is a contraction of FORMula TRANslation. FORTRAN is one of the earliest programming languages and was designed specifically for solving scientific and engineering computational problems.

This text utilizes the Fortran 95/2003/2008 standard. Older versions of Fortran, like Fortran 77, are not referenced. The older Fortran versions have less features and require additional, often burdensome, formatting requirements.

## 1.3 Complete Fortran 95/2003/2008 Documentation

This text is not a comprehensive or complete reference to the Fortran 95/2003/2008 language. The entire GNU Fortran compiler documentation is available on-line at the following location:

**<http://gcc.gnu.org/onlinedocs/gcc-5.3.0/gfortran/>**

If this location changes, a web search will be able to find the new location.

## 1.4 What Is A Program

A *computer program* is a series of instructions which enables the computer to perform a designated task. As this text will demonstrate, a computer must be told what to do in precise, step-by-step detail. These steps might include obtaining data, arithmetic operations (addition, subtraction, multiplication, division, etc.), data storage, and information output. The computer will perform these tasks as instructed, even if they don't always make sense. Consequently, it is the programmer who must develop a solution to the problem.

## 1.5 Operating System

The Operating System, or OS, is an interface between the user and the hardware (CPU, memory, screen, disk drive, etc.). The OS is responsible for the management of the hardware, coordination of activities, and the sharing of the resources of the computer that acts as a host for computing applications run on the machine. The common operating systems include various versions of Windows, MAC OS X, and UNIX/Linux. Programs written in Fortran will work on these operating systems.

## 2 Computer Organization

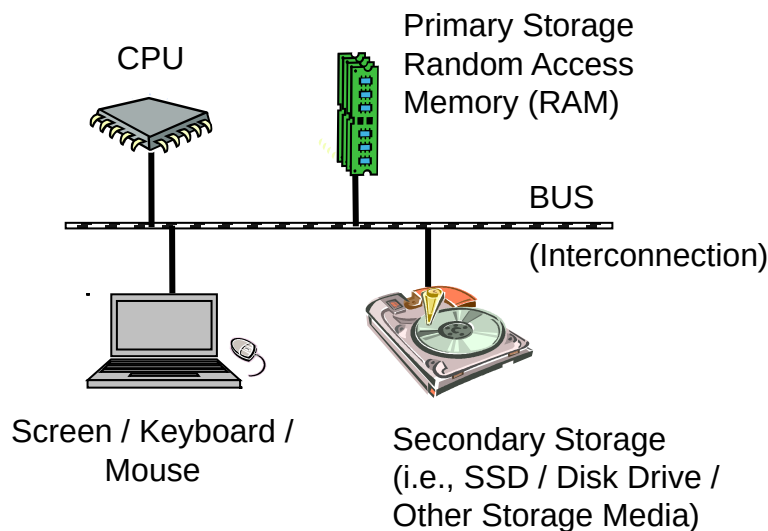
---

Before writing programs, it is useful to understand some basics about how a computer is organized. This section provides a brief, high-level overview of the basic components of a computer and how they interact.

### 2.1 Architecture Overview

The basic components of a computer include a Central Processing Unit (CPU), Primary Storage or Random Access Memory (RAM), Secondary Storage, and Input/Output devices (i.e., screen, keyboard, and mouse), and an interconnection referred to as BUS. The secondary storage may be a Solid-State Drive (SSD), disk drive, or other type of secondary storage media.

A very basic diagram of a computer architecture is as follows:

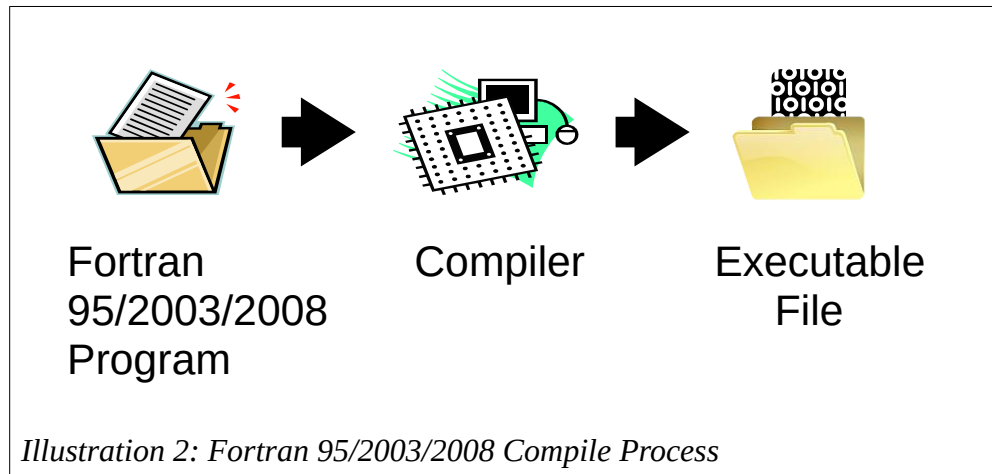


*Illustration 1: Computer Architecture*

Programs and data are typically stored on the disk drive. When a program is executed, it must be copied from the disk drive into the RAM memory. The CPU executes the program from RAM. This is similar to storing a term paper on the disk drive, and when writing/editing the term paper, it is copied from the disk drive into memory. When done, the updated version is stored back to the disk drive.

## 2.2 Compiler

Programs can be written in the Fortran programming language. However, the CPU does not read Fortran directly. Instead, the Fortran program that we create will be converted into binary (1's and 0's) by the *compiler*. These 1's and 0's are typically referred to as machine language. The CPU will read the instructions and information, represented in binary as machine language, and perform the commands from the program.



The compiler is a program itself and is required in order to create the files needed to execute programs written in Fortran 95/2003/2008.

## 2.3 Information Representation

All information, including numbers, characters, and instructions are represented in the computer in binary (1's and 0's). The information, numbers in this example, is converted into binary representation (1's and 0's) for storage in the computer. Fortunately, this is generally done transparently.

### 2.3.1 Decimal Numbers

Before discussing binary numbers, a brief review of the decimal system is presented. The number "1234" as,

Thousands	Hundreds	Tens	Ones
$10^3$	$10^2$	$10^1$	$10^0$
1000	100	10	1
1	2	3	4

Which means,

$$1234 = 1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1$$

$$1234 = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

The decimal system is *base 10* using the digits 0 through 9.



### 2.3.2 Binary Numbers

A bit is the basic unit of information in computing and digital communications. A bit can have only one of two values, 0 or 1. The term bit is a contraction of binary digit.

The binary system, as well as its math, operates in *base 2*, using two symbols, 0 and 1.

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
128	64	32	16	8	4	2	1
0	0	0	0	1	1	0	1

In base 2, we put the digits 0 or 1 in columns  $2^0$ ,  $2^1$ ,  $2^2$ , and so on. For example,

$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 1$$

Which in decimal is  $13_{10}$ .

A set of 8 bits is referred to as a byte. Computer data is typically allocated in bytes or sets of bytes.

### 2.3.3 Character Representation

Characters are represented using the American Standard Code for Information Interchange (ASCII). Refer to Appendix A.

## 2.4 Exercises

Below are some quiz questions based on this chapter.

### 2.4.1 Quiz Questions

Below are some quiz questions.

- 1) How is information represented in the computer?
- 2) What does the Fortran *compiler* do?
- 3) What architecture component connects the memory to the CPU?
- 4) What are the following binary values in decimal?
  - a)  $0000101_2$
  - b)  $0001001_2$
  - c)  $0001101_2$
  - d)  $0010101_2$
- 5) How are characters represented in the computer?

- 6) Where are programs stored when the computer is turned off?
- 7) Where must programs be located when they are executing?

## 3 Getting Started

---

This section provides a brief overview of how to get started. This includes the general process for creating a very simple program, compiling, and executing the program. Some detailed steps regarding working with files, obtaining the compiler, and compiling a program are included in *Appendix B, Windows Start-up Instructions*.

### 3.1 Required Skills

Before starting, you should have some basic computer skills, including the following:

- Ability to use a web browser
- Basic understanding of hierarchical computer directory structure
- File manipulation (create, delete, rename, move, etc.)
- Usage of the command prompt (Windows) or terminal interface (Unix, MAC)
- Ability to edit a text file
  - Includes selecting and learning a text editor (i.e., Notepad, Notepad++, emacs, etc.)

If you are unsure about any of these requirements you will need to learn them before continuing. Fortunately, they are not difficult. Additionally, there are numerous tutorials available on the Web.

The following sections assume that the Fortran 95/2003/2008 compiler is installed and available. For additional information regarding obtaining and installing the compiler, refer to Appendix B. The Fortran 95/2003/2008 compiler is available for download at no cost.

### 3.2 Program Formats

Fortran 95/2003/2008 programs must be written and formatted in a specific manner. The following sections summarize the basic program elements followed by a simple example.

#### 3.2.1 Program Statement

A Fortran 95/2003/2008 program is started with a program statement, '**program <name>**', and ended with an end program statement, '**end program <name>**'. Refer to the example first program to see an example of these statements. The program name for **<name>** is chosen by the program author and would typically reflect something related to what the program does.

The name used may not be used again for other program elements (such as variables described in the next chapter). The program name must start with a letter, followed by letters, numbers, or an underscore (" \_ ") and may not be longer than 32 characters. Capital letters are treated the same way as lower-case letters. Refer to the sample program in the following sections for an example.

### 3.2.2 Comments

Comments are information for the programmer and are not read by the computer. For example, comments typically include information about the program. For programming assignments, the comments should include the programmer name, assignment number, and a brief description of the program. In Fortran, the exclamation mark (!) denotes a comment. Any characters after the exclamation mark (!) are ignored by the compiler and thus are comments as shown in following example.

### 3.2.3 Simple Output

A program can display a simple message to the screen by using the *write* statement. For example:

```
write (*,*) "Hello World"
```

Will display the message **Hello World** to the screen. Additional information regarding the write statement and outputting information is provided in later chapters.

### 3.2.4 Example – First Program

The following trivial program illustrates the initial formatting requirements.

```
! Simple Example Program  
program first  
  
    write (*,*) "Hello World."  
  
    end program first
```

In this example, the program is named 'first'. This file, provided as input to the compiler, is typically referred to as the *source* file.

## 3.3 Text Editor

The first step is to create a text file named **hw.f95** using a text editor. It is useful to place programs and various files into a working directory. This way the files can be easily found and not confused with other, unrelated files and data. The **hw.f95** file should be created and placed in a user created working directory. For this example, a working directory named 'fortran' was created and the file is then created and edited in that location.

A file name is typically comprised of two parts; a name and an extension. In this example, the file name is **hw** and the extension is **.f95**. The usual extension for this and future programs will be **.f95** which indicates that the file is a Fortran 95/2003/2008 source file.

The following examples will use the **hw.f95** file name. If desired, a different file name may be used. However, the name will need to be adjusted for the compiler and execute steps in the following sections.

### 3.4 Compiling

Once the program is typed into a file, the file must be *compiled*. Compiling will convert the human-readable Fortran program, or source file, into a machine-readable version (in binary).

In order to compile, the command prompt (Windows) or terminal interface (Unix, MAC) is required. This interface will allow commands to be typed directly into the computer (instead of using a mouse). Once started, it is typically necessary to change directories (from the default location) to the location of where the **hw.f95** source file was located (from the previous steps). Changing directories is typically done with a **cd <directoryName>** command. For example, **cd fortran** (which is the name of the directory used in this example). The prompt typically changes to include the current directory location.

In the example below, the commands typed by the user are displayed in bold. The regular (non-bolded) text refers to prompts or other information displayed by the computer (which need not be typed).

To compile the example program, the following command would be entered:

```
C:\fortran> gfortran -o hw hw.f95
```

This command will tell the '**gfortran**' compiler to read the file **hw.f95** and, if there are no errors, create an output file referred to as an executable file. On Windows based machines the executable file is named **hw.exe**. And on Unix or Mac based machines, the executable is named **hw** (no extension). If there is an error, the compiler will generate an error message, sometimes cryptic, and provide a line number. Such errors are usually the result of mistyping one of the instructions. Any errors must be resolved before continuing.

#### 3.4.1 Advanced Compiler Options

In addition, to the basic compilation options, sometimes additional instructions, or options, may be required. For example, when using arrays (chapter 9), an option for bounds checking is typically desired.

For example, to compile with bounds checking, the following command might be entered:

```
C:\fortran> gfortran -fcheck=bounds -o hw hw.f95
```

This command will tell the '**gfortran**' compiler to include bounds checking. More information regarding arrays and bounds checking is addressed in chapters 13 and 14.

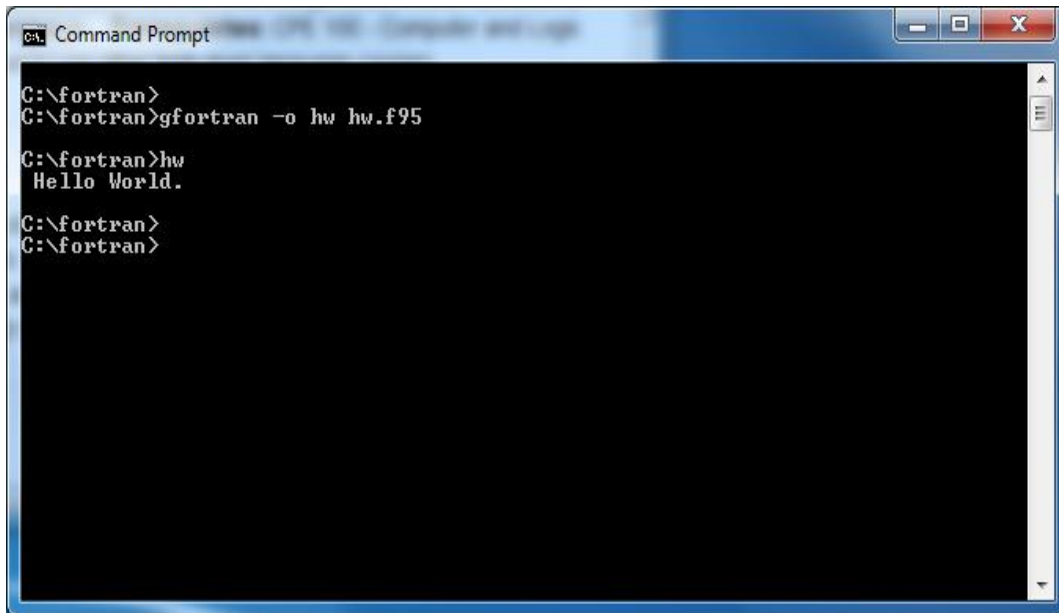
### 3.5 Executing

To execute or run a program on a Windows based machine, type the name of the executable file. For example, to execute or run the **hw.exe** program:

```
C:\fortran> hw
Hello World.
C:\fortran>
```

Which will execute the example program and display the “Hello World.” message to the screen.

A more complete example is as follows:



```
C:\fortran>
C:\fortran>gfortran -o hw hw.f95
C:\fortran>hw
Hello World.
C:\fortran>
C:\fortran>
```

It is not necessary to type the extension (i.e., “.exe”) portion of the file name. It should be noted that the space prior to the “H” is not produced by the program, but is how the system displays output.

To execute or run a program on a Unix or MAC based machine, type “./” and the name of the executable file. For example, to execute or run the **hw** program:

```
c:\fortran> ./hw
Hello World.
c:\fortran>
```

The output ('Hello World.' as displayed on the screen) will be the same for Windows, Unix, or MAC based machines.

## 3.6 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 3.6.1 Quiz Questions

Below are some quiz questions.

- 1) What is the typical name of the input file for the compiler?
- 2) What is the typical name of the output file from the compiler?
- 3) Fortran program must start with and end with what statement?
- 4) How are Fortran comments marked?

- 5) What is the typical Fortran 95/2003/2008 source file extension?
- 6) What is the typical Fortran 95/2003/2008 compiler output file, or executable file, extension (after the program is compiled)?

### **3.6.2      *Suggested Projects***

Below are some suggested projects.

- 1) Create a working directory for the storage of program files (on the computer being used).
- 2) Obtain and install the GNU Fortran 95/2003/2008 compiler on a suitable computer. Refer to Appendix B as needed.
- 3) Type in the hello world program, compile, and execute the program.
- 4) Update the example program to display your name in addition to the Hello World message.





## 4 Fortran 95/2003/2008 – Basic Elements

---

Before beginning to writing programs, it is necessary to know some of the basic elements of the Fortran language. This section describes some of the basic elements of Fortran. Additional information will be added in later sections.

### 4.1 Variables

The basic concept in a program is the concept of a variable. Variables in a program are like variables in an algebraic expression. They are used to hold values and then write mathematical expressions using them. Fortran allows us to have variables of different types.

A variable can hold one value at a time. If another value is placed in the variable, the previous value is over-written and lost.

Variable Name →

42
----

Variables must be declared at the start of the program.

#### 4.1.1 Variable Names

Each variable must be named. The variable name is how variables, which are memory locations, are referred to by the program. A variable name must start with a letter, followed by letters, numbers, or an underscore (“\_”) and may not be longer than 32 characters. Capital letters are treated the same way as lower-case letters, (i.e., “AAA” is the same variable as “aaa”).

For example, some valid variable names are as follows:

```
x
today
next_month
summation10
```

Some invalid examples include:

```
1today
this_is_a_variable_name_with_way_way_too_many_characters_in_it
next@month
next month
today!
```

Note that the space (between next and month) or the special character, @, is not allowed. Additionally, each variable must have a type associated as explained in the following sections.

### 4.1.2 Keywords

In programming, a keyword is a word or identifier that has a special meaning in a programming language. For example, in the “hello world” Fortran program from the previous chapter, the word *program* has a special meaning in that it is used to note the start or beginning of a program. Additionally, the word *write* has a special meaning to note an output action (e.g., writing some information to an output device, like the screen).

Such keywords are reserved in that they cannot be used for anything else such as variable names. That is, a variable name of *program* or *write* is not allowed.

As additional Fortran 95/2003/2008 statements and language constructs are explained, more keywords will be identified. In general, words used for Fortran language statements, attributes, and constructs will likely be keywords. A complete list of keywords or reserved words is located in Appendix G.

## 4.2 Data Types

Fortran, like many other high level programming languages, supports several different *data types* to make data manipulation easier. The most frequently used data types are integer and floating-point. Other data types are complex numbers, characters and logical data.

In a Fortran statement, data can appear either as a literal (e.g., an actual value such as 3.14159, 16, -5.4e-4) or as a variable name which identifies a location in memory to store the data.

The five basic Fortran 95/2003/2008 data types are as follows:

Type	Description
<b>integer</b>	Variable that is an integer or whole number (not a fraction) that can be positive, negative, or zero.
<b>real</b>	Variable that can be set to a real number.
<b>complex</b>	Variable that can be set to a complex number.
<b>character</b>	Variable that is a character or sequence of characters.
<b>logical</b>	Variable that can only be set to <i>.true.</i> or <i>.false.</i>

It is also possible to have derived types and pointers. Both of these can be useful for more advanced programs and are described in later chapters.

### 4.2.1 Integer

An integer<sup>1</sup> is a whole number (not a fraction) that can be positive, negative, or zero. Examples include the numbers 10, 0, -25, and 5,148. Integers are the numbers people are most familiar with, and they serve a crucial role in mathematics and computers. All integers are whole numbers, so operations like one divided by two (1/2) is 0 since the result must be a whole number. For integer division, no rounding will occur as the fractional part is truncated.

1 For more information regarding integers, refer to: <http://en.wikipedia.org/wiki/Integer>

### 4.2.2 Real

A real number<sup>2</sup> includes the fractional part, even if the fractional part is 0. Real numbers, also referred to as floating-point numbers, include both rational numbers and irrational numbers. Examples of irrational numbers or numbers with repeating decimals include  $\pi$ ,  $\sqrt{2}$  and  $e$ . Additional examples include 1.5, 5.0, and 3.14159. Fortran 95/2003/2008 will accept 5. as 5.0. All examples in this text will include the “.0” to ensure clarity.

### 4.2.3 Complex

A complex number<sup>3</sup>, in mathematics, is a number comprising a real number and an imaginary number. It can be written in the form of  $a + bi$ , where  $a$  and  $b$  are real numbers, and the  $i$  is the standard imaginary unit with the property that  $i^2 = -1.0$ . The complex numbers contain the ordinary real numbers, but extend them by adding in extra numbers like an extra dimension. This data type is not used extensively, but can be useful when needed.

### 4.2.4 Character

A character<sup>4</sup> is a symbol like a letter, numerical digit, or punctuation. A string<sup>5</sup> is a sequence or set of characters. Characters and strings are typically enclosed in quotes. For example, the upper case letter “Z” is a character and “Hello World” is a string. The characters are represented in a standardized format referred to as ASCII.

### 4.2.5 Logical

A logical<sup>6</sup> is only allowed to have two values, *true* or *false*. A logical can also be referred to as a boolean. In Fortran, the true and false values are formally expressed as **.true.** or **.false.** which are also called logical constants. The leading and trailing . (period) are required for the true and false constants.

### 4.2.6 Historical Data Typing

Unless a variable was explicitly typed, older versions of Fortran implicitly assumed a type for a variable depending on the first letter of its name. Thus, if not explicitly declared, a variable whose name started with one of the letters **I** through **N** was assumed to be an integer; otherwise it was assumed to be real. To allow older code to run, Fortran 95/2003/2008 permits *implicit typing*. However, this is poor practice, can be confusing, and often leads to errors. So, we will include the **IMPLICIT NONE** statement at the start of all programs. This turns off implicit typing and the compiler will identify and flag any variable not defined. This will help make some errors, such as misspelling a variable name, significantly easier to locate.

2 For more information regarding real numbers, refer to: [http://en.wikipedia.org/wiki/Real\\_numbers](http://en.wikipedia.org/wiki/Real_numbers)

3 For more information regarding complex numbers, refer to: [http://en.wikipedia.org/wiki/Complex\\_number](http://en.wikipedia.org/wiki/Complex_number)

4 For more information regarding characters, refer to: [http://en.wikipedia.org/wiki/Character\\_\(computing\)](http://en.wikipedia.org/wiki/Character_(computing))

5 For more information regarding strings, refer to: [http://en.wikipedia.org/wiki/String\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/String_(computer_science))

6 For more information regarding logicals, refer to: [http://en.wikipedia.org/wiki/Boolean\\_data\\_type](http://en.wikipedia.org/wiki/Boolean_data_type)

## 4.3 Declarations

**Fortran variables must be declared before executable statements. This section provides an introduction to how variables are declared.**

### 4.3.1 Declaring Variables

Declaring variables formally defines the data type of each variable and sets aside a memory location. This is performed by a type declaration statement in the form of:

```
<type> :: <list of variable names>
```

The type must be one of the predefined data types (integer, real, complex, character, logical) as outlined in the previous section. Declarations are placed in the beginning of the program (after the program statement).

For example, to define an integer variable *today*,

```
integer :: today
```

Additional examples include:

```
integer :: today, tomorrow, yesterday
real :: ans2
complex :: z
logical :: answer
character :: myletter
```

The declarations can be entered in any order.

Additional information regarding character variables is provided in chapter 11.

### 4.3.2 Variable Ranges

The computer has a predefined amount of space that can be used for each variable. This directly impacts the size, or range, of the number that can be represented.

For example, an integer value can range between  $-2,147,483,648$  and  $+2,147,483,647$ . Fortunately, this is large enough for most purposes.

The range for real values is more complex. The range is approximately  $\pm 1.7 \times 10^{\pm 38}$  supporting about 7 digits of precision.

### 4.3.3 Type Checking

The variable type declaration is enforced by the compiler. For example, if a variable is declared as an integer, only an integer value (a whole number) is allowed to be assigned to that variable. Attempting to assign a value of 1.75 to an integer variable could cause problems related to loss of precision. This restriction is related to the fact that the internal representations for various types are very different and not directly compatible. The compiler can sometimes recognize a type mismatch and implicitly (automatically) perform a conversion. If this is done automatically, it is not always clear and could lead to errors. As such, it is generally considered poor programming practice.

Conversions between types should be performed explicitly. Later chapters provide specific examples of how this can be accomplished.

When initially learning to program, this may seem quite annoying. However, this type mismatch can cause subtle errors that are difficult to find.

### 4.3.4 Initialization

It is possible to declare a variable and set its initial value at the same time. This initialization is not required, but can sometimes be convenient.

For example, to define an integer variable **today'sdate** and set it to the 15<sup>th</sup> of the month:

```
integer :: today'sdate=15
```

Additional examples include:

```
integer :: today'sday=15, tomorrow=16, yesterday=14  
real :: ave = 5.5
```

Spaces or no spaces are allowed between the variable name. The variable declaration may or may not include an equal sign (for initialization). Commas are used to separate multiple variable declarations on the same line. Variables initialized at declaration can be changed later in the program as needed.

### 4.3.5 Constants

A *constant* is a variable that cannot be changed during program execution. For example, a program might declare a variable for  $\pi$  and set it to 3.14159. It is unlikely that a program would need to change the value for  $\pi$ . The parameter qualifier will declare the variable as a constant, set the initial value, and not allow that initial value to be altered during the program execution.

For example, the declarations:

```
real, parameter :: pi = 3.14159  
integer, parameter :: width = 1280
```

will set the variable **pi** to 3.14159 and **width** to 1280 and ensure that they cannot be changed while the program is executing.

## 4.4 Comments

As previously noted, comments are information for the programmer and ignored by the compiler. The exclamation mark (!) denotes a comment. Any information after the exclamation mark (!) is ignored by the compiler. In general, comments typically include information about the program. For example, a comment might include the last modification date, programmer name, and details about the update. For programming assignments, the comments might include the programmer name, assignment number, and a description of the program. The comments might include information about the approach being used, source of formulas (if applicable), or maybe data requirements such as using positive values for some geometric formulas. Commenting such reference information is strongly encouraged and will be addressed in future sections.

## 4.5 Continuation Lines

A statement must start on a new line. If a statement is too long to fit on a line, it can be continued on the next line with an ampersand ('&'). Even shorter lines can be split and continued on multiple lines for more readable formatting.

For example,

```
A = 174.5 * year                &
      + count / 100.0
```

Is equivalent to the following

```
A = 174.5 * year + count / 100.0
```

Note that the '&' is *not* part of the statement.

### 4.5.1 Example

The following trivial program illustrates the program formatting requirements and variable declarations.

```
! Example Program

program example1

  implicit none
  integer :: radius, diameter
  integer :: height=100, width=150
  real :: area, perimeter
  real :: length = 123.5, distance=413.761
  real, parameter :: pi = 3.14159
  character(11) :: msg = "Hello World"

      write (*,*) "Greeting: ", msg

end program example1
```

In this example, a series of variables are defined (as examples) with most not used. The program will display **Greeting: Hello World** when executed. The following chapters will address how to use the variables to perform calculations and display results. Additional information regarding character variables is provided in chapter 11.

## 4.6 Declarations, Extended Size Variables

The size or range of a number that can be held in a Fortran variable is limited. Special declarations can be used to provide variables with extended or larger ranges. Only integer and real variables are addressed here.

### 4.6.1 *Integers*

As previously noted, the range of an integer value can range between  $-2,147,483,648$  and  $+2,147,483,647$ . In the unlikely event that a larger range is required, a special declaration can be used to extend the range. The **kind** specifier is used with the integer declaration.

For example, to declare a variable **bignum** with an extended range, the integer declaration would be as follows:

```
integer*8 :: bignum
```

or

```
integer(kind=8) :: bignum
```

Both of these equivalent declarations use more space for the variables (8 bytes instead of the normal 4) in order to provide a larger range. The extended range of integer variables declared with the **\*8** or **kind=8** is  $-9,223,372,036,854,775,808$  to  $9,223,372,036,854,775,807$ .

### 4.6.2 *Real*

As previously noted, the range is approximately  $\pm 1.7 \times 10^{\pm 38}$  supporting about 7 digits of precision. If more precision is required, the **kind** specifier can be used.

For example, to declare a real variable **rnum** with an extended range, the declaration would be as follows:

```
real*8 :: rnum
```

or

```
real(kind=8) :: rnum
```

Both of these equivalent declarations use more space for the variables (8 bytes instead of the normal 4) in order to provide a larger range. The extended precision of real variables declared with the **\*8** or **kind=8** is approximately  $-2.2 \times 10^{-308}$  to  $+1.8 \times 10^{+308}$  which supports a much larger range with about 15 digits of precision.

## 4.7 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 4.7.1 *Quiz Questions*

Below are some quiz questions.

- 1) What are the five Fortran 95/2003/2008 data types?
- 2) What should a Fortran variable name start with?

3) What data type are each of the following numbers (integer or real)?

475	_____
19.25	_____
19123	_____
5.0	_____
123.456	_____

- 4) Write the statements required to declare **value** as an integer and **count** as a real.
- 5) Write the statements required to declare **rate** as a real variable initialized to 7.5.
- 6) Write the statements required to declare **e** as a real constant initialized to 2.71828183.

### 4.7.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the *example1* example program, compile, and execute the program.
- 2) Update the example program (from 1) to display your name (instead of the 'Hello World' message).



## 5 Expressions

---

This section describes how to form basic Fortran 95/2003/2008 expressions and perform arithmetic operations (i.e., add, subtract, multiple, divide, etc.). Expressions are formed using literals (actual values), variables, and operators (i.e., +, -, \*, /, etc.). The previous chapter provides an explanation of what variables are and a summary of the five Fortran data types.

### 5.1 Literals

The simplest expression is a direct value, referred to as a *literal*. Since literals are actual values, not variables, they cannot be changed. There are various types of literal constants described in the following sections that correspond to the Fortran data types.

#### 5.1.1 Integer Literals

The following are some examples of *integer constants*:

```
1
0
-100
32767
+42
```

An integer must be a whole number (with no fractional component).

#### 5.1.2 Real Literals

The following are some examples of *real constants*:

```
1.0
-0.25
3.14159
```

The real number should include the decimal point (i.e., the “.”). A real number includes the fractional part, even if the fraction is 0. Fortran will accept a number with the “.” and no further digits. For example, 5. is the same as 5.0. All examples in this text will include the “.0” to ensure clarity.

#### 5.1.2.1 E-Notation

For larger real numbers, *e-notation* may be useful. The e-notation means that you should multiply the constant by 10 raised to the power following the "E". This is sometimes referred to as scientific notation.

The following are some real constants using e-notation:

**2.75E5**  
**3.333E-1**

Hence, 2.75E5 is  $2.75 \times 10^5$  or 275,000 and 3.333E-1 is  $3.333 \times 10^{-1}$  or 0.3333 or approximately one third.

### 5.1.3 *Complex Literals*

A *complex constant* is designated by a pair of constants (integer or real), separated by a comma and enclosed in parentheses. Examples are:

**(3.2, -4.1)**  
**(1.0, 9.9E-1)**

The first number denotes the real part and the second the imaginary part. Although a complex number always consists of two elements, it is considered a single value.

### 5.1.4 *Character Literals*

A *character constant* is either a single character or a set of characters, called a string. A character is a single character enclosed in quotes. A string consists of an arbitrary sequence of characters also enclosed in quotes. Some examples include:

**"X"**  
**"Hello World"**  
**"Goodbye cruel world!"**  
**"Have a nice day"**

Character and string constants (enclosed with quotes) are case sensitive. So, character "X" (upper-case) is not the same as "x" (lower-case).

A problem arises if you want to have a quote in the string itself. A double quote will be interpreted as a single within a string. The two quotes must be together (no spaces between). For example, the string:

**"He said "wow" when he heard"**

Would be displayed as

**"He said "wow" when he heard"**

The double-quote is sometimes referred to as an escape character. Strings and characters must be associated with the character data type.

### 5.1.5 *Logical Constants*

The fifth type is the *logical constant*. These can only have one of two values:

```
.true.
.false.
```

The dots enclosing the true and false are required.

## 5.2 Arithmetic Operations

This section summarizes the basic arithmetic operations.

### 5.2.1 *Assignment*

In programming, *assignment* is the term for setting a variable equal to some value. Assignment is performed with an equal (=) sign. The general form is:

```
variable = expression
```

The expression may be a literal, variable, an arithmetic formula, or combination of each. Only one assignment to a single variable can be made per line.

For example, to declare the variable **answer1** as a real value,

```
real :: answer1
```

and to set it equal to 2.71828183, it would be:

```
answer1 = 2.71828183
```

The value for **answer1** can be changed as often as needed. However, it can only hold one value at a time.

### 5.2.2 *Addition*

The Fortran addition operation is specified with a plus sign (+). For example, to declare the variables, **mysum**, **number1**, **number2**, and **number3**,

```
integer :: mysum, number1=4, number2=5, number3=3
```

and calculate the sum,

```
mysum = number1 + number2
```

which will set the variable **mysum** to 9 in this example. The data types of the variables, integer in this example, should be the same. Multiple variables can be added on one line. The line can also include literal values. For example,

```
mysum = number1 + number2 + number3 + 2
```

which will set the variable **mysum** variable to 14. Additionally, it will write over the previous value of 9.

### 5.2.3 Subtraction

The Fortran subtraction operation is specified with a minus sign (-). For example, to declare the variables, *ans*, *value1*, *value2*, and *value3*,

```
real :: ans, value1=4.5, value2=2.5, value3=1.0
```

and calculate the difference,

```
ans = value1 - value2
```

which will set the variable *ans* to 2.0. The data types of the variables, real in this example, should be the same. Multiple variables can be subtracted on one line. The line can also include literal values. For example,

```
ans = value1 - value2 - value3
```

which will set the variable *ans* to 1.0. Additionally, it will over-write the previous value of 2.0.

### 5.2.4 Multiplication

The Fortran multiplication operation is specified with an asterisk (\*). For example, to declare the variables, *ans*, *value1*, *value2*, and *value3*,

```
real :: ans, value1=4.5, value2=2.0, value3=1.5
```

and calculate the product,

```
ans = value1 * value2
```

which will set the variable *ans* to 9.0. The data types of the variables, real in this example, should be the same. Multiple variables can be multiplied on one line. The line can also include literal values. For example,

```
ans = value1 * value2 * 2.0 * value3
```

which will set the variable *ans* to 27.0. Additionally, it will over-write the previous value of 9.0.

### 5.2.5 Division

The Fortran division operation is specified with a slash symbol (/). For example, to declare the variables, *ans*, *value1*, *value2*, and *value3*,

```
real :: ans, value1=10.0, value2=2.5, value3=2.0
```

and calculate the quotient,

```
ans = value1 / value2
```

which will set the variable *ans* to 4.0. The data types of the variables, real in this example, should be the same. Multiple variables can be divided on one line.

For example,

```
ans = value1 / value2 / value3
```

which will set the variable **ans** to 2.0. Additionally, it will over-write the previous value of 4.0.

### 5.2.6 Exponentiation

Exponentiation means “raise to the power of”. For example, 2 to the power of 3, or  $2^3$  is  $(2 * 2 * 2)$  which is 8. The Fortran exponentiation operation is specified with a double asterisks (\*\*).

For example, to declare the variables, **ans** and **value1**,

```
real :: ans, value1=2.0
```

and calculate the exponentiation,

```
ans = value1 ** 3
```

which will set the variable **ans** to 8.0. When using exponentiation, pay close attention to the data types. For example, raising an integer variable to the power 0.5 would produce a truncated integer result.

## 5.3 Order of Operations

Fortran follows the standard mathematical order of operations or precedence of operations. That is, multiplication and division are performed before addition and subtraction. Further, in accordance with mathematical standards, the exponentiation operation is performed before multiplication and division.

The following table provides a partial summary of the basic Fortran 95/2003/2008 precedence levels:

Precedence Level	Operator	Operation
1 <sup>st</sup>	-	unary -
2 <sup>nd</sup>	**	exponentiation
3 <sup>rd</sup>	* /	multiplication and division
4 <sup>th</sup>	+ -	addition and subtraction

For operations of the same precedence level, the expression is evaluated left to right. Parentheses may be used to change the order of evaluation as necessary. For example, declaring the variables **ans1**, **ans2**, **num1**, **num2**, and **num3**.

```
integer :: ans1, ans2, num1=20, num2=50, num3=10
```

and calculating the **ans1** and **ans2**, as follows:

```
ans1 = num1 + num2 * num3  
ans2 = (num1 + num2) * num3
```

will set to **ans1** to 520 and **ans2** to 700 (both integers).

## 5.4 Intrinsic Functions

Intrinsic functions are standard built-in functions that are provided by Fortran. These include a rich set of standard functions, including the typical mathematical standard functions. Intrinsic functions can be used in expressions as needed. Most intrinsic functions accept one or more arguments as input and return a single value.

### 5.4.1 Mathematical Intrinsic Functions

The intrinsic or built-in functions include the standard mathematical functions such as sine, cosine, tangent, and square root.

For example, the cosine of  $\pi$  is -1.0. Declaring and initializing the variables *z* and *pi* as follows,

```
real :: z
real, parameter :: pi = 3.14159
```

and then performing the calculation of the cosine the variable *pi* as follows,

```
z = cos(pi)
```

which will set *z* to -1.0. The variable *pi* is the input argument.

### 5.4.2 Conversion Functions

Other intrinsic functions include functions to change the type of variables or values. The basic conversion functions are as follows:

Function	Explanation
<code>real(&lt;integer argument&gt;)</code>	Convert the <integer argument> to a real value
<code>int(&lt;real argument&gt;)</code>	Convert the <real argument> to an integer, truncates the fractional portion
<code>nint(&lt;real argument&gt;)</code>	Convert the <real argument> to an integer, rounds the fractional portion

For example, given the following variable declarations,

```
integer :: inum1=10, inum2, inum3
real :: rnum1, rnum2 = 4.8, rnum3 = 5.8
```

and calculate the *rnum1*, *inum2*, and *inum3*,

```
rnum1 = real(inum1)
inum2 = int(rnum2)
inum3 = nint(rnum3)
```

which will set to *rnum1* to 10.0, *inum2* to 4, and *inum3* to 6.

### 5.4.3 Summary

A summary of some of the more common intrinsic functions include:

Function	Description
COS(W)	Returns real cosine of real argument W in radians.
INT(A)	Converts real argument A to integer, truncating (real part) towards zero.
MOD(R1,R2)	Returns remainder after division of R1 on division by R2. Result, R1 and R2 should be all integer or all real types.
NINT(X)	Returns the nearest integer to real value X (thus rounding up or down as appropriate).
REAL(A)	Converts integer argument A to real.
SIN(W)	Returns real sine of real argument W in radians.
SQRT(W)	Returns the real square root of real argument W; W must be positive.
TAN(X)	Returns the real tangent of real argument X in radians.

A more complete list of intrinsic functions is located in Appendix D.

## 5.5 Mixed Mode

In general, mathematical operations should be performed on variables of the same type. When both integer and real values or variables are used in the same statement, it is called mixed mode.

Real and integer operations:

```
1/2 = 0
1.0 + 1/4 = 1.0
1.0 + 1.0/4 = 1.25
```

Any integers values are converted to real only when mixed-mode is encountered on the same operation type. Conversion may also occur on assignment.

Unexpected conversions can cause problems when calculating values. In order to avoid such problems, it is strongly recommended to not use mixed-mode. There are a series of rules associated with mixed mode operations. In some circumstances, these rules can be confusing. For simplicity, those rules are not covered in this text.

If it is necessary to perform calculations with different data types, such as integers and reals, the intrinsic or built-in conversion functions should be used to ensure correct and predictable results. This also allows the programming greater control of when types are converted. In very complex calculations, this would help ensure clarity and address precision issues. Further recommendations to address highly precise calculations are not addressed in this text.

## 5.6 Examples

Below is an example program that calculates velocity based on acceleration and time. The program declares the appropriate variables and calculate the velocity.

```

program findvelocity
! Program to calculate the velocity from the
! acceleration and time

! Declare variables
implicit none
real :: velocity, acceleration = 128.0
real :: time = 8.0

! Display initial header
write (*,*) "Velocity Calculation Program"
write (*,*)

! Calculate the velocity
velocity = acceleration * time

write (*,*) "Velocity = ", velocity

end program findvelocity

```

Additional information regarding how to perform input and output in the next chapter. The comments are not required, but help make the program easier to read and understand.

## 5.7 Exercises

Below are some quiz questions and project suggestion based on this chapter.

### 5.7.1 Quiz Questions

Below are some quiz questions.

- 1) What is the *assignment* operator?
- 2) What is the exponentiation operator?
- 3) How can an integer variable be converted to a real value?
- 4) How can a real variable be converted to a integer value?
- 5) What are the two logical constants?
- 6) List three intrinsic functions.



- 7) Write the single Fortran statement for each of the following formulas. You may assume all variables are already declared as real values. Additionally, you may assume the variable PI is set as a parameter and initialized to 3.14159.

$$x1 = \left(\frac{\pi}{6}\right) (3a^2 + 3b^2 + c^2)$$

$$x2 = -\frac{2a}{c} \cos(b) \sin(b)$$

$$x3 = \frac{-b + \sqrt{(b^2 - 4ac)}}{2a}$$

### 5.7.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the velocity program, compile, and execute the program. Change the declared values, compile, and execute the modified program. Verify the results of both executions with a calculator.
- 2) Write a program to calculate and display the difference between time as read from a sundial and a clock (which is due to the irregular movement of the Sun). The difference can be calculated with the Equation of Time<sup>7</sup> which is as follows:

$$b = 2\pi(n - 81) / 365$$

$$e = 9.87 \sin(2b) - 7.53 \cos(b) - 1.5 \sin(b)$$

Where,  $n$  is the day number. For example,  $n = 1$  for January 1,  $n = 2$  for January 2, and so on. The program should read the value for  $n$  (1-365) from the user as an integer. The program should perform the appropriate type conversions, perform the required calculations, and display the original  $n$  value, the calculated  $b$  (for reference), and final  $e$  value which represents the time difference in minutes. Test the program on a series of different values. The formulas provided are an approximation. For more accurate results, refer to the referenced Wikipedia entry for the complete formula.

<sup>7</sup> For more information, refer to: [http://en.wikipedia.org/wiki/Equation\\_of\\_time](http://en.wikipedia.org/wiki/Equation_of_time)



## 6 Simple Input and Output

---

Simple, unstructured, input and output can be performed with the *write* and *read* statements as explained in the following sections. In a later chapter, a more structured approach will be presented in later sections.

### 6.1 Output – Write

As noted from the first program, simple output can be performed by using the *write* statement. For example:

```
write (*,*) "Hello World"
```

Which will send the message, referred to as a *string*, **Hello World** to the screen. The first “\*” means the default output device, which is the screen or monitor. The second “\*” refers to the 'free format'. Thus, the “(\*,\*)” means to send it to the screen in 'free format'.

The free format allows the Fortran compiler to determine the appropriate format for the information being displayed. This is easy, especially when first getting started, but does not allow the program much control over how the output will be formatted or displayed on the screen.

Additionally, the value held by declared variables can be displayed. For example, to declare the variables *num1*, *num2*, and *num3*.

```
integer :: num1=20, num2=50, num3=10
```

the write statement to display *num1* would be,

```
write (*,*) num1
```

The free format allows the Fortran compiler to determine the appropriate output format for the information being displayed.

A write statement with no strings or variables,

```
write (*,*)
```

will display a blank line.

Multiple variables and strings can be displayed with one write statement. For example, using the previous declarations;

```
write (*,*) "Number 1 = ", num1, "Number 2 = ", num2
```

The information inside the quotes is displayed as is, including capitalization and any spelling errors. When the quotes are not used, it is interpreted as a variable. If the variable is not declared, a compiler error will be generated. The value assigned to each variable will be displayed. A value must have been assigned to the variable prior to attempting to display.

### 6.1.1 Output – Print

In addition to the write statement, a print statement can be used. The print statement will send output only to the screen. Thus, it is a more restrictive form of the write statement.

As with the write statement, multiple variables and strings can be displayed with one print statement. For example, using the previous declarations,

```
print *, "Number 1 = ", num1, "Number 2 = ", num2
```

The information inside the quotes is displayed as is, including capitalization and any spelling errors. When the quotes are not used, it is interpreted as a variable. If the variable is not declared, an error will be generated. If the variable is defined, the value assigned to that variable will be displayed.

In general, all examples will use the write statement.

## 6.2 Input – Read

To obtain information from the user, a *read* statement is used. For example, to declare the variables *num1*, *num2*,

```
integer :: ans1, ans2
```

then read a value for *ans1* from the user,

```
read (*,*) ans1
```

Which will read a number from the user entered on the keyboard into the variable *ans1*. The (\*,\*) means to send it to read the information in 'free format'. The free format allows the Fortran compiler to determine the appropriate format for the information being read.

Multiple variables can be read with one read statement. For example, using the previous declarations,

```
read (*,*) ans1, ans2
```

will read two values from the user into the variables *ans1* and *ans2*.

Since the read is using free format, two numbers will be required. The numbers can be entered on the same line with one or more spaces between them or on separate lines. The read will wait until two numbers are entered.

When reading information from the user, it is usually necessary to provide a prompt in order to ensure that the user understands that input is being requested by the program. A suitable write statement with an appropriate string, followed by a read statement will ensure that the user is notified that input is being requested.

For example, to read a date, a program might request month, day, and year as three separate variables. Given the following declarations,

```
integer :: month, day, year
```

the program might prompt for and read the data in the following manner,

```
write (*,*) "Enter date (month, day, and year)"  
read (*,*) month, day, year
```

Since the program is requesting three integers, three integers must be entered before the program continues. The three numbers may be entered on one line with a single space between them, with multiple spaces or tab between them, or even on three different lines as in the following examples:

```
Enter date (month, day, and year)
10 17 2009
```

```
Enter date (month, day, and year)
10
17
2009
```

```
Enter date (month, day, and year)
10      17      2009
```

The type of number requested here is an integer, so integers should be entered. Providing a real number or character (e.g., letter) would generate an error. Later chapters will address how to deal with such errors.

### 6.3 Example

Below is an example program that calculates the area of a circle. The program will declare the appropriate variables, read the radius, calculate the circle area, and display the result.

```
program circle
! Program to calculate the area of a circle

! Declare variables
implicit none
real :: radius, area
real, parameter :: pi = 3.14159

! Display initial header and blank line
write (*,*) "Circle Area Calculation Program"
write (*,*)

! Prompt for and read the radius
write (*,*) "Enter Circle Radius"
read (*,*) radius

! Calculate the circle area
area = pi * radius**2

! Display result
write (*,*) "Circle Area:  ", area

end program circle
```

The comments are not required, but help make the program easier to read and understand. If the program does not work at first, the comments can aid in determining the problem.

## 6.4 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 6.4.1 Quiz Questions

Below are some quiz questions.

- 1) What does the `(*,*)` mean?
- 2) Provide the statement to output a message “Programming is Fun!”
- 3) What are the statements to declare and read the value for a person's age in years?

### 6.4.2 Suggested Projects

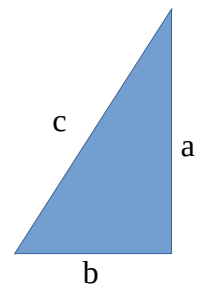
Below are some suggested projects.

- 1) Type in the circle area program, compile and execute the program. Test the program on several sets of input.
- 2) Modify the circle area program to request a circle diameter. The formula for circle area must be adjusted accordingly. Recall that radius = diameter divided by two. Test the program on several sets of input.
- 3) Type in the velocity program from the previous chapter and update to prompt for and request input for the acceleration and time, and then display the results. Test the program on several sets of input.
- 4) Write a Fortran program to read the length of the ***a*** and ***b*** sides of a right triangle and compute the perimeter length. The program should prompt for input and display the values for sides ***a***, ***b***, ***c***, and the perimeter with appropriate headings.

The formulas for the ***c*** side and perimeter are:

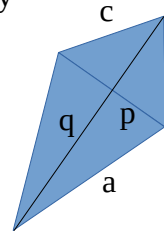
$$c = \sqrt{a^2 + b^2}$$

$$\text{perimeter} = a + b + c$$



Test the program on several sets of input.

- 5) Write a Fortran program to compute geometric information for a kite. The program should read the ***a***, ***c*** and ***p*** lengths and compute the ***q*** length. The program should display an appropriate prompt, read the values, compute the answer, and display the original input and the final result.



The formula for the ***q*** length is as follow:

$$q = \sqrt{a^2 - \frac{p^2}{4}} + \sqrt{c^2 - \frac{p^2}{4}}$$

Test the program on several sets of input.





## 7 Program Development

---

Writing or developing programs is easier when following a clear methodology. The main steps in the methodology are:

- Understand the Problem
- Create the Algorithm
- Implement the Program
- Test/Debug the Program

To help demonstrate this process in detail, these steps will be applied to a simple problem to calculate and display the period of a pendulum.

As additional examples are presented in later chapters, they will be explained and presented using this methodology.

### 7.1 Understand the Problem

Before attempting to create a solution, it is important to understand the problem. Ensuring a complete understanding of the problem can help reduce errors. The first step is to understand what input is required and what information the program is expected to produce. In this example, the formula for the period of a pendulum is:

$$Period = 2 \pi \sqrt{\frac{L}{g}} \left( 1 + \frac{1}{4} \sin^2\left(\frac{\alpha}{2}\right) \right)$$

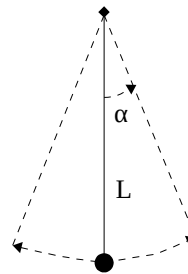
Where:

$g$  = 980 cm/sec<sup>2</sup>

$\pi$  = 3.14159

$L$  = Pendulum length (cm)

$\alpha$  = Angle of displacement (degree)



Both  $g$  (gravity) and  $\pi$  should be declared as constants. The formula is a simplified version of the more general case. As such, for very large, very small, or zero angle values the formula will not provide accurate results. For this example, that is acceptable.

As shown, the pendulum is attached to a fixed point, and set into motion by displacing the pendulum by an angle,  $\alpha$ , as shown in the diagram. The program must define the constants for  $g$  and  $\pi$ , declare the variables, display appropriate prompts, read the values for  $L$  and  $\alpha$ , then calculate and display the original input and the period of the pendulum with the given length and angle of displacement.

## 7.2 Create the Algorithm

The algorithm is the name for the *ordered sequence of steps involved in solving the problem*. That sounds good, but it is a fancy way of saying that an algorithm is just a step-by-step procedure to solve a problem. Once the program is understood, the steps can be developed to solve that specific problem. There can be multiple correct solutions to a given problem.

The process for creating an algorithm can be different for different people. In general, some time should be devoted to thinking about a possible solution. This may involve working on some possible solution on a scratch piece of paper. Once a possible solution is selected, that solution can be developed into an algorithm. The algorithm can be written down, reviewed, and refined. This algorithm is the outline of the program.

For this problem, the variables and constants must be declared, the applicable headers and prompts displayed, and the values for  $L$  and  $\alpha$  read from the user. The degree entered by the user should be converted to radians<sup>8</sup>, which is required by the sin function. The formula to convert degrees to radians is as follows:

$$\text{radians} = \text{degrees} * \frac{\pi}{180}$$

Then the period can be calculated based on the provided formula and the results displayed. Formalizing this, the following steps can be developed and written down as follows:

```
! declare variables
!   real constants -> gravity, pi
!   reals -> angle, length, alpha
! display initial header
! prompt for and read the length and angle values
! convert degrees to radians
! calculate the period
! display the results
```

While this is a fairly straightforward algorithm, more complex problems would require more extensive algorithms. Examples in later chapters will include more complex programs. For convenience, the steps are written as program comments. This will allow the addition of the code to the basic algorithm.

## 7.3 Implement the Program

Based on the algorithm, the following program can be created.

```
program period
! Program to calculate the period of a pendulum

! declare variables
!   real constants -> gravity, pi
!   reals -> angle, length, alpha
implicit none
real :: angle, length, pperiod, alpha
```

<sup>8</sup> For more information, refer to: <https://en.wikipedia.org/wiki/Radian>

```

real, parameter :: gravity=980.0, pi=3.14159

! display initial header
    write (*,*) "Pendulum Period Calculation Program"
    write (*,*)

! prompt for and read the length and angle values
    write (*,*) "Enter Length and Angle values:"
    read (*,*) length, angle

! convert degrees to radians
    alpha = angle * pi / 180.0

! calculate the period
    pperiod = 2.0 * pi * sqrt(length/gravity) *
              ( 1.0 + 1.0/4.0 * sin(alpha/2.0)**2 ) &

! display the results
    write (*,*) "The period is:", pperiod

end program period

```

The indentation is not required, but helps make the program easier to read. Note that the “2”, “1”, and “4” in the algorithm are entered as 2.0, 1.0, and 4.0 to ensure consistent data typing (i.e., all reals). When 1 divided by 4 is entered as “1/4” instead of “1.0/4.0” the result will be 0 because that would be integer division.

## 7.4 Test/Debug the Program

Once the program is written, testing should be performed to ensure that the program works. The testing will be based on the specific parameters of the program. In this example, each of the three possible values for the discriminant should be tested.

```

C:\mydir> period

Pendulum Period Calculation Program

Enter Length and Angle values:
120.0  15.0

The Period is:      2.20801973

C:\mydir>

```

For this program, the results can be verified with a calculator. A series of different values should be used for testing. If the program does not work, the program comments provide a checklist of steps and can be used to help debug the program.

### 7.4.1 Error Terminology

In case the program does not work, it helps to understand some basic terminology about where or what the error might be.

#### 7.4.1.1 Compiler Error

Compiler errors are generated when the program is compiled. This means that the compiler does not understand the instructions. The compiler will provide a list of errors with the line number of each error. It is recommended to address the errors from the top down. Resolving an error at the top can clear multiple errors further down.

Typical compiler errors include misspelling a statement and/or omitting a variable declaration. For example, if the correct Fortran statement “**write** (\*,\*)” is entered incorrectly as “**wrote** (\*,\*)”, an error will be generated.

In this case, the compiler error displayed will appear as follows:

```
c:\mydir> gfortran -o period period.f95
period.f95:13.1:

  wrote (*,*)
1
Error: Unclassifiable statement at (1)
```

The first digit, 13 in this example, represents the line number where the error occurred. Using a text editor that displays line numbers, the statement that caused the error can be quickly found and corrected.

If the declaration for the variable *length* is omitted, the error would appear as follows:

```
c:\mydir> gfortran -o period period.f95
period.f95:17.18:

  read (*,*) length, angle
1
Error: Symbol 'length' at (1) has no IMPLICIT type
```

In this case, the error is shown on line 18 (first digit after the “:”). However, the actual error is that the variable *length* is not declared. Each error should be reviewed and evaluated.

#### 7.4.1.2 Run-time Error

A run-time error is something that causes the program to crash. For example, if a number is requested and a letter is entered, it will cause a run-time error.

For example, the period program expects two real numbers to be entered. If the user enters letters, *x* and *y*, in this example, an error will be generated during the execution of the program as follows:

```

c:\mydir> period
Pendulum Period Calculation Program

Enter Length and Angle values:
x y
At line 17 of file period.f95 (unit = 5, file = 'stdin')
Fortran runtime error: Bad real number in item 1 of list input

```

The program was expecting numeric values and letters were provided. Since letters are not meaningful in this context, it is an error and the program “crashes” or stops with an error message.

Later chapters will provide additional information on how to deal with such errors. Until then, providing the correct data type will avoid this kind of error.

### 7.4.1.3 Logic Error

A logic error is when the program executes, but does not produce the correct result. For example, coding a provided formula incorrectly or attempting to compute the average of a series of numbers before calculating the sum would be considered a logic error.

For example, the correct formula for the period of a pendulum is as follows:

```

ppperiod = 2.0 * pi * sqrt(length/gravity) *      &
              ( 1.0 + 1.0/4.0 * sin(alpha/2.0)**2 )

```

If the formula is typed incorrectly or incompletely as follows:

```

ppperiod = 2.0 * pi * sqrt(length/gravity) *      &
              ( 1.0 + 1/4 * sin(alpha/2.0)**2 )

```

The 1 over 4 is entered as “1/4” which are interpreted as integers. As integers, “1/4” results in 0. The compiler will accept this, perform the calculations, and provide an incorrect result.

The program would compile and execute as follows.

```

c:\mydir> period
Pendulum Period Calculation Program

Enter Length and Angle values:
120.0 15.0
The period is: 2.19865513

```

However, an incorrect answer would be generated as shown. This is why testing the program is required. Logic errors can be the most difficult to find.

One of the best ways to handle logic errors is to avoid them by careful development of the algorithm and writing the code.

If the program has a logic error, one way to find the error is to display intermediate values. Further information will be provided in later chapters regarding advice on finding logic errors.

## 7.5 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 7.5.1 Quiz Questions

Below are some quiz questions.

- 1) What are the four program development steps?
- 2) What are the three types of errors?
- 3) If a program to compute the area of a rectangle uses a formula, ***height*** × ***height*** × ***width***, what type of error would this be?
- 4) Provide an example of that would generate a compiler error.

### 7.5.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the pendulum period calculation program, compile, and execute the program. Test the program using several different input values.
- 2) Create a program to prompt for and read the circle area from the user and calculate the circumference of a circle using the following formula:

$$\text{circumference} = 2 \sqrt{\pi \text{ CircleArea}}$$

Test the program using several different input values.

- 3) Create a program to prompt for and read the radius of a sphere from the user and calculate the surface area of the sphere using the following formula:

$$\text{sphere Surface Area} = 4 \pi r^2$$

Test the program using several different input values.

- 4) Create a program to prompt for and read the radius of a sphere from the user and calculate the sphere volume using the following formula:

$$\text{sphere Volume} = (4 \pi / 3) r^3$$

Test the program using several different input values.

## 8 Selection Statements

---

When writing a program, it may be necessary to take some action based on the outcome of comparing the values of some variables. All programming languages have some facility for decision-making. That is, doing one thing if some condition is true and (optionally) doing something else if it is not.

Fortran IF statements and/or CASE statements are used to allow a program to make decisions.

### 8.1 Conditional Expressions

The first step is to compare two values. Values may be literals, variables, or expressions. These values are compared with a relational operator and are referred to as *operands*. Relational operators are used between variables or operands of matching types. That is real to real, integer to integer, logical to logical, and character/string to character/string.

The basic relational operators are:

Relational Operation	Relational Operator (normal)	Relational Operator (alternate)
Greater than	>	<b>.gt.</b>
Greater than or equal	>=	<b>.ge.</b>
Less than	<	<b>.lt.</b>
Less than or equal	<=	<b>.le.</b>
Equal to	==	<b>.eq.</b>
Not equal to	/=	<b>.ne.</b>

The normal form will be used for examples in this text. However, the alternate form may be used at any time. The alternate forms may be required to support older Fortran programs.

A relational operation is used to form a conditional expression. The result of a conditional expression must always result in either a *true* or *false* result.

The “==” (two equal signs) is used to compare. The “=” (single equal) is used for assignment (setting a variable). The “==” does not change any values, while the “=” does.

For example, given the declaration of,

**integer :: gameLives**

it might be useful to know if the current value of **gameLives** is greater than 0.

In this case, the conditional expression would be,

```
(gameLives > 0)
```

Which will result in a *true* or *false* result based on the value of the variable *gameLives*.

## 8.2 Logical Operators

Logical operators are used between two logical variables or two conditional expressions. They are:

Logical Operator	Explanation
<b>.and.</b>	the result is true if <b>both</b> operands are true
<b>.or.</b>	the result is true if <b>either</b> operand is true
<b>.not.</b>	logical negate (if true, makes false and if false, makes true)

Logical operators are used to combine conditional expressions as needed to form a more complex conditional expression. For example, given the declaration of,

```
integer :: gameLives, extraLives
```

it might be useful to know if the current value of *gameLives* and *extraLives* are both 0 which would indicate the game is over. In this case, the relational operator would be **AND** with the complete conditional expression,

```
( (gameLives == 0) .and. (extraLives == 0) )
```

which will result in a *true* or *false* result. Since the **AND** logical operation is used, the final result will be *true* only if both conditional expressions are *true*.

Another way of check the status to determine if the game should continue might be,

```
( (gameLives > 0) .or. (extraLives > 0) )
```

which still results in a *true* or *false* result. However, since the **OR** logical operation is used, the final result will be *true* if either conditional expressions is *true*.

The relational operators (e.g., <, <=, >, >=, ==, !=) have higher precedence than logical operators (AND, OR, NOT). This means each of the smaller conditional expressions will be completed before the logical operation is applied.

A conditional expression can be a combination of multiple conditional expressions combined with logical operators.

## 8.3 IF Statements

IF statements are used to perform different computations or actions based on the result of a conditional expression (which evaluates to true or false). There are a series of different forms of the basic IF statement. Each of the forms is explained in the following sections.



### 8.3.1 IF THEN Statement

The IF statement, using the conditional expression, is how programs make decisions. The general format for an IF statement is as follows:

```
if ( <conditional expression> ) then
    <fortran statement(s)>
end if
```

Where the <fortran statements> may include one or more valid Fortran statements.

For example, given the declaration of,

```
integer :: gameLives
```

based on the current value of *gameLives*, a reasonable IF statement might be;

```
if ( gameLives == 0 ) then
    write (*,*) "Game Over."
    write (*,*) "Please try again."
end if
```

which will display the message “Game Over.” and “Please try again.” on the next line if the value of *gameLives* is equal to 0.

#### 8.3.1.1 IF THEN Statement, Simple Form

Additionally, another form of the IF statement includes

```
if ( <conditional expression> ) <fortran statement>
```

In this form, only a single statement is executed if the conditional expression evaluates to true. The previous example might be written as;

```
if ( gameLives == 0 ) write (*,*) "Game Over."
```

In this form, no “then” or “end if” are required. However, only one statement can be executed.

### 8.3.2 IF THEN ELSE Statement

The IF THEN ELSE statement expands the basic IF statement to also allow a series of statements to be performed if the conditional expression evaluates to *false*.

The general format for an IF THEN ELSE statement is as follows:

```
if ( <conditional expression> ) then
    <fortran statement(s)>
else
    <fortran statement(s)>
end if
```

Where the <fortran statements> may include one or more valid Fortran statements.

For example, given the declaration of,

```
integer :: gameLives
```

based on the current value of **gameLives** is, a reasonable IF THEN ELSE statement might be:

```
if ( gameLives > 0 ) then
    write (*,*) "Still Alive, Keep Going!"
else
    write (*,*) "Extra Life Granted."
    gameLives = 1
end if
```

Which will display the message “Still Alive, Keep Going!” if the value of **gameLives** is greater than 0 and display the message “Extra Life Granted.” if the value of **gameLives** is less than or equal to 0.

### 8.3.3 IF THEN ELSE IF Statement

The IF THEN ELSE IF statement expands the basic IF statement to also allow a series of IF statements to be performed in a series.

The general format for an IF THEN ELSE IF statement is as follows:

```
if ( <conditional expression> ) then
    <fortran statement(s)>
else if ( <conditional expression> ) then
    <fortran statement(s)>
else
    <fortran statement(s)>
end if
```

Where the <fortran statements> may include one or more valid Fortran statements.

For example, given the declaration of,

```
integer :: gameLives
```

based on the current value of **gameLives**, a reasonable IF THEN ELSE IF statement might be:

```
if ( gameLives > 0 ) then
    write (*,*) "Still Alive, Keep Going!"
else if ( gameLives < 0 ) then
    write (*,*) "Sorry, game over."
else
    write (*,*) "Extra Life Granted."
    gameLives = 1
end if
```

Which will display the message “Still Alive, Keep Going!” if the value of **gameLives** is greater than 0, display the message “Sorry, game over.” if the value of game lives is < 0, and display the message “Extra Life Granted.” if the value of **gameLives** is equal to 0.

## 8.4 Example One

As previously described, writing or developing programs is easier when following a methodology. As the program becomes more complex, using a clear methodology is even more important. The main steps in the methodology are:

- Understand the Problem
- Create the Algorithm
- Implement the Program
- Test/Debug the Program

To help demonstrate this process in detail, these steps will be applied to a familiar problem as an example. The example problem is to calculate the solution of a quadratic equation in the form:

$$ax^2 + bx + c = 0$$

Each of the steps, as applied to this problem, will be reviewed.

### 8.4.1 Understand the Problem

Before creating a solution, it is important to understand the problem. Ensuring a complete understanding of the problem can help reduce errors.

It is known that the solution to the quadratic equation is as follows:

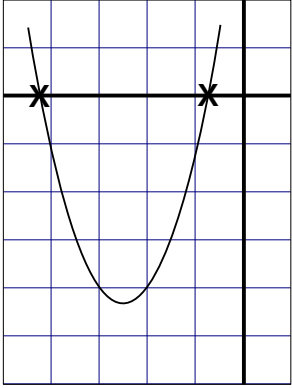
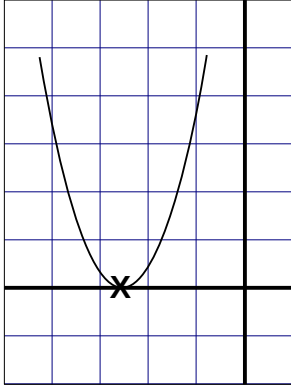
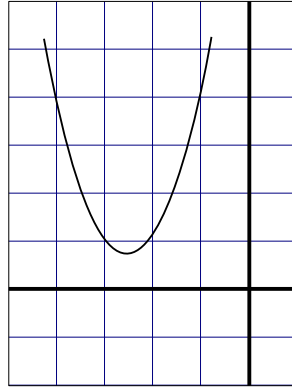
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In the quadratic equation, the term  $(b^2 - 4ac)$  is the *discriminant* of the equation. There are three possible results for the discriminant as described below:

- If  $(b^2 - 4ac) > 0$  then there are two distinct real roots to the quadratic equation. These two solutions represent the two possible answers. If the equation solution is graphed, the curve representing the solution will cross the  $x$ -axis (i.e., representing  $x=0$ ) in two locations.
- If  $(b^2 - 4ac) = 0$  then there is a single, repeated root to the equation. If the equation solution is graphed, the curve representing the solution will cross the  $x$ -axis in one location.
- If  $(b^2 - 4ac) < 0$  then there are two complex roots to the equation. If the equation solution is graphed, the curve representing the solution will not cross the  $x$ -axis and therefore there no real number solution. However, mathematically the square root of a negative value will provide a complex result. A complex number includes a real component and an imaginary component.

A correct solution must address each of these possibilities. For this problem, it is appropriate to use real values.

The relationship between the discriminant and the types of solutions (two different solutions, one repeated solution, or no real solutions) is summarized in the below table:

Positive Discriminant	Zero Discriminant	Negative Discriminant
Two real solutions	One real solution	Two complex solutions
Example: $3x^2 + 9x + 3 = 0$	Example: $2x^2 + 4x + 2 = 0$	Example: $3x^2 + 3x + 3 = 0$
		
Two distinct x-intercepts	One x-intercept	No x-intercept
Root 1 = -0.3819 Root 2 = -2.618	Root 1 = -1.0	Root = -0.5 + 0.866i Root = -0.5 - 0.866i

The examples provided above are included in the example solution in the following sections.

### 8.4.2 Create the Algorithm

The algorithm is the name for the ordered sequence of steps involved in solving the problem. The variables must be defined and an initial header displayed. For this problem, the **a**, **b**, and **c** values will need to be read from the user. Formalizing this, the following steps can be developed.

```
! declare variables
!   reals -> a, b, c, discriminant, root1, root2
! display initial header
! read the a, b, and c values
```

Then, the discriminant can be calculated.

Based on the discriminant value, the appropriate set of calculations can be performed.

```
! calculate the discriminant
! if discriminant is 0,
!   calculate and display root
```

```

! if discriminant is >0,
!   calculate and display root1 and root2
! if discriminant is <0,
!   calculate and display complex root1 and root2

```

For convenience, the steps are written as program comments.

### 8.4.3 *Implement the Program*

Based on the algorithm, the following program can be created.

```

program quadratic
! Quadratic equation solver program

! declare variables
!   reals -> a, b, c, discriminant, root1, root2
implicit none
real :: a, b, c
real :: discriminant, root1, root2

! display initial header
write (*,*) "Quadratic Equation Solver Program"
write (*,*) "Enter A, B, and C values"

! read the a, b, and c values
read (*,*) a, b, c

! calculate the discriminant
discriminant = b ** 2 - 4.0 * a * c

! if discriminant is 0,
!   calculate and display root
if ( discriminant == 0 ) then
    root1 = -b / (2.0 * a)
    write (*,*) "This equation has one root:"
    write (*,*) "root = ", root1
end if

! if discriminant is >0,
!   calculate and display root1 and root2
if ( discriminant > 0 ) then
    root1 = (-b + sqrt(discriminant)) / (2.0 * a)
    root2 = (-b - sqrt(discriminant)) / (2.0 * a)
    write (*,*) "This equation has real roots:"
    write (*,*) "root 1 = ", root1
    write (*,*) "root 2 = ", root2
end if

! if discriminant is <0,
!   calculate and display complex root1 and root2

```

```

if ( discriminant < 0 ) then
    root1 = -b / (2.0 * a)
    root2 = sqrt(abs(discriminant)) / (2.0 * a)
    write (*,*) "This equation has complex roots:"
    write (*,*) "root 1 = ", root1, "+i", root2
    write (*,*) "root 2 = ", root1, "-i", root2
end if

end program quadratic

```

The indentation is not required, but does help make the program easier to read.

#### 8.4.4 Test/Debug the Program

Once the program is written, testing should be performed to ensure that the program works. The testing will be based on the specific parameters of the program. In this example, each of the three possible values for the discriminant should be tested.

```

C:\mydir> quad
Quadratic Equation Solver Program
Enter A, B, and C values
2 4 2
This equation has one root:
root = -1.0000000

C:\mydir> quad
Quadratic Equation Solver Program

Enter A, B, and C values
3 9 3
This equation has has real roots:
root 1 = -0.38196602
root 2 = -2.6180339

C:\mydir> quad
Quadratic Equation Solver Program
Enter A, B, and C values
3 3 3
This equation has complex roots:
root 1 = -0.50000000 +i 0.86602539
root 2 = -0.50000000 -i 0.86602539
C:\mydir>

```

Additionally, these results can be verified with a calculator.

### 8.5 SELECT CASE Statement

A SELECT CASE statement, often referred to as a CASE statement, is used to compare a given value with preselected constants and take an action according to the first constant to match. A CASE

statement can be handy to select between a series of different possibilities or cases.

The select case variable or expression must be of type integer, character, or logical. A real type is not allowed. Based on the selector, a set of one or more of Fortran statements can be executed.

The general format of the SELECT CASE statement is:

```

select case (variable)
  case (selector-1)
    <fortran statement(s)-1>
  case (selector-2)
    <fortran statement(s)-2>
    .
    .
    .
  case (selector-n)
    <fortran statement(s)-n>
  case default
    <fortran statement(s)-default>
end select

```

where <fortran statement(s)-1>, <fortran statement(s)-2>, <fortran statement(s)-3>, ..., <fortran statement(s)-n> and <fortran statement(s)-default> are sequences of one or more executable statements. The selector-1, selector-2, selector-3, ..., and selector-n are called selector lists. Each CASE selector list may contain a list and/or range of integers, character or logical constants, whose values may not overlap within or between selectors. A selector-list is either a single or list of values, separated by commas. Each selector list must be one of the following forms.

```

( value )
( value-1 : value-2 )
( value-1 : )
( : value-2 )

```

where value, value-1, and value-2 are constants or literals. The type of these constants must be identical to that of the selector.

- The first form has only one value
- The second form means all values in the range of value-1 and value-2 (inclusive). In this form, value-1 must be less than value-2
- The third form means all values that are greater than or equal to value-1
- The fourth form means all values that are less than or equal to value-2

In order, each selector expression is evaluated. If the variable value is the selector or in the selector range, then the sequence of statements in <fortran statement(s)> are executed.

If the result is not in any one of the selectors, there are two possibilities:

- if CASE DEFAULT is there, then the sequence of statements in statements-DEFAULT are executed, followed by the statement following END SELECT
- if the CASE DEFAULT is not there, the statement following END SELECT is executed

The constants listed in selectors must be unique. The CASE DEFAULT is optional. But with a CASE DEFAULT, you are guaranteed that whatever the selector value, one of the labels will be used. The place for CASE DEFAULT can be anywhere within a SELECT CASE statement; however, putting it at the end would be more natural.

For example, given the declarations,

```
integer :: hours24, hours12, year
logical :: isAM
```

the following case statement,

```
select case (hours24)
  case (0)
    hours12 = 12
    isAM = .true.
  case (1:11)
    hours12 = hours24
    isAM = .true.
  case (12)
    hours12 = hours24
    isAM = .false.
  case (13:23)
    hours12 = hours24 - 12
    isAM = .false.
end select
```

might be useful to convert 24-hour time into 12-hour time. In this example, a logical variable *isAM* is used to indicate AM (true) or PM (false).

Additionally, the selectors can be combined and separated by commas. For example, given the declarations,

```
integer :: monthnumber, daysinmonth
```

the following case statement,

```
select case (monthnumber)
  case (1,3,5,7,8,10,12)
    daysinmonth = 31
  case (2)
    if (mod(year,4)==0) then
      daysinmonth = 29
    else
      daysinmonth = 28
    end if
  case (4,6,9,11)
    daysinmonth = 30

  case default
    write (*,*) "Error, month number not valid."
end select
```



might be useful to determine the number of days in a given month. The leap-year calculation is not complete, but is adequate if the range of the year is sufficiently limited.

## 8.6 Example Two

A typical problem is to assign grades based on a typical grading standard.

### 8.6.1 *Understand the Problem*

For this example, the program will assign grades using the following grade scale:

A	B	C	D	F
A ≥ 90	80 - 89	70 - 79	60 - 69	≤ 59

The program will read three test scores, compute the average, and display the appropriate grade based on the average.

### 8.6.2 *Create the Algorithm*

The algorithm is the name for the ordered sequence of steps involved in solving the problem.

For this problem, the variables will be declared and an initial header displayed. Then, the *test1*, *test2*, and *test3* values will need to be read from the user.

```
! declare variables
!   reals -> test1, test2, test3
!   integer -> testave
! display initial header
! read the test1, test2, and test3 values
```

Next, the average can be calculated. The average will be converted to the nearest integer and, based on that, the appropriate grade can be determined and displayed. Formalizing this, the following steps can be developed.

```
! calculate the testave and convert to integer
! determine grade
!   A - >= 90
!   B - 80 to 89
!   C - 70 to 79
!   D - 60 to 69
!   F - <= 59
```

For convenience, the steps are written as program comments.

### 8.6.3 *Implement the Program*

Based on the algorithm, the following program can be created.

```

program grades

! declare variables
implicit none
real :: test1, test2, test3
integer :: testave

! display initial header
write (*,*) "Grade Assignment Program"
write (*,*)
write (*,*) "Enter test 1, test 2 and test 3 values"

! read the test1, test2, and test3 values
read (*,*) test1, test2, test3

! calculate the average and convert to integer
testave = nint ((test1 + test2 + test3)/3.0)

! determine grade
!   A → >= 90, B → 80-89, C → 70-79, D → 60-69, F → <= 59

    select case (testave)
    case(90:)
        write (*,*) "Grade is: A"
    case(80:89)
        write (*,*) "Grade is: B"
    case(70:79)
        write (*,*) "Grade is: C"
    case(60:69)
        write (*,*) "Grade is: D"
    case(:59)
        write (*,*) "Grade is: F"
    end select

end program grades

```

The indentation is not required, but does help make the program easier to read.

#### 8.6.4 *Test/Debug the Program*

Once the program is written, testing should be performed to ensure that the program works. The testing will be based on the specific parameters of the program.

In this example, each of the three possible values for the discriminant should be tested.

```

C:\mydir> grade
Grade Assignment Program

Enter test 1, test 2 and test 3 values
70 80 90

```

**Grade is: B**

**C:\mydir>**

The program should be tested with a series of data items to ensure appropriate grade assignment for each grade. Test values for each grade should be entered for the testing.

## 8.7 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 8.7.1 Quiz Questions

Below are some quiz questions.

- 1) List the six relational operators?
- 2) List the three basic logical operators?
- 3) For each of the following, answer **.true.** or **.false.** in the space provided.

```
logical :: b1 = .true., b2=.false., b3=.true.
integer :: i=5, j=10
```

( b1 .or. b2 )	_____
( b1 .or. b3 )	_____
( b1 .and. b2 )	_____
( (b1 .or. b2) .and. b3 )	_____
( b1 .or. (b2 .and. b3) )	_____
( .not. ( i < j ) )	_____
( j < i )	_____

- 4) Write the Fortran IF THEN statements to display the message "Game Over" if the integer variable **lives** is  $\leq$  to 0. You may assume the variable **lives** is already declared as an integer and initialized.
- 5) Write the Fortran IF THEN statements to check the integer variable **num** and if the value is  $< 0$ , take the absolute value of the number and display the message, "Variable num was made positive". You may assume the variable **num** is already declared as an integer and initialized.

- 6) Write the Fortran statements to compute the formula  $z = \frac{x}{y}$  assuming the values for integer variables  $x$ ,  $y$ , and  $z$  are previously set. However, if  $y = 0$ , do not compute the formula, set  $z = 0$ , and display an error message, "Z not calculated". You may assume the variables  $x$ ,  $y$ , and  $z$  are already declared as integers and initialized.
- 7) Write the statements required to compute the following formula using real variables  $f$ ,  $x$ , and  $y$ . Use a single IF THEN ELSE IF statement. You may assume the values for  $f$ ,  $x$ , and  $y$  have already been declared as real values and initialized.

$$f(x) = \begin{cases} x^2 * y & \text{if } x \leq 0.0 \\ x * y & \text{if } x > 0.0 \end{cases}$$

### 8.7.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the quadratic equation program, compile, and execute the program. Provide input values that will check each of the possible outputs.
- 2) Write a Fortran program to prompt for and read the year that a person was born. The year must be between 1900 and 2015 (inclusive). If an invalid entry is read, the program should display the message, "Sorry, that is not a valid year." and re-prompt. If the correct value is not provided after 3 attempts, the program should display the message "Sorry, you're having problems. Program terminated." and terminate. Once a valid year is read, the program should display the year and a message "is a leap year" or "is not a leap year". Include appropriate declarations, prompts, read statements, calculations, and write statements. Test the program on a series of input values and verify that the output is correct.
- 3) Type in the grades program, compile, and execute the program. Test the program on a series of input values that will check each grade.
- 4) Modify the grades program to handle the following grade assignment;

A	A-	B+	B	B-	C+	C	C-	D	F
≥94	93-90	89-87	86-84	83-80	79-77	76-74	73-70	69-60	≤59

Compile, and execute the program. Test the program on a series of input values that will check each grade.

- 5) Write a Fortran program to prompt and read the Fahrenheit as an integer, convert to Celsius, and display the result as a real. The formula to convert a Fahrenheit temperature to a Celsius temperature is as follows:

$$celsius = \left(\frac{5}{9}\right) (fahrenheit - 32)$$

The Fahrenheit value must be between -50 and 150 (inclusive). If the Fahrenheit value is out of range, the program should display an error message, "Temperature out of range", and terminate. The calculations must be performed as real. Include program statements, appropriate declarations, prompts, read statements, calculations, and write statements. Test the program on a series of input values.

- 6) Write a Fortran program that reads an item cost (real numbers) and amount tendered (real number) and compute the correct change. The correct change should be returned as the number of twenties, tens, fives, ones, quarters, dimes, nickels, and pennies. The main program should ensure that the amount paid exceeds the item cost and, if not, display an appropriate error message. Test the program multiple times using a series of input values.
- 7) Write a Fortran to program that reads a number from the user that represents a television channel and then uses a CASE construct to determine the call letters for that station.

Channel	Call Letters	Affiliation
3	KVBC	NBC
5	KVVU	FOX
8	KLAS	CBS
10	KLVX	Public
13	KTNV	ABC

The program should display an appropriate message if an invalid or unassigned channel is entered. Test the program on a series of input values that will show each station.



## 9 Looping

---

When a series of Fortran statements need to be repeated, it is referred to as a *loop* or *do-loop*. A Fortran *do-loop* is a special control statement that allows a Fortran statement or set of statements to be executed multiple times. This repetition can be based on a set number of times, referred to as counter controlled, or based on a logical condition, referred to as conditionally controlled. Each of these looping methods is explained in the following sections.

### 9.1 Counter Controlled Looping

A counter controlled loop repeats a series of one or more Fortran statements a set number of times. The general format of the counting loop is:

```
do count_variable = start, stop, step
    <fortran statement(s)>
end do
```

where the count variable must be an integer variable, start, stop, and step are integer variables or integer expressions. The step value is optional. If it is omitted, the default value is 1. If used, the step value cannot be zero. The <fortran statement(s)> is a sequence of statements and is referred to as the *body* of the do-loop. You can use any executable statement within a do-loop, including IF-THEN-ELSE-END IF and even another do-loop. Before the do-loop starts, the values of start, stop, and step are computed exactly once. More precisely, during the course of executing the do-loop, these values will not be re-computed.

The count variable receives the value of start variable or expression. If the value of control-var is less than or equal to the value of stop-value, the <fortran statement(s)> part is executed. Then, the value of step (1 if omitted) is added to the value of control-var. At the end, the loop goes back to the top and compares the values of control-var and stop-value.

If the value of control-var is greater than the value of final-value, the *do-loop* completes and the statement following *end do* is executed.

For example, with the declarations,

```
integer :: counter, init=1, final=10, sum=0
```

the following do-loop,

```
do counter = init, final
    sum = sum + counter
end do
write (*,*) "Sum is: ", sum
```

will add the numbers between 1 and 10 which will result in 55. Since the step was not specified, it is defaulted 1.

Another example, with the declarations,

```
integer :: counter, init=1, final=10, step=2
```

and the following do-loop,

```
do counter = init, final, step
  write (*,*) counter
end do
```

will display the odd numbers between 1 and 10 (1, 3, 5, 7, 9).

Another example would be to read some numbers from the user and compute the sum and average of those numbers. The example asks the user how many numbers, reads that many numbers, computes the sum, computes the average, and displays the results.

```
program calcAverage
implicit none
integer :: count, number, sum, input
real    :: average

  write (*,*) "Enter count of numbers to read"
  read (*,*) count

  sum = 0
  do number = 1, count
    read (*,*) input
    sum = sum + input
  end do

  average = real(sum) / real(count)
  write (*,*) "Average = ", average

end program calcAverage
```

The use of the function *real()* converts the sum and count variables from integers to real values as required for the average calculation. Without this conversion, sum/count division would be interpreted as dividing an integer by an integer, yielding an integer result.

A final example of a counter controlled loop is to compute the factorial of a positive integer. The factorial of an integer *n*, written as *n!*, is defined to be the product of 1, 2, 3, ..., *n-1*, and *n*. More precisely,  $n! = 1 * 2 * 3 * \dots * n$ .

```
integer :: factorial, n, i

factorial = 1
do i = 1, n
  factorial = factorial * i
end do
```



The do-loop above iterates  $n$  times. The first iteration multiplies factorial with 1, the second iteration multiplies factorial with 2, the third time with 3, ..., the  $i^{\text{th}}$  time with  $i$  and so on. Thus, the values that are multiplied with the initial value of factorial are 1, 2, 3, ...,  $n$ . At the end of the do-loop, the value of factorial is  $1 * 2 * 3 * \dots * n$  which is  $n!$ .

## 9.2 EXIT and CYCLE Statements

The exit and cycle statements are used to modify the execution of a do-loop. The *exit* statement is used to exit a loop. The exit can be used alone, but it is typically used with a conditional statement to allow exiting a loop based on a specific condition. The *exit* statement can be used in a counter controlled loop or a conditionally controlled loop.

For example, given the following declarations,

```
integer :: i
```

the following loop,

```
do i = 1, 10
    if (i == 5) exit
    write (*,*) i
end do
```

will display the numbers from 1 to 4 skipping the remaining iterations. Since the variable  $i$  is checked before the write statement, the value is not displayed with  $i$  is 5 and the loop is exited without completing the remaining iterations. While it is possible to have multiple *exit* statements, typically only one is used. However, multiple exit statements may be required for more complex problems.

The *cycle* statement will skip the remaining portion of the do-loop and start back at the top. The *cycle* statement can be used in a counter controlled loop or a conditionally controlled loop. If the cycle statement is used within a counter controlled loop, the next index counter is updated to the next iteration, which could terminate the loop.

For example, given the following declarations,

```
integer :: i
```

the following loop,

```
do i = 1, 10
    if (i == 5) cycle
    write (*,*) i
end do
```

will display the numbers from 1 to 4 and 6 to 10.

## 9.3 Counter Controlled Example

In this example, we will write a Fortran program to find the difference between the sum of the squares and the square of the sum of the first  $N$  natural numbers.

### 9.3.1 Understand the Problem

In order to find the difference between the sum of the squares and the square of the sum of the first  $N$  natural numbers, we will need to find both the sum of the squares and the square of the sum. For example, the sum of the squares of the first ten natural numbers is,

$$1^2 + 2^2 + \cdots + 10^2 = 385$$

The square of the sum of the first ten natural numbers is,

$$(1 + 2 + \cdots + 10)^2 = 55^2 = 3025$$

Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is  $3025 - 385 = 2640$ .

The program will display the  $N$  value, the sum of the squares, the square of the sum, and the difference for each number from 2 to a given  $N$  value. The program should prompt for and read the  $N$  value. The program will display appropriate headers.

### 9.3.2 Create the Algorithm

For this problem, first we will need to read the  $N$  value. Then, we will loop from 1 to the  $N$  value and find both the sum of the squares and the square of the sum.

```
! declare variables
!   integer -> i, n, SumOfSqrs, SqrOfSums
! display initial header
! prompt for and read the n value
! loop from 1 to n
!   compute sum of squares
!   compute sums
! square the sums
! compute difference between sum of squares and square of sums
! display results
```

For convenience, the steps are written as program comments.

### 9.3.3 Implement the Program

Based on the algorithm, the program below could be created.

```
program SOSdifference

! declare variables
implicit none
integer :: i, n, SumOfSqrs=0, SqrOfSums=0, difference

! display initial header
```

```

write (*,*) "Example Program"
write (*,*) "  Difference between sum of squares "
write (*,*) "  and square of sums"
write (*,*)

! prompt for and read the n value
write (*,*) "Enter N value: "
read (*,*) n

! loop from 1 to n

  do i = 1, n
    ! compute sum of squares
    SumOfSqrs = SumOfSqrs + i**2

    ! compute square of sums
    SqrOfSums = SqrOfSums + i
  end do

! square the sums
SqrOfSums = SqrOfSums**2

! compute difference between sum of squares and square of sums
difference = SqrOfSums - SumOfSqrs

! display results
write (*,*) "Difference: ", difference

end program SOSdifference

```

The spacing and indentation are not required, but help to make the program more readable.

### 9.3.4 *Test/Debug the Program*

For this problem, the testing would be to ensure that the results match the expected values. Some expected results can be determined with a calculator or a spreadsheet. If the program does not provide the correct result, one or both of the intermediate results, *SumOfSqrs* or *SqrOfSums*, may be incorrect. These values can be displayed with a temporary write statement to determine which might not be correct. If the problem is still not found, the intermediate values calculated during the loop can also be displayed with a write statement. The output can be reviewed to determine what the program is doing (and what may be wrong).

## 9.4 Conditional Controlled Looping

A conditional controlled loop repeats a series of one or more Fortran statements based on a condition. As such, the loop may execute an indeterminate number of times.

One form of the conditional loop is:

```
do while (conditional expression)
  <fortran statement(s)>
end do
```

In this form, the conditional expression is re-checked at the top of the loop on each iteration.

A more general format of the conditional loop is:

```
do
  <fortran statement(s)>
end do
```

As is, this loop will continue forever. Probably not so good. A selection statement, such as an IF statement, and an *exit* statement would be used to provide a means to terminate the looping.

For example,

```
do
  <fortran statement(s)>
  if (conditional expression) exit
  <fortran statement(s)>
end do
```

Would stop looping only when the conditional expression evaluates to true. The exit statement can be used multiple times in different locations as needed. An IF statement, in any form, can be used for either the exit or cycle statements.

For example, a conditional loop could be used to request input from the user and keep re-prompting until the input is correct.

```
integer :: month

do
  write (*,*) "Enter month (1-12): "
  read (*,*) month
  if (month >= 1 .and. month <= 12) exit
  write (*,*) "Error, month must be between 1 and 12."
  write (*,*) "Please re-enter."
end do
```

This will keep re-prompting an unlimited number of times until the correct input (a number between 1 and 12) is entered.

Since a counter controlled DO loop requires an integer loop counter, another use of conditional loops would be to simulate a real counter. For example, to display the values from 1.5 to 4.5 stepping by 0.25, the following conditional loop could be used.

```
real :: currValue = 1.5

do while (currValue <= 4.5)
    write (*,*) "Value = ", currValue
    currValue = currValue + 0.25
end do
```

The values of *currValue* and amount of the increment, set as 0.25, can be adjusted as needed.

## 9.5 Conditionally Controlled Loop Example

In this example, we will write a Fortran program that will read a valid date from the user. The date will consist of three values, one for each of the month, day, and year. This example will use some of the previous example fragments.

### 9.5.1 *Understand the Problem*

For this limited example, we will request a date where the year is between 1970 and 2020. The month must be between 1 and 12. The date will depend on the month since some months have 30 or 31 days. February has either 28 days or 29 days if it is a leap year. Due to the limited allowable range of the year, the determination of a leap year can be performed by checking if the year is evenly divisible by 4 (which implies a leap year).

### 9.5.2 *Create the Algorithm*

For this problem, we will need to read the three values (month, day, and year). Then, we will check the values to ensure that date is valid. Then, we will check the month first and then the year since it will be used to check the date.

The months January, March, May, July, August, October, and December have 31 days. The months April, June, September, and November have 30 days. February has 28 days unless the year is evenly divisible by 4, in which case February has 29 days.

```
! declare variables; integer -> month, day, year
! display initial header

! loop
!     request month, day, and year
!     read month, day, and year
!     check month (1-12)
!     check year (1970-2020)
!     check day
!         1,3,5,7,8,10,12 -> 31 days
!         4,6,9,11 -> 30 days
!         2 -> if modulo of year/4 is 0 -> 29 days
```

```

!           2 → if modulo of year/4 is not 0 → 28 days
!           if invalid, display error and loop to try again
!       end loop
! display results

```

For convenience, the steps are written as program comments.

### 9.5.3 *Implement the Program*

Based on the algorithm, the program below could be created.

```

program dateCheck

! declare variables
implicit none
integer :: month, day, year, dayMax

! display initial header
write (*,*) "Date Verification Example"

! loop
do
    ! request month, day, and year
    write (*,*) "Enter month, day, and year"

    ! read month, day, and year
    read (*,*) month, day, year

    ! check month (1-12)
    if ( month < 1 .or. month > 12 ) then
        write (*,*) "Error, invalid month"
        cycle
    end if

    ! check year (1970-2020)
    if ( year < 1970 .or. year > 2020 ) then
        write (*,*) "Error, invalid year"
        cycle
    end if

    ! check day
    !     1,3,5,7,8,10,12 → 31 days
    !     4,6,9,11 → 30 days
    !     2 → if modulo of year/4 is 0 → 29 days
    !     2 → if modulo of year/4 is not 0 → 28 days

```

```

        select case (month)
            case (1,3,5,7,8,10,12)
                dayMax = 31
            case (2)
                if (mod(year,4)==0) then
                    dayMax = 29
                else
                    dayMax = 28
                end if
            case (4,6,9,11)
                dayMax = 30
        end select

        ! if invalid, display error and loop to try again
        if ( day < 1 .or. day > dayMax ) then
            write (*,*) "Error, invalid day."
            cycle
        end if

        exit

!     end loop
    end do

! display results
    write (*,*) "Valid Date is:", month, day, year

end program dateCheck

```

The spacing and indentation are not required, but help to make the program more readable.

### 9.5.4 *Test/Debug the Program*

For this problem, the testing would be to ensure that the results match the expected value. This will require entering a series of different dates and verifying that the displayed output is correct for the given input data.

## 9.6 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 9.6.1 *Quiz Questions*

Below are some quiz questions.

- 1) What will happen when an *exit* statement is executed?
- 2) How many *exit* statements can be included in a loop?

- 3) What will happen when a *cycle* statement is executed?
- 4) How many *cycle* statements can be included in a loop?
- 5) If there are multiple *cycle* statements in a loop, which one will be executed?
- 6) What is the output of the following Fortran statements. Assume **sum** and **i** are declared as integers.

```
sum = 0
do i = 1, 5
    sum = sum + i
end do
write (*,*) "The SUM is:", sum
```

- 7) What is the output of the following Fortran statements. Assume **i** and **j** are declared as integer.

```
write (*,*) "start"
do i = 1, 3
    do j = 1, 2
        write (*,*) i, " * ", j, " = ", (i*j)
    end do
end do
write (*,*) "end"
```

- 8) Are the following Fortran statements **valid** or **invalid**? If valid, what will happen?

```
do i = 3, 2
    write (*,*) i
end do
```

- 9) Are the following Fortran statements **valid** or **invalid**? If valid, what will happen?

```
do i = 3, 2
    if ( i == 3 ) then
        write (*,*) i
    end do
end if
```

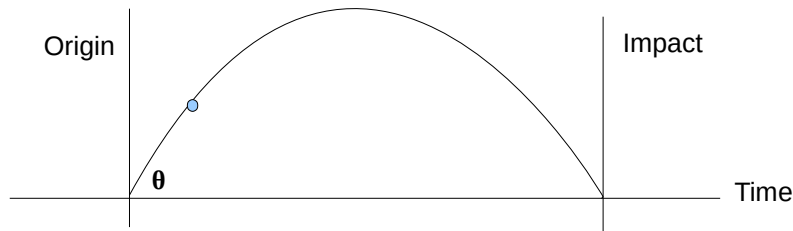
- 10) What is the limit of statements that can be included in a loop?
- 11) When IF statements (any form) are nested inside a loop, what must be done to ensure the statements are valid?



### 9.6.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the difference program, compile, and execute the program. Test the program on a series of different input values.
- 2) Type in the date check program, compile, and execute the program. Test the program on a series of different input values.
- 3) Write a program to calculate the range that a ball would travel when it is thrown with an initial velocity  $v_0$  and angle  $\theta$ . Based on an initial velocity provided by the user, calculate the range every 5 degrees for angles between 5 and 85 degrees. If we assume negligible air friction and ignore the curvature of the earth, a ball that is thrown into the air from any point on the earth's surface will follow a parabolic flight path.



The range (distance between the initial origin and final impact) is determined by the formula:

$$range = -\frac{2v_0^2}{g} \cos \theta \sin \theta$$

where  $v_0$  is the initial velocity of the ball,  $\theta$  is the angle of the throw, and  $g$  is the acceleration due to the earth's gravity. The value for gravity should be defined as a constant and set to -9.81 meters per second.

*Note*, the intrinsic trigonometric functions work in radians, so the angle in degrees will need to be converted to radians for the calculations. To convert degrees to radians:

$$radians = degrees \left( \frac{\pi}{180} \right)$$

Test the program on a series of different input values.



## 10 Formatted Input/Output

---

Fortran uses a **FORMAT** statement to allow control of how data is displayed or read. This is useful when very specific input or output is required. For example, displaying money figures typically require exactly two decimal places. There are format specifiers for each data type; integer, real, character, logical, and complex.

### 10.1 Format

The format specifiers, separated by commas, are contained in a pair of parenthesis as a string literal. There are multiple possible ways to define a format. However, we will focus on the easiest, most direct method. The format specifier will replace the second “\*” in the read or write statements. For example:

```
read (*,'(<format specifiers>')') <variables>
write (*,'(<format specifiers>')') <variables/expressions>
```

The following sections explain the options for the format specifiers.

### 10.2 Format Specifiers

The format specifiers tell the system exactly how the input or output should be handled. Each value being read or written requires some amount of space. For example, an integer of four digits requires at least four spaces or positions to print. Therefore, the number of positions to be used is a key part of the specifier.

The following convention of symbols:

- w** → the number of positions to be used
- m** → the minimum number of positions to be used
- d** → the number of digits to the right of the decimal point
- n** → the number or count
- c** → column number
- r** → repeat count

The following is a summary of the most commonly used format specifiers:

Description	Specifier
Integers	<i>rIw</i> or <i>rIw.m</i>
Real	<i>rFw.d</i>
Logicals	<i>rLw</i>
Characters	<i>rA</i> or <i>rAw</i>

Horizontal Positioning (space)	<b><i>nX</i></b>
Horizontal Positioning (column)	<b><i>Tc</i></b>
Vertical Spacing	<b><i>n/</i></b>

In addition, each specifier or group of specifiers can be repeated by preceding it with a repeat count. Format specifiers for complex numbers will be addressed in later chapters.

### 10.3 Integer Format Specifier

The integer format specifier ***rIw*** or ***rIw.m*** is used to tell the system exactly how many positions should be used to either read or write an integer variable. The ***w*** is the width or how many total places are used. If the number is negative, the sign uses a place. The ***m*** is optional and can be used to set a minimum number of digits to display, which will display leading zeros if needed in order to display the minimum number of digits. The ***r*** is the number of times the format specifier should be repeated.

A format of '***(i6)***' would look like:

<b><i>x</i></b>	<b><i>x</i></b>	<b><i>x</i></b>	<b><i>x</i></b>	<b><i>x</i></b>	<b><i>x</i></b>
← <b><i>w</i></b> →					

For example, given the declarations,

```
integer :: num1=42, num2=123, num3=4567
```

the following write statement can be used to display the value in variable ***num1*** with no leading or trailing spaces.

```
write (*,'(i2)') num1
```

Which will display “42”.

Multiple variables can be displayed. For example, to display the values in variables ***num1*** and ***num2***, with no leading or trailing spaces.

```
write (*,'(i2,i3)') num1, num2
```

Which will display “42123” with no spaces between the two different values. However,

```
write (*,'(i2,i4)') num1, num2
```

will display “42 123” with one space between the values. Further,

```
write (*,'(i5,i5,i5)') num1, num2, num3
```

will display “ 42 123 4567” where each variable uses 5 spaces. And, finally,

```
write (*,'(i6.4)') num1
```

will display “ 0042”.

## 10.4 Real Format Specifier

The real format specifier **rFw.d** is used to tell the system exactly how many positions should be used to either read or write a real variable. The **w** is the width or how many total places are used, including the decimal point. If the number is negative, the sign uses a place. The **d** is how digits are displayed after the decimal point, which does not count the decimal point. The **r** is the number of times the format specifier should be repeated.

A format of '**f6.2**' would look like:

$x$	$x$	$x$	$.$	$x$	$x$
				$\leftarrow d \rightarrow$	
$\leftarrow w \rightarrow$					

For example, given the declarations,

```
real :: var1=4.5, var2=12.0, var3=2145.5713
```

the following write statement can be used to display the value in variable **var1** with no leading or trailing spaces.

```
write (*,'(f3.1)') var1
```

Which will display “4.5” with no leading spaces. Multiple variables can be displayed. For example, to display the values in variables **var1** and **var2**.

```
write (*,'(f5.2,f8.3)') var1, var2
```

Which will display “ 4.50 12.000”. Another example with three variables, **var1**, **var2**, and **var3**, is as follows:

```
write (*,'(f10.4,f10.4,f10.4)') var1, var2, var3
```

Which will display “ 4.5000 12.0000 2145.5713” where each variable uses 10 spaces with each having exactly 4 digits after the decimal point.

Although we may print a number using as many positions as you want, this is only for input/output formatting. The number of positions or size is **not** the precision (*i.e.*, the number of significant digits) of that number. The default precision of real numbers is about seven significant digits. This is the *precision* of real numbers. However, we can print a real number using 50 positions in which 25 positions are for the fractional part. This is only a way of describing the appearance and does not change the precision of real numbers.

## 10.5 Horizontal Positioning Specifiers

The horizontal positioning specifiers, **nX** and **Tc**, are used to provide specific horizontal spacing or placement as needed. The **nX** specifier is used to insert **n** spaces. The **Tc** specifier is used to move directly to the specified column **c**.

For example,

```
write (*,'(a,2x,a,t20,a)') "X", "Y", "Z"
```

will display the following.

```
      X      Y              Z
```

The initial **X** and **Y** are separated by two spaces. The final **Z** is placed in the 20 column.

## 10.6 Logical Format Specifier

The logical format specifier **rLw** is used to tell the system exactly how many positions should be used to either read or write a logical variable. The **w** is the width or how many total places are used. The **r** is the number of times the format specifier should be repeated. Since a logical variable can only be set to the logical constants **.true.** or **.false.**, the width will specify how many of the characters of the logical constants will be read or displayed.

For example, given the declarations,

```
logical :: dooropen=.true., windowopen=.false.
```

the following write statement can be used to display the value in the logical variables **dooropen** and **windowopen** with no leading or trailing spaces. It should be noted that only a **T** or **F** will be displayed. The following write statement,

```
write (*,'(l1,lx,l1)') dooropen, windowopen
```

will display **"T F"**. *Note*, the **l1** format is lower-case L and number 1.

The size or width can be adjusted as needed. For example, the following write statement,

```
write (*,'(l3,2x,l3)') dooropen, windowopen
```

will display **" T F"**, which will display a total of 8 characters; two spaces and a **T** from the right justified L3 (\_\_\_T), followed by 2 spaces from the 2X (\_\_\_), and then two spaces followed by an **F** from the L3 (\_\_\_F).

## 10.7 Character Format Specifier

The character format specifier **rAw** is used to tell the system exactly how many positions should be used to either read or write a character variable. The **w** is the width or how many total places are used. If the width is not specified, the existing length of the string is used. The **r** is the number of times the format specifier should be repeated.

A format of **'(a6)'** would look like:

<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>
←	<i>w</i>				→

For example, given the declarations,

```
character(len=11) :: msg = "Hello World"
```

the following write statement can be used to display the string in variable *msg* with no leading or trailing spaces. The following write statement,

```
write (*,'(a11)') msg
```

will display “**Hello World**”. The count is not required when using the character format specifier. For example, the statement,

```
write (*,'(a)') msg
```

will display the same “**Hello World**” string. Multiple variables or strings can be displayed. Also, the count can be used to display a portion of the string. For example, to display the string in variable *msg* and the string “Goodbye cruel world”.

```
write (*,'(a9,2x,a)') msg, "Goodbye cruel world"
```

Which will display “**Hello Wor    Goodbye cruel world**” to the screen. Note that for the first string variable, *msg*, only the first 9 characters of the 11 total characters are displayed since the A9 format was used.

## 10.8 Advance Clause

The advance clause instructs the computer whether or not to advance the cursor to the next line. The possible values are “yes” and “no”. If the advance clause is not included, the default value is “yes”. This clause is useful when prompting for user input to allow the input to be entered on the same line as the prompt. When using the advance clause, the free format (\*) is not allowed. A format must be included.

For example, the period program from the previous chapter included the statements:

```
! prompt for and read the n value  
write (*,'(a)') "Enter count to sum: "  
read (*,*) n
```

Which, when executed, the input is entered on the line following the prompt.

```
c:\mydir> sums  
Example Program  
  Difference between sum of squares  
  and square of sums  
  
Enter count to sum:  
10  
Difference:           2640
```

When the advance clause is used with the setting of “no”, as follows:

```
! prompt for and read the n value
  write (*,'(a)', advance="no") "Enter count to sum: "
  read (*,*) n
```

The resulting execution would be as follows:

```
c:\mydir> sums
Example Program
  Difference between sum of squares
  and square of sums

Enter count to sum: 10
Difference:          2640
```

Which allows the input to be entered on the same line as the prompt.

## 10.9 Example

This example will read a date from the user (month, day, and year, on the same line), determine the day of week (for that month/day/year). Then, the program will display the original input date (numeric form) and the formatted date. The original input date will be displayed with the applicable fields including the leading 0's (i.e., 01/01/2010).

### 10.9.1 Understand the Problem

For this problem, we will read the three numbers for the date from the user. The verification of the date information is left as an exercise.

To calculate the day on which a particular date falls, the following algorithm may be used (the divisions are integer divisions):

```
a = (14 - month) / 12
y = year - a
m = month + 12*a - 2
daynum = [ date + y + y/4 - y/100 + y/400 + (31*m/12) ] mod 7
```

The value of *daynum* is 0 for a Sunday, 1 for a Monday, 2 for a Tuesday, etc.

### 10.9.2 Create the Algorithm

For this problem, first we will need to read the date. The verification of the date entered and error checking is left as an exercise. Then, the original input date can be displayed, in numeric form, formatted appropriately. For a date, this would mean two digits for the month, a “/”, two digits for the day, a “/”, and four digits for the year. When the day is only one digit, for example 5, it is customary to display a “05” so the program will ensure this occurs.

```
! declare variables
!   integer -> month, day, year
!   integer -> a, y, m, daynum (for calculations)
! display initial header
```



```
! prompt for month, day, and year
! read month, day, and year
! display formatted numeric month/day/year
```

Then the program can calculate the day of the week (based on the formula) and convert the resulting number (0-6) into a date string and display the result.

```
! calculate day of week
! convert day of week (0-6) to string
! convert month (1-12) to string
! display formatted string for day, month, and year
```

For convenience, the steps are written as program comments.

### 10.9.3 *Implement the Program*

Based on the algorithm, the below program could be created.

```
program dateFormatter

! declare variables
implicit none
integer :: month, day, year
integer :: a, m, y, d
character(9) :: amonth, day_of_week

! -----
! display initial header
    write (*,*) "Date Formatting Example"

! prompt for month, day, and year
    write (*,'(a)',advance="no") "Date (month, day, year):"

! read month, day, and year
    read (*,*) month, day, year

! -----
! display formatted numeric month/day/year

    write (*,*) "-----"
    write (*,*) "Input Date: "
    write (*,'(5x, i2.2, a, i2.2, a, i4)') month, "/", &
        day, "/", year

! -----
! calculate day of week

    a = (14 - month) / 12
    y = year - a
    m = month + 12 * a - 2
```

```

        d = mod ( (day + y + y/4 - y/100 + y/400 + (31*m/12)), 7)

! -----
!      convert day-of-week integer to day-of-week string

      select case (d)
        case (0)
          day_of_week = "Sunday  "
        case (1)
          day_of_week = "Monday  "
        case (2)
          day_of_week = "Tuesday "
        case (3)
          day_of_week = "Wednesday"
        case (4)
          day_of_week = "Thursday "
        case (5)
          day_of_week = "Friday  "
        case (6)
          day_of_week = "Saturday "
      end select

! -----
! convert month (1-12) to string
      select case (month)
        case (1)
          amonth = "January  "
        case (2)
          amonth = "February "
        case (3)
          amonth = "March    "
        case (4)
          amonth = "April    "
        case (5)
          amonth = "May      "
        case (6)
          amonth = "June     "
        case (7)
          amonth = "July     "
        case (8)
          amonth = "August   "
        case (9)
          amonth = "September"
        case (10)
          amonth = "October  "
        case (11)
          amonth = "November "
        case (12)
          amonth = "December "

```

```

        end select

! -----
! display formatted string for day, month, and year

        write (*,'(/a)') "Formatted Date:"
        write (*,'(5x, a, a, a, 1x, i2.2, a, i4/)')      &
                trim(day_of_week), " ", trim(amonth),    &
                day, " ", year

    end program dateFormatter

```

The spacing and indentation is not required, but help to make the program more readable. The *trim()* intrinsic function removes any trailing spaces from the input string. Additional information regarding handling character data types is provided in the following section.

### 10.9.4 Test/Debug the Program

For this problem, the testing would be to ensure that the output formatting is correct. Since there is no error checking on the input, only correct dates should be entered. Test the program on a series of different input values and verify that the output is correct for those input values.

## 10.10 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 10.10.1 Quiz Questions

Below are some quiz questions.

- 1) What is the format specifier for each of the following:
  - a) integer values
  - b) real values
  - c) logical values
  - d) horizontal spacing (i.e., spaces)
  - e) a new line
  - f) characters/strings
- 2) Describe the output of the following code fragment (1 pts each):  
 Note, show blanks with an \_ (underscore) character.
 

```

write (*,'(a5)') "Hello World"
write (*,'(a)') "Hello World"

```
- 3) Describe the output of the following code fragment (3 pts):  
 Note, show blanks with an \_ (underscore) character.
 

```

integer :: number = 5
write (*,'(i5.3)') number

```

- 4) What is the write statement and format specifier to output the integer variable **num1** which contains a value between 0 and 999 (right justified, no leading zero's, no additional spaces).
- 5) What is the write statement and format specifier to output the real value of **pi** which has been initialized to 3.14159 (right justified, no additional spaces)?
- 6) What is the *single* write statement and format specifier to output "Programming" and "Is Fun!" on two different lines?
- 7) What is the single write statement and format specifier to output "Enter Number:" and leave the cursor on the current line?

### 10.10.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the date formatting program example, compile, and execute the program. Test the program on a series of different input values and verify that the output is correct for those input values.
- 2) Update the date formatting program to perform complete error checking on the date entered. That is, the program should check for appropriate values for month (between 1 and 12), check for appropriate values for date (between 1 and 31), including checking for a valid date for the specific month, and ensure that the value for year is between 1970 and 3000 (inclusive). For example, April 31 is not a valid date. Additionally, the program should check for a leap year to see if February has 28 or 29 days for that year. Test the program on a series of different input values and verify that the output is correct for those input values.
- 3) Write a Fortran program that displays an amortization schedule. The program should read the loan amount, annual interest rate, and the loan term in months (from a single line).

The formula for calculating the monthly payment is:

$$payment = amount * \left( irate * \frac{(1 + irate)^{term}}{(1 + irate)^{term} - 1} \right)$$

*Note*, the annual interest rate, **irate**, in the formula must be converted to a monthly rate (divided by 12) and then divided by 100 (to convert from percentage). During the time period, **term**, some of each monthly payment will be used to pay the interest and some will be used to reduce the outstanding balance. The monthly interest amount can be calculated by multiplying the monthly interest rate times outstanding balance. The amounts must be lined up with only two digits for cents. The payment number must display three digits, including leading zeros if necessary. Test the program on a series of different input values and verify that the output is correct for those input values.

Output will consist of appropriate headings and aligned and formatted columns for payment number, monthly payment, principal paid, interest paid, and outstanding balance. A sum will appear at the bottom of each column.

Loan amounts shall not exceed \$250,000 and the maximum loan term shall not exceed 360 months. Be sure to follow the requirement that the three input variables will be formatted on a single line.

- 4) Write a Fortran program that calculates and displays compounded interest. The program should read the initial principal amount, interest rate percentage, and the term (number of years). The program should display a summary of the input and the yearly compounded interest. Refer to the example output for formatting.

The formula for compounding interest is:

$$\text{value} = \text{principal}(1 + \text{interest})^{\text{year}}$$

*Note*, the interest rate percentage read from the user must be converted to a number (i.e., divided by 100). The output must be formatted in a manner similar to the example output. This includes ensuring that the dollar amounts are displayed with the appropriate two decimal points. Test the program on a series of different input values and verify that the output is correct for those input values.



# 11 Characters and Strings

---

Fortran was originally developed for scientific and engineering application requiring significant mathematical calculations. However, the Fortran 95/2003/2008 language includes extensive character and string handling capabilities.

## 11.1 Character and String Constants

A character is a single character or symbol, typically enclosed in quotes. For example, letters (“A”-“Z” and “a” - “z”), punctuation (“!”, “,”, “?”, etc.) , symbols, (“@”, “#”, “>”, etc.), and digits “1”, “2” are characters.

Some examples include:

```
"X"  
"z"  
"5"
```

Character and string constants are case sensitive. So, character “X” (upper-case) is not the same as “x” (lower-case). When a digit is enclosed in quotes, it is treated as a character and consequently arithmetic operations (addition, subtraction, etc.) are not allowed.

A string is a series of characters. A string consists of an arbitrary sequence of characters also enclosed in quotes. Some examples include:

```
"Hello World."  
"456"  
"1 2 3"  
"456?"  
"Goodbye cruel world!!"  
"Have a nice day?"
```

Since digits enclosed in quotes are not numeric values, the strings “1 2 3” and “456?” are allowed.

A problem arises if you want to have a quote in the string itself. A double quote will be interpreted as a single within a string. The two quotes must be together (no spaces between). For example, the string:

```
"He said "wow" when he heard"
```

Would be displayed as

```
"He said "wow" when he heard"
```

The double-quote is sometimes referred to as an escape character. Strings and characters must be associated with the character data type.

## 11.2 Character Variable Declaration

A character variable is a variable that can contain a set of 1 or more characters. Character variables must have a defined length. All declarations are placed in the beginning of the program (after the program statement). Declaring character variables formally defines the type and sets aside memory.

This is performed with a type declaration statement in the form of:

```
<type> :: <list of variable names>
```

For character variables, the type is “character”. For example, to define a character variable to hold the day of week (i.e., “Wednesday”), the following declaration,

```
character(len=9) :: dayofweek
```

Would define the variable, *dayofweek*, with a maximum length of 9 possible characters.

Additional examples include:

```
character(len=3) :: symbol1, symbol2  
character :: symbol3  
character(1) :: symbol4, symbol5  
character(30) :: symbol6, symbol7
```

The declarations can be entered in any order, however they must be at the beginning of the program.

The “len=” is optional and can be omitted. When the length is omitted entirely, the default length is set to 1. This, “character”, “character(len=1)”, and “character(1)” are all the same.

When multiple variables are included in a single declaration, they must all be the same length. If different lengths are required, separate declaration lines are required.

## 11.3 Character Variable Initialization

It is possible to declare a character variable and to set an initial value at the same time. This initialization is not required, but can sometime be convenient. For example, to define a character variable, *dayofweek*, and set it to the day of week:

```
character(len=9) :: dayofweek="Wednesday"
```

Additional examples include:

```
character(9) :: thismonth="June", lastmonth, nextmonth="July"  
character :: ltr1="A", ltr2="b"
```

Spaces or no spaces between the variables, equal signs, semicolons, and commas are allowed. Variables initialized at declaration can be changed during program execution as needed.

## 11.4 Character Constants

It is possible to declare a character variable, set its initial value, and ensure that the value cannot be changed. For example, to define a character constant, *language*,

```
character(len=7), parameter :: language="English"
```



To save counting characters, the “\*” can be used. For example,

```
character(len=*), parameter :: university="UNLV"
```

This instructs the Fortran compiler to count the characters and set the appropriate length.

## 11.5 Character Assignment

Assignment is a term for setting a character variable equal to some value (character or string).

Assignment is performed with an equal (=) sign. For example, given the declaration,

```
character(9) :: thismonth
```

A value can be assigned to the variable as follows,

```
thismonth = "September"
```

When character variables are assigned, they are filled from the left and automatically padded with blanks if necessary. For example, if the variable *thismonth* is reset

```
thismonth = "May"
```

The variable *thismonth* contains “May ” (e.g., “May” with an additional 6 blanks).

## 11.6 Character Operators

The only character operator is “//” (concatenation) which simply concatenates two strings together. For example,

```
"University of " // "Nevada Las Vegas"
```

Characters variables and literals may be used with concatenation. For example, given the following declaration,

```
character(len=6) :: str1="ABCDEF", str2="123456"  
character(len=12) :: str3
```

The following statements

```
str3 = str1 // str2
```

will set *str3* to “ABCDEF123456”.

## 11.7 Character Substrings

A substring is a subset or part of a string. A substring can be selected based on its position within the string with the first character corresponding to 1, the second character corresponding to 2, and so forth. The substring is selected or specified with a start and stop position in the form of (**start:stop**). The stop must be greater than or equal to the start position.

For example, given the following declaration,

```
character(len=6) :: str1="ABCDEF", str2="123456", str3
```

The following statements

```
str3 = str1(1:3) // str2(4:6)
```

will set **str3** to “ABC456”.

## 11.8 Character Comparisons

The standard relational operators (“==”, “>”, “>=”, etc.) have some limitations when character data is used. Simple comparisons, such as,

```
"A" < "D"  
"ABC" == "ABC"
```

will work as expected. That is, both will evaluate to true.

However, when comparing, the following characters, each will evaluate to false.

```
"A" > "a"  
"20" < "100"  
"ABCD" <= "ABC"
```

This is a result of the relational operations referring to the assigned values (based on their location in the ASCII table located in Appendix A).

Comparisons between digits, “0” - “9”, will work relative to each other. Comparisons between upper-case letters, “A” - “Z”, will also work relative to each other. Comparisons between lower-case letters, “a” - “z”, will also work relative to each other. Since the lower case letters are after the upper case letters in the table an upper-case letter will be less than a lower-case letter. The digits are in the table before the letters (upper- and lower-case), so they will evaluate as less than letters. This must be taken into account when dealing with character comparisons.

## 11.9 Intrinsic Character Operations

There are a number of character oriented intrinsic operations. Some of the basic character oriented functions include:

Function	Description
ACHAR(I)	Returns the character represented by integer argument I based on the ASCII table (Appendix A). Integer argument I must be between 1 and 127.
IACHAR(C)	Returns the integer value of the character argument C represented by ASCII table (Appendix A).
LEN(STR)	Returns an integer value representing the length of string argument STR.
LEN_TRIM(STR)	Returns an integer value representing the length of string

	argument STR excluding any trailing spaces.
TRIM(STR)	Returns string based on the string argument STR with any trailing spaces removed.
ADJUSTL(STR)	Return a string modified by removing leading spaces. Spaces are inserted at the end of the string as needed.
ADJUSTR(STR)	Return a string modified by removing trailing spaces. Spaces are inserted at the beginning of the string as needed.

A complete list of intrinsic functions can be found in Appendix D.

## 11.10 Example

This example will scan a string and convert all lower-case letter to upper-case.

### 11.10.1 Understand the Problem

For this problem, the string will be read from the user with a maximum of 80 characters. Any lower case letters encountered will be converted to upper-case. All other characters (digits, symbols, etc.) will be left alone. To determine if a character is lower-case, we can see if it is between “a” and “z”. The final string will be displayed back to the screen. Based on the ASCII table in Appendix A, there is a specific, fixed difference between each upper and lower-case letter. Thus, in order to convert a lower-case character to upper-case, that difference can be subtracted. However, in order to perform the subtraction, each character needs to be converted into an integer (based on its value in the ASCII table). The ICHAR() intrinsic function performs this conversion. After the conversion (subtraction), the integer must be converted back into its corresponding character, which can be accomplished with the ACHAR() intrinsic function. These functions work on a single character/integer, so each character will need to be addressed individually.

### 11.10.2 Create the Algorithm

For this problem, first we will need to prompt for and read the input string. Then any trailing blanks will be removed and the final length can be determined. Based on that length, each character will be accessed and converted if needed.

```

! declare variables
!   integer -> string1, string2, I, strlen
! display initial header
! prompt for string
! read string
! trim any trailing blanks
! determine length of string
! loop
!   access each character
!   if check lower-case ("a" - "z") -> convert to upper-case
! display final string

```

For convenience, the steps are written as program comments.

**11.10.3 Implement the Program**

Based on the algorithm, the below program could be created.

```

program caseConverter

! declare variables
implicit none
integer :: i, strlen
character(80) :: string1

! -----
! display initial header
      write (*,'(a,/)' ) "Case Conversion Example"

! -----
! prompt for string
! read string
      write (*,'(a)',advance="no") "Enter String (80 char max): "
      read (*,'(a)') string1

! -----
! trim any trailing blanks
! determine length of string
      strlen = len(trim(string1))

! loop
      do i = 1, strlen

!      access each character
!      if check lower-case -> convert to upper-case

          if ( string1(i:i) >= "a" .and. string1(i:i) <= "z" ) then
              string1(i:i) = achar( iachar(string1(i:i)) - 32)
          end if

      end do

! -----
! display final string

      write (*,'(/,a)') "-----"
      write (*,'(a,/,2x,a,/)' ) "Final String: ", string1

end program caseConverter

```

The spacing and indentation are not required, but help to make the program more readable.

### 11.10.4 Test/Debug the Program

For this problem, the testing would ensure that the output string is correct for the given input.

For example, the following output,

```
c:\mydir> case
Case Conversion Example

Enter String (80 char max): Hello World!?

-----
Final String:
    HELLO WORLD!?
```

Each lower-case letter was converted to upper-case while the upper-case, space, and punctuation were unchanged. The program should be executed with a series of test inputs to verify the correct output.

## 11.11 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 11.11.1 Quiz Questions

Below are some quiz questions.

- 1) What is the declaration for a character variable, *msg*, to contain the string “Hello World!”?
- 2) Given the following conditional expressions,

```
"D" > "c"
"100" < "20"
"Da" > "cA"
"20" < "10"
"d" > "C"
"20" < "100"
"ABBC" <= "ABCD"
```

state which will evaluate to true and which to false.

- 3) Given the following Fortran statements,

```
character(len=6) :: str1="abcdef", str2="ABCDEF"
character(len=6) :: str3="123456", str4="78910"
character(len=12) :: astr1, astr2, astr3, astr4

astr1 = str1(1:3)
astr2 = str3(4:6)
astr3 = str3 // str4
astr4 = str2(4:6) // str3(1:3) // str1(2:3)
```

provide the resulting strings (*astr1*, *astr2*, *astr3*, and *astr4*).

- 4) How can the integer value (based on the ASCII table) of a character be obtained?
- 5) How can integer value be converted to a character (based on the ASCII table)?

### 11.11.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the case conversion example program, compile, and execute the program. Test the program on a series of different input values and verify that the output is correct for those input values.
- 2) Update the case conversion example program to convert any upper-case characters to lower-case characters. Test the program on a series of different input values and verify that the output is correct for those input values.
- 3) Write a program to read a string and count the vowels ("a", "e", "i", "o", and "u"). The program should provide a count for each vowel and a total count of vowels. The program should ensure that the vowels are counted for both upper and lower-case. Test the program on a series of different input values and verify that the output is correct for those input values.
- 4) Write a program to read 5 strings ( $\leq 80$  characters each) and display the strings in alphabetical order. Test the program with a variety of different input strings, including digits, upper-case, and lower-case characters. Test the program on a series of different input values and verify that the output is correct for those input values.

## 12 File Operations

---

File operations allow Fortran programs to read from files and/or write to files. The basic read and write statements for file operations are the same as previously used with some additional information or clauses.

### 12.1 File Open

A file must be opened before information can be written or read from a file. In order to open a file, the operating system needs some information about the file in order to correctly identify the file and establish the access parameters (i.e., read, write, etc.). The open statement “clauses” provide that information to the operating system.

The file open statement is as follows:

```
open (unit=<unit number>, file=<file name>,           &
      status=<file status>, action=<file action>,       &
      position=<file position>, iostat=<status variable>)
```

The following table summarizes the various open statement clauses.

Clause	Explanation
<b>unit</b>	Unit number for subsequent file operations (i.e., read, write, etc.). Typically an integer between 10 and 99.
<b>file</b>	Name of file to be opened. Can be character literal or a character variable.
<b>status</b>	Status of file. Allowable options “old”, “new”, or “replace” “old” → the file must already exist. “new” → a new file will be created. “replace” → a new file will be created, replacing an existing one if necessary.
<b>action</b>	Action or open operation. Allowable options are “read”, “write”, or “readwrite”. “read” → read data from a file. “write” → write data to a file. “readwrite” → simultaneously read data from and write data to a file.
<b>position</b>	Position or place to start. Allowable options are “rewind” (beginning), “append” (end).

<b>iostat</b>	Name of variable for system to place a status code indicating the status (success or failure) of the operation. If the status variable is set to 0, the operation is successful. If the status variable is set to >0, an error occurred and the operation was unsuccessful.
---------------	---

The unit number assigned should be between 10 and 99.

## 12.2 File Write

The file must be opened for “write” or “readwrite” access before any information can be written to the file. The general form of the write statement is as follows:

```
write (unit=<unit number>, fmt=<format statement>,          &
      advance="no", iostat=<variable>)                      &
      <variables/expressions>
```

The write statement is the same as the simple write; however, the unit number must be the number assigned during the open operation. Normally, the next write will be on the next line. The **advance="no"** is optional. If it is included, the next write will be on the same line where the previous line stopped.

For example to open a file named temp.txt and place the string “Fortran Example” and the numbers 42, and 3.14159 on separate lines, the following declarations:

```
integer :: myanswer=42, myopenstatus, mywritestatus
real, parameter :: pi=3.14159
character(15) :: mymessage="Fortran Example"
character(8) :: myfilename="temp.txt"
```

and the following Fortran statements,

```
open (unit=10, file=myfilename, status="replace",          &
      action="write", position="rewind",                    &
      iostat=myopenstatus)
if (myopenstatus > 0) stop "Cannot open file."
write (10, '(a/, i5/, f7.5)', iostat=mywritestatus)        &
      mymessage, myanswer, pi
```

would write the file information to the file.

## 12.3 Stop Statement

The Fortran **stop** statement, as used in the previous example, will immediately terminate the program. The optional string, as used in the previous example, will be displayed. This is useful for ending the program when certain error critical conditions prevent any further progress.



## 12.4 File Read

The file must be opened for “read” or “readwrite” access before any information can be read from the file. The general form of the write statement is as follows:

```
read (unit=<unit number>, fmt=<format statement>,          &  
      iostat=<variable>) <variables>
```

The read is the same as the simple read; however, the unit number must be the number assigned during the open operation. If the status variable is set to less than 0, that is an indication that the end of the file has been reached. For example, if the file *numbers.dat* exists and has two numbers (on separate lines), the following declarations,

```
integer :: num1, num2, myopenstatus, myreadstatus  
character(11) :: myfilename="numbers.txt"
```

and the following Fortran statements,

```
open (unit=12, file=myfilename, status="old",              &  
      action="read", position="rewind",                    &  
      iostat=myopenstatus)  
if (myopenstatus > 0) stop "Cannot open file."  
  
read (12, '(i5)', iostat=myreadstatus) num1  
read (12, '(i5)', iostat=myreadstatus) num2
```

would read information from the file.

## 12.5 Rewind

An open file can be reset back to the beginning. This might be useful if the file needs to be read twice. The rewind statement will reset the file read pointer and subsequent reads will start back at the beginning. The general form of a rewind statement is:

```
rewind(<unit number>)
```

Where the unit number was assigned during the initial open. The file must be open when the rewind is executed for the rewind to work correctly.

## 12.6 Backspace

When reading from a file, each successive read will return the next line from the file. The computer keeps track of which lines have been read and will automatically return the next line. It is possible to read a line and then backspace and re-read the line again with the backspace statement.

The general form of a backspace statement is:

```
backspace(<unit number>)
```

Where the unit number was assigned during the initial open. The file must be open when the backspace is executed for the backspace to work. It should be noted that this operation is not used very often.

## 12.7 Close File

An open file should be closed when it is no longer needed. The general form of a close statement is:

```
close(<unit number>)
```

Where the unit number was assigned during the initial open.

For the previous examples,

```
close(10)  
close(12)
```

would be used to close the opened files.

## 12.8 Example

In this example, we will write a Fortran program to read an input file and write a line number and the original line to an output file.

### 12.8.1 Understand the Problem

For this problem, we will read the file names from the user and open the files. Then we will read a line from the input file, and write the line number and line to the output file. When done, we will close the file.

### 12.8.2 Create the Algorithm

For this problem, first, we will need to prompt for and read the file names from the user and ensure that they open correctly. If the file cannot be opened, an error message will be displayed and the file names will be re-read.

```
! declare variables  
!   integer -> i, rdopst, wropst  
!   character -> line  
! display initial header  
! loop  
!   prompt for input file name  
!   read input file name  
!   open input file (read access)  
!   if open unsuccessful, display error message  
!   otherwise, end loop  
! loop  
!   prompt for output file name  
!   read output file name  
!   open output file (write access)  
!   if open unsuccessful, display error message  
!   otherwise, end loop
```

Once the file is open, a line will be read from the input file, and the line number and the line will be written to the output file. For this example, we will assume that a line will have 132 or less characters.

This process will continue until there are no more lines in the input file.

```
! loop
!   read line from input file
!   if end of file, exit loop
!   write line number and line to output file
! close files
```

For convenience, the steps are written as program comments.

### 12.8.3 *Implement the Program*

Based on the algorithm, the below program could be created.

```
program linenumbers

! declare variables
implicit none
integer :: i, rdopst, wropst, rdst
character(30) :: rdfile, wrfile
character(132) :: line

! display initial header
write (*,*) "Line Number Example"

! -----
! prompt for input file name
do
    write (*,'(a)', advance="no") "Input File Name: "

    ! read input file name
    read (*,*) rdfile

    ! open input file (read access)
    ! if open unsuccessful, display error message
    ! otherwise, end loop
    open(12, file=rdfile, status="old",           &
         action="read", position="rewind",       &
         iostat=rdopst )
    if (rdopst==0) exit

    write (*,'(a/,a)') "Unable to open input file.", &
        "Please re-enter"
end do

! -----
! prompt for output file name
do
    write (*,'(a)', advance="no") "Output File Name: "
```

```

        ! read output file name
        read (*,*) wrfile

        ! open output file (read access)
        ! if open unsuccessful, display error message
        !     otherwise, end loop

        open(14, file=wrfile, status="replace",           &
              action="write", position="rewind",         &
              iostat=wropst )
        if (wropst==0) exit

        write (*,'(a, a/,a)') "Unable to open ",         &
                               "output file.", "Please re-enter"
    end do

! -----

    i = 1
    do
        ! read line from input file
        read (12, '(a)', iostat=rdst) line

        ! if end of file, exit loop
        if (rdst >0) stop "read error"
        if (rdst <0) exit

        ! write line number and line to output file
        write (14, '(i10,2x,a)') i, line
        i = i + 1
    end do

! close files
    close(12)
    close(14)

end program linenumbers

```

The spacing and indentation are not required, but help to make the program more readable.

### 12.8.4 *Test/Debug the Program*

For this problem, testing would involve executing the program with various input files and verifying that the line numbers are correctly added.

## 12.9 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 12.9.1 Quiz Questions

Below are some quiz questions.

- 1) What must occur before a file can be read or written?
- 2) What is the range of the valid unit numbers for a file open?
- 3) For the following statements:

```
integer :: opnstat
character(20) :: filename="file.txt"

open (14, file=filename, status="old", action="read",
      &          position="rewind", iostat=opnstat)

if ( opnstat > 0 ) then
    write (*, *) "Error, can not open file."
    stop
end if
```

- a) What is the name of the file being opened?
  - b) What is the *unit* number that will be used for subsequent write operations?
  - c) Does the error message get printed if the file does not exist (yes/no)?
  - d) What does it mean when the status variable is  $> 0$ ?
- 4) Assume the file **answers.txt** exists, has been successfully opened using unit number **20**, and contains the following data:

```
"line 1 data1=23 data2 =034 data3 =05123"
```

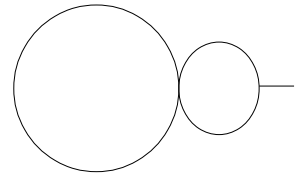
What is the read statement required to get data1, data2, and data3 into the integer variables **num1**, **num2**, and **num3** respectively?

### 12.9.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the line numbers program, compile, and execute the program. Test the program on a number of different input values.

- 2) Imagine the movement of a small circle that rolls on the outside of a rigid circle. Imagine now that the small circle has an arm, rigidly attached, with a plotting pen fixed at some point. That is a epicycloid, commonly called a Spirograph<sup>9</sup>. Write a Fortran 95/2003/2008 program to generate the (x,y) coordinates for a Spirograph drawing.



First, the program should prompt for an output file name, read the file name, and open the file. If the file cannot be opened, the program should display an appropriate error message and re-prompt. Next, the program should prompt for a read radius 1 (fixed circle), radius 2 (moving circle), and offset position (rigid arm length). The radius 1, radius 2, and offset position values must be between -100 and +100 (inclusive). If any value is out of range, the program should re-prompt. If the user does not enter a valid file name or number of input values after three tries, the program should terminate with an appropriate error message. That is, three errors are acceptable, but if a fourth error is made, the program should terminate. The prompts and subsequent reads should be on the same line (see example).

Then, the program should generate (x,y) points based on one the following formulas:

$$x = ((radius_1 + radius_2) * \cos(step)) + \left( offsetposition * \cos\left( (radius_1 + radius_2) * \frac{step}{radius_2} \right) \right)$$

$$y = ((radius_1 + radius_2) * \sin(step)) + \left( offsetposition * \sin\left( (radius_1 + radius_2) * \frac{step}{radius_2} \right) \right)$$

The step should start at 0.0 and stop at 360.0, stepping by 0.1. All writes should use a formatted output (not the '\*'). Then the program should close the output file and inform the user the plotting is completed. Test the program on a number of different input values.

- 3) Write a Fortran program that plays the Chaos Game. To play the Chaos Game, plot 3 points A, B, C, and an arbitrary initial point X1. Then, generate a random number between 1 and 3 representing A (1), B (2), or C (3). If A comes up, plot the midpoint of the line joining X1 to A. If B comes up, plot the midpoint of the line joining X1 to B; the case with C is similar. Call this new point X2. Generate another random number between 1 and 3. Plot the midpoint of the line joining X2 to either A, B or C depending on the random number. Call this new point X3. Repeat this process *N* times. Test the program on a number of different input values.

Refer to Appendix C for more information regarding generating random numbers.

**Note**, the correct output of this program is shown on the cover page. The image is referred to as a Sierpinski Triangle<sup>10</sup>.

<sup>9</sup> For more information, refer to: <http://en.wikipedia.org/wiki/Spirograph>

<sup>10</sup> For more information, refer to: [https://en.wikipedia.org/wiki/Sierpinski\\_triangle](https://en.wikipedia.org/wiki/Sierpinski_triangle)

## 13 Single Dimension Arrays

---

An array is a collection or set of data. A variable can hold a single value. An array can hold multiple values. The type (i.e., integer, real, etc.) of each item in the array must be the same. In order to access specific elements, an *index* or *subscript* is used. The index specifies which element or value in the array is being accessed.

An array is considered a direct access structure since any element can be accessed directly without accessing any other elements.

The most basic form of an array is a single dimension array. A single dimension array can be thought of as a single column in a spreadsheet. The column name, like **A**, is the array name and the row number is like the index. For example, a spreadsheet column might appear as follows:

	A
1	<value>
2	<value>
3	<value>
4	<value>
5	<value>
6	<value>
7	<value>
8	<value>
9	<value>
10	<value>

A single dimension array is logically the same and might look like:

Index	Array Name
1	<value>
2	<value>
	...
	...
<b><i>n-1</i></b>	<value>
<b><i>n</i></b>	<value>

The array name is chosen by the programmer and thus is not limited to **A**. The specific syntax requires an index or subscript to specify which element of the array to access. By default, the first element is at index=1, the next at index=2, and so forth. This can be changed if needed.

## 13.1 Array Declaration

An array must be declared before use. The type of the array is defined followed by the size or dimension. There are two ways to declare an array; static and dynamic.

### 13.1.1 Static Declaration

A static declaration means that the size or dimension of the array must be defined initially (before the program is compiled). The size definition cannot be altered. The general form an array declaration is,

```
type, dimension(extent) :: name1, name2, ..., nameN
```

where type is the data type (integer, real, etc.) of the array. The dimension specifies the extent or size, and **name1, name2, ..., nameN** are names of one or more arrays being declared.

For example, to declare an array of 1000 integers,

```
integer, dimension(1000) :: nums1
```

will create an array, **nums1**, with space for 1000 integer values.

In this example, the extent is 1000 which means that the array indexes will range from 1 to 1000. The extent can be changed by specifying the extent as:

```
smaller-integer : larger-integer
```

When only one number is specified, the smaller-integer is assumed to be 1. When both numbers are specified, the array index's will range between the smaller-integer and the larger-integer. For example, a declaration of:

```
integer, dimension(-5:5) :: ranges
```

will create an array, **ranges**, with indexes between -5 and 5 (inclusive). Using index values not within the specified range will result in an error. If the compiler option, **fcheck=bounds**, to verify subscript bounds is used, this error will be trapped at run-time. However, if the run-time bounds checking is not turned on, the error may not be obvious and could result in unrelated problems such as other variables being over-written or even security holes in some cases.

### 13.1.2 Static Array Declaration

Static array declarations are appropriate when the maximum size of the array is known ahead of time. The size of a static array is set by the declaration and once the array is declared, the size cannot be changed. Optionally, the array value can be initialized by the declaration. For example, to declare an array named **costs** with 4 elements and set the four elements to 10.0, 15.0, 20.0, and 25.0,

```
real, dimension(4) :: costs=(/10.0, 15.0, 20.0, 25.0/)
```

The numbers, enclosed between the “/”s are assigned in order. So, **costs(1)** is set to 10.0, **costs(2)** is set to 15.0, **costs(3)** is set to 20.0, and **costs(4)** is set to 25.0.



Additionally, after declaration, it is possible to initialize all elements of an array to a single value with an assignment statement. For example,

```
costs = 0.0
```

will set all four elements of the *costs* array to zero.

### 13.1.3 *Dynamic Array Declaration*

A dynamic declaration means that the size or dimension of the array can be set when the program is executed. Dynamic array declarations are appropriate when the maximum size of the array is not known ahead of time and can be determined based on information obtained when the program is executing. However, once set, the size cannot be altered. When using a dynamic declaration, the array type and name must be defined. This only specifies the name and type of the array, but does not reserve any space for the array. During the program execution, the array must be allocated. The allocation will create the space for the array. Only after the array has been allocated can it be used.

For example, to declare an array,

```
integer, dimension(:), allocatable :: nums2
```

reserves the name *nums2*, but does not reserve any space for values.

#### 13.1.3.1 *Dynamic Array Allocation*

To allocate the space for the array, the *allocate* statement must be used. Before an array can be allocated, it must be declared as allocatable. The general form of the *allocate* statement is:

```
allocate(<array name>, stat=<status variable>)
```

The status variable must be an integer variable and will be used by the system to place a status code indicating the status (success or failure) of the operation. If the status variable is set to 0, the allocation was successful. If the status variable is set to >0, an error occurred and the allocation was not successful.

For example, given the declarations,

```
integer, dimension(:), allocatable :: nums2  
integer :: allst
```

the following *allocate* statement allocates space for 1000 numbers in array *nums2*,

```
allocate(nums2(1000), stat=allst)
```

The size, 1000 in this example, can be a variable, but it must be an integer. The status variable *allst* will be set to 0 if the allocation is successful. However, if the status variable *allst* is set to a value >0, an error occurred and the allocation was not successful.

## 13.2 Accessing Array Elements

To access elements in an array, the array name and an index must be specified. The index must be an integer or integer expression and enclosed in parentheses. The general format is,

```
array-name(<integer expression>)
```

For example, given the declaration,

```
real, dimension(10) :: times
```

would declare an array with ten elements. To place a value 121.3 in the first array element,

```
times(1) = 121.3
```

And to place 98.125 in the fifth element,

```
times(5) = 98.125
```

The index in these examples is a literal. However, the index can be an integer variable or integer expression.

For example, given the following declarations,

```
real, dimension(10) :: temps  
integer :: i=5, j
```

would declare an array with ten elements. To place a value 98.6 in the fifth array element,

```
temps(i) = 98.6
```

To access the fifth element, subtract 3.0 and place the result in the sixth element,

```
temps(i+1) = temps(i) - 3.0
```

To set all elements of the temps array to 0.0, a loop could be used as follows:

```
do i = 1, 10  
    temps(i) = 0.0  
end do
```

Array elements can be accessed as many times as needed.

### 13.2.1 Array Bounds

When an array is declared for a specific size, only that many elements can be used. For example, if an array is declared with 10 elements, only 10 elements are available. Given the following declaration,

```
real, dimension(10) :: expArr
```

would declare an array with ten elements.

To place a value 42.5 in the first array element and 73.5 in the last array element.

```
expArr(1) = 42.5
expArr(10) = 73.5
```

However, if an array element is accessed that is outside the declared bounds, it is an error. For example,

```
expArr(11) = 99.5
```

would be an error. These kinds of errors would be difficult to find. However, if the bounds checking is turned on (as noted in chapter 3), when the program is executed the error will be noted.

To compile with bounds checking turned on, the following compile command should be used:

```
C:\fortran> gfortran -fcheck=bounds -o hw hw.f95
```

This command will tell the '**gfortran**' compiler to include bounds checking.

In general, using the bounds checking can slow a program down. However, this is not a significant issue when learning to write programs.

### 13.3 Implied Do-Loop

An implied do-loop is a special form of a loop that can be performed on one line. This can be useful for accessing elements in an array. For example, assuming *i* is declared as an integer, the following code,

```
write (*,*) nums(1), num(2), num(3), num(4), num(5)
```

would display each element of the 5 element array to the screen.

The same thing can be accomplished with an implied do-loop as follows:

```
write (*,*) (nums(i), i=1,5)
```

Both forms of the loop will display the same results. If necessary, a step can be used. If the step is omitted, as in the example, the step is defaulted to 1.

### 13.4 Intrinsic Functions

There are a number of intrinsic, or built-in, functions that can be used for arrays.

Function	Description
MAXVAL(ARR1)	Returns the maximum value in array ARR1. Type of value returned is based on the type of the argument array ARR1.
MINVAL(ARR1)	Returns the minimum value in array ARR1. Type of value returned is based on the type of the argument array ARR1.
SUM(ARR1)	Returns the sum of values in array ARR1. Type of value returned is based on the type of the argument array ARR1.

A complete list of intrinsic functions can be found in Appendix D.

### 13.5 Initializing Arrays

An array can be initialized when it is declared. Each element in the array can be initialized to a single value or each element to a different value. The following declaration,

```
real, dimension(10) :: numbers = 1.0
```

will initialize each of the 10 elements in the array numbers to 1.0.

To initialize each element to a different value, each value needs to be specified.

For example, the following declaration,

```
real, dimension(5) :: numbers = (/ 1.0, 2.0, 3.0, 4.0, 5.0 /)
```

will initialize each of the 5 elements in the array numbers to 1.0, 2.0, 3.0, 4.0, and 5.0 respectively.

The implied do-loop may also be used. For example, in the following declaration,

```
integer, dimension(5) :: numbers = (/ (i, i=1,5) /)
```

will initialize each of the 5 elements in the array numbers to 1, 2, 3, 4, and 5 respectively.

### 13.6 Example

In this example we will write a Fortran program to read a series of numbers from a file and compute some statistical information including minimum, maximum, sum, average, and standard deviation<sup>11</sup>.

The standard deviation is calculated as follows:

$$\text{standard deviation} = \sqrt{\frac{\sum_{i=1}^n (\text{average} - \text{list}(i))^2}{n - 1}}$$

As such, the average must be calculated before the standard deviation.

#### 13.6.1 Understand the Problem

The program will display an initial header and get the file name. Specifically, we will need to prompt for the file name, read the file name, and verify that the file is available by attempting to open the file. Then, the program will read the numbers from the file and store them in an array. For this problem, there will be no more than 5000 numbers in the file. After the numbers are in the array, the minimum, maximum, and sum will be found. Next, the average can be computed. Finally, the standard deviation can be calculated (using the provided formula). The numbers should be displayed, ten per line, followed by the results.

11 For more information, refer to: [http://en.wikipedia.org/wiki/Standard\\_deviation](http://en.wikipedia.org/wiki/Standard_deviation)

### 13.6.2 Create the Algorithm

After the header is displayed, the program should prompt for the file name, read the file name, and verify that the file is available by attempting to open the file. If the file cannot be opened, the program will display an error message and re-prompt. If the user does not enter correct information after three tries, the program should terminate with an appropriate error message. That is, three errors are acceptable, but if a fourth error is made, the program will terminate.

```
! declare variables
!   integer -> i, ncount, errs, opstat, rdstat
!   real -> min, max, sum, stdsum
!   real -> array for
!   character -> filename(20)
! display initial header
! loop
!   prompt for file name
!   read file name
!   attempt to open file
!   if file open successful, exit loop
!   display error message
!   count error
!   if >3 errors, terminate program
! end loop
```

Then, the program will loop to read the numbers from the file and store them in an array. The program will check for any read errors (status variable > 0) and for the end of file (status variable < 0).

If a valid number is read, it will be counted and placed in the array.

```
! loop
!   read from the file
!   if error on read, terminate program
!   if end of file, exit loop
!   increment number counter
!   place number in array
! end loop
```

Next, another loop will be used to find the minimum, maximum, and sum of the numbers. To find the minimum and maximum values, we will assume that the first element in the array is the minimum and maximum. Then, the program will check each number in the array. If the number from the array is less than the current minimum value, the current minimum value will be updated to the new value. Same for the maximum, if the number from the array is more than the current maximum value, the current maximum value will be updated to the new value.

```
! initialize min, max, and sum
! loop
!   check for new min
!   check for new max
!   update sum
! end loop
```

Once the sum is available, the average can be computed. Finally, a loop will be used to calculate the

summation for the standard deviation.

```
! calculate average
! initialize stdsum
! loop
!     calculate average – array item
!     update stdsum
! end loop
```

Once the summation is completed, the standard deviation can be computed and the final results displayed. As per the example specifications, the numbers should be displayed 10 per line.

One way to handle this is to display numbers on the same line (with the *advance*=“no” clause) and every 10<sup>th</sup> line display a new line.

```
! calculate standard deviation
! loop to display numbers, 10 per line
! display results
! end program
```

For convenience, the steps are written as program comments.

### 13.6.3 Implement the Program

Based on the algorithm, the program below could be created.

```
program standardDeviation

! declare variables
implicit none
integer :: i, ncount=0, errs=0, opstat, rdstat
real :: num, min, max, sum, average, stdsum, std
real, dimension(5000) :: numbers
character(20) :: filename

! display initial header
write (*,*) "Standard Deviation Program Example."

! loop
do

    ! prompt for file name
    write (*,'(a)', advance="no") "Enter File Name:"

    ! read file name
    read (*,*) filename

    ! attempt to open file
    open(42, file=filename, status="old",           &
         action="read", position="rewind",         &
         iostat=opstat )
```

```

! if file open successful, exit loop
if (opstat==0) exit

! display error message
write (*,'(a)') "Error, can not open file."
write (*,'(a)') "Please re-enter."

! count error
errs = errs + 1
! if >3 errors, terminate program
if (errs > 3) then
    write (*,'(a)') "Sorry you are having problems."
    write (*,'(a)') "Program terminated."
    stop
end if

! end loop
end do

! loop
do
    ! read file
    read (42, *, iostat=rdstat) num
    ! if error on read, terminate program
    if (rdstat>0) stop "Error on read."

    ! if end of file, exit loop
    if (rdstat<0) exit

    ! increment number counter
    ncount = ncount + 1

    ! place number in array
    numbers(ncount) = num

! end loop
end do

! initialize min, max, and sum
min = numbers(1)
max = numbers(1)
sum = 0.0

! loop
do i = 1, ncount

    ! check for new min and new max
    if (numbers(i) < min) min = numbers(i)
    if (numbers(i) > max) max = numbers(i)

```

```

        ! update sum
        sum = sum + numbers(i)

! end loop
end do

! calculate average
average = sum / real(ncount)

! initialize stdsum
stdsum = 0.0

! loop
do i = 1, ncount

        ! calculate (average - array item)^2 and update sum
        stdsum = stdsum + (average - numbers(i))**2

! end loop
end do

! calculate standard deviation
std = sqrt ( stdsum / (real(ncount)-1) )

! display results
write (*,'(a)') "-----"
write (*,'(a)') "Results:"

do i = 1, ncount
        write (*,'(f8.2,2x)', advance="no") numbers(i)
        if (mod(i,10)==0) write (*,*)
end do

write (*,'(a, f8.2)') "Minimum = ", min
write (*,'(a, f8.2)') "Maximum = ", max
write (*,'(a, f8.2)') "Sum = ", sum
write (*,'(a, f8.2)') "Average = ", average
write (*,'(a, f8.2)') "Standard Deviation = ", std

end program standardDeviation

```

The spacing and indentation are not required, but help to make the program more readable.

### 13.6.4 *Test/Debug the Program*

For this problem, the testing would involve executing the program using a file with a set of numbers where the correct results are either known ahead of time or can be calculated by hand in order to verify that the results are accurate.



## 13.7 Arrays of Strings

An array may also contain characters or strings. The declaration and access of array elements is the same. However, the string size must be included in the declaration and cannot be easily changed once defined.

For example, to declare an array to hold 100 titles where each title is a maximum of 40 characters,

```
character(40), dimension(100) :: titles
```

Setting an element is the same. For example, to set the first element of the array to a simple message,

```
titles(1) = "Programming is fun!"
```

The message must contain 40 or less characters.

Character arrays may be statically or dynamically declared as noted in the previous sections.

## 13.8 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 13.8.1 Quiz Questions

Below are some quiz questions.

- 1) Explain why an array is considered a direct access structure.
- 2) Can arrays hold integer values (yes/no)?
- 3) Can arrays hold real values (yes/no)?
- 4) Write the declarations for the following:
  - a) An integer constant, **SIZE1**, set to 100.
  - b) An array, **rvalues**, with 10 real elements.
  - c) An array, **inums**, with **SIZE1** integer arguments.
  - d) An array, **counts**, with 10 real elements whose subscript values range from 0 to 9.
- 5) Given the declarations and the following code:

```
integer, dimension(4) :: arr  
integer :: k
```

```
arr(1) = 10  
arr(2) = 15  
arr(3) = 20  
arr(4) = 25  
k = 3
```

- a) What does **arr(1)** equal?

- b) What does `arr(1) + arr(2)` equal?
  - c) What does `arr(1+2)` equal?
  - d) What does `arr(k)` equal?
- 6) When can an array be allocated (two options)?
- 7) Given the following statements:

```
integer, dimension(5) :: nums
integer :: i=1

nums = 0

do
    if ( i == 5 ) exit
    if ( mod(i,2) == 0 ) then
        nums(i) = 99
    else
        nums(i) = i
    end if
    i = i + 1
end do
```

- a) What values are in the array after execution?
- b) What is the `(nums(i), i=1,5)` referred to as?
- c) What does `write (*,'(1x,i2)') (nums(i), i=1,5)` display? *Note:* use an underscore (“\_”) to show the spaces.

### 13.8.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the standard deviation program, compile, and execute the program. Test the program on a series of different input values.
- 2) Write a Fortran program to cube a series of integer numbers from a file and cube each number. The program should also calculate the real average of the original numbers and the real average of the cubed numbers. Test the program on a series of different input values.
- 3) Write a Fortran program to generate a series of real values between 1.0 and 100.0 and store the numbers in an array. Then, the program should compute the *norm* of a vector (single dimension array). The formula for *norm* is as follows:

$$|norm| = \sqrt{a_1^2 + a_2^2 + a_3^2 + \dots + a_n^2}$$

Refer to Appendix C for more information regarding generating random numbers. Test the program on a series of different input values.

- 4) Write a Fortran program to read a series of numbers from a file, store the numbers in an array, and then sort the numbers using the following selection sort algorithm:

```

for i = len downto 1
  big = arr(1)
  index = 1
  for j = 1 to i
    if arr(j) > big
      big = arr(j)
      index = j
    end_if
  end_for
  arr(index) = arr(i)
  arr(i) = big
end_for

```

You will need to convert the above pseudo-code algorithm into Fortran code. Test the program on a series of different input values.



## 14 Multidimensional Arrays

---

A more advanced array is a multidimensional array. A multidimensional array can be thought of as multiple columns in a spreadsheet. The column name, like **A**, **B**, **C**, etc., are the array columns and the number is like the row.

For example, a two-dimensional array might look like:

Array Name	index	1	2
	1	<value>	<value>
	2	<value>	<value>
	3	...	...
		...	...
		<value>	<value>
	<b><i>n</i></b>	<value>	<value>

The specific syntax requires an index or subscript to specify which element of the array to access. The indexing for a two dimension array is:

Array Name	index	1	2
	1	arr(1,1)	arr(1,2)
	2	arr(2,1)	arr(2,2)
	3	arr(3,1)	arr(3,2)
		...	...
		...	...
	<b><i>n</i></b>	arr( <b><i>n</i></b> ,1)	arr( <b><i>n</i></b> ,2)

By default, the first element is at index=1, the next at index=2, and so forth. This default (where the first number is at index 1) can be changed if needed.

### 14.1 Array Declaration

Multidimensional array declaration is very similar to single-dimension array declaration. Arrays must be declared before use. The type of the array is defined followed by the size or dimension, which in this case requires a size for each dimension. As before, there are two ways to declare an array: static and dynamic.

### 14.1.1 Static Declaration

A static declaration means that the size or dimension of the array must be defined initially (before the program is compiled). The size definition cannot be altered. The general form of an array declaration is,

```
type, dimension(extent,extent) :: name1, name2, ... , nameN
```

where *type* is the data type (integer, real, etc.) of the array. The dimension specifies the size, and name1, name2, ... , nameN are names of one or more arrays being declared.

For example, to declare a two-dimensional array 100 by 100,

```
integer, dimension(100,100) :: nums1
```

will create an array, *nums1*, with space for a total of 10,000 integer values.

In this example, the extent for each dimension is 100, which means that each of the two dimension's indexes will range from 1 to 100. Each or both extents can be changed by specifying the extents as:

```
(smaller-integer:larger-integer, smaller-integer:larger-integer)
```

When only one number is specified, the smaller-integer is assumed to be 1. When both numbers are specified, smaller and larger index, the dimension of the array will range between the smaller-integer and the larger-integer. For example, a declaration of:

```
integer, dimension(0:9,0:9) :: ranges
```

will create an array, *ranges*, with both indexes between 0 and 9 (inclusive). Using index values not within the specified range will result in an error.

### 14.1.2 Dynamic Declaration

The same as single-dimension, a dynamic declaration means that the dimension of the array can be set when the program is executed. Once set, the dimensions cannot be altered. When using a dynamic declaration, the array type and name must be defined, which specifies only the name and type of the array, but does not reserve any space for the array. Then, during program execution, the array can be allocated which will create the space for the array. Only after the array has been allocated can it be used.

For example, to declare an array,

```
integer, dimension(:,:), allocatable :: nums2
```

reserves the name *nums2*, but does not reserve any space for values.

### 14.1.3 Dynamic Array Allocation

To allocate the space for the array, the allocate statement must be used. Before an array can be allocated, it must be declared as allocatable.

The general form of the allocate statement is:

```
allocate(<array name>, <dimension>, stat=<status variable>)
```

The status variable must be an integer variable and will be used by the system to place a status code indicating the status (success or failure) of the operation. As before, if the status variable is set to 0, the

allocation was successful. If the status variable is set to >0, an error occurred and the allocation was not successful.

For example, given the declarations,

```
integer, dimension(:, :), allocatable :: nums2  
integer :: allstat
```

the following allocate statement allocates space for 10,000 numbers in array *nums2*,

```
allocate(nums2(100,100), stat=allstat)
```

The size, 100 by 100 in this example, can be a parameter or variable, but it must be an integer. The variable *allstat* will be set to 0 if the allocation is successful and >0 if the allocation failed.

## 14.2 Accessing Array Elements

To access elements in an array, the array name and the an index must be specified. The index must include an integer or integer expression for each dimension enclosed in parentheses. The general format is,

```
array-name(<integer expression>, <integer expression>)
```

For example, given the declaration,

```
real, dimension(10,5) :: table1
```

would declare an array, *table1*, with a total of 50 elements. To place a value 121.3 in the first row and first column,

```
table1(1,1) = 121.3
```

And to place 98.125 in the tenth row and fifth column,

```
table1(10,5) = 98.125
```

The index in these examples is a literal. However, the index can be an integer variable or integer expression. For example, given the following declarations,

```
real, dimension(10,10) :: tmptable  
integer :: i=2, j=3
```

would declare an array, *tmptable*, with one hundred elements.

To place a value 98.6 in the second row, fourth column,

```
tmptable(i,j+1) = 98.6
```

To access the same element, subtract 3.0 and place the result back into the same location,

```
tmptable(i,j+1) = tmptable(i,j+1) - 3.0
```

To set all elements of the **tmptable** array to 0.0, a nest loop could be used as follows:

```
do i = 1, 10
  do j = 1, 10
    tmptable(i,j) = 0.0
  end do
end do
```

In addition, the entire array can be set to 0 in the following statement,

```
tmptable = 0.0
```

Array elements can be accessed as many times as needed.

## 14.3 Example

In this example, we will write a Fortran program that will request a count, generate count (**x,y**) random points, and perform a Monte Carlo  $\pi$  estimation based on those points. All **x** and **y** values are between 0 and 1. The main routine will get the count and then use a subroutine to generate the random (**x,y**) points and a function, to perform the Monte Carlo  $\pi$  estimation. For this example, the count should be between 100 and 1,000,000.

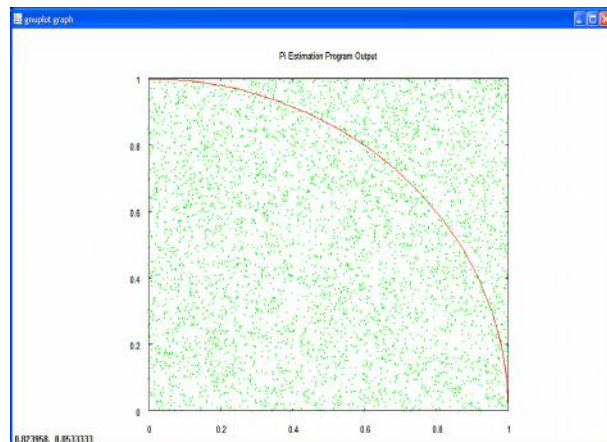
### 14.3.1 Understand the Problem

Based on the problem definition, the main routine that will read and verify the count value. If the count is not between 100 and 1,000,000, the routine will re-prompt until the correct input is provided. Once a valid count value is obtained, then the main will allocate the array and call the two subroutines. The first subroutine will generate the random (**x,y**) points and store them in an array. The second subroutine will perform the Monte Carlo  $\pi$  estimation.

Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to compute their results. Suppose a square is centered on the origin of a Cartesian plane, and the square has sides of length 2. If we inscribe a circle in the square, it will have a diameter of length 2 and a radius of length 1. If we plot points within the upper right quadrant, the ratio between the points that land within the inscribed circle and the total points will be an estimation of  $\pi$ .

$$est \pi = 4 \left( \frac{\text{samples inside circle}}{\text{total samples}} \right)$$

As more samples are taken, the estimated value of  $\pi$  should approach the actual value of  $\pi$ . The Pythagorean theorem can be used to determine the distance of the point from the origin. If this distance is less than the radius of the circle, then the point must be in the circle. Thus, if the square root of  $(x^2 + y^2) < 1.0$ , the random point is within the circle.





Finally, the figure we are discussing, a square centered on the origin with an inscribed circle is symmetric with respect to the quadrants of its Cartesian plane. This works well with the default random number generations of values between 0 and 1.

### 14.3.2 Create the Algorithm

The main routine will display an initial header, get and check the count value, and allocate the array. The program will need to ensure that the array is correctly allocated before proceeding. Then the program can generate the (x,y) points. Based on the problem definition, each point should be between 0.0 and 1.0, which is provided by default by the Fortran random number generator. Next, the program can perform the Monte Carlo *pi* estimation. This will require the already populated (x,y) points array and the count of points. Each point will be examined to determine the number of points that lie within the inscribed circle. The Pythagorean theorem allows us to determine the distance of the point from the origin (0.0,0.0). Thus, for each point, we will calculate the  $\sqrt{(x^2+y^2)}$  and if the distance is less than the circle radius of 1.0, it will be counted as inside the circle.

Then, the estimated value of  $\pi$  can be calculated based on the formula:

$$est \pi = 4 \left( \frac{\text{samples inside circle}}{\text{total samples}} \right)$$

When completed, the program will display the final results. The basic algorithm is as follows:

```
! declare variables
! display initial header
! prompt for and obtain count value
!   loop
!       prompt for count value
!       read count value
!       if count is correct, exit loop
!       display error message
!   end loop
! allocate two dimension array
! generate points
!   loop count times
!       generate x and y values
!       place (x,y) values in array at appropriate index
!   end loop
! set count of samples inside circle = 0
!   loop count times
!       if [ sqrt (x(i)**2 + y(i)**2) < 1.0 ]
!           increment count of samples inside circle
!       end loop
! display results
```

For convenience, the steps are written as program comments.

### 14.3.3 Implement the Program

The following code results from the implementation of the preceding algorithm.

```

program piestimation

! declare variables
implicit none
integer :: count, alstat, i, incount
real :: x, y, pi_est, pt
real, allocatable, dimension(:, :) :: points

! display initial header
write (*, '(a)') "Program Example – PI estimation."

! prompt for and obtain count value
do
    ! prompt for count value
    write (*, '(a)', advance="no")
        "Enter Count (100 - 1,000,000): "

    ! read count value
    read (*, *) count

    ! if count is correct, exit loop
    if ( count >= 100 .and. count <= 1000000 ) exit

    ! display error message
    write (*, '(a,a,/a)') "Error, count must be ",
        "between 100 and 1,000,000.",
        "Please re-enter."
end do

! allocate two dimension array
allocate (points(count,2), stat=alstat)
if (alstat /= 0 ) then
    write (*, '(a,a,/a)') "Error, unable to",
        " allocate memory.", "Program terminated."
    stop
end if

! generate_points
call random_seed()

! loop count times
do i = 1, count

    ! generate x and y values
    call random_number(x)
    call random_number(y)

```

```

        ! place (x,y) values in array
        pts(i,1) = x
        pts(i,2) = y

    end do

! perform monte carlo estimation
! set count of samples inside circle = 0
incount = 0

! loop count times
do i = 1, count

! if [ sqrt (x(i)2 + y(i)2) < 1.0 ]
!     increment count of samples inside circle

        pt = pts(i,1)**2 + pts(i,2)**2
        if (sqrt(pt) < 1.0) incount = incount + 1

    end do

    pi_est = 4.0 * real(incount) / real(count)

! display results
    write (*,'(a, f8.2)') "Count of points: ", count
    write (*,'(a, f8.2)') "Estimated PI value: ", pi_est

end program piestimation

```

The spacing and indentation are not required, but help to make the program more readable.

### 14.3.4 *Test/Debug the Program*

For this problem, the testing would involve executing the program using a series of different count values and ensure that the  $\pi$  estimate is reasonable and improves with higher count values.

## 14.4 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 14.4.1 *Quiz Questions*

Below are some quiz questions.

- 1) Can a multidimensional array simultaneously hold both integer and real values (yes/no)?
- 2) Multiple choice: what is the order of the indexes?
  - a) (row, column)

- b) (column, row)
- c) (row, row)
- d) (column, column)
- e) user-selectable

*Note*, there is only one correct answer.

3) Given the following code:

```
real, dimension(5,3) :: mdarr
integer :: i, j

do i = 1, 5
    do j = 1, 3
        mdarr(i,j) = real(i+j)
    end do
end do
```

- a) How many values, total, can be stored in the **mdarr** array?
  - b) Show the contents of every cell in the **mdarr**.
  - c) What does **mdarr(2,1)** contain?
  - d) What does **mdarr(1,3)** contain?
  - e) What does **mdarr(4,3)** contain?
- 4) How can an unsuccessful multidimensional dynamic allocation be detected?

### 14.4.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the  $\pi$  estimation program, compile, and execute the program. Test the program on a series of different point count values. Demonstrate that larger point values provide a better estimation.
- 2) Update the  $\pi$  estimation program to ensure that a valid count value is obtained within three tries. If there are more than three errors, the program should display an error message and terminate.
- 3) Update the  $\pi$  estimation program to display the estimated  $\pi$  value 10 times. In order to perform this, the count value can be divided by 10 and the current estimated  $\pi$  value displayed.
- 4) Write a program to statically declare a 100x100 two-dimensional array of real values. The program should populate the array with random numbers between 0 (inclusive) and 1 (exclusive). Refer to Appendix C for information regarding generating random numbers. The program should scan the array to find and display the maximum value and the location of that value (i.e., the row and column where the value was found).

- 5) Update the find maximum program (from the previous question) to declare the array dynamically and allow the user to enter the row and column dimensions and ensure that each is between 10 and 1000. Once entered, the program should allocate the array and find and display the maximum and minimum values and their locations (row and column).
- 6) Write a Fortran program to construct an odd-order Magic Square<sup>12</sup>. The algorithm for constructing a  $N \times N$  **odd** ordered Magic Square is as follows:
  - First, place a **1** in the middle of the top row.
  - After placing an integer, **k**, move up one row and one column to the right to place the next integer, **k+1**, unless the following occurs:
    - If a move takes you above the top row in the  $j^{\text{th}}$  column, move to the bottom of the  $j^{\text{th}}$  column and place the integer there.
    - If a move takes you outside to the right of the square in the  $i^{\text{th}}$  row, place the integer in the  $i^{\text{th}}$  row at the left side.
    - If a move takes you to an already filled square or if you move out of the square at the upper right hand corner, place **k+1** immediately below **k**.

Test the program and compare the results to the Wikipedia example.

<sup>12</sup> For more information, refer to: [http://en.wikipedia.org/wiki/Magic\\_square](http://en.wikipedia.org/wiki/Magic_square)



## 15 Subprograms

---

Until now, all of the programs have essentially been single, fairly small programs. However, as we scale up into larger programs, this methodology will become more difficult. When developing larger programs, it becomes necessary to break larger programs up into multiple, smaller more manageable pieces. Then, during program development, it is possible to focus on each subsection or piece individually and then combine the results into a final complete program. And, for very large projects, multiple people may work on different parts of the program simultaneously.

Some of the key advantages of developing a program using functions and/or subroutines include:

- Structured development of programs
- Reuse of subprograms
- Isolation of subprograms

Fortran subprograms are the mechanism to break a large program into multiple smaller parts. This allows for a more comprehensive program design.

### 15.1 Subprogram Types

There are two types of Fortran subprograms: *functions*, and *subroutines*, each of which is explained in the following sections.

### 15.2 Program Layout

The functions and subroutines can be defined as either internal or external. Internal functions and subroutines are defined within the program statement (i.e., before the “end program <name>” statement). The basic layout for both internal and external subprograms are as follows:

```
program <name>

    <declarations>
    <program statements>

contains

    <internal functions or subroutines>

end program <name>

    <external functions or subroutines>
```

Where a combination of both or either internal or external routines is allowed.

### 15.2.1 *Internal Routines*

Internal routines require the keyword “contains” to separate them from the program code. Primarily, internal routines will be used in this text for simplicity. There is no limit to the number of internal routines. However, if too many routines are included the file will become large and such large files can be difficult to work with.

### 15.2.2 *External Routines*

External functions are defined outside the program statement (i.e., after the “end program <name>” statement) or in another file. For larger programs, external routines would be used extensively. However, additional set-up statements, including an *external declaration* and an *interface block*, are required. The definition and use of external routines are not addressed in this chapter.

## 15.3 Arguments

When writing and using Fortran subprograms, it is typically necessary to provide data to a subprogram and/or to obtain results back from the functions or subroutines. This information, in the form of variables, is referred to as an argument or arguments. The argument or arguments in the calling routine are referred to as *actual arguments*, and the argument or arguments in the function or subroutine are referred to as *formal arguments*. The formal arguments take on the values that are passed from the calling routine.

The only way to transfer values in to or out of a subroutine is through the arguments. A function typically passes values in through the arguments with a single return value (via the function name). All other variables are independent and isolated.

### 15.3.1 *Argument Intent*

Subprograms often return values by altering or updating some of the arguments. When passing a variable, the information (value or values) can be passed into the function or subroutine. This is referred to as “intent(in)”. If the variable is to be set by the function or subroutine, that is referred to as “intent(out)”. If the variable contains a value or multiple values (i.e., an array) that are to be passed into the function or subroutine and altered in some way by the function or subroutine and returned back to the calling routine, that is referred to as “intent(inout)”.

## 15.4 Variable Scope

The variable *scope* refers to where a given variable can be accessed. Scope rules tell us if an entity (i.e., variable, parameter, and/or function) is visible or accessible at certain places. Places where an entity can be accessed, or visible, is referred as the *scope* of that entity. The variables defined in a subprogram are generally not visible to the calling routine. Thus, a variable *x* in the calling routine is different than a variable *x* in the subprogram.

## 15.5 Using Functions and Subroutines

Before a function or subroutine can be used, it must be defined or written. Once defined, the function or subroutine can be used or *called*. A function or subroutine is called by using its name as we have

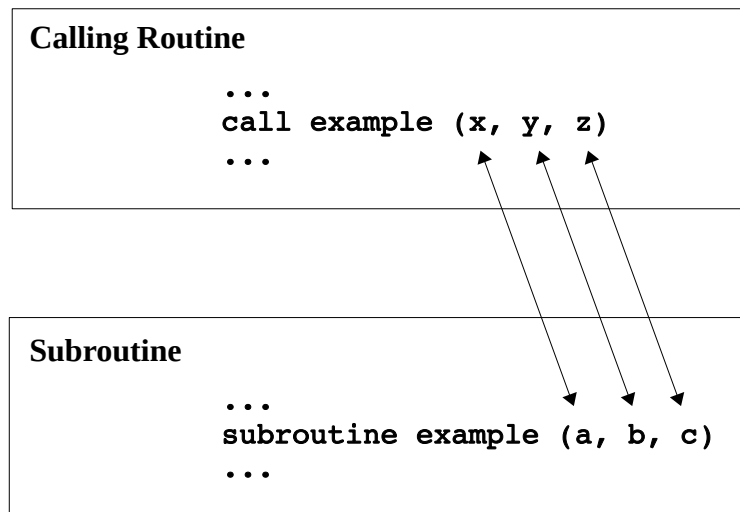


done with the intrinsic functions. When a program uses a subroutine, it is called with a *call statement*. When a program uses a function, it is used by name and returns a result which must be assigned somewhere appropriate (e.g., like a variable).

### 15.5.1 Argument Passing

When using functions or subroutines, information (values, variables, etc.) is typically passed to or from the routines. Argument association is a way of passing values from actual arguments to formal arguments. If an actual argument is an expression, it is evaluated and passed to the corresponding formal argument. If an actual argument is a variable or constant, its value is passed to the corresponding formal argument. There must be a one-to-one correspondence between the actual argument (calling routine) and the formal argument (function/subroutine).

The arguments in the call are matched up to the arguments in the function/subroutine by **position**. Each of the arguments is matched by its corresponding position. The names of the variables do not need to match, however, the data types must match. For example, given the following subroutine call and subroutine,



each variable is matched by its corresponding position.

Other variables in either the calling routine (top) or the subroutine (bottom) are isolated from each other. As such, the same variable name can be re-used in both the calling routine and the subroutine (and refer to different values).

## 15.6 Functions

A function is a special type of Fortran subprogram that is expected to return a single result or answer. A function will typically accept some kind of input information and based on that information, return a result. The two types of Fortran functions are described in the following sections.

### 15.6.1 Intrinsic Functions

As described previously, an *intrinsic function* is a built-in function that is already available. Some of the intrinsic functions already described include *sin()*, *cos()*, *tan()*, *real()*, *int()*, and *nint()*. A more comprehensive list is contained in Appendix D.

### 15.6.2 User-Defined Functions

A user-defined function is a function that is written by the user for specific or specialized requirements. The general form of a user-defined function is as follows:

```

<type> function <name> ( <arguments> )
<declarations>

    <body of function>

    <name> = expression
    return
end function <name>

```

The <type> is one of the Fortran data types: real, integer, logical, character, or complex. It is possible to place the type declaration on a separate line from the function statement.

The information, in the form of arguments, is passed from the calling routine to the function. Each of the passed arguments must be declared and the declaration must include the type and the intent. The arguments in the calling routine and the function must match and are matched up by position.

An example function to convert a Fahrenheit temperature to Celsius temperature would be as follows:

```

real function fahr_to_celsius(ftemp)
real, intent(in) :: ftemp

    fahr_to_celsius = (ftemp - 32.0) / 1.8

    return
end function fahr_to_celsius

```

Which, given a Fahrenheit temperature, will return the Celsius temperature. The single input argument, *ftemp*, is declared to be a real value and “intent(in)”, which means that the value is expected to be coming into the function and cannot be changed. The final value is returned to the calling routine by assigning a value to the function name, *fahr\_to\_celsius*, in this example.

#### 15.6.2.1 Side Effects

A *side-effect* is when a function changes one or more of its input arguments. Since the arguments can be declared as “intent(out)” or “intent(inout)”, the function could change the arguments. In general, this is considered poor practice and should be avoided. None of the examples in this text will include or utilize side-effects.

## 15.7 Subroutines

A subroutine is a Fortran subprogram that can accept some kind of input information and based on that information, return a result or series of results.

The general form of a subroutine is as follows:

```
subroutine <name> ( <arguments> )
<declarations>

    <body of subroutine>

    return
end subroutine <name>
```

The information, in the form of arguments, is passed from the calling routine to the subroutine. Each of the passed arguments must be declared and the declaration must include the type and the intent. The arguments in the calling routine and the subroutine must match and are matched up by position.

For example, given the following simple program to find the sum and average of three numbers.

```
program subExample

implicit none
real :: x1=4.0, y1=5.0, z1=6.0, sum1, ave1
real :: x2=4.0, y2=5.0, z2=6.0, sum2, ave2

    call sumAve(x1, y1, z1, sum1, ave1)
    write (*,'(a,f5.1,3x,a,f5.1)') "Sum=", sum1,           &
                                   "Average=", ave1
    call sumAve(x2, y2, z2, sum2, ave2)
    write (*,'(a,f5.1,3x,a,f5.1)') "Sum=", sum2,           &
                                   "Average=", ave2

contains

subroutine sumAve (a, b, c, sm, av)
real, intent(in) :: a, b, c
real, intent(out) :: sm, av

    sm = a + b + c
    av = sm / 3.0

    return
end subroutine sumAve

end program subExample
```

The arguments in the first call (*x1*, *y1*, *z1*, *sum1*, and *ave1*) are matched up to the arguments in the subroutine (*a*, *b*, *c*, *sm*, and *av*) by *position*. That is, the *x1* from the call is matched with the *a* in the subroutine. The arguments in the second call (*x2*, *y2*, *z2*, *sum2*, and *ave2*) are again matched up to the arguments in the subroutine (*a*, *b*, *c*, *sm*, and *av*) by *position*. While the names of the variables do not

need to match, the data types must match. Variables declared in a function or subroutine are not the same as variables in the calling routine. This is true, even if they are the same name!

## 15.8 Example

In this example, we will write a Fortran program to simulate the dice game of Twenty-Six<sup>13</sup>, which is a single player betting game with 10 dice. The main program will determine how many games to play, track the count of games won and lost, and display some win/loss statistics. A subroutine will be used to play the Twenty-Six game. The subroutine will be called as many times as requested.

The subroutine, *twenty\_six()*, will play the dice game Twenty-Six. To play the game, the player rolls the dice (1 to 6) and this initial roll is used as the “point” number. Then the player throws the ten dice 13 times. The score is the number of times that the point number is thrown. A random number between 1 and 6 will be used for each dice roll.

The routine will determine the payout based on the point count using the following table:

Point Count	Payout
10 or less	10
13	5
26	4
27	5
28	6
29	8
30	10
Other	0

The subroutine should display the dice (all 10 dice for each of 13 rolls), point count, game result, and payout. For example, if the point was 6, the subroutine might display the following:

```

Point: 6
Roll: 1 Dice: 4 6 5 3 3 1 1 3 3 2
Roll: 2 Dice: 1 6 3 3 4 1 4 4 2 6
Roll: 3 Dice: 3 2 6 4 5 3 2 1 5 4
Roll: 4 Dice: 5 6 4 1 4 6 6 2 4 4
Roll: 5 Dice: 4 6 6 4 5 3 6 1 5 5
Roll: 6 Dice: 3 1 4 5 6 5 3 3 3 4
Roll: 7 Dice: 6 6 5 6 1 5 5 6 5 5
Roll: 8 Dice: 4 1 3 4 1 4 4 6 2 5
Roll: 9 Dice: 4 4 2 1 1 4 3 1 5 4
Roll: 10 Dice: 5 6 1 2 4 1 1 2 1 1
Roll: 11 Dice: 2 3 2 4 1 3 3 6 5 1
Roll: 12 Dice: 1 1 6 5 4 5 1 6 6 5
Roll: 13 Dice: 6 4 4 5 3 3 5 3 3 5
Point Count: 22

```

13 For more information, see: <http://www.dice-play.com/Games/TwentySix.htm>

**Game Result: LOSS      Payout = 0**

For this example, the main will track the games won and lost.

### 15.8.1 *Understand the Problem*

The program will display an initial header and get the number of games to play. Specifically, we will need to prompt for the count of games and verify that the count is between 2 and 1,000,000 (arbitrarily chosen). Then, the program will call the *twenty\_six()* subroutine count times. After each game, the main will update the count of games won. The main will also track the payout and bank value status, which is initialized to 100 (chosen arbitrarily) and updated after each game is played.

An example main is provided as follows:

```

program diceGame

!-----
! Fortran program to simulate a dice game of Twenty-Six
!   The main program:
!       displays appropriate headers
!       obtains and checks number of games to play
!       loops to play 'count' number of games times

implicit none
integer, parameter :: initial_bank=100
integer :: num_games, games_won=0, games_lost=0
integer :: i, payout, bank
integer, dimension(13,10) :: dice
real :: win_pct

write (*, '(/a/a/)')                                &
"-----",                                           &
"Dice Game ""Twenty-Six"" Simulator."
do
write (*, '(2x, a )', advance = "no")                &
"Enter number games to simulate: "
read (*,*) num_games
if (num_games >= 2 .and. num_games <= 1000000) exit
write (*, '(2x, a)') "Error, number of ",            &
"games must be between 1 and 1000000."
write (*, '(2x, a)') "Please re-enter."
end do

bank = initial_bank
call random_seed()

do i = 1, num_games
bank = bank - 1
call twentySix (payout)
if (payout > 0) then

```

```

        games_won = games_won + 1
    else
        games_lost = games_lost + 1
    end if
    bank = bank + payout
end do

win_pct = ( real(games_won) / real(num_games) ) * 100.00

write (*,'(/a,/a/,3(2x,a,i9/),2(2x,a,i8/),2x,a,f4.1,a)') &
    "-----" &
    "Games Statistics:", &
    "Game Count:    ", num_games, &
    "Games Won:     ", games_won, &
    "Games Lost:    ", games_lost, &
    "Initial Bank:  ", initial_bank, &
    "Final Bank:    ", bank, &
    "Win Percentage: ", win_pct, "%"

contains

! *****
!     subroutine(s) goes here...
! *****

end program diceGame

```

Refer to Appendix C for additional information regarding the random number generation and initialization of the built-in random number generator.

### 15.8.2 Create the Algorithm

Since the main is provided, the algorithm will focus on the twenty-six game. Since the built-in random number generator provides random numbers between 0.0 and 1.0, they will need to be scaled and converted to an integer between 1 and 6 (for a dice). The initial point value must first be established followed by a loop to throw the ten dice 13 times in accordance with the game rules. The results will be stored in a two-dimensional array. While not strictly required, it does provide an additional example of how to use a two-dimensional array. Finally, the payout will be determined based on the game rules.

```

! Randomly select a number from 1 to 6 as the "point" number
! Throw ten dice 13 times
!     results go into dice(13,10) array
! Score is the number of times that the point number
!     is thrown
! determine payout

```

For convenience, the steps are written as program comments.

### 15.8.3 Implement the Program

Based on the algorithm, the below program could be created.

```
! *****
! Subroutine to simulate twenty-six game.
!   Randomly select a number from 1 to 6 as the "point" number
!   Throw ten dice 13 times
!       results go into dice(13,10) array
!   Score is the number of times that the point number is
thrown

subroutine twentySix (payout)
implicit none
integer, dimension(13,10) :: dice
integer, intent(out) :: payout
integer :: point, pnt_cnt
real :: x
integer :: i, j

!   determine point
call random_number(x)
point = int(x*6.0) + 1

!   roll dice
pnt_cnt = 0

do i = 1, 13
    do j = 1, 10
        call random_number(x)
        dice(i,j) = int(x*6.0) + 1
        if (dice(i,j) == point) pnt_cnt = pnt_cnt + 1
    end do
end do

! determine payout
select case (pnt_cnt)
case (:10)
    payout = 10
case (13,27)
    payout = 5
case (26)
    payout = 4
case (28)
    payout = 6
case (29)
    payout = 8
case (30)
    payout = 10
case default
    payout = 0
end select
```

```

end select

write (*,'(/,5x,a/,5x,a,i2/,5x,a,i2)')           &
      "-----",                               &
      "Point: ", point

do i = 1, 13
  write (*,'(8x, a, i2, 2x, a, 10(2x, i1),/)',      &
        advance="no") "Roll: ", i, "Dice: ",      &
        (dice(i,j), j=1,10)
end do

write (*,'(/,5x,a,i2)') "Point Count: ", pnt_cnt

if (payout > 0) then
  write (*,'(5x,a,i2)')                               &
    "Game Result: WIN      Payout = ", payout
else
  write (*,'(5x,a,i2)')                               &
    "Game Result: LOSS      Payout = ", payout
end if

write (*,'(5x,a,i6)') "Bank:", bank

return
end subroutine twentySix

```

The spacing and indentation are not required, but help to make the program more readable.

### 15.8.4 Test/Debug the Program

For this problem, the testing would involve executing the program using a file with a set of numbers where the correct results are either known ahead of time or can be calculated by hand in order to verify that the results are accurate.

## 15.9 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 15.9.1 Quiz Questions

Below are some quiz questions.

- 1) What are the two types of Fortran subprograms?
- 2) How many values does a user-defined function typically return?
- 3) In the function call, **ans = power(x, y)**, what are **x** and **y** are called?
- 4) In the function heading, **integer function power(a, b)**, what are **a** and **b** are called?



- 5) In the function, **integer function power(a, b)**, what is the *type* of the value returned?
- 6) Is it possible to pass integer arguments to a real function (yes/no)?
- 7) The subprogram section (where functions and subprograms are defined) is marked by what keyword?
- 8) What is the output of the following function:

```
integer function power(a, b)
integer, intent(in) :: a, b

    power = a ** b
    return
end function power
```

- a) with the input of **a** = 2 and **b** = 3?
  - b) with the input of **a** = 3 and **b** = 2?
- 9) Given the following program?

```
program quiz
implicit none
real :: temp=80.0, temp1=50.0, temp2

    write (*, '(f5.2, 2x, f5.2)') temp, temp1
    temp2 = fahrToCelsius(temp1)
    write (*, '(f5.2, 2x, f5.2)') temp, temp2

contains

real function fahrToCelsius(temp)
real, intent(in) :: temp

    fahrToCelsius = (temp - 32.0) / 1.8
    return
end function fahrToCelsius

end program quiz
```

- a) What is the name of the function?
  - b) Is the above program correct (yes/no)?
  - c) Does the variable **temp** in the main and the variable **temp** in the function refer to the same value?
  - d) What is the output?
- 10) What is meant by the term *variable scope*?

- 11) What is the correct *intent* for the following situations:
- A variable passed to a function that will not be changed in the function.
  - A variable that will be set in the subroutine and returned. No value is passed in.
  - A variable that will be passed into a subroutine, updated, and returned back to the calling routine.
- 12) What is meant by the term *side-effect*?

### 15.9.2 Suggested Projects

Below are some suggested projects.

- Type in the dice game program example, compile, and execute the program. Test the program by playing it for a series of rounds. Ensure the scoring is correct.
- Write a main program and an integer Fortran function, ***gSeries()***, to compute the following geometric series:

$$g = \sum_{n=0}^{n-1} x^n = 1 + x + x^2 + x^3 + \dots + x^{(n-1)}$$

The arguments for the call, in order, are as follows: ***n*** (integer value). The function should return an integer result (of the formula based on the ***n*** value). The main should call the function with several different values.

- Write a main program and a real function, ***harmonicMean()***, to compute the harmonic mean of a series of real numbers. The real numbers are passed to the function in an array along with the count.

$$\text{harmonic mean} = \frac{N}{\left( \frac{1}{x_1} + \frac{1}{x_2} + \dots + \frac{1}{x_N} \right)}$$

The arguments for the call, in order, are as follows; array of numbers (with ***count*** real values), count (integer). The function should return a real result (of the formula). The main should call the function with several different values.

- Write a main program and a subroutine, ***CircleStats()***, that, given an array containing a series of circle diameter's (real values), will compute the area of each circle in a series of circles and store them into a different array (real values). The subroutine should also compute the real average of the circle areas. The arguments for the call, in order, are as follows; circle diameter's array (***count*** real values), circle areas array (***count*** real values), count (integer), areas average (real). The main program should declare the array and initialize the array with a series of values for circle areas. The program results should be verified with a calculator.

- 5) Write a main program and a subroutine, **ReadCoord()**, to read an  $(x, y, z)$  coordinate from the user. The subroutine must prompt for and read  $(x, y, z)$  and ensure that the  $x$ ,  $y$ , and  $z$  values are between 0 and 100 (inclusive). The values may be prompted for and read together, but the prompt should leave the cursor on the same line. The subroutine should re-prompt for all three if the input data is not correct. If the user provides valid data, the  $(x, y, z)$  values should be returned with a logical for valid data set to *true*. If the user does not provide valid data entry after **three** tries, the subroutine should display an error message and set the logical for valid data to *false*. The arguments for the call, in order, are as follows;  $x$  value (integer),  $y$  value (integer),  $z$  value (integer), and valid data flag (logical value). The main program should call the subroutine three times and display the results for each call.
  
- 6) Write a main program and a subroutine, **Stats()**, that, given an array containing a series of numbers (real values), will find and display the following real values; minimum, median, maximum, sum, and average. The display must use a formatted write(). The real values will not exceed 100.0 and should display three digits decimal values (i.e., **nnn.xxx**). The arguments for the call, in order, are as follows; array of numbers (*count* real values), count (integer). The main program should populate the array with random numbers and call the subroutine.



## 16 Derived Data Types

---

A derived data type is a user-defined combination of the intrinsic data types. The derived data types are a convenient way to combine or group variables about a particular item.

For example, a 'student' might include a name, identification number, final score, and grade. Each of these pieces of information can be represented with individual variables (as outlined in previous section) as follows:

```
character(50) :: name
integer :: id
real :: score
character(2) :: grade
```

However, for multiple students, multiple sets of variables would be required. This can become cumbersome and confusing.

By using a derived data type, these separate pieces of information can be more easily grouped together. The details on defining, declaring and using derived data types are provided in the following sections.

### 16.1 Definition

Before a derived data type can be used, it must be defined. The definition will establish which pieces of information will be grouped together. Each piece of information included in the definition is referred to as a component.

```
type type_name
    <component definitions>
end type type_name
```

For example, to declare the student type described previously, the following declaration would be appropriate:

```
type student
    character(50) :: name
    integer :: id
    real :: score
    character(2) :: grade
end type student
```

The indentation is not required, but does make the definition easier to read. The fields (name, id, score, grade) are called components. These components together make up the information for a 'student'.

The type definition is required only once at the beginning of the program. Once defined, the type definition cannot be changed. More specifically, additional components cannot be added unless the definition is updated and the program is recompiled.

This definition will establish a template as follows:

student	name	
	id	
	score	
	grade	

Once defined, the template can be used to declare variables. Each variable declared with this definition will be created based on the definition which includes these four components.

## 16.2 Declaration

Once a derived data type is defined, variables using that definition can be declared. The general format for a declaration is as follows:

```
type (<type_name>) :: <variable_name(s)>
```

For example, to declare two students, the following declaration could be used:

```
type (student) :: student1, student2
```

This declaration will declare two variables, *student1* and *student2*, each with the set of components defined in the type definition. The definition can be thought of as the cookie cutter and the declaration is the cookie. Only after a variable has been declared, can values be set for that variable.

## 16.3 Accessing Components

Once some variables using the derived data type have been declared, the individual components can be accessed. To access a component of a derived data type, specify the variable name followed by a “%” (percent sign) followed by the component name. The general format is:

```
<variable_name>%<component_name>
```

For example, to set all components for the student *student1*, the following

```
student1%name = "Joseph"  
student1%id = 1234  
student1%score = 99.99  
student1%grade = "A"
```

Each component for *student1* is set individually. Not every component must be set. Of course, as with other variables, any component that has not been set cannot be used.

This previous declaration and these assignments will establish a variable as follows:

student	name	Joseph
	id	1234
	score	99.99
	grade	A

It is possible to assign all components to another variable of the same derived data type. For example, to set **student2** to be the same as **student1**, an assignment is used as follows:

```
student2 = student1
```

This will copy all components from the variable **student1** into the variable **student2** (since both **student1** and **student2** are of the same derived data type).

## 16.4 Example One

In this example, we will write a simple program to read two times from the user, time one and time two, and calculate the sum of the two times. For this example, the time will consist of hour, minutes, seconds in 24-hour format. For this exercise, the hours may exceed 23 when the times are summed. The program should declare the appropriate variables using a derived data type, use a subroutine to read a time (which should be called twice), and another subroutine to calculate the sum of the times. The subroutine to read the times must perform appropriate error checking. The main should display both the times and the final time sum.

### 16.4.1 Understand the Problem

The main is expected to define the appropriate derived data type for time, declare some variables of that type and call the subroutines. The first subroutine will read a time from the user which will consist of hour, minutes, and seconds in 24-hour format. This subroutine will be called twice. The second subroutine will add the times together and provide a result.

The first subroutine to read a time from the user is expected to perform error checking on the data entered by the user. Specifically, this requires that hours range from 0 to 23, minutes range from 0 to 59, and seconds range from 0 to 59. Values outside these ranges, 60 seconds for example, are not valid. For this simple example, we will re-prompt for incorrect data entry (until correct data is provided).

The second subroutine will add the two times and must ensure that the correct ranges for seconds and minutes are maintained. When adding the two times, it is possible to add the seconds, minutes, and hours. However, if the sum of the two seconds values exceeds 60, the seconds must be adjusted and the minutes must be updated accordingly (add one extra minute). This applies to the minutes as well. However, when added in this exercise, the sum of the final time may exceed 23 hours.

For example, given time one as 14 hours, 47 minutes and 22 seconds (i.e., 14:47:22) and time two as 18 hours, 22 minutes, and 50 seconds, (i.e., 18:22:50), the total time would be 33 hours, 10 minutes and 12 seconds (i.e., 33:10:12).

### 16.4.2 Create the Algorithm

For this example, there are three parts; the main, the read time subroutine, and the time summation subroutine. The basic steps for the main include:

```
! define derived data type for time
!   must include → hours, minutes, seconds
! declare variables, including → time1, time2, and timesum
! display initial header
! call subroutine to read time1
! call subroutine to read time2
! call subroutine to add times
! display results
```

The basic steps for the read time subroutine include:

```
! subroutine header and appropriate declarations
! loop
!   prompt for time
!   read time (hours, minutes, seconds)
!   check time entered
!   → if [ hours(0-23), minutes(0-59), seconds (0-59) ] exit
!   display error message
! end loop
```

The basic steps for the time summation subroutine include:

```
! subroutine header and appropriate declarations
! add the seconds
! add the minutes
! add the hours
! if seconds > 59, then
!   subtract 60 from seconds
!   add 1 to minutes
! if minutes > 59, then
!   subtract 60 from minutes
!   add 1 to hours
```

For convenience, the steps are written as program comments.

### 16.4.3 Implement the Program

Based on the algorithm, the below program could be created.

```
program timeSummation

! define derived data type for time (hours, minutes, seconds)
implicit none

type time
```



```

        integer :: hours, minutes, seconds
end type time

! declare variables
!   includes → time1, time2, and timesum
type(time) :: time1, time2, timesum

! display initial header
write (*,'(/,a,/)' ) "Time Summation Example Program."

! call subroutine to read each time
call readtime(time1)
call readtime(time2)

! call subroutine to add times
call addtimes(time1, time2, timesum)

! display results
write (*,'(/,a,i2.2,a1,i2.2,a1,i2.2)' ) "Time One: ", &
    time1%hours, ":", time1%minutes, ":", time1%seconds
write (*,'(a,i2.2,a1,i2.2,a1,i2.2)' ) "Time Two: ", &
    time2%hours, ":", time2%minutes, ":", time2%seconds
write (*,'(a,i2.2,a1,i2.2,a1,i2.2,/)' ) "Time Sum: ", &
    timesum%hours, ":", timesum%minutes, ":", &
    timesum%seconds

contains

! *****
! Subroutine to prompt for, read, and check
!   a time (hours:minutes:seconds) in 24-hour format.

subroutine readtime ( timeval )
type(time), intent(out) :: timeval

    do
!   prompt for time
        write (*,'(a)',advance="no") &
            "Enter time (hh mm ss): "

!   read time (hours, minutes, seconds)
        read (*,*) timeval%hours, timeval%minutes, &
            timeval%seconds

!   check time entered
        if ( timeval%hours >= 0 .and. &
            timeval%hours <= 23.and. &
            timeval%minutes >= 0 .and. &
            timeval%minutes <= 59 .and. &
            timeval%seconds >= 0 .and. &
            timeval%seconds <= 59 ) exit
    end do
end subroutine readtime

```

```

!      display error message
!      write (*,'(a,/,a)')
!      "Error, invalid time entered.",
!      "Please re-enter time."

      end do

      return
end subroutine readtime

! *****
! Subroutine to add two times.
! Ensures seconds and minutes are within range (0-59)
! Hours may exceed 23

! subroutine header and appropriate declarations
subroutine addtimes ( tm1, tm2, tmsum )
type(time), intent(in) :: tm1, tm2
type(time), intent(out) :: tmsum

! add the seconds, minutes, hours
tmsum%seconds = tm1%seconds + tm2%seconds
tmsum%minutes = tm1%minutes + tm2%minutes
tmsum%hours = tm1%hours + tm2%hours

! if seconds > 59, subtract 60 from seconds and add 1 to minutes
if (tmsum%seconds > 59) then
    tmsum%seconds = tmsum%seconds - 60
    tmsum%minutes = tmsum%minutes + 1
end if

! if minutes > 59, subtract 60 from minutes and add 1 to hours
if (tmsum%minutes > 59) then
    tmsum%minutes = tmsum%minutes - 60
    tmsum%hours = tmsum%hours + 1
end if

      return
end subroutine addtimes

end program timeSummation

```

If the program does not work at first, the comments can aid in determining the problem.

#### 16.4.4 Test/Debug the Program

For this problem, the testing would involve executing the program and entering a series of various time values to ensure that the results are correct. If the program does not work initially, the functionality of each subroutine should be checked. The times read from the user can be displayed to the screen to

ensure they are correct. Once the times are correct, the add times subroutine can be checked. Each of the time sums can be displayed to help determine where the error might be.

## 16.5 Arrays of Derived Data

In addition to declaring single variables based on the derived data type definition, it is possible to declare an array based on the derived data type definition. For example, to declare an array named *class* to hold 30 elements of type *student*, the following declaration can be used:

```
type(student), dimension(30) :: class
```

Each element of the array *class* will be of the type *student* and include each of the defined components (name, id, score, grade in this example). For an array of *type(student)*, the layout would be as follows:

class(1)	name	
	id	
	score	
	grade	
class(2)	name	
	id	
	score	
	grade	
class(3)	name	
	id	
	score	
	grade	
...		...

To access elements in the array, an index must be used. After the index, the desired component would be specified. For example, to set values for the third student, the following statements could be used.

```
class(3)%name = "Fred"  
class(3)%id = 4321  
class(3)%score = 75.75  
class(3)%grade = "C"
```

As with any array, the index can be an integer variable.

As with single variables, it is possible to assign all components of an array element to another array element, or another variable of the same derived data type. The following declarations and code could be used to swap the location of the fifth student and the eleventh student.

```
type student  
character(50) :: name
```

```

        integer :: id
        real :: score
        character(1) :: grade
    end type student

    type(student), dimension(30) :: class
    type(student) :: temp

    temp = class(5)
    class(5) = class(11)
    class(11) = temp

```

This code fragment will copy all components from the fifth array element (of type *student*) into a temporary variable (also of type *student*). Then, the eleventh array element can be copied into the fifth array element (thus overwriting all previous values). And, finally, the eleventh array element can be set to the original values from the fifth array element, which are held in the temporary variable.

## 16.6 Example Two

In this example, we will write a simple program to perform some processing for students. The student information will be stored in an array of derived data types. There will be no more than 50 students per class. The main will call a subroutine to read student information (name and score) and another subroutine to set the student grades. Finally, the main will call a function to calculate the class average. The main will display the average. Routines for displaying the students are left as an exercise.

### 16.6.1 Understand the Problem

The main is expected to define the appropriate derived data type for student, declare some variables of that type and call the subroutines. The first subroutine will read student information, including a name (up to 60 characters) and score from the user. Names and scores should continue to be read until a blank name is entered. The score value must be between 0.0 and 100.0 (inclusive). For this simple example, we will re-prompt for incorrect data entry (until correct data is provided). The routine must return the count of students entered. The second subroutine sets the grades based on the following standard scale.

A	B	C	D	F
A ≥ 90	80 - 89	70 - 79	60 - 69	≤ 59

When determining the final grade, the program should round up when appropriate. The main will call a function to calculate and return the average of the scores. Additionally, the main will display the final average.

### 16.6.2 Create the Algorithm

For this example main, part of the main includes a declaration, display header, call read time subroutine, and the call the time summation subroutine. The basic steps for the main include:

```

! define derived data type for student
!     must include → name, id, grade
! declare variables
!     includes → array for up to 50 students
! display initial header
! call subroutine to read student information
! call subroutine to set grades
! use function to calculate average of scores
! display average

```

The basic steps for the read student information subroutine include:

```

! subroutine header and appropriate declarations
! loop
!     prompt for student name
!     read name
!     if name is empty, exit loop
!     loop
!         prompt for student score
!         read score
!         check score entered
!         if [score is between 0.0 and 100.0, inclusive] exit
!         display error message
!     end loop
!     update count of students
!     place values in student array
! end loop

```

The basic steps for the set grades subroutine include:

```

! subroutine header and appropriate declarations
! loop
!     read score / set grade
!         ≥ 90 → A; 80 - 89 → B; 70 - 79 → C; 60 - 69 → D; ≤ 59 → F
! end loop

```

When determining the final grade, the nearest integer intrinsic function, `nint()`, can be used to perform the appropriate rounding.

The basic steps for the calculate average score function include:

```

! function header and appropriate declarations
! loop
!     sum scores
! end loop
! calculate and return average

```

For convenience, the steps are written as program comments.

### 16.6.3 *Implement the Program*

Based on the algorithm, the below program could be created.

```

program classScores
! define data type for student, includes → name, id, grade

implicit none
type student
    character(60) :: name
    real :: score
    character(1) :: grade
end type

! declare variables, including → array for up to 50 students
type(student), dimension(50) :: class
integer :: count
real :: average

! display initial header
    write (*,'(/,a,/)' ) "Student Information Example Program."

! call subroutine to read student information
    call readStudents (class, count)

! call subroutine to set grades
    call setStudentGrades (class, count)

! use function to calculate average of scores
    average = classAverage (class, count)

! display average
    write (*,'(/,a, f5.1)' ) "Final Class Average: ", average

contains

! *****
! Subroutine to read student information (name and score).
! A blank name will stop input
! The score must be between 0.0 and 100.0 (inclusive)

subroutine readStudents (class, count)
type(student), dimension(50), intent(out) :: class
integer, intent(out) :: count = 0
character(60) :: tempname
real :: tempscore

    do
! prompt for student name and read name
        write (*,'(a)',advance="no") "Enter Student Name: "
        read (*,'(a60)' ) tempname

! if name is empty, exit loop
        if ( len_trim(tempname) == 0 ) exit
    
```

```

do
!      prompt for student score and read score
      write (*,'(a)',advance="no")           &
           "Enter Student Score: "
      read (*,*) tempscore

!      check score entered
      if ( tempscore >= 0.0 .and.             &
          tempscore <= 100.0 ) exit

!      display error message
      write (*,'(a,/,a)')                   &
           "Error, invalid score.",          &
           "Please re-enter time."

      end do

!      update count of students and place in student array
      count = count + 1
      class(count)%name = tempname
      class(count)%score = tempscore

      end do

      return
end subroutine readStudents

! *****
! Subroutine to set student grades.
!      ≥ 90 → A; 80 - 89 → B; 70 - 79 → C; 60 - 69 → D; ≤ 59 → F

subroutine setStudentGrades (class, count)
type(student), dimension(50), intent(inout) :: class
integer, intent(in) :: count
integer :: i

!      check each score / set each grade
do i = 1, count
      select case ( nint(class(i)%score) )
      case (90:)
            class(i)%grade = "A"
      case (80:89)
            class(i)%grade = "B"
      case (70:79)
            class(i)%grade = "C"
      case (60:69)
            class(i)%grade = "D"
      case (:59)
            class(i)%grade = "F"
      end select
end do

```

```

        return
    end subroutine setStudentGrades

! *****
! Function to calculate average score.

real function classAverage (class, count)
type(student), dimension(50), intent(in) :: class
integer, intent(in) :: count
integer :: i
real :: sum = 0.0

!     sum scores
    do i = 1, count
        sum = sum + class(i)%score
    end do

! calculate and return average
    classaverage = sum / real(count)

    return
end function classAverage

end program classScores

```

If the program does not work at first, the comments can aid in determining the problem.

### 16.6.4 *Test/Debug the Program*

For this problem, the testing would involve executing the program and entering a series of student data values to ensure that the results are correct. If the program does not provide the correct results, each of the subroutines and the function results should be verified individually. Each can be checked by displaying the intermediate results to the screen. In this manner, the subroutine or function that is not working correctly can be quickly identified. Once identified, some additional write statements inside the subprogram can be used to help identify the specific problem. The testing and debugging process is left to the reader as an exercise.

## 16.7 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 16.7.1 *Quiz Questions*

Below are some quiz questions.

- 1) An item in a derived data type is called?
- 2) How are components of a derived data type accessed?



- 3) Define a derived data type, **circle**, to store information about a circle. Must include a circle name (max 20 characters), size (radius – a real value) and the position in 3-dimensional space (**x**, **y**, and **z** → all integer values).
- 4) Write the declaration necessary to declare two variables named **ring1** and **ring2** of type **circle** (from previous question).
- 5) Define a user-defined type, **planet**, to store information for a planet. Include a name (15 characters), radius (real value), area (real value), and volume (real value). Additionally, write the declaration necessary to declare two variables named **earth** and **mars** of type **planet**.
- 6) Define a user-defined type named **date** for processing dates consisting of a month name (10 characters), month (integer), date of month (integer), and year (integer).
- 7) Based on the previous question
  - a) Write the statements to declare a variable named **today** and set it to 12/25/2013.
  - b) Write the statements to declare a variable named **newyear** and set it to January 1, 2011.

### 16.7.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the time summation program, compile and execute the program. Test on several sets of input including the example in the explanation.
- 2) Write a Fortran program to read and display information about a set of planetary bodies. The program should read the information from a file, allow the user to select a display option, and call the appropriate routine to display that information.

The main program should call a series of subroutines as follows:

- Subroutine **readPlanets()** to prompt for a file name of the planets file, open the file (including error checking), and read the file into an array. Three errors are allowed, but if a fourth error is made, the routine should terminate the program.
- Subroutine **calcPlanetArea()** to calculate the planet area based on the diameters.
- Function **getUserOption()** to display a list of options, read the selection option.
- Subroutine **displayPlanetMinMax()** to display the minimum and maximum based on an option as follows:
  - Option (1) → Smallest and Largest Planets (based on area)
  - Option (2) → Coldest and Hottest Planets (based on average temperature)
  - Option (3) → Planets with Shortest and Longest Days (based on day length)
- Subroutine **printPlanetsSummary()** to display the planet name, distance from sun, and planet size.

The output should be formatted as appropriate. Test on several sets of input values and verify that the output is correct for the given input values.

- 3) Modify the planet program (from previous question) to sort the planets based on the radius. Test on several sets of input values and verify that the output is correct for the given input values.
- 4) Type in the time class scores program, compile and execute the program. Test on several sets of input values.
- 5) Modify the class scores program to assign grades based on the following scale:

<b>F</b>	<b>D</b>	<b>C-</b>	<b>C</b>	<b>C+</b>	<b>B-</b>	<b>B</b>	<b>B+</b>	<b>A-</b>	<b>A</b>	<b>A+</b>
<b>0-59</b>	<b>60-70</b>	<b>70-72</b>	<b>73-76</b>	<b>77-79</b>	<b>80-82</b>	<b>83-86</b>	<b>87-89</b>	<b>90-92</b>	<b>93-96</b>	<b>97-100</b>

Test on several sets of input values and verify that the output is correct for the given input.

- 6) Modify the class scores program to read the name and score file. The program should include prompting for a file, opening the file, and reading the file contents into the class array. In order to complete this exercise, create a file containing some random names and scores. Test on several sets of input values.

## 17 Modules

---

For larger programs, using a single source file for the entire program becomes more difficult. Fortunately, large programs can be split into multiple source files, each file can contain a subset of subroutines and/or functions. There must be a main or primary source file that includes the main program. The secondary file or files is referred to as a module or modules. Additionally, the modules can then be more easily used in other, different programs ensuring that the code can be easily re-used. This saves time and money by not re-writing routines. This section provides a description of the formatting requirements and an example of how to set-up the modules.

### 17.1 Module Declaration

The secondary source file or module must be formatted in a specific manner as follows:

```
module <name>
    <declarations>
contains
    <subroutine and/or function definitions>
end module <name>
```

For example, to declare a module named **stats** that includes a function to find the average of the numbers in an array, the following module declaration might be used.

```
module stats
! note, no global variables used in this module

contains

! *****
! Simple function to find average of len values in an array.

real function average(array, len)
real, intent(in), dimension(1000) :: array
integer, intent(in) :: len
integer :: i
    real :: sum = 0.0

    do i = 1, len
        sum = sum + array(i)
    end do
    average = sum / real(len)
end function average

end module stats
```

This example assumes the real array contains **len** number of values up to a maximum of 1000 values.

## 17.2 Use Statement

Once the module is defined, the routines from the module can be included by using the **use** statement. The use statement or statements must be at the beginning of the applicable source file. For example, below is a simple main that uses the previous stats module.

```

program average
use stats

implicit none
real, dimension(1000) :: arr
integer :: i, count
real :: ave

! -----
! Initialize array with some values.

    count = 0
    do i = 1, 20
        arr(i) = real(i) + 10.0
        count = count + 1
    end do

! -----
! Call function to find average and display result.

    ave = arraverage(arr, count)

    write (*, '(/, a, f10.2, /)') "Average = ", ave

end program average

```

The use statement is included before the variable declarations. Any number of use statements for defined modules may be included.

## 17.3 Updated Compilation Commands

For a large program that is split between multiple source files, the compilation process must be updated. The compilation process refers to the steps required to compile the program into a final executable file. Each module unit must be compiled independently. This allows the programmer to focus on one module, or set of routines, at a time. Further, for very large projects, multiple programmers can work on separate modules simultaneously.

The initial step is to compile each module. Assuming the module from the earlier section is named **stats.f95**, the command to compile a module is:

```
gfortran -c stats.f95
```

which will read the source file (**stats.f95**) and create two new files: an object file **stats.o** and a module file **stats.mod**. The name of the object file is based on the name of the source file. The

name of the module file is based on the module name. While they are the same name in this example, that is not a requirement.

The compile command is required for each module.

Once all the modules are compiled and the `.o` and `.mod` files are available, the main file can be compiled. This step reads the `.o` and `.mod` files for each module and builds the final executable file. For example, the command to compile the main file for the previous array average example is:

```
gfortran -o main main.f95 stats.o
```

For multiple modules, each of the `.o` files would be listed. In this example, the `stats.mod` file is read by the `gfortran` compiler. While not explicitly listed, the `.mod` files are required and used at this step.

## 17.4 Module Example Program

The following is an example program to compute the surface area and volume of a sphere. This is a fairly straightforward problem focusing more on the creation and use of a module for some routines.

### 17.4.1 Understand the Problem

This problem will be divided into two parts, the main program source file and a secondary source file containing the subroutines and functions. While this problem is not large enough to require splitting into multiple files, it is split to provide an example of how to use modules.

The formulas for the surface area and volume of a sphere are as follows:

$$\text{surfaceArea} = 4.0 * \pi * \text{radius}^2$$

$$\text{volume} = \frac{4.0 * \pi}{3.0} * \text{radius}^3$$



The value of  $\pi$  will be defined as a constant and set to 3.14159.

For this example, the main program will display some initial headers and read the radius from the user. Once the radius is read, the main program will call functions for the surface area and the volume, and a subroutine to display the results.

### 17.4.2 Create the Algorithm

Based on the problem definition, the steps for the main are:

```
! display header and read radius  
! call functions for sphere volume and surface area  
! call routine to display formatted results
```

The module will contain the functions and subroutine. The first function will compute the sphere volume. The single step is:

```
! compute the volume of a sphere with given radius.
!   sphere volume = [ (4.0 * pi) / 3.0 ] * radius^3
```

The second function will compute the sphere surface area. The single step is:

```
! compute the volume of a sphere with given radius
!   sphere volume = 4.0 * pi * radius^2
```

The subroutine will display the formatted results.

### 17.4.3 *Implement the Program*

The program is presented in two parts corresponding to the main program and the secondary module routines. While this example is not really long or complex enough to require multiple files, the program is split in order to provide an example using a separate module file.

#### 17.4.3.1 *Main Program*

Based on the algorithm, the below program could be created.

```
program sphere
  use sphereRoutines

  implicit none
  real :: radius, spVolume, spSurfaceArea

! -----
! Display header and read radius

  write (*,'(a/)' ) "Sphere Example Program"
  write (*,'(a)', advance="no") "Radius: "

  read (*,*) radius

! -----
! Call functions for sphere volume and surface area

  spVolume = sphereVolume(radius)
  spSurfaceArea = sphereSurfaceArea(radius)

! -----
! Call routine to display formatted results.

  call displayResults(radius, spVolume, spSurfaceArea)

end program sphere
```

The name of the module, **sphereRoutines** in this example, must be the name of the secondary source file.

### 17.4.3.2 Module Routines

Based on the algorithms for the two functions and subroutine, the below module program could be developed. In this example, the declaration for  $\pi$  is defined as a global variable. This shares the variable between all the subroutines and functions in the module. Use of global variables is typically limited. This provides an example of an appropriate use of a global variable.

```
! Example secondary source file.

module sphereRoutines
implicit none                      ! needed in every module

! Global declarations, if any, go here.

real, parameter :: pi = 3.14159

! *****
! Subroutines and functions are included after
! the 'contains'.

contains

! *****
! Compute the volume of a sphere with given radius.
!   sphere volume = [ (4.0 * pi) / 3.0 ] * radius^3

real function sphereVolume (radius)
real, intent(in) :: radius

    sphereVolume = ( ( 4.0 * pi ) / 3.0 ) * radius ** 3

    return
end function sphereVolume

! *****
! Compute the volume of a sphere with given radius.
!   sphere volume = 4.0 * pi * radius^2

real function sphereSurfaceArea (radius)
real, intent(in) :: radius

    sphereSurfaceArea = 4.0 * pi * radius ** 2

    return
end function sphereSurfaceArea

! *****
! Simple routine to display results.

subroutine displayResults(rad, vol, area)
real, intent(in) :: rad, vol, area
```

```

        write (*,'(/, a)') "-----"
        write (*, '(a)' ) "Results:"
        write (*,'(3x, a, f10.2)') "Sphere Radius = ", rad
        write (*,'(3x, a, f10.2)') "Sphere Volume = ", vol
        write (*,'(3x, a, f10.2, /)')          &
            "Sphere Surface Area = ", area

        return
    end subroutine displayResults

! *****

end module sphereRoutines

```

In a more complex program, multiple module files might be used. The grouping should be based on the logical relationship of the routines. A more complicated program would require a more comprehensive design effort.

#### 17.4.4 *Compile the Program*

The commands to compile the module file and then the main file are as follows:

```

gfortran -c sphereroutines.f95
gfortran -o modmain modmain.f95 sphereroutine.o

```

The first will create files **sphereroutines.o** and **sphereroutines.mod**. The second will read the files **modmain.f95** and **sphereroutines.o** then create the executable **modmain.exe** file.

#### 17.4.5 *Test/Debug the Program*

For this problem, testing would involve executing the program and entering a series of radius values and to ensure that the results are correct. If the program does not provide the correct results, each of the functions and the subroutines could be verified individually. Each can be checked by displaying the intermediate results to the screen. In this manner, the subroutine or function that is not working correctly can be quickly identified. Once identified, some additional write statements inside the subprogram can be used to help identify the specific problem.

### 17.5 Exercises

Below are some quiz questions and project suggestions based on this chapter.

#### 17.5.1 *Quiz Questions*

Below are some quiz questions.

- 1) What is the primary purpose of using a module?
- 2) In the main program, what statement is used to include the modules?



- 3) In the secondary source file, what statements is used to define and name the module?
- 4) How many main programs are allowed?
- 5) How many modules are allowed?
- 6) Is the **contains** key word needed in a module file (yes or no)?

### **17.5.2      *Suggested Projects***

Below are some suggested projects.

- 1) Type in the array average main program and the array average module, compile and execute the program. Test on several sets of input including the example in the explanation.
- 2) Type in the sphere volume and surface area main program and the sphere volume and surface area module, compile and execute the program. Test on several sets of input including the example in the explanation.
- 3) Update the planets program from the previous chapter, problem #2, and break the program into a main file and a module file for the functions and subroutines.



## 18 Recursion

---

The Google search result for recursion, shows *Recursion, did you mean recursion?*

Recursion is a powerful general-purpose programming technique and is used for some important applications including search and sorting methods. Recursion is the idea that a function may call itself (which is the basis for the joke).

Recursion can be very confusing in its simplicity and power. The examples in this section will not be enough in themselves for the reader to obtain recursive enlightenment. The goal of this section is to provide an introduction to the concept on recursion. The simple examples here, which are used to introduce recursion, are meant to help demonstrate the form and structure for recursion. More complex examples (than will be discussed here) should be studied and implemented in order to ensure a complete appreciation for the power of recursion.

The calling process previously described supports recursion without any changes.

A recursive routine must have a recursive definition that includes:

1. base case, or cases, that provide a simple result (that defines when the recursion should stop).
2. rule, or set of rules, that reduce toward the base case.

This recursive definition is referred to as a recursive relation.

### 18.1 Recursive Subroutines

A recursive subroutine declaration must be preceded by the keyword **recursive**. For example:

```
recursive subroutine <name> ( <arguments> )
<declarations>

    <body of subroutine>

    return
end subroutine <name>
```

Based on this declaration the subroutine can call itself. The routine must ensure that it eventually stops calling itself. Arguments are passed in the standard way.

The calling routine does not need any special declarations in order to call a recursive routine.

## 18.2 Recursive Print Binary Example

This section provides an example recursive subroutine to accept a decimal number and print that number in binary (i.e., using 1's and 0').

It is assumed the reader has a basic understanding of binary<sup>14</sup> representation. This information is summarized in the chapter on Computer Organization. Additionally, there are many references available on the Internet.

### 18.2.1 Understand the Problem

For this example, the problem is divided into two parts, the main program and the recursive subroutine. The main program will handle the prompting and reading of the decimal number including error checking and re-prompting as needed. The recursive subroutine will display the binary value. Since the error checking is already performed, the recursive subroutine will assume valid input. For more complex examples, the routine may need to perform basic error checking.

### 18.2.2 Create the Algorithm

One basic algorithm to convert a decimal number into a binary number, is successive integer division by 2. For example, given the number 13, 13 divided by 2 is 6 with a remainder of 1. Next, the 6 is divided by 2 giving 3 with a remainder of 0. Again, the 3 is divided by 2 providing a 1 with a remainder of 1. The final 1 is divided by 2 resulting in a 0 with a remainder of 1. With a final result of 0, the algorithm is completed. The process is shown as follows:

$$\frac{13}{2} = 6 \quad \text{remainder } 1$$

$$\frac{6}{2} = 3 \quad \text{remainder } 0$$

$$\frac{3}{2} = 1 \quad \text{remainder } 1$$

$$\frac{1}{2} = 0 \quad \text{remainder } 1$$

The remainders, always 0 or 1, represent the binary value. However, the resulting remainders are generated in backwards order. As such, the resulting remainders 1, 0, 1, and 1 in this example must be reversed for a final value of 1101<sub>2</sub> (as noted in chapter 2).

This process can be converted into a recursive relation as follows:

$$\text{printBinary}(n) = \begin{cases} \text{if } n \leq 1 & n \\ \text{if } n > 1 & \text{printBinary}(n/2) \\ & \text{output mod}(num, 2) \end{cases}$$

<sup>14</sup> For more information regarding binary representation, refer to: [http://en.wikipedia.org/wiki/Binary\\_number](http://en.wikipedia.org/wiki/Binary_number)

This definition assumes that the value of  $n$  is positive. The recursive relation can be used by directly converting the algorithm into code.

### 18.2.3 *Implement the Program*

Based on the recursive definition, a simple recursive subroutine can be created. In order to demonstrate the recursive subroutine, a main program is provided that will read a decimal value from user and ensure it is between 0 and 1,000,000, and then call the recursive routine.

```
! Simple recursive program to print a decimal number in binary
program binary
implicit none
integer :: decimalNumber

    write (*,'(a)') "Decimal to Binary Conversion Example"

    do
        write (*,'(a)', advance="no")           &
            "Enter Decimal Number (0 - 1,000,000): "
        read (*,*) decimalNumber
        if (decimalNumber >= 0 .and.             &
            decimalNumber <= 1000000) exit

        write (*,'(a)') "Error, decimal value out of range."
        write (*,'(a)') "Please re-enter."
    end do

    write (*,'(/a, i7, a)', advance="no")       &
        "The decimal value of ", decimalNumber, " is "

    call printBinary(decimalNumber)
    write (*,'(/)')

contains

! *****
! Print binary subroutine.

recursive subroutine printBinary(num)
integer, intent(in) :: num

    if (num > 1) call printBinary(num/2)

    write (*,'(i1)', advance="no") mod(num,2)

    return
end subroutine printBinary

! *****
end program binary
```

The spacing and indentation are not required, but help to make the program more readable. The main program ensures that the recursive routine is not called in invalid values (i.e., values  $\leq 0$ ).

### 18.2.4 *Test/Debug the Program*

For this problem, the testing would involve executing the program and entering a series of decimal values and ensure that the results are correct. The Windows calculator provides simple convert-to-binary function that can be used for verification.

If the program does not provide the correct results, the input to the recursive subroutine should be verified (via write statements). Next, some additional write statements in the recursive subroutine can be added to provide insight into what is being done for each call.

## 18.3 Recursive Functions

A recursive function declaration must be preceded by the keyword **recursive**. In addition, the keyword **result** must be explicitly added to the function declaration. The **result** keyword is used to specify a single variable for the return value. Similar to a standard function, the result variable must be set to a return value to the calling routine.

The function type specifies the type of the result variable. For example:

```
<type> recursive function <name> (<arg's>) result (<variable>)
<declarations>

    <body of function>

    <variable> = expression

    return
end function <name>
```

The main does not need any special declarations.

## 18.4 Recursive Factorial Example

This section provides an example recursive function to compute the mathematical factorial<sup>15</sup> function. It is assumed the reader is familiar with the factorial function.

### 18.4.1 *Understand the Problem*

The factorial function is mathematically defined as follows:

$$n! = \prod_{k=1}^n k$$

Or more familiarly, you might see 5! as:

$$n! = 5 \times 4 \times 3 \times 2 \times 1$$

For this example, the problem is divided into two parts, the main program and the recursive function.

<sup>15</sup> For more information regarding factorial, refer to: <http://en.wikipedia.org/wiki/Factorial>

The main program will handle the prompting and reading of the  $n$  value. This will include error checking and re-prompting as needed. The recursive function will compute and return the factorial value. Since the error checking is performed, the recursive function will assume valid input.

For more complex examples, the function itself may need to perform the error checking. As such, a simple helper function could be used to verify the input value or values before calling the recursive function.

### 18.4.2 Create the Algorithm

A typical recursive relation for factorial is as follows:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times \text{factorial}(n-1) & \text{if } n \geq 1 \end{cases}$$

This definition assumes that the value of  $n$  is positive.

It must be noted that this function could easily be computed with a loop. However, the reason this is done recursively is to provide a simple example of how a recursive function is developed using a familiar mathematical function.

### 18.4.3 Implement the Program

Based on the recursive definition, a simple recursive function can be created. In order to demonstrate the recursive function, a main program is provided that will read the decimal value from user and ensure it is between 1 and 15, and then call the recursive routine.

The recursive function declaration uses an input argument,  $n$ , and a result argument, **ans**, in this example. The input argument must be declared as **intent(in)** in the standard manner. However, the result argument is an **out** by definition and will assume the type of the function itself, integer in this example.

For the recursive factorial function, the basic algorithm is provided as part of the recursive definition. The example main will read the  $n$  value from the user, call the factorial function, and display the results.

```
! Simple recursive function example.

program recursionExample
implicit none
integer :: num, numFact
    write (*,'(a/)' ) "Recursion Example"

    do
        write (*,'(a)', advance="no") "Enter N (1-15): "
        read (*,*) num
        if (num >= 1 .and. num <= 15) exit
        write (*,'(a)') "Error, N out of range."
        write (*,'(a)') "Please re-enter."
    end do

    numFact = fact(num)
```

```

        write (*,'(a, i2, a, i10,/)' ) "Factorial of ", num, &
            " is ", numFact

contains

! *****
! Factorial function

integer recursive function fact(n) result (ans)
implicit none
integer, intent(in) :: n

    if (n == 1) then
        ans = 1
    else
        ans = n * fact(n-1)
    end if

    return
end function fact

! *****

end program recursionExample

```

The spacing and indentation are not required, but help to make the program more readable. The main program ensures that the recursive routine is not called in invalid values (i.e., values  $\leq 0$ ).

#### 18.4.4 *Test/Debug the Program*

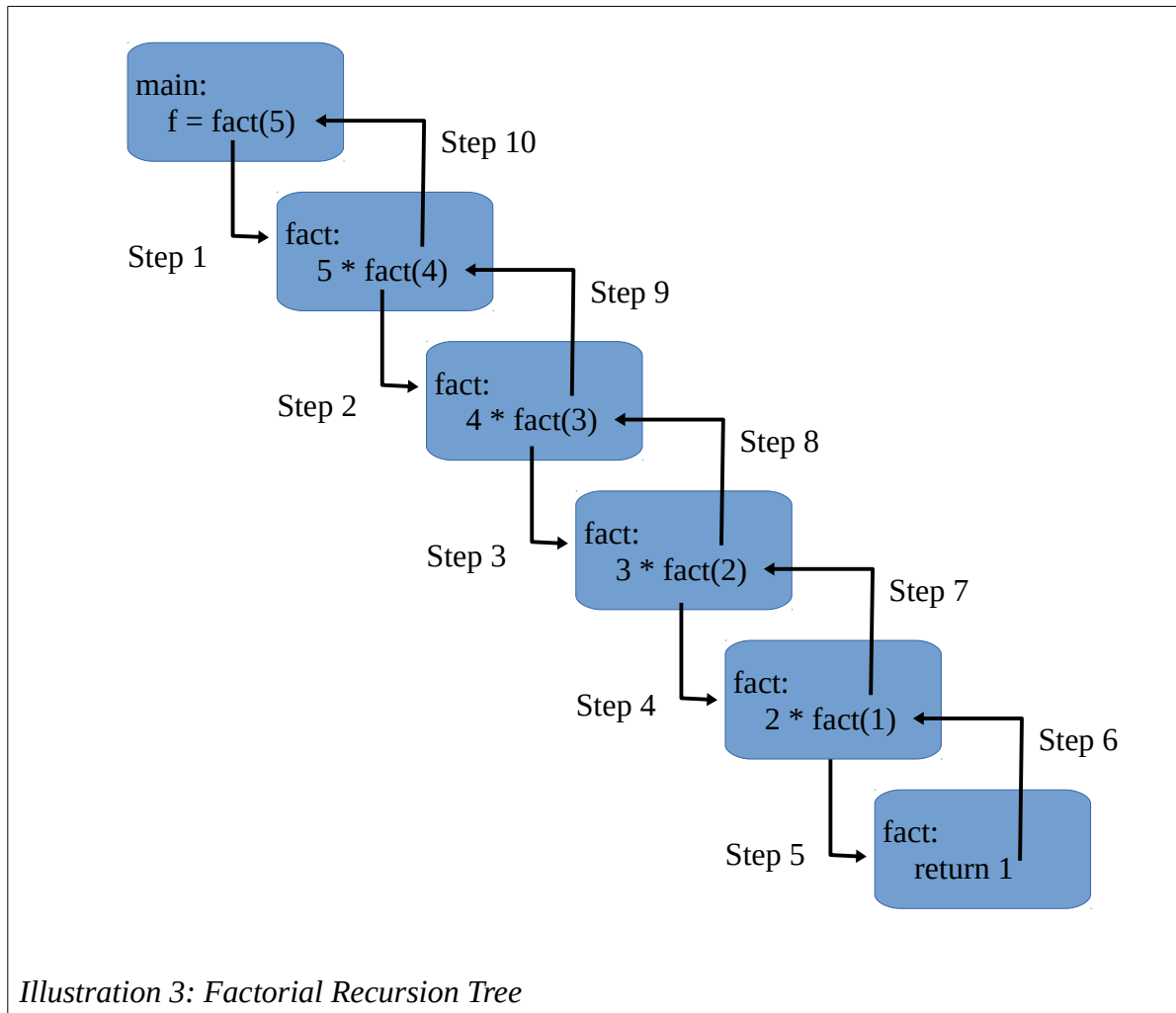
For this problem, the testing would involve executing the program, entering a number, and ensuring that the result is correct. The Windows calculator includes a factorial function which can be used to verify the result.

If the program does not provide the correct result, the input to the recursive function should be verified (via write statements). Next, some additional write statements in the recursive function can be added to provide insight into what is being done for each call.



## 18.5 Recursive Factorial Function Call Tree

In order to better understand recursion, a recursion tree can help show how the recursive calls interact.



When the initial call to factorial function occurs from main, the main will start into the **fact()** function (shown as step 1). Since the argument of 5 is not a base case, the **fact()** function must call **fact()** again with the argument of  $n-1$  or 4 in this example (step 2). And, again, since 4 is not the base case, the **fact()** function must call **fact()** again with the argument of  $n-1$  or 3 in this example (step 3).

This process continues until the argument passed into the **fact()** function meets the base case which is when the arguments is equal to 1 (shown as step 5). When this occurs, only then is a return value provided to the previous call (step 6). This return argument is then used to calculate the previous multiplication which is 2 times 1 which will return a value to the previous call (as shown in step 7). This process will continue (steps 8, 9, and 10) until the main has a final answer.

Since the code being executed is the same, each instance of the **fact()** function is different from any other instance only in the arguments and any local values (none in this example).

It should also be noted that the height of the recursion tree is directly associated with the amount of memory used by the recursive function. For problems where the recursion tree is very large, this can have a negative impact on overall performance of a recursive routine.

## 18.6 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 18.6.1 Quiz Questions

Below are some quiz questions.

- 1) What are the two requirements for a recursive definition?
- 2) In recursion, the case for which a solution is obtained directly is called what?
- 3) What keyword is required for a recursive subroutine?
- 4) What two keywords are required for a recursive function?
- 5) What special requirements are required of the calling routine in order to call a recursive subroutine or recursive function?
- 6) For a recursive routine, what would happen if the routine does not stop recursing?
- 7) Create a recursion tree for the recursive Fibonacci function (as described in the following suggested projects section) with the input of 13. *Note*, the recursive Fibonacci function requires two recursive calls for the non-base case step.

### 18.6.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the print binary main program and recursive subroutine. Test on several data sets and verify that the program produces the correct results.
- 2) Type in the factorial main program and recursive function. Test on several data sets and verify that the program produces the correct results.
- 3) The recursive definition of Fibonacci function is as follows:

$$fib(n) = \begin{cases} 1 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ fib(n-1)+fib(n-2) & \text{if } n \geq 2 \end{cases}$$

Create a main program to read the  $n$  value from the user and ensure it is between 1 and 40. Develop recursive function, `fib`, to recursively compute the Fibonacci number based on the provided definition. *Note*, the recursive Fibonacci function requires two recursive calls for the non-base case step.

- 4) Develop a recursive subroutine to recursively print a star tree. Based on an initial value,  $n$ , the star tree should be displayed. For example, for an  $n$  value of 5, the program should output something similar to the following:

**Recursive Subroutine Program**

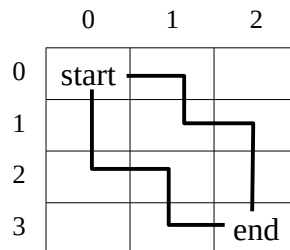
**Enter Number of Stars: 5**

**Star Tree:**

```
* * * * *
* * * *
* * *
* *
*
```

Create a main program to read the  $n$  value from the user and ensure it is between 1 and 50. Develop recursive subroutine, **printStars()**, to recursively print the star tree as shown. The subroutine should print one line per call. For successive recursive calls, the  $n$  value passed as an argument should be decremented. The based case would be one (1) star.

- 5) Write a program using a recursive function to determine the number of possible paths through a two-dimensional grid. The only allowed moves are one step to the right or one step down. For example, given a grid as follows:



Moving from the starting location, (0,0) in this example, going to the end location, (3,2) in this example, can be performed in 10 different ways. Two, of the ten, different ways are shown in the example above. The function must be recursive.

Create a main program to read the initial grid coordinates and ensure that they are valid (positive values) and that the end coordinates are greater than the start coordinates. Create a recursive function, **countPaths()**, to determine the number of possible paths through a two-dimensional grid. The function will accept a start coordinate (row,col) and a final end coordinate (row,col).

- 6) The Tower of Hanoi is a mathematical puzzle that consists of three pegs, and a number of disks of different sizes which can slide onto any peg. The puzzle starts with the disks neatly stacked in order of size on one peg, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another peg, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the pegs and sliding it onto another peg, on top of the other disks that may already be present on that peg.
- No disk may be placed on top of a smaller disk.



The following is a recursive definition for the problem:

$$hanoi(n, from, to, by) = \begin{cases} write(\text{move the disc from } from \text{ to } to) & \text{if } n=1 \\ \begin{matrix} hanoi(n-1, from, by, to) \\ hanoi(1, from, to, by) \\ hanoi(n-1, by, to, from) \end{matrix} & \text{if } n>1 \end{cases}$$

Create a main program to read and validate the number of disks, ***n***, from the user and ensure it is between 1 and 10. Develop recursive function, ***hanoi***, to recursively compute a solution to the Tower of Hanoi problem.

## 19 Character String / Numeric Conversions

---

Character string values, such as "123" cannot be used to perform numeric operations such as addition or multiplication. As such, for more complex programs, there is sometimes the need to convert between a character string representing a numeric value and an actual real or integer number.

These conversions can be performed using what is referred to as an internal read or an internal write. Basically, the read or write functions and associated format statements can be used to perform basic conversions. Instead of reading from an open file, the read and write operations can read and write directly from and to variables. The specified format provides guidance for the conversion result.

Based on the input, a conversion may not be possible. For example, the character string "3.14" can be converted into the real value of 3.14. However, the character string "3.1z4" could not be converted since the 'z' is not a legal numeric value.

If a conversion is not possible, an error would be generated. If not handled, such an error would crash the program. In order to address and handle any potential errors, the **iostat** parameter for the read/write operation is used as previously described in the file operations chapter.

### 19.1 Character String to Numeric Conversion

A character string can be converted into an integer or real value using an internal read operation. The string is provided as the input instead of a file unit number. The numeric variable is provided as the location for the result of the read operation. The format will provide guidance for the conversion.

The following is a simple example that will declare two strings and convert the first into an integer value and the second into a real value. Additionally, a third string conversion is performed on an invalid numeric string (to better show the error handling).

```
! Example program to use an internal read for
! character / numeric conversion.

program cvtExample 1

implicit none
integer :: cvtErr
character(4) :: iString = "1234"
character(7) :: rString = "3.14159"
character(7) :: badString = "3.14z59"
integer :: iNum1, iNum2
real :: pi, tau

      write (*, '(a, /)') "Example Conversion Program."

! -----
```

```

! Convert string to an integer value.

      read (iString, '(i10)', iostat=cvtErr) iNum1

      if (cvtErr == 0 ) then
          iNum2 = iNum1 * 2
          write (*,'(a, i5, /, a, i5, /)')      &
              "num1 = ", iNum1, "num2 = ", iNum2
      else
          write (*,'(a, /)') "Error, invalid integer string."
      end if

! -----
! Convert string to a real value.

      read (rString, '(f17.6)', iostat=cvtErr) pi

      if (cvtErr == 0 ) then
          tau = pi * 2.0
          write (*,'(a, f5.3, /, a, f5.3, /)')      &
              "pi = ", pi, "tau = ", tau
      else
          write (*,'(a, /)') "Error, invalid real string."
      end if

! -----
! Convert string to a real value.

      read (badString, '(f12.4)', iostat=cvtErr) pi

      if (cvtErr == 0 ) then
          tau = pi * 2.0
          write (*,'(a, f5.3, /, a, f6.3)')      &
              "pi = ", pi, "tau = ", tau
      else
          write (*,'(a, /)') "Error, invalid real string."
      end if

end program cvtExample1

```

The specific formats used on the read operations in the example are wider or larger than the expected number (which is allowed). Should a smaller format be used, it would either truncate the value or possibly generate a conversion error. To ensure appropriate conversion, the final values should be verified against the expected result.

An example of the output for this program is as follows:

**Example Conversion Program.**

```

num1 = 1234
num2 = 2468

```

```
pi = 3.142
tau = 6.283
```

```
Error, invalid real string.
```

The multiplication by 2 for each of the numeric values was performed only as an example since multiplication can only be performed on numeric data types (i.e., integer, real, or complex).

## 19.2 Numeric to Character String Conversion

An integer or real value can be converted into a character string using a write operation. The string is provided as the output variable instead of a file unit number. The numeric variable is provided as the input for the write operation.

The following is a simple example that will convert an integer into a string and a real into a string. Some numeric operations are performed on the numeric values and then the resulting strings are concatenated with another string. Concatenation can only be performed on character data types.

```
! Example program to use an internal write for
! character / numeric conversion.

program cvtExample2
implicit none
integer :: cvtErr
character(50) :: str1, str2, msg1, msg2
integer :: iNum=2468
real :: pi = 3.14, tau

        write (*,'(a, /)') "Example Conversion Program."

! -----
! Convert integer value to a string.

        iNum = iNum / 100
        write (str1, '(i3)', iostat=cvtErr) iNum

        if (cvtErr == 0 ) then
            msg1 = "My age is " // str1
            write (*,'(a, a)')          &
                "Message 1 = ", msg1
        else

            write (*,'(a, /)') "Error, invalid conversion."
        end if

! -----
! Convert real value to a string.

        tau = pi * 2.0
```

```

write (str2, '(f5.3)', iostat=cvtErr) tau

if (cvErr == 0 ) then
    msg2 = "The value of TAU is " // str2
    write (*,'(a, a, /)')          &
        "Message 2 = ", msg2
else
    write (*,'(a, /)') "Error, invalid conversion."
end if

end program cvtExample2

```

An example of the output for this program is as follows:

**Example Conversion Program.**

```

Message 1 = My age is 124
Message 2 = The value of TAU is 6.283

```

Once the numeric values are converted into strings, the character functions and character operations can be used as needed.

## 19.3 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 19.3.1 Quiz Questions

Below are some quiz questions.

- 1) Which operation, internal read or internal write, is required to convert a character string containing a numeric value?
- 2) Which operation, internal read or internal write, is required to convert a real value into a character string?
- 3) Provide an appropriate statement to convert the character string *sNum*="123" into an integer variable *iNum*. The error status should be written to the variable *cvErr*. You may assume all variables are already declared and initialized.
- 4) Provide an appropriate statement to convert the integer variable *iNum*=234 into a character string *sNum*. The error status should be written to the variable *cvErr*. You may assume all variables are already declared and initialized.

### 19.3.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the character string to numeric values conversion example program. Update the values of the character strings with different data and verify that the program produces the correct results.



- 2) Update the program from the previous question to read a series of character strings from the user, attempt conversion for each character string to a real value. If invalid, the user should be re-prompted. If valid, the sum of the values should be maintained. When the user enters a 0 , the program should stop reading numbers, display the final sum, and terminate.
- 3) Type in the numeric values to character string values conversion example program. Update the values of the real and integer values with different numbers and verify that the program produces the correct results.
- 4) Develop a program to read a series of integer values, sum the values, and convert the final sum into a character string. The string should be concatenated with the string “The sum is ” and displayed to the terminal using a single character string variable.
- 5) Write a program to prompt the user for an integer number between 100 and 999, re-prompting until a valid number is provided. When a valid number is provided, create a file name in the form of “file<number>.txt”, open/create the file, and write all numbers from 1 to the number, one per line, and close the file. For example, if 42 is entered, the file **file42.txt** should be created and contain the numbers 1, 2, 3, ..., 42 (one per line).



## 20 System Services

---

The term system services generally refers to asking the operating system for information.

The read, write, and file operations (open, read, write, close) are common system services and have already been addressed in a previous chapter.

Other system services include obtaining the date and/or time from the operating system and obtaining the command line arguments (if any). The term *command line arguments* is used to refer to information entered on the command line after the program name. This allows the user to provide some information to the program before it starts (or as the program is started), which might save time as compared to prompting for and interactively reading the information at run-time.

While there are many system services, only these basic ones are presented for reference. These system services may be useful when working on more complex problems. Additionally, the calls and usage for other system services is very similar to how these are performed.

### 20.1 Date and Time

The date and time functions are combined into a single system service call. The date and time values can be obtained as character strings, as integers, or both simultaneously. The options for date and time as explained in the next section followed by an example.

It must be noted that if the operating system has an incorrect date or time, the values returned to the program will also be incorrect.

#### 20.1.1 Date and Time Options

The date and/or time values are obtained from the operating system using the `get_date_time()` system service call. The argument or arguments for the system service call must specify at least one of the following options:

- **date** = <character(8)>
- **time** = <character(10)>
- **zone** = <character(5)>
- **values** = <integer values array>

As noted, each option must provide a location of where to place the results of the specified size and date type. The options are comma separated, similar to the read and write calls noted in a previous chapter. At least one argument must be included in the call.

The zone, or time zone, option will provide the time difference between local time and Coordinated Universal Time (UTC<sup>16</sup>). The character string will provide a result in *hours:minutes* format and the integer values will be in minutes only. However, the minutes can be easily converted to hours.

16 For more information regarding coordinated universal time, refer to:  
[http://en.wikipedia.org/wiki/Coordinated\\_Universal\\_Time](http://en.wikipedia.org/wiki/Coordinated_Universal_Time)

The options and associated values returned are more fully described in the following table.

Option	Data Type	Description
<b>date</b>	character(8)	The string returned will be in the form YYYYMMDD, where YYYY is year, MM is month, and DD is date.
<b>time</b>	character(10)	The string returned will be in the form HHMMSS.SSS where HH is hour, MM is minute, SS is second, and SSS is milliseconds.
<b>zone</b>	character(5)	The string returned will be in the form of $\pm$ HHMM, where HHMM is the time difference between local time and Coordination Universal Time.
<b>values</b>	integer array, 8 elements	The values will be returned in the 8 value integer array as follows: <ul style="list-style-type: none"> <li>• values(1) → year</li> <li>• values(2) → month (1-12)</li> <li>• values(3) → date (1-31)</li> <li>• values(4) → time zone difference (minutes)</li> <li>• values(5) → hour (0-23)</li> <li>• values(6) → minutes (0-59)</li> <li>• values(7) → seconds (0-59)</li> <li>• values(8) → milliseconds (0-999)</li> </ul>

Each argument is optional, but at least one argument must be included. Multiple arguments are allowed.

### 20.1.2 Date and Time Example Program

The following is an example program that obtains the date and time information from the operating system in various formats. The final results are shown for reference.

```
! Example program to obtain the date and time from the system.

program timeDateExample

! -----
! Declarations.

implicit none
```

```

integer, dimension(8) :: valuesArr
character(len=8) :: today
character(len=10) :: now
character(len=5) :: myzone
integer :: i

! -----
! Display simple header.

        write (*,'(a)')                                &
            "Example Program for Date and Time Functions."

! -----
! Get date, time, and zone from system as characters.
! Display to screen for reference.

        call date_and_time(date=today)
        write (*,'(/a, a)') "Today is: ", today

        call date_and_time(time=now, zone=myzone)
        write (*,'(a, a)') "Time is: ", now
        write (*,'(a, a/)') "Time Zone is: ", myzone

! -----
! Get all date values from the system as integers.
! Display to screen for reference.

        call date_and_time(values=valuesArr)

        write (*,'(a)') "Values Array:"

        write (*,'(a, i4)') "Date, Year:           ", valuesArr(1)
        write (*,'(a, i2)') "Date, Month:          ", valuesArr(2)
        write (*,'(a, i2)') "Date, Day:             ", valuesArr(3)
        write (*,'(a, i2)') "Time, Hour:            ", valuesArr(5)
        write (*,'(a, i2)') "Time, Minutes:         ", valuesArr(6)
        write (*,'(a, i2)') "Time, Seconds:         ", valuesArr(7)
        write (*,'(a, i3)') "Time, Millseconds:     ", valuesArr(8)

        write (*,'(/,a, i8)')                                &
            "Time difference with UTC in minutes: ",    &
            valuesArr(4)

        write (*,'(a, i2, a1, i2.2 ,/)')                &
            "Time difference with UTC in hours: ",      &
            valuesArr(4)/60, ":", mod(valuesArr(4), 60)

end program timeDateExample

```

While this program does not really use the time or date values for anything meaningful, it does provide

an example of how the information is obtained for use in other, more complex programs. The output of this example program is shown as follows:

```

Today is: 20131212
Time is: 154032.491
Time Zone is: -0800

Values Array:
Date, Year:      2013
Date, Month:     12
Date, Day:       12
Time, Hour:      15
Time, Minutes:   40
Time, Seconds:   32
Time, Millseconds: 491

Time difference with UTC in minutes:    -480
Time difference with UTC in hours:      -8:00

```

The UTC for Las Vegas, Nevada is indeed, -8 hours as shown. The results for the UTC will be based on the actual geographic location of where the system executing the program is located.

## 20.2 Command Line Arguments

The usage of command line arguments, information entered on the command line after the program name, can be very useful in specific circumstances. By allowing the user to provide some information on the command line, it saves the effort of entering the information interactively after the program starts. The term command line arguments is sometimes referred to as command line options.

For example, when starting a Fortran program for a simple game, the persons name could be provided on the command line. If the name is provided, the program could use that name. Otherwise, the program could use a generic name such as 'player'. The command line might appear as follows:

```
c:\fortran> tictactoe ed
```

On Unix based systems, this might be:

```
% ./tictactoe ed
```

Where **tictactoe** is the name of the program and 'ed' is the single command line argument.

Multiple command line arguments can be provided, but must be separated with a space or multiple spaces. There is no predefined required format. The formatting or ordering requirements are up to the program.

The **gfortran** compiler requires command line arguments for the various options including the input file name (i.e., tictactoe.f95) and the output file name (i.e., -o tictactoe) specification. For example,

```
c:\fortran> gfortran tictactoe.f95 -o tictactoe
```

For the **gfortran** compiler, there is no required order for the command line arguments. However, a valid output file name must follow the '-o' argument.

The error handling for command line arguments is typically handled differently. Many programs check the command line arguments, and if incorrect for any reason, display an error message and terminate the program. This is what the **gfortran** compiler does when invalid or incorrect arguments are provided.

It would be possible to create a program that could verify arguments, and if incorrect, display an error message and then request correct input from the user. The handling of the command line arguments processing is entirely the responsibility of the program and not the operating system.

### 20.2.1 Argument Count

There are generally two steps to obtaining the command line arguments. The first step is getting the argument count or the number of arguments entered on the command line. The previous **tic-tac-toe** example has one argument (i.e., 'ed'). The previous **gfortran** example had three (i.e., 'tictactoe.f95', '-o', and 'tictactoe').

The argument count is obtained using the **command\_argument\_count()** system service as follows:

```
integer :: myCount
myCount = command_argument_count()
```

Which will return the count in the variable **myCount** (as shown above). The count will be zero if no arguments are entered.

### 20.2.2 Get Arguments

Once the argument count is available, the actual arguments can be obtained. The arguments are always returned as character values. If the argument is meant to be used as a real or integer value, it must be converted. The Character String / Numeric Conversions chapter provides a description of how this can be accomplished.

When obtaining the command line arguments, the **get\_command\_argument()** system service is used. An argument is returned into a specified character variable. The character variable should be large enough (i.e., able to hold enough characters) to store the expected argument. The actual length of the returned argument can optionally be provided. Additionally, if the character variable is too small, the returned result will be truncated and the status set accordingly to indicate an error.

The options and associated values returned are described in the following table.

Option	Data Type	Description
<b>length</b>	integer	Input integer argument indicating which argument should be returned. Must be between 1 and the <b>command_argument_count()</b> value.
<b>value</b>	character(*)	Output character variable of where to store the N <sup>th</sup> argument as specified by the length value (above). The variable must be declared with an appropriate size.

<b>length</b>	integer	Output integer argument for the actual length of the string returned by the value argument (above).
<b>status</b>	integer	Output integer argument for the returned status value. A return status value of 0 is success and -1 is fail.

The first two arguments are required and the final two arguments are optional.

### 20.2.3 *Command Line Arguments, Example Program*

This simple example obtains the command line argument count and displays the arguments to the screen for reference. In this example, the program will expect a real value as the first argument and an integer value as the second argument (if a second argument is provided). Any additional arguments, while not used, are still displayed the screen along with the argument count value.

For this example, since the number of arguments is unknown ahead of time, an array to hold the arguments is allocated at run-time. While this is not necessary, it does help provide a more complete example. Such a process would only be appropriate if a varying number of command line arguments is desired.

```

! Example program to demonstrate how to obtain the
! command line arguments from the system.

program argsExample

implicit none
integer :: argCount, allocStatus, rdErr, i, iNum
real :: rNum
character(len=80), dimension(:), allocatable :: args

! -----
! Get command line argument count from system.

    argCount = command_argument_count()

    if (argCount == 0) then
        write (*,'(a)') "No command line arguments provided."
        stop
    end if

! -----
! Allocate an array to hold the arguments.

    allocate(args(argCount), stat=allocStatus)

    if (allocStatus > 0) then

```



```

        write (*,'(a)')
        "Allocation error, program terminated."
    stop
end if

! -----
! Get each argument, one at a time, from system.

    do i = 1, argCount
        call get_command_argument(number=i,value=args(i))
    end do

! -----
! Display arguments to screen.

    if (argCount == 0) then
        write (*,'(a)') "No command line arguments provided."
    else
        if (argCount == 1) then
            write (*,'(a, i1, a)') "There was ",      &
                argCount, " command line argument."
        else
            write (*,'(a, i2, a)') "There were ",      &
                argCount, " command line arguments."
        end if

        write (*,'(/,a)') "The arguments were: "
        do i = 1, argCount
            write (*,'(a, a)') "  ", trim(args(i))
        end do

        write (*,*)
    end if

! -----
! Convert a string to a numeric value using an internal read.

    if (argCount >= 1) then
        read (args(1), '(f12.5)', iostat=rdErr) rNum
        if (rdErr == 0 ) then
            write (*,'(a, f12.5)')
                "Argument 1 - Real Number = ", rNum
            &
        else
            write (*,'(a)') "Error, invalid real value."
        end if
    end if

    if (argCount >= 2) then
        read (args(2), '(i10)', iostat=rdErr) iNum
        if (rdErr == 0 ) then
            write (*,'(a, i10)')
                &

```

```

                                "Argument 2 - Integer Number = ", iNum
                                else
                                write (*,'(a)') "Error, invalid integer value."
                                end if
                            end if

                            write (*,*)

                        end program argsExample

```

An example of the output for this program with valid command line arguments provided is shown below. The executable name of the example program is '**args**'.

```

c:\fortran> args 3.14 23 hello world
There were 4 command line arguments.

The arguments were:
3.14
23
hello
world

Argument 1 - Real Number =      3.14000
Argument 2 - Integer Number =      23

c:\fortran>

```

Another example of the output for this program with invalid command line arguments provided is shown as follows:

```

c:\fortran> args 3.14 23 hello world
There were 4 command line arguments.

The arguments were:
hello
3.14
23
world

Error, invalid real value.
Error, invalid integer value.

c:\fortran>

```

*Note*, the order for the valid and invalid arguments was chosen arbitrarily.

## 20.3 Exercises

Below are some quiz questions and project suggestions based on this chapter.

### 20.3.1 Quiz Questions

Below are some quiz questions.

- 1) Can the date alone be obtained (i.e., without obtaining the time) using the **get\_date\_time()** system service (yes/no)?
- 2) When using the **get\_date\_time()** system service to obtain the integer date and time values, what is the index number for the:
  1. The month value.
  2. The hour value.
  3. The year value.
- 3) How can the integer time zone value be converted to *hours:minutes* format?
- 4) Provide the **get\_date\_time()** system service call to obtain the current date string into the already declared character variable **dStr**.
- 5) Provide the **get\_date\_time()** system service call to obtain the current time string and time zone string into the already declared character variables **timeStr** and **zoneStr**.
- 6) Provide the **get\_date\_time()** system service call to obtain the current date/time integer values, and the current date string into the already declared 8 element integer **valuesArr** array.
- 7) If there are no arguments entered on the command line, what is returned by the **get\_command\_argument()** system service?
- 8) Provide an appropriate statement to obtain the 3<sup>rd</sup> command line argument, the argument length, and a status value. You may assume the variables for the command line argument (character), named **args**, the length (integer), named **arg3length**, and the status (integer), named, **arg3stat** are already declared appropriately.

### 20.3.2 Suggested Projects

Below are some suggested projects.

- 1) Type in the get date and time example program. Update the output to more clearly display only the current date and time on one line. Remove the unnecessary or redundant output.
- 2) Type in the get command line arguments example program. Execute the program using a variety of different expected and unexpected arguments.
- 3) Develop a program that reads command line arguments for a title and time (hours and minutes). Calculate and display the difference between the current time and the time from the command line. If the command line arguments are incomplete or incorrect, an error message should be displayed and the program, terminated. The program should use 24-hour format. One possible approach to determining the time difference would be to convert each time into minutes (from hours and minutes), compute the difference, and convert the difference in minutes back to hours

and minutes. The final output should display the title, the system time, the entered time, and the time difference. All times should be displayed in *hours:minutes* format.

- 4) Write a program to get the date from the system and display a formatted date. The formatted date should be: <day of week>, <month name> <day>, <year>. For example, 12/2/2013 should be displayed as **Monday, December 2, 2013**. Each string should include only the appropriate spacing. Specifically, there should be no extra spaces between the words or numbers (including the date value, 2 in the example).

To calculate the day on which a particular date falls, the following algorithm may be used.

*Note*, all variables are integers and the divisions are integer divisions.

$$a = \frac{14 - \text{month}}{12}$$

$$y = \text{year} - a$$

$$m = \text{month} + 12 \times a - 2$$

$$\text{dayNum} = \left[ \text{day} + y + \frac{y}{4} - \frac{y}{100} + \frac{y}{400} + \left( 31 \times \frac{m}{12} \right) \right] \bmod 7$$

Where the *month*, *day*, and *year* variables are the integer date values (12, 2, and 2013 in the previous example). The final value of *dayNum* is 0 for a Sunday, 1 for a Monday, 2 for a Tuesday, and so forth with the maximum value of 6 for Saturday.

- 5) Update the program from the previous question to write the formatted date to a file. The file name should be created based on the current date in the following format “file<MMDDYY>.txt”. For example, for a date of 12/02/2013, the file should be named **file120213.txt** and contain the character string **Monday, December 2, 2013**.
- 6) Write a program to obtain the current date and attempt to open a file based on the date in the following format, “file<MMDDYY>.txt”. For example, for a date of 12/02/2013, the file should be named **file120213.txt**. If the file exists, the contents should be displayed to the screen. If the file does not exist, an appropriate error message should be displayed.

## 21 Appendix A – ASCII Table

---

This table lists the American Standard Code for Information Interchange (ASCII) characters or symbols and their decimal numbers.

Char.	Dec.
	32
!	33
"	34
#	35
\$	36
%	37
&	38
'	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
:	58
;	59
<	60
=	61
>	62
?	63

Char.	Dec.
@	64
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90
[	91
\	92
]	93
^	94
_	95

Char.	Dec.
`	96
a	97
b	98
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111
p	112
q	113
r	114
s	115
t	116
u	117
v	118
w	119
x	120
y	121
z	122
{	123
	124
}	125
~	126
	127



## 22 Appendix B – Start-Up Instructions

---

The following provides some specific instructions for getting started. These instructions are specifically geared for using a MacOS or Windows based PCs. This includes all versions of Windows from Windows XP, Windows Vista, Windows 7, Windows 8, and Windows 10.

The basic process is very similar to Linux (which is not covered here).

### 22.1 Working Files

Before working with Fortran program files, you should decide where you will be working (default drive, USB drive, network drive, etc.) and create a working directory where your files will be placed. In general, it will be easier if your Fortran files are not mixed with other, unrelated files. This directory should be someplace you can easily get to. That might be on your home workstation/laptop, on a network drive, or on a USB drive.

### 22.2 Obtaining The Compiler

First, you will need to download and install the GNU Fortran compiler. The main web page for the GNU Fortran compiler is:

<http://gcc.gnu.org/fortran/>

This page provides general information regarding the Fortran compiler development effort, project objectives, and current status.

More specific information, and the Windows compiler binaries, can be found on the gFortran Wiki page, which is located at:

<http://gcc.gnu.org/wiki/GFortran>

This page contains links to the Gfortran binaries. On this page, click on the link labeled:

**Binaries for Windows, Linux, and MacOS**

Which will display the page for the GNUBinaries for various platforms, including Windows, MacOS, and Linux. Click on the appropriate link Windows, MacOS, GNU/Linux.

#### 22.2.1 Windows Download and Install

When selected, Windows link will show the various Windows installer options. For standard Windows (XP/Vista/7/8), the heading **MinGW build** (“native Windows” build), includes a link for the latest **installer**. Click on the link and download the Windows installation program. You will need to access this file to perform the actual installation. There is no need to install the MinGW build.

This version will work on all supported Windows versions including Windows Vista, Windows XP, Windows 7, Windows 8, and Windows 10.

After downloading, install the compiler. The installation can be accomplished by double-clicking on the downloaded file. As with all Windows installs, it will require System Administrator privileges. Once the installation has been completed, the downloaded installation file can be deleted.

### 22.2.2 *MacOS Download and Install*

When selected, MacOS link will show the various MacOS installer options. Choose the appropriate download link based on your specific MacOS version (from the listed options). After the download is complete, click on the downloaded file and perform the installation. The installation will require the system password. Once the installation has been completed, the downloaded installation (.dmg) file can be deleted.

In order to develop programs on the MacOS, Xcode<sup>17</sup>, the integrated development environment, is required. The Xcode application can be downloaded from the Apple App Store or via iTunes. An Apple account is required. If you do not have an Apple account, you can obtain a free developer account (developer.apple.com).

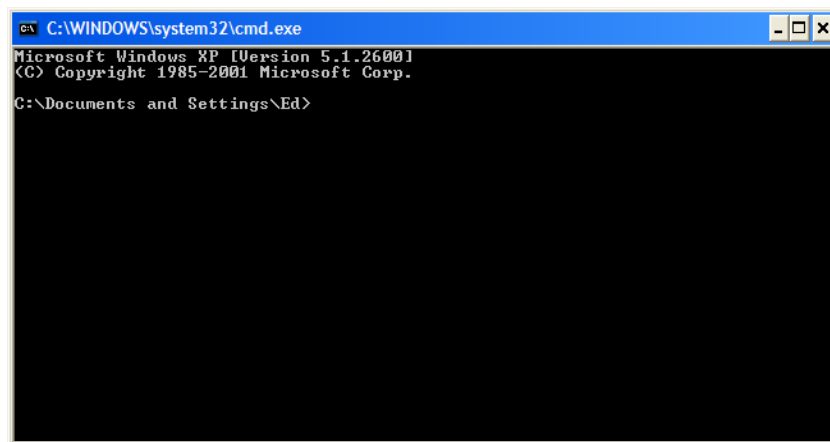
## 22.3 Windows Compilation

Please note, the Fortran compiler download and installation (previous section) must have been completed before starting this step. In order to compile and work with programs under Windows, it will be necessary to provide typed commands. This is done from within the “Command Prompt” utility.

### 22.3.1 *Command Prompt Window*

The first step is to open the Windows Command Prompt Window. In Windows 7, the “Command Prompt” is usually under **Programs** → **Accessories** → **Command Prompt**. In Windows 8 or Windows 10, the “Command Prompt” can be found by using Search and is listed under the **Windows System** heading.

The open Command Prompt window for any Windows version will look similar to the following:



Once the Command Prompt is open, the device and directory for the working files can be set.

<sup>17</sup> For more information, refer to: <https://en.wikipedia.org/wiki/Xcode>



### 22.3.2 Device and Directory

In order to compile and work with programs under Windows, a working directory should be established. This is where the program files will be stored. First, establish the **device** of the working directory. If the files are on C:\ (the main hard drive), the device is already set. If the working directory is located on a network drive, alternate drive, or USB drive, the device will need to be set.

Using the “My Computer”, determine the device or drive letter where your directory is located. Type that letter at the prompt in the Command Prompt window. For example, if your device is **K**, you would type “**k:**”.

At the drive prompt (i.e., “C:\>” or “K:\>”) you will need to change directory into the directory where your files are (if you created one). The “cd <dir\_name>” command can be used. For example, if the directory is named cs117, the command would be “**cd cs117**”.

For example:

```
C:\Documents and Settings\Ed\My Documents> k:
C:\> cd cs117
```

At this point, typing **dir** (for directory) will provide a list of files in the directory. While the format will appear different, the files are the same as shown in the **My Computer** listing.

### 22.3.3 Compiler Installation Verification

To ensure the compiler is installed correctly, open the “Command Prompt”, and type **gfortran** at the prompt.

```
C:\cs117> gfortran
gfortran: fatal error: no input files
compilation terminated.
```

The “no input files” message means that the installation was completed successfully. It is not necessary to set the device or directory in order to perform this verification.

However, if the following message is displayed,

```
C:\cs117> gfortran
'gfortran' is not recognized as an internal or external command,
operable program or batch file.
```

it means that the Fortran compiler is not installed. The installation issue must be addressed before continuing. Once the installation is completed successfully, the compilation steps detailed in Chapter 3 can be completed.

### 22.3.4 Compilation

Once the Fortran compiler is installed, programs can be compiled. Open the “Command Prompt”, and set the device and directory as appropriate.

At this point the program can be compiled, using “gfortran”. The optional “-o” qualifier is used to set the name of the output file. To execute, you can type the name of the executable (that was specified with the “-o”).

To compile the example program, the following command would be entered:

```
C:\cs117> gfortran -o hw hw.f95
```

This command will tell the 'gfortran' compiler to read the file **hw.f95** and, if there are no errors, create an executable file named **hw.exe**. If there is an error, the compiler will generate an error message, sometimes cryptic, and provide a line number. Such errors are usually the result of mistyping one of the instructions. Any errors must be resolved before continuing.

### 22.3.5 Executing

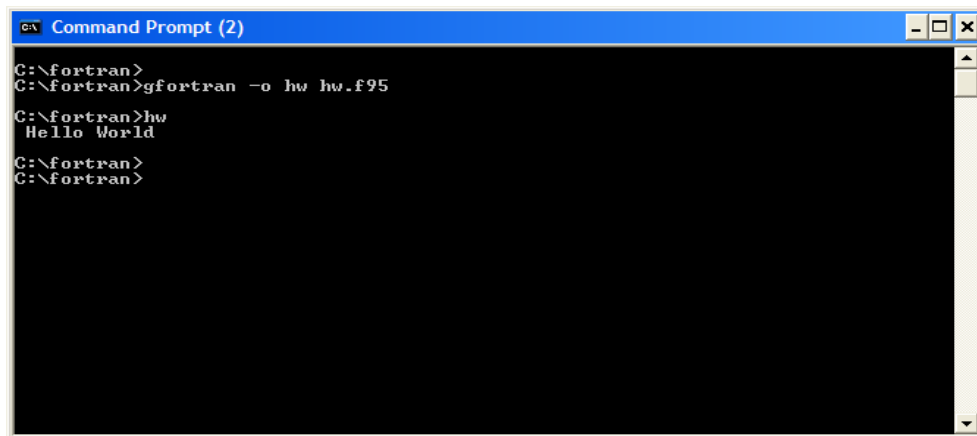
To execute or run a program, type the name of the executable file. For example, to execute or run the **hw.exe** program:

```
C:\cs117> hw
Hello World
C:\cs117>
```

Which will execute the example program and display the “Hello World” message to the screen.

### 22.3.6 Example

A more complete example is as follows:



```
Command Prompt (2)
C:\fortran>
C:\fortran>gfortran -o hw hw.f95
C:\fortran>hw
Hello World
C:\fortran>
C:\fortran>
```

It is not necessary to type the extension (i.e., “.exe”) portion of the file name.

## 22.4 MacOS Compilation

Please note, the Fortran compiler and xCode download and installation (previous section) must have been completed before starting this step. In order to compile and work with programs under MacOS, we will need to provide typed commands to the computer. This is done from within the Terminal window.

### 22.4.1 *Terminal Window*

The first step is to open the Terminal window. The open Terminal window will look similar to the following:



The screen background may be black depending on the default configuration settings. Once the Terminal is open, the directory for the working files can be set as shown in the following section.

### 22.4.2 *Device and Directory*

In order to compile and work with programs under MacOS, a working directory should be established. This is where the program files will be stored.

At the Terminal window prompt (i.e., “**eds-iMac:~ ed\$**” in this example) you will need to change directory into the directory where your files are (if you created one). The Terminal prompt will be different for each user and includes a combination of the machine name and the current user account name. The “**cd <dir\_name>**” command can be used. For example, if the directory is named **cs117**, the command would be “**cd cs117**”.

For example:

```
eds-iMac:~ ed$ cd cs117
eds-iMac:cs117 ed$
```

The prompt will change to show the new directory name (second line). At this point, typing **ls** (for directory) will provide a list of files in the directory. While the format will appear different, the files are the same as shown in the **Finder** listing.

### 22.4.3 *Compiler Installation Verification*

To ensure the compiler is installed correctly, open the Terminal window, and type **gfortran** at the prompt.

```
eds-iMac:cs117 ed$ gfortran
gfortran: fatal error: no input files
compilation terminated.
```

The “no input files” message means that the installation was completed successfully.

However, if the following message is displayed,

```
eds-iMac:cs117 ed$ gfortran
-bash: gfortran: command not found
```

it means that the Fortran compiler is not installed. The installation issue must be addressed before continuing.

If the following message is displayed,

```
eds-iMac:cs117 ed$ gfortran -o hw hw.f95
xcode-select: note: no developer tools were found at
'/Applications/Xcode.app', requesting install. Choose an option
in the dialog to download the command line developer tools.
```

it means that the Fortran compiler is installed, but Xcode is not installed. Xcode must be successfully installed in order to continue. Once the Fortran and Xcode installations are completed successfully, the compilation steps detailed in Chapter 3 can be performed.

### 22.4.4 *Compilation*

Once the Fortran compiler and Xcode are installed, programs can be compiled. Open the Terminal window and change directory as appropriate.

At this point the program can be compiled, using “gfortran. The optional “-o” qualifier is used to set the name of the output file. To execute, you can use the name of the executable (that was specified with the “-o”).

To compile the example program, the following command would be entered:

```
eds-iMac:cs117 ed$ gfortran -o hw hw.f95
```

This command will tell the 'gfortran' compiler to read the file **hw.f95** and, if there are no errors, create an executable file named **hw** (no extension). If there is an error, the compiler will generate an error message, sometimes cryptic, and provide a line number. Such errors are usually the result of mistyping one or more of the instructions. Any errors must be resolved before continuing.

### 22.4.5 Executing

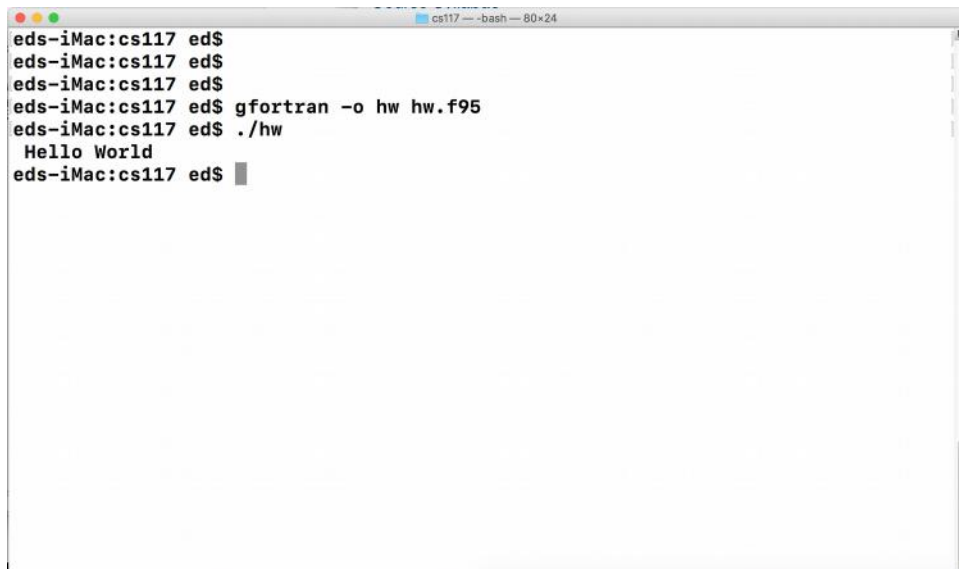
To execute or run a program, type “./” and the name of the executable file. For example, to execute or run the *hw* program:

```
eds-iMac:cs117 ed$ ./hw
Hello World
eds-iMac:cs117 ed$
```

Which will execute the example program and display the “Hello World” message to the screen.

### 22.4.6 Example

A more complete example is as follows:



```
eds-iMac:cs117 ed$
eds-iMac:cs117 ed$
eds-iMac:cs117 ed$
eds-iMac:cs117 ed$ gfortran -o hw hw.f95
eds-iMac:cs117 ed$ ./hw
Hello World
eds-iMac:cs117 ed$
```

The “./” is required when executing programs.



## 23 Appendix C – Random Number Generation

---

Generating random numbers is a common requirement for many problems. The following provides a summary of utilizing the built-in Fortran random number generator routines.

### 23.1 Initialization

The first step in generating random numbers is to initialize the Fortran random number generator, `random_seed()`. The most basic initialization is performed as follows:

```
call random_seed()
```

This will initialize the random number generator with a default seed. As such, each execution will re-generate the same series of random numbers for each execution. While this may not appear very random, since successive executions generate the same series of random numbers, the testing is more repeatable.

### 23.2 Generating Random Number

To request a random number, a real variable must be declared and then passed to the following call to the built-in `random_number()` routine. For example:

```
call random_number(x)
```

The random number between 0.0 and 1.0 such that  $0.0 \leq \text{random number} < 1.0$ .

In order to obtain a larger number, it can be multiplied by an appropriate scale. For example, to simulating the roll of a dice, a random integer number between 1 and 6 would be required. The following code would obtain the random number and then scale it and convert to integer as required.

```
call random_number(x)
die = int(x*6.0) + 1
```

Since the 0.0 is a possible value and 1.0 is not ( $0.0 \leq \text{random number} < 1.0$ ) 1 is added to ensure that 0 cannot be assigned to the variable **die**. Further, since .999 is the largest possible value (since 1.0 is not), and  $.999 * 6$  will generate 6 (5.9 truncated to 5 with 1 added).

### 23.3 Example

A simple example program to generate 100 random integer numbers, each between 1 and 100, is as follows:

```

program rand

implicit none
integer, parameter :: rcount=100
integer :: i
integer, dimension(rcount) :: nums
real :: x

    call random_seed()

    do i = 1, rcount
        call random_number(x)
        nums(i) = int(x*100.0) + 1
    end do

    write (*,'(a)') "Random Numbers:"
    do i = 1, rcount
        write (*,'(i3,2x)', advance="no") nums(i)
        if (mod(i,10)==0) write (*,*)
    end do

end program rand

```

The call to *random\_seed()* must be performed before the call to *random\_number()*. Additionally, the call to *random\_seed()* can only be performed once.

The output of this example program is shown below:

```

Random Numbers:
100  57  97  75  37  49   8   1  35  35
 22  14  91  39  45  67   2  66  65  33
 86  41  21  97  60  68  46  34  11  76
 61  72  90  66  16  62  98 100  26  56
 66  56  98  91  66  73  41  93  15  68
 77  34  12  62  83  95  74  50  38  43
 56 100 100  75  96  10  74  76  95  71
 82  56   7  49  60  14  59  52  89  31
 67  67  51  27   8  11  55  38   2  80
 63  78  96  12  32  60   5  12  22  11

```

Each execution will generate the same series of random numbers.



## 23.4 Example

In order to generate different random number for successive executions, the seed must be initialized with a different set of seed values each time.

The following example simulates the roll of two dice, which requires two random integer numbers, each between 1 and 6.

```

program diceRoll

implicit none
integer :: m, die1, die2, pair
real :: x
integer :: i, n, clock
integer, dimension(:), allocatable :: seed
character(10) :: nickname

    call random_seed(size = n)
    allocate(seed(n))
    call system_clock(count=clock)
    seed = clock + 37 * (/ (i-1, i=1, n) /)
    call random_seed(put = seed)
    deallocate(seed)

    call random_number(x)
    die1 = int(x*6.0) + 1

    call random_number(x)
    die2 = int(x*6.0) + 1

    write (*, '(2(a,1x,i1/),a,1x,i2)') "Dice 1:", die1,      &
        "Dice 2:", die2, "Dice Sum", (die1+die2)

end program diceRoll

```

Will generate different values each execution. For example, three executions of the example program are shown below:

```

C:\fortran> dice
Dice 1: 5
Dice 2: 4
Dice Sum  9

C:\fortran> dice
Dice 1: 2
Dice 2: 4
Dice Sum  6

```

```
C:\fortran> dice
Dice 1: 1
Dice 2: 6
Dice Sum  7

C:\fortran>
```

The dice values will be different for each execution.

## 24 Appendix D – Intrinsic Functions

---

The following is a partial listing of the Fortran 95/2003/2008 intrinsic functions. Only the most common intrinsic functions are included in this section. A complete list can be found on-line at the GNU Fortran documentation web page.

### 24.1 Conversion Functions

The following table provides a list of intrinsic functions that can be used for conversion between different data types.

Function	Description
INT(A)	Returns the integer value of real argument A, truncating (real part) towards zero.
NINT(X)	Return the nearest integer value (with appropriate rounding up or down) of the real argument X.
REAL(A)	Returns the real value of integer argument A.

### 24.2 Integer Functions

The following table provides a list of intrinsic functions that can be used for integers.

Function	Description
ABS(A)	Returns the integer absolute value of integer argument A.
MOD(R1,R2)	Return the integer remainder of integer argument R1 divided by integer argument R2.

## 24.3 Real Functions

The following table provides a list of intrinsic functions that can be used for reals.

Function	Description
ABS(A)	Returns the real absolute value of real argument A.
ACOS(W)	Returns the real inverse cosine of real argument W in radians.
ASIN(W)	Returns the real inverse sine of real argument W in radians.
ATAN(X)	Returns the real inverse tangent of real argument X in radians.
COS(W)	Returns the real cosine of real argument W in radians.
LOG(W)	Returns the real natural logarithm of real argument W. Real argument W must be positive.
MOD(R1,R2)	Return the real remainder of real argument R1 divided by real argument R2.
SIN(W)	Returns the real sine of real argument W in radians.
SQRT(W)	Returns the real square root of real argument W. Real argument W must be positive.
TAN(X)	Returns the real tangent of real argument X in radians.

## 24.4 Character Functions

The following table provides a list of intrinsic functions that can be used for characters/strings.

Function	Description
ACHAR(I)	Returns the character represented by integer argument I based on the ASCII table (Appendix A). Integer argument I must be between 1 and 127.
IACHAR(C)	Returns the integer value of the character argument C represented by ASCII table (Appendix A).
LEN(STR)	Returns the integer value representing the length of string argument STR.
LEN_TRIM(STR)	Returns the integer value representing the length of string argument STR excluding any trailing spaces.
LGE(STR1,STR2)	Returns the logical true, if $STR1 \geq STR2$ and false otherwise.
LGT(STR1,STR2)	Returns the logical true, if $STR1 > STR2$ and false otherwise.

LLE(STR1,STR2)	Returns the logical true, if STR1 $\leq$ STR2 and false otherwise.
LLT(STR1,STR2)	Returns the logical true, if STR1 $>$ STR2 and false otherwise.
TRIM(STR)	Returns the string based on the string argument STR with any trailing spaces removed.
ADJUSTL(STR)	Return a string modified by removing leading spaces. Spaces are inserted at the end of the string as needed.
ADJUSTR(STR)	Return a string modified by removing trailing spaces. Spaces are inserted at the beginning of the string as needed.

## 24.5 Complex Functions

The following table provides a list of intrinsic functions that can be used for complex numbers.

Function	Description
AIMAG(Z)	Returns the real value of the imaginary part of the complex argument Z.
CMPLX(X,Y)	Returns the complex value with real argument X and the real part and real argument Y as the imaginary part.
REAL(A)	Returns the real value of the real part of the complex argument Z.

## 24.6 Array Functions

The following table provides a list of intrinsic functions that can be used for arrays.

Function	Description
MAXLOC(A1)	Returns the integer location or index of the maximum value in array A1.
MAXVAL(A1)	Returns the maximum value in array A1. Type of value returned is based on the type of the argument array A1.
MINLOC(A1)	Returns the integer location or index of the minimum value in array A1.
MINVAL(A1)	Returns the minimum value in array A1. Type of value returned is based on the type of the argument array A1.
SUM(A1)	Returns the sum of values in array A1. Type of value returned is based on the type of the argument array A1.

## 24.7 System Information Functions

The following table provides a list of intrinsic functions that obtain information from the system.

Function	Description
COMMAND_ARGUMENT_COUNT()	Returns the number of command line arguments.
GET_COMMAND_ARGUMENT(NUMBER, VALUE, LENGTH, STATUS)	<p>Returns the command line arguments, if any.</p> <ul style="list-style-type: none"> <li>NUMBER, integer argument of the number to return. Must be between 1 and COMMAND_ARGUMENT_COUNT().</li> <li>VALUE, character(*), N<sup>th</sup> argument</li> <li>LENGTH, integer, length of argument returned in VALUE</li> <li>STATUS, integer, status, 0=success and -1=VALUE character array is too small for argument, other values=retrieval failed</li> </ul>
CPU_TIME(TIME)	Returns the amount of CPU time expended on the current program in seconds. TIME is return as a real value.
DATE_AND_TIME (DATE, TIME, ZONE, VALUES)	<p>Return the date and time.</p> <ul style="list-style-type: none"> <li>DATE(), character(8), string in the form YYYYMMDD, where YYYY is year, MM is month, and DD is date.</li> <li>TIME(), character(10), string in the form HHMMSS.SSS where HH is hour, MM is minute, SS is second, and SSS is millisecond.</li> <li>ZONE(), character(5), string in the form of ±HHMM, where HHMM is the time difference between local time and Coordination Universal Time.</li> <li>VALUES(), integer array where <ul style="list-style-type: none"> <li>VALUES(1) → year</li> <li>VALUES(2) → month (1-12)</li> <li>VALUES(3) → date (1-31)</li> <li>VALUES(4) → time zone difference (minutes)</li> <li>VALUES(5) → hour (0-23)</li> <li>VALUES(6) → minutes</li> <li>VALUES(7) → seconds (0-59)</li> <li>VALUES(8) → milliseconds (0-999)</li> </ul> </li> </ul> <p>Each argument is optional, but at least one argument must be included.</p>

## 25 Appendix E – Visualization with GNUplot

---

The Fortran language does not have any built-in graphics capabilities. To support some basic data visualization a plotting application, GNUplot, can be used. GNUplot is a free, open source plotting program that can plot data files and display user-defined functions.

This appendix provides a very brief summary of some of the basic GNUplot functions as applied to plotting data from simple programs in this text. An general example program to plot a simple function is provided for reference.

### 25.1 Obtaining GNUplot

The first step is to obtain and install GNUplot.

GNUplot is available at,

**`http://www.gnuplot.info/`**

To ensure that GNUplot is installed correctly, for Windows machine, you can enter command prompt and type,

**`wgnuplot`**

which will start GNUplot by opening a new window. To exit, type **`exit`** at the prompt.

Complete documentation, tutorials, and examples are available at that site. Additionally, there are many other web sites dedicated to GNUplot.

### 25.2 Formatting Plot Files

Since our specific use of GNUplot will involve plotting a data file created with a Fortran program, it will be necessary to provide some information and directions for GNUplot in the file. That information is provided in a header (first few lines) and a footer (last few lines). As such, the program must first write the header, write the data, typically in the form of data points to be plotted, and write a final footer. The header and footer may be very different based on what is being plotted.

### 25.2.1 Header

The header will provides some guidelines on how the output should look, including title (if any), axes (if any), labels (if any), and plotting color(s). Additionally, comments can be included with a “#” character. Comments are useful to provide information about the contents of the plot file or nature of the data being plotted. A typical header might be as follows:

```
# Example Plot File
set title "CS 117 Plot Function"
plot "-" notitle with dots linewidth 2 linecolor 2
```

Once the header is written, a series of data points can be written.

### 25.2.2 Footer

The footer is used to formally tell GNUplot there are no more points. Additionally, the “pause” directive can be used to ensure that any plots displayed from the command line are left on the screen. A typical footer might be as follows:

```
end
pause -1
```

Nothing after the footer will be read by GNUplot.

## 25.3 Plotting Files

In order to display a plot file on a Windows machine, at the command prompt, type,

```
wgnuplot file.plt
```

which will start GNUplot and instruct GNUplot to read the file “file.plt”. The file name can be anything and the file extension is not required to be “.plt”.

If the header or footer commands are incorrect or the data points are invalid, nothing will be displayed. To investigate, the file can be opened with a text editor.

## 25.4 Example

This section provides an example program that plots a simple function.

$$y = \sin(x) * \left( \frac{1 - \cos(x)}{3.0} \right)$$

The program, data file, and final output are presented for reference.



### 25.4.1 Plot Program

The following is an example program that generates points, opens a plot file, write the header, data points, and footer.

```

program plotExample

implicit none
real, dimension(200) :: x, y
integer :: i, opnstat

! Generate x and y points
do i = 1, 200
    x(i) = i * 0.05
    y(i) = sin(x(i)) * (1-cos(x(i)/3.0))
end do

! Output data to a file
open (12, file="data.plt", status="replace",      &
      action="write", position="rewind",      &
      iostat=opnstat)
if (opnstat > 0) stop "error opening plot file."

! Write header
write (12, '(a)') "# Example Plot File"
write (12, '(a)') "set title ""Example Plot Function"" "
write (12, '(a,a)') "plot ""-"" notitle with dots ",
                    "linewidth 2 linecolor 2"

! Write points to file.
do i=1,100
    write (12,*) x(i), y(i)
end do

! Write footer and close file
write (12, '(a)') "end"
write (12, '(a)') "pause -1"

close(12)

end program plotExample

```

The data file name can be changed or read from the user.

### 25.4.2 Plot File

The output file from this program appears as follows:

```
# Example Plot File
set title "Example Plot Function"
plot "-" notitle with dots linewidth 2 linecolor 2
  5.0000001E-02  6.9413913E-06
  0.1000000      5.5457884E-05
  0.1500000      1.8675877E-04
  0.2000000      4.4132397E-04
  0.2500000      8.5854460E-04

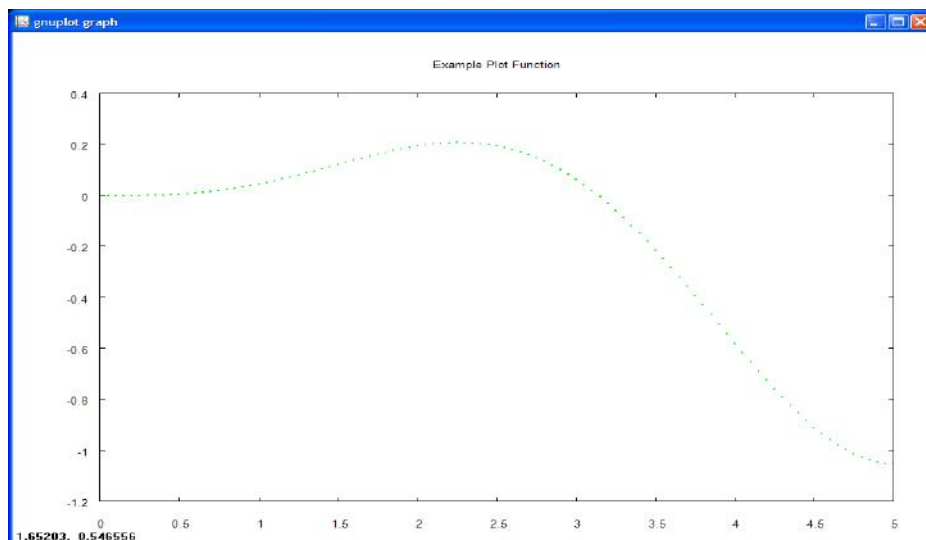
      [many points not displayed due to space considerations]

  4.900000      -1.043852
  4.950000      -1.048801
  5.000000      -1.050716
end
pause -1
```

The output file can be edited with a standard text editor. This will allow checking the data file for possible errors if it does not display correctly.

### 25.4.3 Plot Output

The plot output is as follows:



On Windows machines, the plot can be printed by right clicking on the blue bar (on top). This will display a menu, with “Print” as one of the options.

## 26 Appendix F – Quiz Question Answers

---

This appendix provides answers for the quiz questions in each chapter.

### 26.1 Quiz Question Answers, Chapter 1

There are no quiz questions in Chapter 1.

### 26.2 Quiz Question Answers, Chapter 2

Quiz question answers for chapter 2 are as follows:

- 1) In the computer, information is represented in binary.
- 2) Convert the Fortran program into binary or machine language.
- 3) The BUS connects the memory to the CPU.
- 4) The answer are as follows:
  - a) The binary value  $0000101_2$  is 5 in decimal.
  - b) The binary value  $0001001_2$  is 9 in decimal.
  - c) The binary value  $0001101_2$  is 13 in decimal.
  - d) The binary value  $0010101_2$  is 21 in decimal.
- 5) Characters are represented in binary using binary ASCII. Refer to Appendix A for the ASCII table.
- 6) Programs are stored on the secondary storage device (SSD, disk drive, or other storage media).
- 7) Program must be in primary storage or RAM in order to execute.

### 26.3 Quiz Question Answers, Chapter 3

Quiz question answers for chapter 3 are as follows:

- 1) The input file for the compiler is the Fortran program file.
- 2) The output from the compiler is the executable program.
- 3) Fortran programs must start with the **program <name>** statement and end with the **end program <name>** statement.
- 4) Comments are marked with an explanation point (!).
- 5) The typical file extension for a Fortran 95/2003/2008 program is “.f95”.
- 6) The typical file extension of the compiler output file is “.exe” for Windows and no extension for Unix based machines (i.e., MAC and Ubuntu).

## 26.4 Quiz Question Answers, Chapter 4

Quiz question answers for chapter 4 are as follows:

- 1) The five Fortran data types are **integer**, **real**, **character**, **logical**, and **complex**.
- 2) A Fortran variable name must start with a letter. Variable names may include numbers but must start with a letter.
- 3) The implied data types are integer, real, integer, real, and real.
- 4) The statements are:

```
integer :: value  
real :: count
```

- 5) The statements are:

```
real :: rate = 7.5
```

- 6) The statements are:

```
real, parameter :: e = 2.71828183
```

## 26.5 Quiz Question Answers, Chapter 5

Quiz question answers for chapter 5 are as follows:

- 1) The assignment operator is = (equal sign).
- 2) The exponentiation operator is \*\* (double asterisks, no space).
- 3) An integer variable can be converted to a real with the real conversion function. For example,

```
realVar = real(intVar)
```

- 4) An integer variable can be converted to a real with the real conversion function. For example,

```
intVar = int(realVar)
```

*Note*, in this example precision may be lost since the fractional part is truncated (not rounded).

- 5) The two logical constants are **.true.** and **.false.** which must include the leading and trailing periods.
- 6) Some intrinsic functions are **real()**, **int()**, **nint()**, **cos()**, **sin()**, **mod()**, **tan()**, and **sqrt()**. There are more as listed in Appendix D.
- 7) The statements are as follows:

```
x1 = (pi / 6.0) * (3.0 * a**2 + 3.0 * b**2 + c**2)  
x2 = -(2.0 * a / c) * cos(b) * sin(b)  
x3 = (-b + sqrt(b**2 - 4.0 * a * c)) / (2.0 * a)
```

## 26.6 Quiz Question Answers, Chapter 6

Quiz question answers for chapter 6 are as follows:

- 1) The “( \*,\* )” means to send it to the screen in 'free format'.
- 2) The statement is as follows:

```
write ( *,* ) "Programming is Fun!"
```

- 3) The statements are as follows:

```
integer :: myage  
write ( *,* ) "Enter Age:"  
read ( *,* ) myage
```

## 26.7 Quiz Question Answers, Chapter 7

Quiz question answers for chapter 7 are as follows:

- 1) The four program development steps are
  1. Understand the Problem
  2. Create the Algorithm
  3. Implement the Program
  4. Test/Debug the Program
- 2) The three types of errors are compiler error, run-time error, and logic error.
- 3) Since the formula is incorrect, that would be a logic error.
- 4) A compiler error is generated when the compiler does not understand the statement. This often occurs when statements are mis-spelled. For example,

```
writte(*,*) "Opps"
```

would generate a compiler error since **write** is mis-spelled.

## 26.8 Quiz Question Answers, Chapter 8

Quiz question answers for chapter 8 are as follows:

- 1) The six relational operators are **>**, **>=**, **<**, **<=**, **==**, and **/=**.
- 2) The three basic logical operators are **.and.**, **.or.**, and **.not.** where the leading and trailing periods are required.
- 3) The answers are **.true.**, **.true.**, **.false.**, **.true.**, **.true.**, **.false.**, and **.false.** where the leading and trailing periods are required.

- 4) The statements are as follows:

```
if (lives <= 0) then
    write (*,*) "Game Over"
end if
```

*Note*, this also could be done as follows:

```
if (lives <= 0) write (*,*) "Game Over"
```

- 5) The statements are as follows:

```
if (num < 0) then
    num = abs(num)
    write (*,*) "Variable num was made positive"
end if
```

- 6) The statements are as follows:

```
if (y /= 0) then
    z = x / y
else
    z = 0
    write (*,*) "Z not calculated"
end if
```

*Note*, another correct solution is as follows:

```
if (y == 0) then
    z = 0
    write (*,*) "Z not calculated"
else
    z = x / y
end if
```

- 7) The statements are as follows:

```
if (x <= 0.0) then
    f = x**2 * y
else
    f = x * y
end if
```

## 26.9 Quiz Question Answers, Chapter 9

Quiz question answers for chapter 9 are as follows:

- 1) When an *exit* statement is executed, the current loop will be exited, thus not completing any remaining iterations.
- 2) There may be an unlimited number of exit statements. Typically, there is only one.
- 3) When a *cycle* statement is executed, the remaining statements in the loop are skipped and the

next iteration of the loop is started (from the beginning of the loop).

- 4) There may be an unlimited number of *cycle* statements. Typically, there is only one.
- 5) If there are multiple *cycle* statements, only the first executed *cycle* statement will be executed.
- 6) The output is as follows:

**The SUM is:            15**

- 7) The output is as follows:

```

start
1      *      1      =      1
1      *      2      =      2
2      *      1      =      2
2      *      2      =      4
3      *      1      =      3
3      *      2      =      6
end

```

- 8) The statements are valid, however since the initial value of *i* is greater than the stop value, no statements in the loop will be executed.
- 9) The statements are valid, however since the initial value of *i* is greater than the stop value, no statements in the loop will be executed.
- 10) There is no specified limit.
- 11) When nesting IF statements, the nested IF statement must be completely nested within the loop.

## 26.10 Quiz Question Answers, Chapter 10

Quiz question answers for chapter 10 are as follows:

- 1) The format specifiers are **rIw**, **rFw.d**, **rLw**, **nX**, **/**, and **rA**.
- 2) The output, using an “\_” (underscore) for blanks, is as follows:

```

Hello
Hello_World

```

- 3) The output, using an “\_” (underscore) for blanks, is as follows:

```

__005

```

- 4) The statement is as follows:

```

write (*,'(i3)') num1

```

- 5) The statement is as follows:

```

write (*,'(f7.5)') pi

```

- 6) The statement is as follows:

```
write (*,'(a/,a)') "Programming", "Is Fun!"
```

- 7) The statement is as follows:

```
write (*,'(a)', advance="no") "Enter Number:"
```

## 26.11 Quiz Question Answers, Chapter 11

Quiz question answers for chapter 11 are as follows:

- 1) The declaration is as follows:

```
character (len=12) :: msg = "Hello World!"
```

- 2) The results are **.false., .true., .false., .false., .true., .false., .true.** where the leading and trailing periods are required.
- 3) The value for `astr1` = “abc”, the value for `astr2` = “456”, the value for `astr3` = “12345678910”, and the value for `astr4` = “DEF123bc”.
- 4) The integer value can be obtained by using the `IACHAR()` function.
- 5) The character can be obtained from the integer value by using the `ACHAR()` function.

## 26.12 Quiz Question Answers, Chapter 12

Quiz question answers for chapter 12 are as follows:

- 1) Before a file can be read or written, it must be opened.
- 2) The recommended range for a file unit number is between 10 and 99 (inclusive).
- 3) The answers as follows:
- a) The name of the file is **file.txt**.
  - b) The unit number used is **14**.
  - c) Yes, the error message will be printed if the file does not exist.
  - d) If the status variable, ***opnstat***, is  $> 0$ , an error on the file open has occurred.
- 4) The read statement is as follows:

```
read (20,'(13x,i2,8x,i3,8x,i5)') num1, num2, num3
```

## 26.13 Quiz Question Answers, Chapter 13

Quiz question answers for chapter 13 are as follows:

- 1) An array is considered a direct access structure since any array value can be directly accessed (without accessing any other locations).



- 2) Yes, an array can hold integer values.
- 3) Yes, an array can hold real values.
- 4) The declarations are as follows:

```
integer, parameter :: SIZE1 = 100
real, dimension(10) :: rvalues
integer, dimension(SIZE1) :: inums
real, dimension(0:9) :: counts
```

- 5) The answers as follows:
  - a) 10.
  - b) 25.
  - c) 20.
  - d) 20.
- 6) An array can be allocated a compile-time or run-time (but not both).
- 7) The answers are as follows:
  - a) The array will contain the following values:

	<b>nums</b>
1	1
2	99
3	3
4	99
5	0

- b) The statements is called an implied do loop.
- c) The statement will display the first 5 values on one line as follows:

```
__1_99__3_99__0
```

## 26.14 Quiz Question Answers, Chapter 14

Quiz question answers for chapter 14 are as follows:

- 1) No, a multi-dimensional array can hold either real values or integers values.
- 2) The answer is **a**.
- 3) The answers are as follows:
  - a) 15.

b) The array contains the following values:

	1	2	3
1	2.0	3.0	4.0
2	3.0	4.0	5.0
3	4.0	5.0	6.0
4	5.0	6.0	7.0
5	6.0	7.0	8.0

c) 3.0.

d) 4.0

e) 7.0

4) An unsuccessful allocation can be detected by checking the status variable for a value  $> 0$ .

## 26.15 Quiz Question Answers, Chapter 15

Quiz question answers for chapter 15 are as follows:

- 1) The two types of Fortran subprograms are subroutines and functions.
- 2) A function returns a single value.
- 3) The variables in the call are referred to as actual arguments.
- 4) The variables in the function heading are referred to formal arguments.
- 5) The return type is integer.
- 6) Yes, it possible to pass integer arguments to a real function.
- 7) The keyword **contains**.
- 8) The answers are as follows:
  - a) 8.
  - b) 9.
- 9) The answers are as follows:
  - a) fahrToCelsius.
  - b) Yes.
  - c) No.
  - d) 10.0.
- 10) The term *variable scope* refers to where a given variable can be accessed.
- 11) The answers are as follows:
  - a) **in**

- b) **out**
- c) **inout**

12) The term side-effect is when a function changes one or more of its input arguments. In general, this is considered poor practice and should be avoided.

## 26.16 Quiz Question Answers, Chapter 16

Quiz question answers for chapter 16 are as follows:

- 1) An item in a derived data type is referred to as a component.
- 2) Components in a derived data type are accessed with the %.
- 3) The definition is as follows:

```
type circle
    character(20) :: name
    real :: radius
    integer :: x, y, x
end type circle
```

- 4) The declaration is as follows:

```
type(circle) :: ring1, ring2
```

- 5) The definition is as follows:

```
type planet
    character(15) :: name
    real :: radius, volume
end type planet

type(planet) :: earth, mars
```

- 6) The definition is as follows:

```
type date
    character(10) :: monthname
    integer :: month, day, year
end type date
```

- 7) The answers are as follows:

- a) The statements are as follows:

```
type(date) :: today

today%monthname = "December"
today%month = 12
today%date = 25
today%year = 2013
```

b) The statements are as follows:

```
type(date) :: newyear

newyear%monthname = "January"
newyear%month = 1
newyear%date = 1
newyear%year = 2011
```

## 26.17 Quiz Question Answers, Chapter 17

Quiz question answers for chapter 17 are as follows:

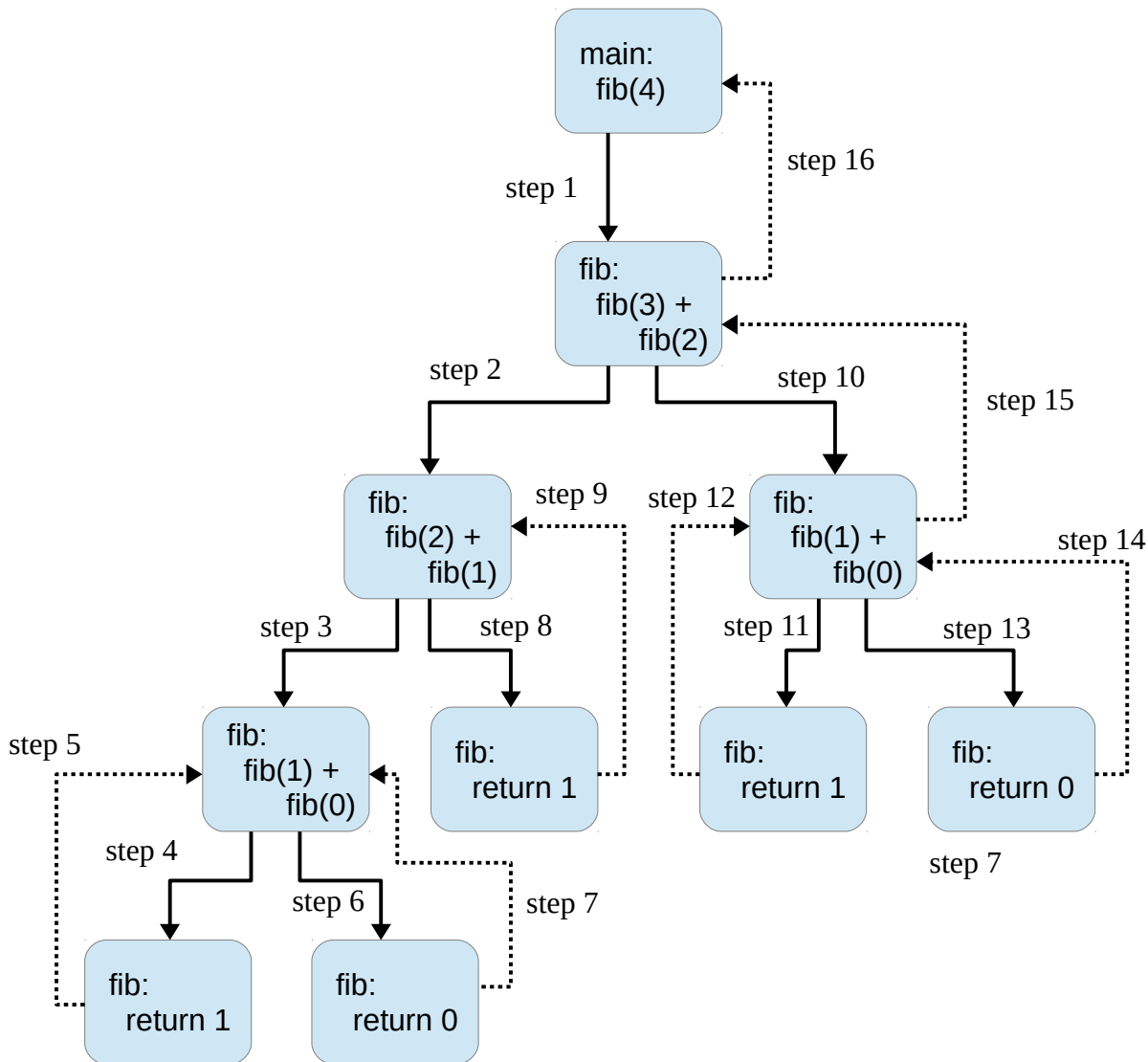
- 1) The primary purpose for using a module is to allow the program to be split into multiple source files.
- 2) The **use <moduleName>** statement must be used in the main program.
- 3) The **module <moduleName>** statement must be used in the main program.
- 4) Only one main program is allowed.
- 5) An unlimited number of modules is allowed.
- 6) Yes, the **contains** statement is required in the module.

## 26.18 Quiz Question Answers, Chapter 18

Quiz question answers for chapter 18 are as follows:

- 1) The two requirements for a recursive definition are a base case and rule, or set of rules, that reduce toward the base case.
- 2) In recursion, the case for which a solution is obtained directly is called the base case.
- 3) The keyword **recursive** is required for a recursive subroutine.
- 4) The two keywords required for a recursive function are **recursive** and **result**.
- 5) There are no special requirements needed in the calling routine in order to call a recursive subroutine or recursive function.
- 6) If a recursive routine does not stop recursing, it will recurse indefinitely, thus hanging the program and producing no results. The program would need to be manually terminated.

7) A complete recursion tree for the recursive Fibonacci function with an input of 4 is as follows:



## 26.19 Quiz Question Answers, Chapter 19

Quiz question answers for chapter 19 are as follows:

- 1) An *internal read* is required to convert an character string contain a numeric value.
- 2) An *internal write* is required to convert a real value into a character string.
- 3) The statement to convert the character string `sNum`="123" into an integer variable `iNum` is as follows:

```
read (sNum, '()', iostat=cvtErr) iNum
```

- 4) The statement to convert the integer variable *iNum*=234 into a character string *sNum* is as follows:

```
write (sNum, '()', iostat=cvtErr) iNum
```

## 26.20 Quiz Question Answers, Chapter 20

Quiz question answers for chapter 20 are as follows:

- 1) Yes, the date alone be obtained (i.e., without obtaining the time) using the `get_date_time()` system service.
- 2) The answers are as follows:
  - a) 2.
  - b) 5.
  - c) 1.
- 3) The hours can be obtained from the minutes by dividing by 60 and the minutes obtained by `mod(hours,60)`.

- 4) The statement is as follows:

```
call get_date_time(date=dStr)
```

- 5) The statement is as follows:

```
call get_date_time(time=timeStr, zone=zoneStr)
```

- 6) The statement is as follows:

```
call get_date_time(values=valuesArr)
```

- 7) If there are no arguments entered on the command line, the `get_command_argument()` system service call will return a 0.

- 8) The statement is as follows:

```
call get_command_argument(number=3, value=arg3,      &  
    length=arg3length, status=arg3stat)
```

## 27 Appendix G – Fortran 95/2003/2008 Keywords

In programming, a keyword is a word or identifier that has a special Fortran 95/2003/2008 meaning. Keywords are reserved in that they cannot be used for anything else such variable names.

The **Type** as listed in the table refers to the following:

- statement → implies a keyword that starts a statement, usually one line unless there is a continuation "&"
- construct → implies multiple lines, usually ending with "end ..."
- attribute → implies it is used in a statement to further clarify or specify additional information.

For reference, below is a partial list of keywords or reserved words. For a complete list of keywords, refer to the on-line GNU Fortran 95/2003/2008 documentation.

Keyword	Type	Meaning
<b>allocatable</b>	attribute	no space allocated here later allocate
<b>allocate</b>	statement	allocate memory space now for variable
<b>assignment</b>	attribute	means subroutine is assignment (=)
<b>backspace</b>	statement	back up one record
<b>call</b>	statement	call a subroutine
<b>case</b>	statement	used in select case structure
<b>character</b>	statement	intrinsic data type
<b>close</b>	statement	close a file
<b>complex</b>	statement	intrinsic data type
<b>contains</b>	statement	internal subroutines and functions follow
<b>cycle</b>	statement	continue the next iteration of a do loop (skipping the subsequent statements)
<b>deallocate</b>	statement	free up storage used by specified variable
<b>default</b>	statement	in a select case structure - all others
<b>do</b>	construct	start a do loop
<b>else</b>	construct	part of if, else if, else, end if
<b>else if</b>	construct	part of if, else if, else, end if
<b>elsewhere</b>	construct	part of where, elsewhere, end where
<b>end do</b>	construct	ends do loop

<b>end function</b>	construct	ends function
<b>end if</b>	construct	ends if
<b>end interface</b>	construct	ends interface
<b>end module</b>	construct	ends module
<b>end program</b>	construct	ends program
<b>end select</b>	construct	ends select case
<b>end subroutine</b>	construct	ends subroutine
<b>end type</b>	construct	ends type
<b>end where</b>	construct	ends where
<b>endfile</b>	statement	mark the end of a file
<b>exit</b>	statement	continue execution outside of a do loop
<b>format</b>	statement	defines a format
<b>function</b>	construct	starts the definition of a function
<b>if</b>	statement and construct	if(...) statement
<b>implicit</b>	statement	"none" is preferred to help find errors
<b>in</b>	a keyword for intent	the argument is read only
<b>inout</b>	a keyword for intent	the argument is read/write
<b>integer</b>	statement	intrinsic data type
<b>intent</b>	attribute	intent(in) or intent(out) or intent(inout)
<b>interface</b>	construct	begins an interface definition
<b>intrinsic</b>	statement	says that following names are intrinsic
<b>inquire</b>	statement	get the status of a unit
<b>kind</b>	attribute	sets the kind of the following variables
<b>len</b>	attribute	sets the length of a character string
<b>logical</b>	statement	intrinsic data type
<b>module</b>	construct	beginning of a module definition
<b>namelist</b>	statement	defines a namelist of input/output
<b>nullify</b>	statement	nullify a pointer
<b>only</b>	attribute	restrict what comes from a module
<b>open</b>	statement	open or create a file
<b>operator</b>	attribute	indicates function is an operator like +
<b>optional</b>	attribute	a parameter or argument is optional



<b>out</b>	a keyword for intent	the argument will be written
<b>print</b>	statement	performs output to screen
<b>pointer</b>	attribute	defined the variable as a pointer alias
<b>private</b>	statement and attribute	in a module
<b>program</b>	construct	start of a main program
<b>public</b>	statement and attribute	in a module - visible outside
<b>read</b>	statement	performs input
<b>real</b>	statement	intrinsic data type
<b>recursive</b>	attribute	allows functions and derived type recursion
<b>result</b>	attribute	allows naming of function result
<b>return</b>	statement	returns from exits subroutine or function
<b>rewind</b>	statement	move read or write position to beginning
<b>select case</b>	construct	start of a case construct
<b>stop</b>	statement	terminate execution of the main procedure
<b>subroutine</b>	construct	start of a subroutine definition
<b>target</b>	attribute	allows a variable to take a pointer alias
<b>then</b>	construct	part of if construct
<b>type</b>	construct	start of user defined type
<b>use</b>	statement	brings in a module
<b>where</b>	construct	conditional assignment
<b>while</b>	construct	a while form of a do loop
<b>write</b>	statement	performs output



# Index

---

ABS(A).....	199p.	Character String / Numeric Conversions.....	169
Accessing Array Elements.....	102, 115	CMPLX(X,Y).....	201
Accessing Components.....	138	Command line arguments.....	175, 178
ACHAR(I).....	86, 200	Command line options.....	178
ACOS(W).....	200	Command_argument_count().....	179
Actual arguments.....	124	COMMAND_ARGUMENT_COUNT().....	202
Addition.....	23	Compiler.....	4
ADJUSTL(STR).....	87, 201	Compiler Error.....	40
ADJUSTR(STR).....	87, 201	Compiling.....	9
Advance clause.....	75	Complex constant.....	22
AIMAG(Z).....	201	Complex Literals.....	22
Allocate.....	101	Complex number.....	15
American Standard Code for Information		Component.....	137
Interchange.....	5	Computer program.....	2
Argument count.....	179	Conditional Controlled Looping.....	63
Argument Intent.....	124	Conditional expression.....	43
Argument Passing.....	125	Conditional Expressions.....	43
Arguments.....	124	Conditionally Controlled Loop Example.....	65
Array Declaration.....	100, 113	Constant.....	17
ASCII.....	5	Contains.....	124
ASIN(W).....	200	Coordinated Universal Time.....	175
Assignment.....	23	COS(W).....	200
ATAN(X).....	200	Counter Controlled Example.....	61
Base 10.....	4	Counter Controlled Looping.....	59
Base 2.....	5	CPU_TIME(TIME).....	202
Binary digit.....	5	Cycle statement.....	61
Binary Numbers.....	5	Date.....	175
Bit.....	5	DATE_AND_TIME.....	202
Boolean.....	15	Decimal Numbers.....	4
Byte.....	5	Declaration.....	138
Call statement.....	125	Declarations, Extended Size Variables.....	18
CASE statement.....	50	Definition.....	137
Central Processing Unit.....	3	Derived Data Types.....	137
Chaos Game.....	98	Direct access structure.....	99
Character.....	15	Discriminant.....	47
Character constant.....	22	Disk drive.....	3
Character format specifier.....	74	Division.....	24
Character Literals.....	22	Do-loop.....	59
Character Representation.....	5	Dynamic Array Allocation.....	101, 114

Dynamic Array Declaration.....	101	Intrinsic Functions.....	26, 126
Dynamic Declaration.....	114	Iostat.....	169
E-notation.....	21	Irrational numbers.....	15
Equal to.....	43	Keyword.....	14, 219
Error Terminology.....	40	Kind.....	19
Escape character.....	22, 83	Kind specifier.....	19
Executable file.....	9	LEN_TRIM(STR).....	86, 200
Exit statement.....	61	LEN(STR).....	86, 200
Exponentiation.....	25	Less than.....	43
External.....	123	Less than or equal.....	43
External declaration.....	124	LGE(STR1,STR2).....	200
External Routines.....	124	LGT(STR1,STR2).....	200
Factorial function.....	162	Literals.....	21
File Open.....	91	LLE(STR1,STR2).....	201
Formal arguments.....	124	LLT(STR1,STR2).....	201
FORMAT statement.....	71	LOG(W).....	200
Functions.....	125	Logic Error.....	41
Get_command_argument().....	179	Logical.....	15
GET_COMMAND_ARGUMENT(NUMBER, VALUE, LENGTH, STATUS).....	202	Logical constant.....	23
Get_date_time().....	175	Logical Constants.....	23
GNUplot.....	203	Logical format specifier.....	74
Greater than.....	43	Logical Operators.....	44
Greater than or equal.....	43	Loop.....	59
Helper function.....	163	Machine language.....	4
Horizontal positioning specifier.....	73	Magic Square.....	121
IACHAR(C).....	86, 200	MAXLOC(A1).....	201
IF statement.....	45	MAXVAL(A1).....	201
IF THEN ELSE IF statement.....	46	MAXVAL(ARR1).....	103
IF THEN ELSE statement.....	45	MINLOC(A1).....	201
IMPLICIT NONE.....	15	MINVAL(A1).....	201
Implicit typing.....	15	MINVAL(ARR1).....	103
Implied do-loop.....	103	MOD(R1,R2).....	199p.
Index.....	99	Module file.....	153
INT(A).....	199	Modules.....	151
Integer.....	14	Monte Carlo.....	116
Integer constants.....	21	Multidimensional Arrays.....	113
Integer format specifier.....	72	Multiplication.....	24
Integer Literals.....	21	NINT(X).....	199
Intent(in).....	124	Not equal to.....	43
Intent(inout).....	124	Object file.....	152
Intent(out).....	124	Operands.....	43
Interface block.....	124	Operating System.....	2
Internal.....	123	Order of Operations.....	25
Internal read.....	169	Parameter.....	17
Internal Routines.....	124	Pi estimation.....	117
Internal write.....	169	Primary Storage.....	3
		Print Binary.....	160

Program Layout.....	123	Subprograms.....	123
Program Statement.....	7	Subroutines.....	127
Pythagorean theorem.....	116	Subscript.....	99
Quadratic equation.....	47	Subtraction.....	24
Random Access Memory.....	3	SUM(A1).....	201
Random_number().....	195	SUM(ARR1).....	103
Random_seed().....	195	System Services.....	175
Rational numbers.....	15	TAN(X).....	200
Read statement.....	32	Terminal window.....	191
Real constants.....	21	Time.....	175
Real format specifier.....	73	TRIM(STR).....	87, 201
Real Literals.....	21	Type Checking.....	16
Real number2.....	15	Type declaration.....	16
REAL(A).....	201	Unit number.....	91
Recursion.....	159	Use Statement.....	152
Recursive Factorial.....	162	User-Defined Functions.....	126
Recursive relation.....	159	Using Functions and Subroutines.....	124
Recursive subroutine.....	159	Values array.....	175
Relational Operation.....	43	Variable Ranges.....	16
Relational Operator.....	43	Variable Scope.....	124
Relational Operator (alternate).....	43	Whole number.....	14
Result variable.....	162	Write statement.....	31
Run-time Error.....	40	Xcode.....	188, 192
Scope.....	124	XCode.....	190
Secondary source file.....	151	Zone.....	175
Secondary Storage.....	3	.and.....	44
SELECT CASE Statement.....	50	.eq.....	43
Selector lists.....	51	.ge.....	43
Side Effects.....	126	.gt.....	43
Side-effect.....	126	.le.....	43
SIN(W).....	200	.lt.....	43
Single Dimension Arrays.....	99	.ne.....	43
Solid-State Drive.....	3	.not.....	44
Source file.....	8	.or.....	44
SQRT(W).....	200	/=.....	43
Static Array Declaration.....	100	<.....	43
Static Declaration.....	100, 114	<=.....	43
Stop Statement.....	92	==.....	43
String.....	31	>.....	43
String5.....	15	>=.....	43
Subprogram Types.....	123		