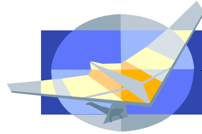


PROGRAM DESIGN
USING C++/F95/MATLAB

BENCHAWAN WIWATANAPATAPHEE
Mahidol University · Thailand

YONG HONG WU
Curtin University of Technology · Australia

Misterkopy Publishing Company
Bangkok

Published by
Misterkopy Publishing Company
Bangkok, Thailand

National Library of Thailand Cataloging in Publication Data

Benchawan Wiwatanapataphee
Program design using C++/F95/Matlab
1. Programming languages
2. Computer programming
I. Title
II. Yong Hong Wu, joint author
005.133
ISBN: 974-94652-7-X

Program Design Using C++/F95/Matlab

Copyright © 2006 by B. Wiwatanapataphee & Y.H. Wu

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, without the prior written permission of the copyright owners

Printed in Thailand by Misterkopy Publishing Company

ISBN : 974-94652-7-X

PREFACE

Over the last few decades, hundreds of high-level computer languages have been developed. However, only a few have achieved broad acceptance. C++, Fortran and MATLAB are among those widely used programming languages. C++ has been used extensively in many computer applications and today most computer operating systems are written in C or C++. Fortran always emphases on computing efficiency and accuracy and today the latest version of Fortran is still one of the dominant programming languages for scientific and engineering applications that require complex numerical computations, while MATLAB is a high level software package with rich collections of built-in functions for mathematical calculations and graphic display of results.

In many computer applications, more than one computer languages are used. For example, in some applications, Fortran is used to implement the computation part, while C++ is used to create user interface and to display computation results graphically. In some other applications, a C++ program or a Fortran program is run to generate numerical results, while the MATLAB package is used to post-process the results and to display the results in graphic form. MATLAB itself can also be used for both computing and graphic display of results. Therefore, it is useful to have a book which covers the essential elements of the three widely used programming languages C++, F95 and MATLAB. With this thought in mind, this book is written with particular emphasis on the fundamental of program design using C++, F95 and MATLAB for scientific computing.

This book is divided into three parts: C++ (chapters 1-8), F95 (chapters 9-15) and MATLAB (chapter 16). All the three parts, in particular the C++ and F95, are written and organized in similar styles and structures. Thus, once you have completely command one of the three programming languages, you will find pretty easy to learn the other two parts. Another feature of the book is that each of the three parts is written independently, and so one could start learning from any of the three parts.

The book was written based on authors' experience in teaching scientific computing and undertaking research in mathematical modelling and numerical simulation. Our experience in teaching the subject and using the programming languages in our research has been built into the book through the design and organization of the contents and the writing of each chapter, section and paragraph. Our aim is twofold, to provide a text which is easy to learn and covers all elements essential to scientific computing, and to provide a quick reference for those who use C++/F95/MATLAB in their research and work.

The book can be used as a text for the first course in scientific computing for first year or second year undergraduate students in particular mathematics and science students. Depending on the specific course requirements, various options may be adopted. The first option is to cover all the three parts by conducting a full semester course with around 45 hours of teaching. The second option is to focus on the teaching of C++ (chapters 1-8) and give only a brief introduction of the other two parts. The third option is to emphasise on the teaching of F95 (chapter 1 plus chapters 9-15) and give only a brief introduction of the other two parts.

Here we take the second part as an example to discuss how to organize the teaching. The second part consists of 7 chapters and can be taught by 21 hours of lecture delivery. The following table is our suggested teaching plan together with the focus for each of the topics.

Hours of Lecture	**Chapter**	**Topics**	**Focus**
1-3	9	Introduction and arithmetic computation	Basic concepts, basic I/O and assignment statements, design simple programs
4-6	10	Control structures	Logical expressions, selection structures, repetition structures, design programs with control structures
7-9	11	Precision control & data type	Determine and use kind type parameter, data types, Design programs with required precision
10-12	12	Formatted I/O, file operation	Control the format of output, Read data from data files
13-15	13	Array processing	Array declaration, whole array operations, Design programs with vector & matrix calculation capacity
16-18	14	Procedures and program design	Define & reference function& subroutines, use of modulus, interface, Design structure of large programs for complex problems & implementation
19-21	15	F95 pointers, dynamic memory allocation	Use of pointers, memory allocation, Design programs with high efficiency and low memory requirement

With the aim of providing a quick reference book for those who use C++/F95/MATLAB in their work, the book has been structured entirely based on the need of mathematical calculations and the key points for each topic are presented with a style which is precise, simple and easy to read. Hence, one can find the required information quickly from the table of contents.

The following two websites may have information for free or trial version of C++ and F95 compilers

http://www.bloodshed.net
http://www.silverfrost.com

B. Wiwatanapataphee & Yong Hong Wu
September 2006

TABLE OF CONTENTS

PREFACE

PART I C++

1 Introduction **3**

1.1 Some Basic Concepts about Computers 3
1.2 Procedure for Solving Mathematical Problems Using Computers 4

2 C++ Arithmetic Computations **8**

2.1 Constants and Variables 8
2.2 Assignment Statements 12
2.3. Stream Input/Output 17
2.4 Initial Values and Named Constants 20
2.5 Construction of a Simple C++ Program 20

3 C++ Control Structures **27**

3.1 Logical Expressions and Calculations 27
3.2 Selection Control 29
3.3 Loop Control 33

4 C++ Functions **45**

4.1 Top-Down Design using Functions 45
4.2 Library Functions and User Defined Functions 45
4.3 Defining a Function 46
4.4 Calling a Function 48
4.5 Recursive Functions 49
4.6 Function Overloading 50
4.7 Storage Classes and Scope 51
4.8 Construction of Projects with Multiple Source Files 51

5 C++ Array Processing 58

5.1 One-Dimensional Arrays 58
5.2 Multi-Dimensional Arrays 59
5.3 Array Operations 61
5.4 Passing Arrays to Functions 63

6 C++ Pointers 68

6.1 Declaration and Initialization of Pointer Variables 68
6.2 Pointer Operators 69
6.3 Function Pointers – Passing a function to another 69
6.4 Passing Arguments to Functions by Reference with Pointer 70
6.5 Pointer Arithmetic 73
6.6 Pointers and Arrays 73
6.7 Dynamic Memory Allocation to Arrays 74

7 C++ File Operations 78

7.1 Include Header Files 78
7.2 Create/Open a Sequential file 78
7.3 Writing Data to and Reading Data from a Sequential File 80
7.4 Application: Calculation of Dominant Eigen Values of Matrices 83

8 C++ Classes and Object-Oriented Program Design 89

8.1 Introduction 89
8.2 Defining a class 89
8.3 Calling Member Functions 91
8.4 Improve Reusability of Classes 94

PART II F95

9 F95 Arithmetic Computations 101

9.1 Constants and Variables 101
9.2 Assignment Statements 104
9.3 Simple Input/Output Statements 107
9.4 Initial Values and Named Constants 108
9.5 Construction of a Complete F95 Program 110

10 F95 Control Structures 116

10.1 Logical Expressions and Logical Variables 116
10.2 Selection Control 118
10.3 Loop Control 123
10.4 Application: Newton's Method for Solving Nonlinear Equations 128

11 F95 Additional Data Types 136

11.1 Control of Precision - Kind Type Parameter 136
11.2 Double Precision Data 139
11.3 Complex Data 140
11.4 Character Data 141
11.5 Application: Composite Simpson's 1/3 Rule for Evaluating Integrals 143

12 F95 Formatted I/O and Files 149

12.1 Formatted Output 149
12.2 Formatted Input 153
12.3 Additional Format Features 155
12.4 File Operations 156
12.5 Application: Runge-Kutta Methods for Solving Initial Value Problems 159

13 F95 Array Processing 165

13.1 One-Dimensional Arrays 165
13.2 Two-Dimensional Arrays 167
13.3 Multi-Dimensional Arrays 168
13.4 Array Operations 168
13.5 Allocatable Arrays 173
13.6 Application: Solution of Tridiagonal Systems of Equations 175

14 F95 Program Design and Subprograms 183

14.1 Top-Down Design: Programs and Subprograms 183
14.2 Function Subprograms 183
14.3 Subroutine Subprograms 185
14.4 Modules 187
14.5 Modules and Explicit Procedure Interfaces 188
14.6 More about Procedures 189
14.7 Application: Solution **of** Linear Systems of Equations by Permuted LU Methods 193

15 F95 Pointers and Dynamic Memory Allocation 202

15.1 Basic Concepts 202
15.2 Using pointers in expressions 205
15.3 Pointers and Arrays 207
15.4 Pointer Arrays as Argument to Procedures 210

PART III MATLAB

16 MATLAB Computation and Graphics 215

16.1 MATLAB Getting Start 215
16.2 MATLAB Arithmetic Computations 217
16.3 MATLAB Control Structures 224
16.4 MATLAB Matrix and Array Calculation 227
16.5 MATLAB M-files : Scripts and Functions 229
16.6 MATLAB Graphics 231

REFERENCES 237

Index 239

Part I C++

C++, evolved from C, was developed in early 1980s. It is initially widely used as the development language of the Unix operating system and today most operating systems are written in C or C++.

C++ programs consist of classes and functions. Each of the functions or classes is used to perform a specific task. Today, there are huge collections of classes and functions available in the C++ standard library, which are usually provided by the compiler vendors. One can program each function/class needed to form a C++ program using the C++ programming language. One can also make use of the existing classes and functions to perform some tasks such as graphic display of results in constructing a complete C++ program.

In this part, you will learn how to use C++ programming language to write a complete C++ program for scientific computing.

CHAPTER 1

INTRODUCTION

1.1 Some Basic Concepts about Computers

A computer is a collection of electronic circuits and devices with ability to remember a sequence of instructions and to obey these instructions at a predetermined point in time. Such a sequence of instructions, written in a computer language, is called a *program*.

Although we do not need to know exactly how a computer works in order to use it, it is useful to create a conceptual model of a computer system, which will enable us to understand more easily and exactly what we are doing when we write a program.

A computer system mainly consists of a *central processing unit* (CPU), *external memory* (such as tapes and disks), *input device* (such as keyboard) and *output device* (such as printer and monitor), as shown in Fig.1.1. Each part is likely to be physically distinguishable from the others in large computer systems. In a microcomputer or minicomputer, all the parts in CPU may be combined in a single, integrated circuit chip.

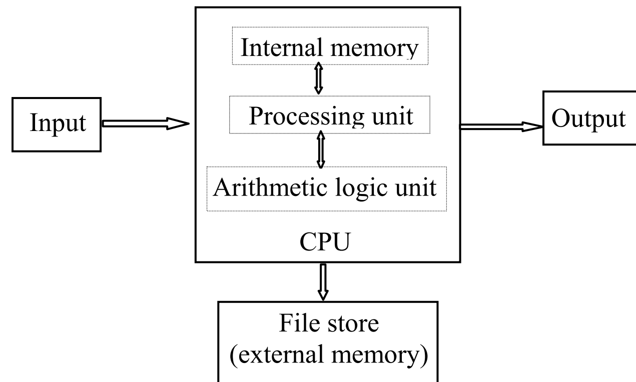


Fig 1.1

The processing unit or the processor is the part of the computer that controls all the other parts. The processor accepts input values and stores them in the memory. It also interprets the instructions in a computer program. If we want to add two values, the processor will retrieve them from the memory (either internal or external) and send them to the arithmetic logic unit (ALU), the ALU performs the desired addition, and the processor then stores the result in the memory. We may also direct (in the program) the processor to print the result on paper or store in external memory such as tapes or disks. The processing unit, internal memory and ALU are collectively called the central processing unit (CPU). Thus, typically, a microprocessor refers to a CPU, but a microcomputer refers to a CPU with input/output capabilities

Memory refers to the device for storing information. There are two main types of information stored in memory, namely a program (instructions which the computer is to obey) and data (values which the computer is to process). Memory includes internal and external memory. Internal memory is built as part of the CPU. When the power is switched off, information stored in internal memory is lost and thus internal memory is of no use for storage of information other than during the running of a program. External memory such as tapes and disks is for large and/or long-term data storage. However, the speed of transfer of information between external memory and the central processor is much times slower than that between the internal memory and the processor.

We may visualize computer memory as a set of boxes; each can store a number, or a word or any other single item that we may wish to store. To distinguish one box from another, each has a label attached with an identifying name. Fig.1.2 shows three boxes (*a,x,p*) containing three different items. Clearly, *a, x* and *p* are the names of the boxes and not their contents. If we want to find out (or use) what is stored in a particular location, we just need to refer to the location name. If we store a new value in some location, then whatever was stored there is destroyed and lost.

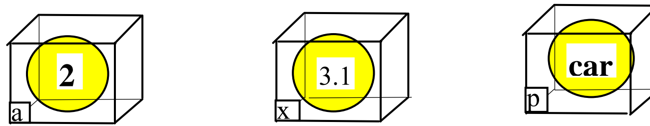


Figure 1.2

When working with computers, you will often hear the terms software and hardware. Software refers to the programs that direct computers to perform operations, compute new values and manipulate data. Hardware refers to the physical components of the computer such as the memory unit, the processor and the ALU.

1.2 Procedure for Solving Mathematical Problems using Computers

Fig 1.3 summarises the procedure

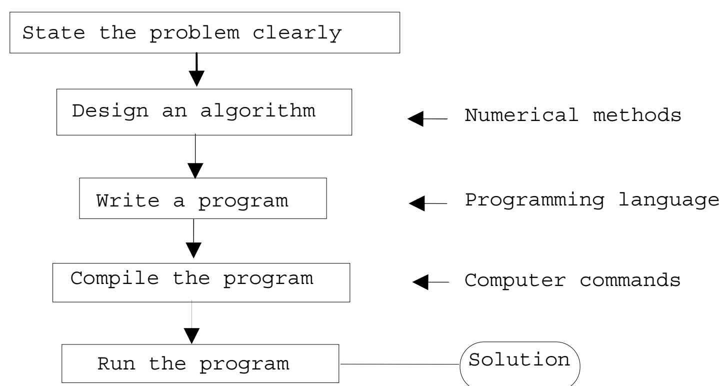


Fig1.3

1) State the Problem Clearly

It is important to give a clear, concise statement of the problem. In particular, it is extremely important to describe clearly what are the given conditions (information or data) and how the expected results are to be presented.

Eg. Given a set of exp. data x_i *, compute the average value*

2) Design an Algorithm

Def. Algorithm : *An algorithm is a sequence of steps that describe how to obtain the solution to a given problem.*

Guidelines for algorithm design

a) Use the **top-down design technique**
Top-down design is composed of two techniques, decomposition and stepwise refinement.

- We first use the Decomposition technique *to break the problem into a series of smaller problems .*

- Then, we use the Stepwise refinement technique *to describe each smaller problem in greater detail.*

The advantage of decomposition is that we can initially think of the overall steps required without getting lost in the details. Details are introduced only as we begin the refinement of our algorithm.

b) Use **flowchart** and **pseudocode** to assist in the design and presentation of algorithms
Two tools can be used to assist us in designing and presenting an algorithm, namely
Pseudocode - *show the steps in a series of English - like language.*
Flowchart - *show the steps in graphic form.*

c) Use **structured algorithm**

- To improve the readability, we should use a set of standard forms (structures) to describe an algorithm. There are three kinds of standard algorithm structures (which will be studied in detail in chapters 2-3)
  i) Sequence structures
  ii) Selection structures
  iii) Repetition structures

- An algorithm formed by a standard structure is called a **structured algorithm**. When a structured algorithm is converted to a computer program, the computer program is called a **structured program**.

Example1.2-1. *Given a set of experiment data x_i, compute the average value*

Algorithm Design

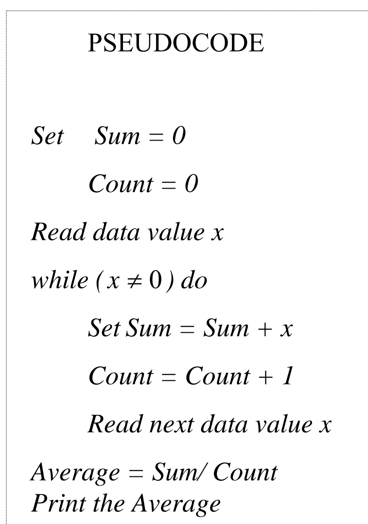
Decomposition: To compute an average, we need to sum and count the values. Thus our decomposition is

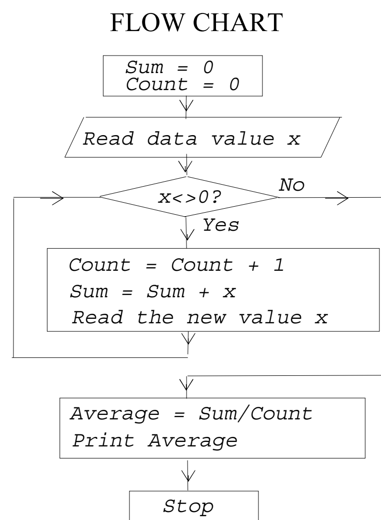
1. Read data values and keep a sum and count
2. Divide the sum by the number of data (count) to get the average
3. Print the average.

Refinement

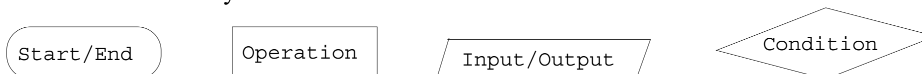
- Step 1 can be refined by setting a count and a sum to zero following by a loop to read the values and update the count and the sum
- The loop should be structured into a while loop, thus we must determine a condition necessary to keep us in loop. For example, if all data values are non-zero, we can put zero at the end of the data to indicate the end of data. So the loop is 'while the data is non zero do ....' .

Representation:





Note: List of some flow chart symbols.



3) Write a Computer Program

Once an algorithm is designed, it must be translated into a program using a standard computer language such as C++.

4) Compile a Computer Program

- Any computer understands only one language known as its machine language. This language is very difficult for humans to understand. It may also differ from machine to machine.
- Thus, normally programs are first written in a high level language (HLL) which is English-like and is much easier to use and understand. Then the HLL program is translated into a set of machine language instructions. This translation is called *compiling* and is performed by a program known as compiler.
- During compilation the source code is parsed to ensure it conforms to the correct syntax for the language. Errors found in the source code are referred to as compile-time error or syntax error.

5) Run/ Execute the Program

- If no compile time error are detected, then the machine language program can be run
- Any errors detected during program execution are referred to as run-time-errors or logical errors.

EXERCISE 1

Q1.1 What are the essential parts of a computer system?
Q1.2 What do we mean by computer memory?
Q1.3 What are the differences between internal memory and external memory?
Q1.4 What do we mean by software and hardware?
Q1.5 What is the definition of algorithm? Briefly describe the top-down technique for the design of algorithms.
Q1.6 How to present an algorithm?
Q1.7 Why do we need to compile a C++ program before we can run the program?

CHAPTER 2

C++ ARITHMETIC COMPUTATIONS

Arithmetic operations (adding, subtracting, multiplying, dividing and etc) are the most fundamental operations performed by computers. To be able to write programs for these operations, we need to know how to store data values in computers, how to implement computations, how to input data values to computers and how to print the computed results. Thus in this chapter, we describe

- Methods for storing data with C++ by using constants and variables
- Assignment statements for arithmetic computations
- Simple input/output statements for introducing data values into computers and printing results
- Construction of a complete C++ program

2.1 Constants and Variables

Numbers can be introduced into a program by either direct use with constants or indirect use with variables.

eg. To calculate the area of a circle of radius r, we can use the following program

```
#include <iostream.h>
int main ()
{
  float r, area;
  cin>>r;
  area=3.14159*r*r;
  cout<<"area="<<area<<endl;
}
```

where 3.14159 is used as a constant;
r is a variable used for storing the value of radius.

Remarks:

(1) The first line `#include <iostream.h>`: is to notify the pre-processor to include the *input-output stream header file* in the program. This statement must be used for any program that reads data from keyboard and print results onto screen.

(2) Every C++ program must have a main function that has the structure of the form:

```
int main ( ){
      c++ statements
}
```

(3) When the 4th statement is executed, the computer will wait for the entry of a value from keyboard. The value, once entered, will be assigned to `r` and execution continues to calculate the area and then the computed result is printed. The input from keyboard and the output are as follows

Input from key board:
```
2.5
```
Output on screen:
```
Area=19.634937
```

a) Constants

- Constants are numbers used directly in C++ statements, such as 3.14159, 2, -2.5 etc. Constants may contain plus or minus signs and decimal points, but they may not contain commas.
- In C++, there are 4 categories of basic intrinsic data types.

Integer data

Integer numbers are whole numbers with or without sign, for examples: 5, -5
Integer constants cannot contain decimal points and commas, Eg, 5. 12,000 are not valid integers. Integer constants are always held exactly in computer memory.

There are four types of integer variables for storing integer data:

- *`int`* (basic integer type)
- *`short int`* (short integer)
- *`long int`* (long integer)
- *`unsigned short int`* / *`unsigned long int`*

The size of each type of integer data depends on the computer used. In general,
`long int` variables can store more digits than *`int`*,
`int` variables can store more digits than *`short int`*,
`unsigned` variables can store only *`unsigned`* data values (cannot store negative values)

Real data

Real numbers are numbers with decimal points. Real constants can be represented in two forms

Decimal form : -12.3, 0.0, etc.
Scientific form : 10.3e8, -5.2e-10 etc. (representing 10.3×10^8 and -5.2×10^{-10})
Exponent must be integer, eg. 3.0e5.6 is invalid real constant.

Real numbers are stored in computers in scientific form, eg. -0.135782123e-5

-	- 0 5	1 3 5 7 8 2 1 2 3

There are limitations on the magnitude and precision of values that can be stored in a computer. All limitations on values depend on the specific computer.

There are four types of real variables for storing real data:

- `float` (single precision)
- `double` (double `precision`)
- `long double`
- `unsigned float`/ `unsigned double` / `unsigned long double`

In general, `float` variables store data with 7 effective digits
`double` variables store data with 15-16 effective digits.

Character data

Character constant has only one character and is always enclosed in single apostrophe, eg. ‘a’, ‘D’, and ‘c’.

There are a number of special character constants which are delineated by a backslash \.

Character	Function
\n	go to next line
\f	go to next page
\’	a single quote

C++ also allows to have character string constant which is enclose in double apostrophe, eg “velocity” and “Australia” are valid C++ character string constants.

A `char` type variable can be used to store one character. To store a character string, one has to use arrays with each element storing one character

Boolean data

Boolean data includes integer numbers 0 and 1, or the C++ literals `true` and `false` which are implicitly promoted to the integers 0 and 1 whenever an arithmetic value is necessary. `bool` type variables can be used to store the integer numbers 0(`false`) or 1(`true`).

b) Variables

A variable represents a memory location that is assigned a name. The memory location can be used to store a value. Once we need the value stored, we reference it with the variable name assigned to the memory location. We can also store a new value in the memory location and in this case, the old value is destroyed and lost.

To store different types of values, we need to use different types of variables. All variables must be declared with a name and a data type before they can be used.

```
type name_variable1, name_variable2, …, name_variableN;
```

In the following, we describe how to name a variable and how to declare the type of a variable in computer programs.

Name A C++ variable name must obey the rules which apply to all C++ names, namely

- It **must begin with a letter (a-z, A-Z)** or an underscore_.
- It may be optionally followed by up to 31 more characters.
- It may only contain the letters a-z and A-Z, the digits 0-9 and the underscore.
- The name is case sensitive.

In C++ , there are various keywords (such as `if`, `do`, `class`, `for`, etc.) which are reserved for certain specific use and should not be used as variable names.

Example: 2X, V.2 and X$ are all not valid C++ variable names.
a, b, x, y, velocity, name_1 are all valid C++ names.

***Type*:** The *type* specifies the data type for which memory space is to be reserved. Variable data types include

```
int,    short int,  long int,
unsigned short int, unsigned long int,
float,  double,    long double,
unsigned float,    unsigned double, unsigned long double,
char and
bool.
```

eg.

`int k, s, d;`	declares *k* , *s* and *d* as `integer` variables.
`long int n1, n2;`	declare *n1* and *n2* as `long integer` variables.
`unsigned int m;`	declares *m* as an `unsigned integer` variable.
`float i, n, x;`	declares *i*, *n*, and *x* as real variables (single precision)
`double x1, x2;`	declares *x1* and *x2* as real variables (double precision)
`char c1, c2;`	declares *c1* and *c2* as character variables .

Note: * In C++, variable declarations can be placed anywhere in a program but they must appear before the variables are used.

* If you omit to declare a variable, it will lead to a syntax error.

2.2. Assignment Statements

There are only two ways in which a variable can be given a value during the execution of a program - by assignment or by a read statement. We will discuss the assignment statement here.

- Arithmetic computations (`+, -, *, /, %`) can be implemented in C++ by using assignment statements.
- General form: `Variable_name = expression;`
- Execution: once an assignment statement is executed, the following two processes occur in the computer
  1) first, calculate the value of the expression
  2) then assign the value to the variable on LHS.

eg.

`x=2.0;`	assigns 2.0 to x
`y=x+2.5;`	evaluates (x+2.5) to yield 4.5, then the value 4.5 is assigned to y
`y=y+1;`	y will be assigned a value equal to its current value plus 1. Thus y will become 5.5.

Remarks (1) Arithmetic expression on the LHS, eg: `x +1= y;` is not allowed.

(2) When a variable appears in both sides of the assignment operator =, we can use short-hand writing,

eg:

$$\begin{array}{ccc} x = x + y; & \Leftrightarrow & x += y; \\ x = x - y; & \Leftrightarrow & x -= y; \\ x = x * y; & \Leftrightarrow & x *= y; \\ x = x / y; & \Leftrightarrow & x /= y; \\ x = x\%y; & \Leftrightarrow & x\% = y; \end{array}$$

where $+=$ is called *addition assignment* ;
$-=$ is called *subtraction assignment, etc.*

(3) To know how to design program to perform arithmetic computing by using assignment statements, we need to know how to translate mathematical formulae to arithmetic expressions and how an expression is evaluated in computers.

(4) The "expression" on the right hand side can also be a assignment statement.

eg 1. a=(b=5);

In this statement, there are two assignment operators. In C++, the operation order of assignment operators is from right to left. Hence, `b=5` will be executed first to yield a value 5 , which is then assigned to `a` and so `a` has the value 5.
eg 2. a=(b=4)+(c=6);

after execution, b has value 4, c has value 6, a has value 10.

2.2.1 Writing Arithmetic Expressions

An expression is a combination of constants, variables, intrinsic functions, operators and parentheses which can be evaluated to give a single value, eg.

2+x (2+x+sin(y))/2.0

where x and y denote variables which have been assigned values previously.

a) Intrinsic (Library) Functions

- Scientific computing usually requires many simple operations such as calculating the sine of an angle. As these operations are so common, they are built as standard functions in header files and we can use them directly in the program. The following is a list of some common functions

Function Name	***Definition***
$fabs(x)$	$\|x\|$
sqrt(x)	$\sqrt{x},\ x \geq 0$
pow(x, y)	x^y
exp(x)	e^x
sin(x)	*Sine of x*
cos(x)	*Cosine of x*
tan(x)	*Tangent of x*
asin(x)	*Arcsin x*
acos(x)	*Arccos x*
atan(x)	*Arctangent of x*
log(x)	*Natural logarithm of x, x>0*
log10(x)	*Base 10 logarithm of x*

Note. To use the above functions, the mathematics header file `<cmath>` (or <math.h> in some C++ compiler) must be included in the program by

```
#include <cmath>
```

- An intrinsic function can be referenced in an expression using the following form :

```
Function_Name (argument, ...);
```

eg.

```
y=sqrt(b*b-4.0*a*c)+1.0 ;
z=sin(x)+2*cos(x);
c=exp(2.5);
```

Notes: 1) The argument can be a constant, a variable or an expression, eg. sqrt(3.0), sqrt(2+x).

2) Some functions require a particular type of input and return a particular type of value, which can be found from most C++ books

b) Arithmetic Operators

Arithmetic calculations can be performed by using the following operators

Operation	Operator	C++ expression	Algebraic expression
Addition	+	`a+b`	a+b
Subtraction	-	`a-b`	a-b
Multiplication	*	`a*b`	$a \times b$
Division	/	`a / b`	$\frac{a}{b}$
Modulus	%	`r%s`	r mod s
Preincrement	++	++i	Increment i by 1 and then the new i value is used in the expression.
Predecrement	--	--i	Decrement i by 1 and then the new i value is used in the expression.
Postincrement	++	i++	The current i value is used in the expression, then i is incremented by 1.
Postdecrement	--	i--	The current i value is used in the expression, then i is decremented by 1.

Eg: `23%5` yields 3 as $23/5 = 4 \times 5 + 3$.

For i=2, j= 2*(++i) yields j=6 and i becomes 3 after execution of the statement.

For i=2, j= 2*(i++) yields j=4 and i becomes 3 after execution of the statement.

2.2.2 Evaluating Arithmetic Expressions

An expression can be evaluated to yield a single value. In order to translate a mathematical formula correctly to a C++ expression to yield an expected value, we need to know the following points.

a) Priorities of Operations

Because several operations can be combined in one arithmetic expression, it is important to know the priorities of the operations (the order in which the operations are performed).

C++ assigns the same priorities to operators as does mathematics, as shown in the following table.

C++ operations	C++ operators	Priorities	
Operations in brackets	( )	Inner most	highest
Increment/Decrement	++ --	Inner most	↓
Intrinsic functions		Left to right	
Multiplication/Division/Modulus	* / %	Left to right	
Addition/Subtraction	+ -	Left to right	lowest

Eg.1 For a=1.0, b=2.0, c=0.5

```
x=(-b + sqrt(pow(b,2) - 4.0*a*c))/(2.0*a);
```
⇒ x =1.0

C+= also provides another operator, comma operator, which connects expressions together as follows

Expression 1, expression 2, ..., expression n

The execution order is expression 1, then expression 2 and so on, and the value of the whole expression is the value of the *expression n*.

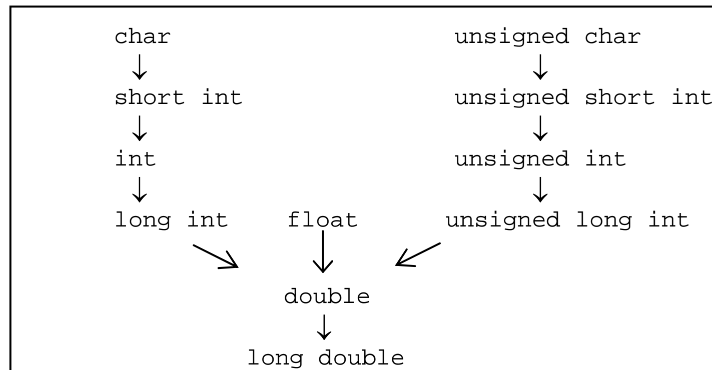
Eg. In the following statement

x=(a=3, 6*5)

x is assigned a value of 30 (the value of the comma expression).

b) Mixed-Mode Operations

When an arithmetic operation is performed using two real numbers, the intermediate result is a real value. If the operands are not of the same type, we have the so-called mix-mode operation. In this case, certain conversion of the data types will be performed before the arithmetic operation. The following diagram shows the conversion direction of data types (from lower level to higher level).



In general, the operand at lower level will be converted to the same level as the other one.

eg.

- For the operation of a *short int* and a *long int*, the *short int* number will be converted to *long int* first.
- If an *int* number is to be added to a *float* number, then both are converted to *double* type first.
- If an *int* type number is to operate with an *unsigned long* type number, then both are converted to *double* type number first.

- C++ also allows transform of data type of expressions by using the transformation operator ( ) via statement of the form (type) expression

  eg.

```
#include <iostream.h>
int main ()
{
 float x;
 int y;
 x=8.69;
 y=(int)x;
 cout <<"x="<<x<<endl;
 cout <<"y="<<y;
}
```

In the above program, `(int)x` transforms the expression to integer and hence the outputs are as follows

```
x=8.69
y=8
```

c) Truncation Error

When a computer stores a real number in an integer variable, it ignores the fractional portion and stores only the whole number portion of the real number, this lost is called truncation. In C++, truncation errors usually arise from the following two sources:

- ***Integer_variable = expression;***

  Once the assignment statement is executed, the expression is first evaluated. If the value of the expression is real, then the real value is to be stored into the integer variable. As an integer variable can only store the whole number part, the fraction portion of the real value is ignored.

  eg. Suppose that *K* is an integer variable, the statement

  ```
  K=3.14*2;
  ```

  assigns an integer value 6 (not 6.28) to *K* after the statement is executed.

- ***Integer Division***

  As the intermediate result of integer division is integer, the fraction portion of the quotient is truncated.

  eg. In computer programs, 3/4 yields 0 instead of 0.75;
  1/2 yields 0 instead of 0.50 and 6/5 yields 1.

  Assuming that *l, n, j* are all integer variables, then for *l*=2, *n*=3, *j*=4,
  l/n**j* yields 0 ;
  but *j***l*/*n* yields 2.

d) Magnitude Limitations, Under Flow and Over Flow

- Every computer has limitations on the maximum & minimum magnitudes of values it can store. This information usually can be obtained from the reference manual of the computer.
- If the magnitude of a value obtained in calculation is smaller than the minimum magnitude which the computer can recognize, an error called underflow error occurs. If the magnitude of a value obtained is larger than the maximum value, overflow error occurs. If these errors occur, firstly check the algorithm for other mistakes.

2.3 Stream Input/Output

2.3.1 Stream Input

A program can read data values from the default input device (usually keyboard) with statement

```
cin >> arg1  >> arg2 ... >> argN;
```

where arguments `arg1 - argN` are the names of variables for which data are to be stored,

"`cin`" is a predefined object; and
"`>>`" is called extraction operator.

eg: `cin >> A >> B;`

* When this statement is executed, the computer first waits for the entry of two values from the default input unit such as keyboard.
* Once two values are entered, the first value is assigned to A and the second to B.

Notes:

* The variables must be separated by "`>>`".
* The input data should be separated by space. If the variable is integer, the corresponding input data should also be integer.
* Each read statement will read as many lines as needed to find new values for the variables.

  eg. `cin>>X>>Y>>Z;` if the input data is $\begin{cases} 1.5 \quad 2.5 \\ 3 \end{cases}$

  then the values stored in variables are X=1.5, Y=2.5, Z=3.

2.3.2 Stream Output

A program can print results stored in `arg1 - argN` to the default output device (using monitor) by using the statement

```
cout << arg1 <<  ... << argN;
```

where arguments `arg1 - argN` are either variables whose values are to be displayed, strings or control characters.

eg.

```
float x, y, average;
x=2;
y=3;
average=(x+y)/2;
cout<<"x="<<x <<" y="<<y <<" (x+y)/2=" <<average << endl;
```

will print

```
x=2    y=3    (x+y)/2=2.5
```

Notes:

* Each expression should be separated by "<<".
* *endl* is the control character that causes the output to move to the next line.

.

2.3.3 More on Stream I/O (format output)

The format of input and output data can be controlled by using the stream manipulators defined in the header file "iomanip". The commonly used stream manipulators are as follows:

Stream manipulators	Function
fixed or setiosflags(ios::fixed)	Floating point display in fixed-point notation.
scientific or setiosflags(ios::scientific)	Floating point display in scientific notation.
setw(n)	Set the total field width for the value to n
setprecision(n)	Set n digits to the right of the decimal point for the data in the default floating output.
left or setiosflags(ios::left)	Left justify output in a field.
right or setiosflags(ios::right)	Right justify output in a field.

Notes:

1. The following statement must be used at the beginning of the program in order to use the above commands.

```
#include  <iomanip>
```

2. A call to the manipulators sets the format for all subsequent output operations until the next manipulator call.

Eg.

```
#include  <iostream>
#include <iosmanip>
void main ( )
{
     double  a = 22.0/7;
     cout << a  << endl;
     cout << setprecision(1) << a  << endl
          << setprecision(2) << a  << endl
          << setprecision(3) << a  << endl;
     cout << setiosflags(ios : : fixed)
```

```
        << setprecision(8) << a << endl;
    cout << setiosflags(ios : : scientific)
        << a << endl;
}
```

Output:

```
3.14286
3
3.1
3.14
3.14285714
3.1428571
```

2.4 Initial Values and Named Constants

Initial Values

In addition to read statements, there is one other method of giving a value to a variable, namely to provide an initial value for a variable as part of the declaration of the variable. This is achieved quite simply by following the name of the variable by an equal sign and the initial value:

```
float a=0.0, b=1.5,c,d,e=1e-6;
int max=100;
```

These initial values will be assigned to the variables by the C++ processor before the execution of the program commences, thus avoiding the need, when the program is executed, either to execute a series of initial assignments or to read an initial set of values.

Named Constants

Frequently, a program will use certain constant values in many places and there is clearly no intention for these values to be altered. C++ allows us a convenient method of dealing with these situations by defining what is called named constant in a declaration statement:

```
const float Pi=3.14159;
const int MAX_Iter=100;
```

A name constant can also be defined by using a define statement, for example

```
#define Pi=3.14159
```

Once a named constant is defined, it is not permitted to attempt to change its value at a subsequent point in the program. The only way that its value can be changed is by modifying the declaration statement accordingly and recompiling the program.

2.5 Construction of a Simple C++ Program

The task of writing a program to solve a particular problem can be broken down into four basic steps.

(1) specify the problem clearly;

(2) analyse the problem, break it down into fundamental elements and then draw up a program design plan (use flow chart or pseudocode to present the plan) ;

(3) code the program according to the plan developed at step 2.

There is also a fourth step which is often the most difficult one of all.

(4) Test the program exhaustively and repeat steps 2 and 3 as necessary until the program works correctly in all situations that you can envisage.

A C++ program is a collection of comments, declarations, a main function, other functions and class definitions. The following figure shows a simple C++ program for reading two real numbers, calculating their sum and printing the results.

```
1   // ******* E2Q1  ********
2   //
3   #include <iostream>
4   using namespace std;
5
6   int main()
7   {
8      float  a,b, absum;
9      cout <<  "input two numbers : \n";
10     cin >> a >> b;
11     absum = a+b;
12     cout <<  "a+b=" <<absum << endl;
13     return 0;
14  }
```

Notes

(1) Lines 1-2 are comment lines. A comment line begins with `//` indicating that the rest of the line is a comment which is for documentation purpose and does not cause the computer to perform any action. A long comment requiring several lines could begin with `/*` and end with */.

(2) Line 3 (`#include <iostream>`) is a preprocessor directive. Lines begin with # are processed by the preprocessor before the program is computed. The line notifies the preprocessor to include the I/O stream header files <iostream> in the program. This header

file is needed for any program that use stream I/O to display data on the screen and read data from the keyboard. Other header files are needed for more complicated programs.

(3) All the elements of the standard C++ library are declared within what is called a namespace with the name *std*. So in order to access its functionality we declare with the expression "using namespace std;" that we will be using these entities.

(4) Line 5 is a blank line which is to make the program easier to read and is ignored by the complier.

(5) Lines 6-15 define a main function. Every C++ program must contain a main function and may also contain some other functions and classes. Once a C++ program is run, it begins executing at the main function regardless its location within the source codes.

(6) A C++ function consists of a function header ("int main()" in this example) followed by a function body enclosed within a left brace { (line 7) and a right brace } (line 14) . The function body is a collection of C++ codes including comment lines, declarations, assignment statements, stream I/O statements and etc. The "return 0; " statement is one of the means to exit a function.

(7) A C++ statement ends with a semicolon (;).

SUMMARY

In this chapter, we learn how to define constants and variables in C++. We discuss how to perform arithmetic operations using assignment statements. Some of the considerations that are unique to computer computations were also discussed: mixed-mode operations, truncation errors, magnitude limitations, underflow and overflow. Statements for reading data from the default input device (keyboard) and for printing answers to screen were also covered. The structure of a complete C++ program is also described with an example.

C++ Syntax Introduced in Chapter Two:

Preprocessor directive	#include <iostream>
Variable declaration	*float* list_of_variable_names; *int* list_of_ variable_ names;
Initial value specification	*type* variable*_name=initial_value*;
Name constant definition	const *type* constant*_name=constant_value;*
Assignment statement	*variable_name=expression;*
Stream input	cin >> arg1 >> arg2 >>... >> argN;
Stream output	cout << arg1 << arg2 <<... << argN;
Arithmetic Operators	+, -, *, / , %
Intrinsic functions	abs, sqrt, exp, pow, sin, cos, asin, tan, atan, log, log10 etc.

End statement	semicolon **;**

Other Key Points

Truncation errors	caused by assigning real values to integer variables and by integer division
Structure of C++ program	see section 2.5

EXERCISE 2

Q2.1. What is the difference between an integer and a real number ? (see section 2.1)

Q2.2 List two advantages of a real variable over an integer variable ? (see sect 2.1)

Q2.3. Write each of the following real constants in scientific form (+0.**** E+**).

12.0, $0.126*10^{-13}$, 3.08, $6.023*10^{23}$, 18900000, -41800

(Ans: 0.1200E+2, 0.1260E-13, 0.3080E+1, ...)

Q2.4. Which of the following are not valid symbolic names of C++ variables? Why?

Area, 2numbers, N/4, A.B, X_C, A15BB, VELOCITY

Q2.5. What is the general form of an assignment statement ? (sect 2.2)

Q2.6. What are C++'s basic arithmetic operators? what are their respectively priorities?

Q2.7. Convert the following formulae into C++ assignment statements (sect 2.2)

(a) $y = \dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$, (b) $z = \dfrac{6\sin(x+y)}{e^{3+a}}$, (c) $z = \sin^{-1}\left(\dfrac{y}{\sqrt{x^2 + y^2}}\right)$

(Ans: (a) y=(-b + sqrt (b*b - 4*a*c)) / (2*a); ....)

Q2.8. What will be printed out by the following program (work it out by hand). (sections 2.2-2.3)

```
#include <iostream>
using namespace std;
int main()
{
   float a, b, p, q, r;
   int  x, y, z;
   a=2.5;  b=4.0;  p=a+b;  x=a+b;
   q=a*b;  y=a*b;  r=p/q;  z=x/y;
```

```
    cout << p<< q<< r<< x<< y<< z<<endl;
    return 0;
}
```

Q2.9. How to assigned an initial value to a variable in a TYPE statement? Give two examples.

Q2.10. How to declare a name constant? Give two examples.

Programming Exercises

Program Debugging - Some guidelines (for questions Q2.11 - Q2.16)

If a program is not working correctly, you should consider taking some of the following steps to isolate the errors.

(1) Check the input potion of your program

- Check the syntax of the read statement.
- Check whether correct values have been assigned to variables in the cin statement. For this purpose, you need to add a cout statement immediately after the cin statement so that from the computer output , you can see whether the values you want to give the variables are being correctly assigned to the variables or not. A common mistake is to enter the data values in the wrong order.

(2) Check the assignment statement

- Double check the placement of parentheses. Be sure that you always have the same number of left parentheses as right parentheses.
- Review each variable name on the right-hand side of the assignment statement to be sure you have spelled it exactly as previously used.
- Make sure all variables on the right-hand side of the assignment statement have been previously assigned a value.
- Be sure that arguments of functions are in the correct order and correct data type. For example, trigonometric functions use angles in radians, not in degree.

(3) Check the Output

- Check the syntax of the output statement
- Do you have the correct variable names listed ?

Q2.11 Run the programs of Q2.8 in computer. If the printed results are different from your solutions worked out by hand, find the mistake you made in your hand calculation and think about why.

To perform computation in computers, you need to

1) Create a C++ file (*Q2_11*.cpp):
   Enter the program (read Section 2.5 about the layout of C++ program)
   Save the program and quit to the command mode.
2) Compile the Program Q2_11.cpp to obtain an executable file
   If syntax errors are detected, you need to check and correct the errors before going to next step.

Remarks: In order for the compiler to understand a C++ program, the program must be written using the correct C++ grammar, which is different to the English grammar. Make sure

- *The order of statements must be correct*
- *Each statement must follow the correct syntax*

3) Run the program
followed by data values required by the cin statements (if there is any) in the program.

Remarks: Once a `cin` statement in the program is executed, the computer will wait for the entry of data values from the input device such as keyboard before executing the next statement. Hence, after running the program, you need to enter data values for the variables listed in the first `cin` statement, then press the ENTER key. Then enter data values for the variables listed in the next cin statement, and so on.

Q2.12. The following simple program contains a number of errors. Identify these errors and then produce a corrected version.

```
// this program contains a number of errors  and  is not a good example of C++

#include <iostream>
using namespace std;
int main ()
{
    float  number;
    cout <<  “type a number : ”;
    cin  >> “number”;
    cout <<  “thank you,  your number is ” << number << endl;
    return 0;
}
```

Run the program on your computer to check that it does indeed work. If it still does not work, then keep correcting it until it does.

Q2.13 Enter the following program exactly as shown

```
// This program contains three major errors.
#include <iostream>
using namespace std;
int main ()
{
   cout <<  please type a number;
   cin >>  number;
   cout << "the number you typed was " << numbr <<endl;
      return 0;
}
```

The program contains source errors, compile the original program, correct only those errors detected by the compiler. Then run it again typing in the value 268 when requested. Was

the answer that was printed correct? If not, why not? How could you improve the program so that the compiler found more of the errors?

Q2.14 Write a program that expects three numbers (two real and one integer) to be entered, but only uses one read statement, and then prints them out so that you can check that they have been input correctly. When typing in the numbers at the keyboard, try (a) typing them all on one line separated by commas or space; (b) typing them one on each separate line.

Q2.15 Write and run a program which will read 6 numbers and find their sum. Test the program with several sets of data.

Q2.16 The following program is intended to swap the values of var_1 and var_2:

```
//  PROGRAM  swap

#include <iostream>
using namespace std;
int main()
{
   float var_1=22.2, var_2=66.6;
   // Exchange values
   var_2=var_1;
   var_1=var_2;
   // Print the new values in var_1 and  var_2
   cout << var_1<< var_2<<endl;
   return 0;
}
```

The program contains an error, however, and will not print the correct values. Find the error and correct it so that it works properly.

CHAPTER
3

C++ CONTROL STRUCTURES

So far our programs have been made up of a few simple statements executed one after another. This kind of structures is called sequence structure. Obviously, in order to write useful programs, we need to be able to

* Execute some statements many times - looping or iteration (need repetition structures).
* Choose between alternative statements based upon a condition - selection (need selection structures).

The rest of the chapter is organized as follows:

Section 3.1 describes how to represent mathematics conditions in C++;
Section 3.2 introduces selection control;
Section 3.3 introduces loop control.

3.1 Logical Expressions and Calculations

Most control structures use a condition to determine which path/action to take in the structure. In C++, a condition is expressed by a logical expression. A logical expression is analogous to an arithmetic expression but is always evaluated to a value either true (1) or false (0). The simplest forms of logical expression are those expressing the relation between two numerical values, namely the relational expression. In general, a condition can be expressed by a composite logical expression which is formed by combining *relational expressions, logical constants* and *logical variables* using *logical operators.*

a) Relational Expression (R.E.)

A relational expression compares the values of two arithmetic expressions using a relational operator, namely

Arithmetic_expression_1 `Relational_Operator` Arithmetic_expression_2

eg. `a > b+1` (condition a>b+1)

- List of Relational Operators:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
= =	equal to
!=	not equal to

- The relational expression can be used to describe simple conditions such as $a > b + 1$. If the condition is true, the expression yields a value 1(true), otherwise a value 0 (false).

  Eg. `A >= 3.5` yields a value 1(true) if `A=4`.

b) Logical Constants and Logical Variables

Logical Constants:
It has been established that evaluating a logical expression will yield a logical value either 1(true) or 0(false).

Logical Variables:
We can declare logical variables to store logical values (1 or 0). Logical variables can be declared in a program using the following statement

```
bool  var_1, var_2;
```

c) Logical Operators and Logical Calculations

By combining the relational expressions together using the logical operators (! , && , | |, etc), we can form a composite logical expression to describe a complicated condition, such as 0 < x+y < 2

- **Definitions of Logical Operators**

Name	*Operator*	*Format*	*Value*
AND	&&	A && B	1(true) only if both expressions A and B are 1(true).
OR	| |	A | | B	1(true) if A or B or both of them are true.
NOT	!	! A	changes the value of the expression A to the opposite value

where A and B can be a logical constant, a logical variable or a relational expression.

- **Priorities of logical Operations**

Type	*Operator*	*Execution order*	
Bracket	( )	1	Highest priority
Arithmetic Cal.		2	↓
Relational Cal.		3	
	!	4	
Logical Cal.	&&	5	
	| |	6	Lowest

Example 3.1.1

For A=3.5, B=5.0, D=1.0 and C=2.5, evaluate (A >= 0.0) && ((A+C) > (B+D)) || !(1).
(Ans, 0 (false))

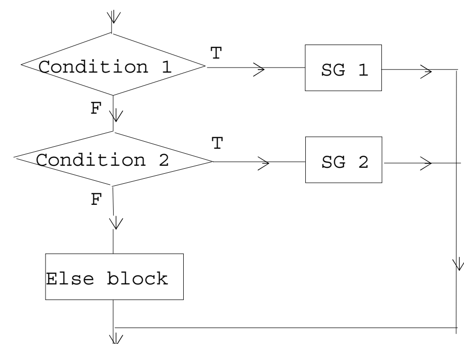
Note: The parentheses shown in the above example are not strictly necessary because the relational operators have a higher priority than logical operators, but to human eyes the inclusion of parentheses makes the true meaning of the expression much clear.

3.2 Selection Control

In practice, most problems require us to choose between alternative courses of action, depending upon circumstances that are not determined until the program is executed. The selection structure is used to choose different paths through our program. It is most commonly described in terms of a *if* construct or a *switch* construct. In the cases in which the alternatives are mutually exclusive and the order in which they are expressed is unimportant, we usually use the *switch* control or otherwise we use the *if* construct.

3.2.1 The *if* Construct

```
if  (condition 1) {
    Statement_group_1(SG1)
}
else if (condition 2) {
    Statement_group_2(SG2)
}
else {
    else_block
}
```



- A *if* construct always begins with a if statement.
- There may be any number of `else if` statements, each followed by a block of statements or there may be none. There may be one else statement followed by a block of statements or there may be none.

eg: For one way selection, we use the following `if` statement

```
if (condition) {
    if_block_SG1)
}
```

For two way selection, we can use the following `if-else` statement

```
if (condition) {
        if_block_SG1 }
else {
        else_block   }
```

- For a better appearance and readability, usually we indent the statement groups a few spaces to the right.
- The *if* construct can appear anywhere in C++ functions.

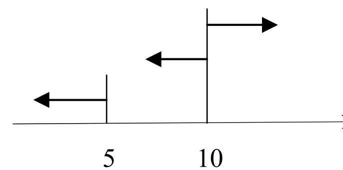
Execution Order

1) Evaluate Condition 1(C.1), if C.1 is 1(true), statement group 1 (SG1) is executed, then exit
   0(false), go to the statement with C. 2 (if there is any)
2) Evaluate C.2, if C.2 is 1(true), SG2 is executed, then exit
   0(false), go to the statement with next condition
3) If none of the conditions are 1(true) then the else block (if there is any) will be executed.

Note: *Statement_group_k* can be executed only if *condition k* is true and all other previous conditions are false.

Example 3.1.2.

Calculate $Y = \begin{cases} 2+x & \text{if } x \le 5 \\ 4.5+0.5x & \text{if } 5 < x \le 10 \\ 19.5-x & \text{if } x > 10 \end{cases}$




```
if (x  <= 5.0)
    y=2.0+x;
else if (x <=10)
    y=4.5+0.5*x;
else
    y=19.5-x;
```

← *(If this statement is executed, it means that the previous condition $x \le 5$ is not true, i.e. 5<x is true. Thus we only need to test the condition $x \le 10$)*

It should also be addressed that, in the two way selection structure, if both the `if_block` and the `else_block` has only one statement assigning different value to the same variable, it can be simplified by using a 3-operands conditional expression.

A 3-operands conditional expression has the form of

Condition ?   expression 2: expression 3

Once the above expression is executed, first the condition is evaluated. If the condition is true, then expression two will be executed and exit, or otherwise expression 3 is executed.

eg 1.

```
max = (a>b)? a: b;
```

is equivalent to the following statement

```
If (a>b) max=a;
else max=b;
```

That is, when the above 3-operands expression is executed, the expression will take the value `a` if the condition (`a>b`) is true or otherwise the value `b`, where `a` and `b` can be constants or arithmetic expressions.

eg 2.

```
cout <<  (a>b)? a : b;
```

will print the larger value of the two values.

3.2.2 The *switch* Construct

In addition to the *if* construct which caters for the ordered choice situation, C++ provides another form of selection, known as the *switch* construct, to deal with the alternative situation in which the various alternatives are mutually exclusive and the order in which they are expressed is unimportant. Its overall structure is shown as follows.

```
switch (case_expression) {
case case_selector_1: {block_1 of statements
                                break;}case
case_selecter_2: {block_2 of statements
                                break;}
                    :
default: {block_D of statements}
}
```

* There may be any number of `case` statements, each followed by a block of statements and terminated by a break statement. There may be one default statement followed by a block of statements or there may be none.

* The *case_expression* is either an integer expression, a character expression; real expressions are prohibited.
* The *case*_selector must match the *case_expression* in data type.
* When the *switch* statement is encountered, the value of *case_expression* is evaluated.
  + If this value matches the *case_value* of a case selector (say *case_selector_k*), then the block immediately after this case selector (*block_k*) will be executed and then exit from the construct.
  + If the value of the *case_expression* does not match any case values given, then the block following the default statement will be executed; if there is no default statement then an exit is made from the case construct without any code being executed.

Example 3.2.2

Write a program to read the coefficients of a quadratic equation $ax^2+bx+c=0$ and print its real roots.

Analysis: The program will use $x = \frac{1}{2a}\left(-b \pm \sqrt{b^2-4ac}\right)$.

$$\text{If } b^2-4ac \begin{cases} \geq 0, \ \exists \text{ two real distinct roots} \\ = 0, \exists \text{ two coincident roots} \\ < 0, \text{no real root} \end{cases}$$

As real arithmetic is only an approximation, we should never compare two real numbers for equality. This is because two numbers which are mathematically equal will often differ very slightly if they have been calculated in a different way. We avoid this difficulty by comparing the difference between two real numbers with a very small positive number ε. Thus we can require the cases as follows

$$b^2-4ac \begin{cases} > \varepsilon, & \exists \text{ two real distinct roots} \\ \in [-\varepsilon,\varepsilon], & \exists \text{ two coincident roots} \\ < -\varepsilon, & \text{no real root} \end{cases} \quad \text{or} \quad \text{int}\left(\frac{b^2-4ac}{\varepsilon}\right) \begin{cases} > 0, & \exists \text{ two real distinct roots} \\ = 0, & \exists \text{ two coincident roots} \\ < 0, & \text{no real root} \end{cases}$$

Hence, we can have the following **structure plan** and the corresponding **program**

```
step1,  read a, b and c
step2,  calculate d=b^2-4ac
step3,  calculate selector [int(d/ε)]
step4,  select case on  selector
          selector>0
              calculate and print two roots
          seletor =0
              calculate and print a single root
```

```
        selector <0
            print "no real root"
```

```
// PROGRAM  example3_2_2
//
#include <cstdlib>
#include <iostream>
#include <cmath>
using namespace std;
int main( )
{
    float a, b, c, d, sqrt_d, x1, x2;
    int  selector;
    // Read Coef
    cout << "please enter the coef. a, b and c\n";
    cin >> a >> b >>c;
    d=pow(b,2)-4.0*a*c;
    if (d==0.0)
        selector=0;
    else if (d<0.0)
        selector=-1;
    else
        selector=1;
    //  Calculate and print roots, if any
    switch (selector) {
        case(1): // Two roots
           sqrt_d=sqrt(d);
   x1=(-b+sqrt_d)/(a+a);
   x2=(-b-sqrt_d)/(a+a);
   cout << "two roots:"<< x1 << "and"<<x2<<endl;
        case(0): // One root
           x1=-b/(a+a);
           cout << "one root: "<<x1<<endl;
        case(-1): // No real root
           cout << "no real root\n"<<endl;
    }
    system("PAUSE");
    return 0;
}
```

3.3 Loop Control

- This section describes the C++ repetition structures that allow us to repeat certain parts of our program.

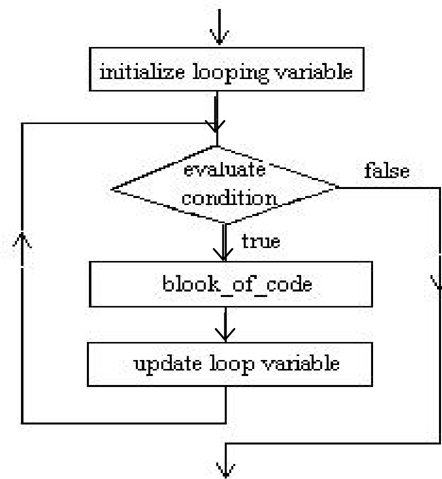
- There are three basic repetition structures: *for* loop, *while* loop and *do-while* loop.
- If the number of iterations is known or can be predetermined, we usually use a count-controlled *for* loop, otherwise a *while* loop or *do-while* loop.

3.3.1 For Loop (Do until the number of count equals to a certain value)

If the number of iterations is known or can be predetermined, we can use the *for* Loop to repeat computing.

(a) General form of the statement in C++

```
for (initialization; loop continuation condition; increment statement)
 {
    block_of_codes
 }
```



- The initialisation statement, idex=initial, is executed at the beginning of the loop, which assign an initial value to the looping variable index. The increment statement may take the form `n++` or `++n` instead of the statement `n=n+1` .
- The initialisation can be omitted. In this case the loop variable must be assigned an initial value before the `for` loop.

(b) Execution of a for loop

1) Assign initial value to loop variable index.
2) Evaluate the loop condition.
3) If the condition is false(0), then exit from the loop,

   otherwise, execute the code between the bracket{ }and then update the looping variable index.
4) Go to step 2).

Notes:

The value of index should not be modified by other statements during the execution of the loop.

Example 3.3.1: Write a C++ program segment to calculate $\sum_{i=1}^{50} i = 1 + 2 + \ldots + 50$ using for loop.

```
sum=0;
for (int count=1; count<= 50; count++)
     sum=sum+count;
```

c) Nested *for* loop

A for loop may be nested within other for loops. If loops are nested, they must use **different indexes** or loop counters. When one loop is nested within another, the inside loop is completely executed each pass through the other loops.

Example 3.2.1.2: Write a C++ program segment to compute the factorials of 4 integers.

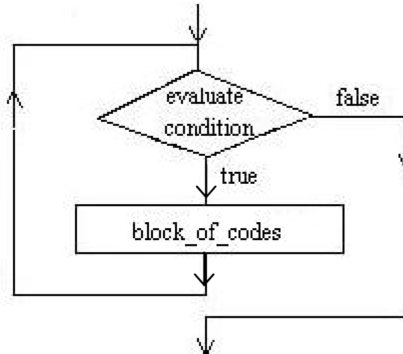
```
long int nfact;
for (int i=1; i<=4; i++){
 cin>>n;
 if (n<0)
    cout<< "invalid n=" <<n<<endl;
 else {
    nfact=1;
    if (n>1) {
       for(int k=1; k<= n; k++)
          nfact=nfact*k;
    }
    cout<< "n =" << n<< "n! =" << nfact<<endl;
 }
}
```

3.3.2 while loop (do while certain condition is true)

The *while* loop identifies a series of steps (statements) that are to be repeated while a certain condition is true. It checks condition for truthfulness before executing any of the code in the loop. If the condition is initially false, the code inside the loop will never be executed.

The C++ statement for a while loop has the form:

```
while ( condition )
{
   block_ of_codes
}
```



Example 3.2.3: Write a C++ program to calculate the average value of a set of data using the *while* loop. Assume that a zero data value indicates that all data have been read.

```
// average value of data
#include <iostream>
using namespace std;
int main()
{
    float x, sum,average_value;

    int  count=0;
    sum=0.0;
    cin>>x;
    while(x!=0.0) {
       sum=sum+x;
       ++ count;
       cin>>x;
    }

    if (count>=1) {
      average_value= sum / count;
      cout << "average=" << average_value<<endl;
      }
    else
      cout << "No data value" <<endl;
    return 0;
}
```

3.3.3 do-while loop

The *do-while* loop identifies a series of steps (statements) that are to be repeated while a certain condition is true. It checks condition for truthfulness after executing the code in the loop. Therefore, the code inside the loop will always be executed at least once.

```
do
{
   block_of_code
} while ( condition );
```

Example 3.2.4 : Write a C++ program to calculate $sum = \sum_{n=1}^{200} n$.

```
#include <iostream>
using namespace std;
int main()
{
    int  n,  sum=0;
    n = 1;

   do{
        sum=sum+n;
         n++;
    } while(n <= 200);

    cout << "sum ="<< sum << endl;
    return 0;
}
```

3.4 The Break and Continue Statements

The break statement

A break statement, **`break;`** once executed within a switch or a loop structure (a while, do...while, for or switch statement), will cause immediate exit from that structure.

Example: In the following program, if the condition (`i==2`) is true, then the `break` statement will be executed which will cause exit from the inner for loop and the execution of the program will continue on the statement `x=1`.

```
for ( ... ; ...; ...)
{
 for (...; ...; ...)
 {
     ....
     If (i==2)
         break ;
     ......
 }
  x=1;
  .....
}
```

The Continue Statement

The continue statement, continue; once executed within a loop structure (a while, do...while, for or switch statement), will skips the remaining statements in the loop cycle and proceeds with the next iteration of the loop.

Example. Consider the following program

```
#include <iostream.h>
int main ()
{
   for (int i=1; i<=10; i++)
   {
       if(i%2==0)
           continue;
       cout <<i<<",  ";
   }
}
```

If the condition `i%2==0` is true (i.e `i` is even), then the `continue` statement is executed and the program skips the print statement. Hence, the above program prints five odd numbers.

```
1, 3,  5, 7, 9,
```

SUMMARY

In this chapter, we study how to express a mathematical condition using a logical expression, how to write C++ statements to choose alternative courses of action and how to repeat certain part of computation using C++.

C++ Syntax Introduced in Chapter Three

Variable declaration — bool *list_of_variable_ names*;

Relational operators — >, >=, <, <=, ==, !=

Logical operators — &&(and) , || (or) , ! (not)

if construct

```
if (logical_expression)  {
       block_of_codes
   }
else if (logical_expression) {
       block_of_codes
       ⋮
   }
else {
       block_of_codes
   }
```

switch construct

```
switch (case_expression) {
case  case_selector_1  :
          { block_1 of C++ statements
            break; }
case  case_selector_2  :
          { block_2 of C++ statements
            break; }
     :
default :
          { block_D }
}
```

for loop

```
for (initialisation; loop continuation condition;
      increment statement)
{
      group_of_statements
}
```

while loop

```
while (condition)
```

```
{
        block_of _codes
}
```

do-while loop

```
do {
        block_of _codes

 } while(condition)
```

EXERCISE 3

Q3.1 What is the difference between a logical operator and a relational operator?

Q3.2 What are the values of the following expressions

(a) 1>2 (b) (1+3) >= 4 (c) 1+3 <= 4 (d) 3>2 && 1+2 <3 || 4<= 3

Q3.3 What is the purpose of the if construct ?

Q3.4. Write logical expressions corresponding to the following conditions.

(a) $x+y>10$ and $x-y \le 0$
(b) $1<a<2$ and $1<b<2$
(c) $1<x<2$ or $x>5$
(d) Either x or y is zero, but not both.
(e) The distance between two points in the plan having coordinates (x1, y1) and (x2, y2) is greater than the distance between (x1, y1) and (x3, y3).

(Ans: (a) x+y > 10.0 && x-y <= 0.0) (b) ....)

Q3.5 If A=2.5, B=7.5, C=5.0, D=6.0 are float numbers and L=1(true), M=0(false) are bool numbers, calculate the values of the following logical expressions.

(a) (A+B) < (C+D) && A = = 3.5 ,
(b) (A+B/2.0) != (C-D) | | C = = D ,
(c) ! L | | C != D && M ,
(d) (C/2.0+D) < A && ! M | | C = =D

(Ans: (a) F, (b) T, (c) F, (d), F)

Q3.6. Write C++ statements that perform the steps in (a) to (d), using the structures indicated.

(a) if statement: If $x>0.0$, add the value of x to sum and increment count by 1.

(b) if statement: If 5.0 < y <10.0, increment y by 2.0, otherwise increment y by 20.0.

(c) if statement:

$$y = \begin{cases} 0 & x \le 0 \\ 1 + x & 0 < x \le 1 \\ 2 & x > 1 \end{cases}$$

(d) count-controlled for Loop: Calculate $y = \sum_{i=1}^{n} i$, $z = n!$

Q3.7. What values will be printed out from each of the following statements.

(a)
```
float x1,x2,x3,x,z,A;
int L;
cin>>x1>>x2>>L;
cin>> x;
cin>> z>>x3>>A;
cout<<L<< "" <<x
    << "" << z
    << "" << A;
```

Data line:
```
0.5  1.0
2
3.0  4.0
6.0  8.0
```

(b)
```
int num=0;
for(int i=1;i<=2;++i){
   for(k=2; k>-2;--k){
       num=i+k;
       cout<<num<<" ";
   }
}
```

(c)
```
float sum=0.0;
 do{
    sum=sum+30.0;
 } while(sum<=100);
 cout<<sum<<endl;
```

Ans: (a) `2 3 4 8`
(b) `3 2 1 0 4 3 2 1`
(c) `120`

Q3.8 What restriction, if any, are there on the case expression in a switch statement?

Q3.9 What forms may a case selector takes? are there are any restrictions on any of these forms?

Q3.10 What is the difference between a counter-controlled for Loop and a while loop? when should the counter-controlled loop be used?

Q3.11 How many times will each of loops controlled by the following for statements be executed?

(a) `for(i=-5; i<=5; ++i)`
(b) `for(j=1; j<=12; j=j+2)`
(c) `for(k=17; k >=15; --k)`
(d) `for(l=17; l>=15; ++l)`

Q3.12 In Q3.11 (a) and (b), what is the value of the for variable after termination of the for loop.

Q3.13 What is an infinite for loop? How can it be avoided?

Q3.14 What is an exit statement used for?

Programming Exercises

Program Debugging - Some guidelines (for Q3.15-3.19)

The most helpful debugging tool is the cout statement. Just knowing that your program is working incorrectly does not tell you where to begin looking for errors. However, if you have the computer, by printing the values of key variables at different points in your program, it becomes easier to isolate the parts of program that are not working correctly. The location of these check-points depend on the program, you need to guess the place which may lead to the error. It is also a good idea to number the check points such as ' Check point 5: x=1.76, y=2.86

(1) If you know that the value of a variable is incorrect, to find the point causing the error, you need to print the intermediate results leading to the final value of the variable.

(2) If you use a if statement, you could check whether the condition is true or not and be sure it is as you expected.

(3) If you believe that the programming error is within a for loop, print the values of key variables at each cycle of the loop which will help you to locate the trouble points.

Q3.15. Design the flow chart and a complete C++ program to calculate and print the maximum, minimum and average values of a series of data. Assume that a data value 10^{20} (1.0e20) indicates that all data have been read. Test your program using the following data: 1.0, 2.8, 9.0, 4.0, 3.2).

Q3.16. Write the pseudocode, flow chart and a complete C++ program for finding the roots of equation $ax^2+bx+c=0$ where $a \neq 0$ using the if construct (read example 3.2.3 in section 3.2.2).

Requirements:

* Read the values of *a, b* and *c* from keyboard;

* In the case of repeated real roots, print 'Repeated real roots x1= x2=......... ;
  In the case of distinct real roots, print 'Distinct real roots x1=....... x2=......;
  In the case of complex roots, print 'Complex roots x1(x2)= .... + (-)... i.

Test your program using the following data: (1) $a = 1$, $b = 1$, c=1.25;
(2) $a = 1$, $b = 3$, $c = -4$.

(**Ans**: (a) $-0.5 \pm i$ (b) 1, - 4)

Q3.17. Write a program which will request a number (1 to 6) to be typed at the keyboard and prints out the corresponding words “one”, ”two” etc. If a number outside this range is typed, print “outside the range 1-6”. Write the program using the switch construct. Can you rewrite the program using the if construct?

Q3.18 Write a program to read a integer value *n*, then calculate $\sum_{i=1}^{n}(2.1*i)$ and $n!$.

***Q3.19 (optional)**

To find a root of a nonlinear equation $f(x)=0$ using fixed-point iteration, we first rewrite the equation as $x=g(x)$. Then, set $x=x_o$ (initial guess) and then perform iteration $x_{i+1} = g(x_i)$ (i = 0, 1, 2, ...) to improve the estimate. If $|x_{i+1} - x_i| < Tol$, we say that the process converges and x_{i+1} is taken as the root. To control the process, we set a limit on the number of iterations (*Max_iter*). If the process does not converge after *Max_iter* iterations, we print an error message and stop computing. The following is an algorithm based on this method

```
Input xo, Tol and  Max_iter.
Set i=1
While (i<Max_iter ) do
    Set x=g(xo)
    If ( |x-x0|<Tol ) then
        output x; ( &  'procedure completed successfully' )
        stop
    else
        Set   i=i+1
        set x0=x
Output ('method failed after Max_iter iterations, Max_iter=', Max_iter)
```

Using above algorithm, write a complete C++ program to find a solution accurate to within 10^{-5} for the equation

$$x^2 - 3x + 2 - e^x = 0$$

(**Hint**. Rewrite the eq. as $x = \frac{1}{3}\left(x^2 + 2 - e^x\right)$ and choose x_o=0.5, *Tol*=10^{-5}, *Max_iter*=100. **Ans**: $x \approx 0.25753$).

CHAPTER 4

C++ FUNCTIONS

4.1 Top-Down Design using Functions

The easiest way to solve most problems is to break them down into smaller sub-problems and deal with each of these in turn, further subdividing these sub-problems as necessary. C++ provides functions to assist in the solution of such sub-problems. Thus, to simplify program logic, a C++ program is normally designed to consist of a main function, a number of other functions, classes, and some global statements external to any functions.

- Execution of the program will start at the beginning of the main function.
- The main function controls the execution order, while each of the other functions is used to perform some specific action. Global statements are introduced to provide global data accessible by all functions located after the statements.
- A program unit (function) needs never be aware of the internal details of any other functions. The only link between a function and a subsidiary function is through the interface of the subsidiary function. This very important principle means that it is possible to write functions totally independent of the main function and of each other. This feature opens up the way for libraries of functions: collections of functions that can be used by more than one program. It also permits large projects to use more than one programmer; what the programmers need to communicate to each other is the information about the interfaces of their functions.

The diagram which outlines the structure of a function is called a STRUCTURE CHART.

4.2 Library Functions and User Defined Functions

There are two kinds of functions: library functions such as *sin* and *sqrt* which are parts of the <cmath> header file, and external functions which are defined by users. A function separates from the other functions and can be called by other functions to perform certain operations and to return the function value computed via the function name.

Example. Calculate the average value of a series of data.

We use a function `average` to calculate the average of *N* values stored in a 1-D array *X*.

use a main function to control the execution order (read *N* & *X*, calculate the average and print the result).

```
/*  Main function for controlling the execution  */

#include <iostream.h>
//using namespace std;
void main()
{
    float average(int, float []);
    int N;
    float X[100];
    cin >>  N;
        for (int i=0; i<N; i++)
             cin >> X[i];
    cout << average(N, X) << endl;
}

/*  Fuction for calculating the average    */

float average(int N, float X[])
{
    float sum_value=0, xbar;
    for(int i=0; i <N;  i++)
         sum_value+=X[i];
    xbar = sum_value/N;
    return  xbar;
}
```

4.3 Defining a Function

A function has the following general form

```
type Function_name(type arg1, type arg2,…,type argN)
{
             C++ statements;
}
```

where ***type*** (int, float and etc.) is the type of the data returned by the function,
dummy arguments, `arg1`, `arg2`,…, can be variables, arrays and functions.

Remarks

(1) If the function is to return a value to the calling statement, it must contain a return statement. The return statement causes execution of the function to return to the point in the calling function at which the function was referenced as though a variable had been inserted in the code at that point, having as its value the value return from the function.

eg.

```
#include <iostream.h>
```

```
int main()
{
    int max(int, int);
    int a, b,c;
    cin >> a >>b;
    c = max(a,b);
    cout << "max ="  << c <<endl;
    return 0;
}

int max(int x, int y)
{
 int z;
  z = (x>y)? x:y;
 return (z);
}
```

The returned value *z* will be passed to the calling statement "`c = max(a,b)`" via the function name `max`.

(2) If no value is to be returned, the function head should be written as

void Function_name(dummy arguments)

eg.

```
void delay(long a)
{
    int i = 0;
    do
    {
     i++;
    } while(i<=a);
}
```

(3) If there is no dummy argument, the function head is

Type Function_name( )

eg.

```
 void  print_message()
 {
     cout << "This is a message.\n";
 }
```

4.4 Calling a Function

To call a function, firstly the function must have already existed. If a library function is to be called, you need to use the `#include` command to include the following header files in the program.

```
#include <iostream.h>     // include standard input/output
#include <cmath.h>        // include math functions
```

If a user defined function is stored in the same file as the calling function and is to be called, a statement in the calling function, called function declaration or function prototype, is needed to tell the compiler the name of the function, the type of data to be returned by the function, the number of parameters and the type of each of the parameters as well as the order. The declaration must precede its use.

eg.

```
float average(int, float[]);
```

Functions can be called in two ways.

(1) Being called as a statement to perform some work, eg. print_message();

(2) Being referenced as operands in expressions. The function name returns one value to the calling function. Functions are referenced in the form:

Function_name (*actual arguments*)

Eg.

```
int main()
{
int max(int, int), a, b, c;
    cin >> a >>b;
    c = max(a,b);
    cout << "max = " << c <<endl;
    return 0;
}
```

- For variable arguments, when a function is referenced,

  the 1st dummy argument will be assigned the value of the 1st actual argument,
  the 2nd dummy argument will be assigned the value of the 2nd actual argument and so on.

  The passing of data is one direction only from calling function to the called function.

- The actual arguments must match the dummy arguments in <u>number</u>, <u>order</u> and <u>type</u>. The argument variable names themselves do not have to match.

4.5 Recursive Functions

In C++, function can call itself. This sometimes can simplify program structure.

eg. $$n! = \begin{cases} 1 & n = 0,1 \\ n \cdot (n-1)! & n > 1 \end{cases}$$

Let fac(n) be a function for calculating n!.

Then $$\begin{aligned} \text{fac}(n) &= n \cdot \text{fac}(n-1) \\ &= n \cdot (n-1) \cdot \text{fac}(n-2) \\ &\vdots \\ &= n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot \text{fac}(1) \end{aligned}$$

Based on the above formula, the C++ program for calculating n! is developed as follows

```
#include <iostream.h>
void main()
{
   float fac(int);
   int n;
   float  x;
   cout <<"Input n :";
      cin >> n;
   x = fac(n);
   cout <<n<<"!=" <<x <<endl;
}
float  fac(int n)
{
   float f;
   if(n<0) cout<<"Data error"<<endl;
   else if (n==0 || n ==1)
           f=1.0;
   else
           f = n*fac(n-1);
   return (f);
}
```

Remarks:

For the recursion to be eventually terminated, the function must include a recursion terminating condition such as

```
else if (n==0 || n==1) f=1.0;.
```

4.6 Function Overloading

- In C, every function must have a unique name. For example, to calculate the cubic of an integer and the cubic of a double, we need to use different function name:

```
int icubic(int)              // for cubic of an integer
double dcubic(double)        // for cubic of a double
```

  This is not very convenient.

- C++ allows several functions to be defined with the same name, as long as these functions have different sets of parameters (in terms of number of parameters or parameter types). This capability is called function overloading.

eg. The following figure uses overloaded cubic functions to calculate the cubic of an integer and the cubic of a double.

```
//overloaded functions
#include <cstdlib>
#include <iostream.h>
using namespace std;
int cubic(int);
double cubic(double);
int main()
{
   cout <<cubic(2)<< endl;      // call int version
   cout <<cubic(2.5)<<endl;     // call double version
   system("PAUSE");
   return 0;
}
// function cubic for int values
int cubic(int x)
{
    cout<<"cubic of int"<< x << "is";
    return x*x*x;
}
double cubic(double x)
{
    cout<<"cubic of double"<< x << "is";
    return x*x*x;
}
```

Remarks

(1) Overloaded functions have the same name but the arguments must be different. For example, in the above example, the argument types are different.

(2) When an overloaded function is called, the C++ compiler selects the proper function by examining the actual arguments with the dummy arguments. The function with

dummy arguments matching the actual arguments in number, and type will be invoked. So, when we reference "cubic(2)", the overloaded cubic function with integer argument is invoked.

(3) Function overloading is usually used to define several functions of the same name that perform similar tasks but on different data types as in the above example. Most functions in the standard C++ library are overloaded for different data types.

4.7 Storage Classes and Scope

(1) Global variables and local variables

- Global variables are defined at location external to any functions. The value of the global variable is assessable to all functions at location after the declaration of the global variable. Thus, global variable declaration must precede its use. Global variables are stored in the area assessable to all functions defined after the variables. Change of the variable in any function will lead to change of the value everywhere. Using global variables, more data can be passed from one function to the others.
- Local variables are defined in function. Its value will be lost after exit from the function.

(2) Static global variables/functions

By preceding global variable/function names with static, we obtain static global variables/functions. The static global variables/functions can only be used in the source file containing the variable/function names.

eg.

```
static int n;
static void staticFunc();
```

(3) Scope

The portion of the program where an identifier can be used is known as scope. For example, when we declare a local variable in a block, it can be used only in that block. A static-storage-class variable's storage is allocated when the program begins execution.

4.8 Construction of Projects with Multiple Source Files

A short program usually can be constructed and saved in one source file. But to develop a long program, it is usually more efficient to organize the program into several parts and save each of them in one source file. All these files have to be included in the same project. To generate an executable file, the following two steps should be taken

(1) compile each of the source files to generate a corresponding object file

(2) link all the object files to generate an executable file.

The most important point which needs to be emphasized, when using project with multiple source files, is that if a global variable has been declared in one of the source files and is to be used in the other source files (say B.cpp and C.cpp), then the variable has to be declared in B.cpp and C.cpp by statement of the following form:

```
extern type variable_name;
```

Remarks: The above declaration does not allocate any space for *variable_name*. It declares that the variable has been created in other program unit and is to be used in this program unit.

Example The following shows a project named "myproject.prj" which consists of two source files: main.cpp and myfunctions.cpp. Contents of main.cpp and myfunction.cpp are as shown below.

```
// main.cpp
#include <cstdlib>
#include <iostream>
using namespace std;
void fenter(float []);
float fsum(float []);
float favg(float []);
int array_size=5;
int main()
{
    float  sum, average;
    float X[array_size];
    // input a set of data into array x
   fenter(X);
    // calculate the sum of  a set of data  x
   sum=fsum(X);
    cout<< "sum of  these values is"<<sum<<endl;
   // calculate the average of a set of data x
   average=favg(X);
    cout<< "Average value of  these values is"<<average<<endl;
    system("PAUSE");
    exit(0);
}
// myfunctions.cpp
#include <iostream.h>
using namespace std;
extern  int array_size;
void fenter(float x[])
{
    for(int i=0; i < array_size ;i++)
    {
        cout<<"Input x["<<i<<"]=";
        cin>> x[i];
    }
```

```
}
float fsum(float x[])
{
    float sumX=0.0;
    for(int i=0; i<array_size;i++)
        sumX += x[i];
    return sumX;
}
float favg(float x[])
{
    float sumX=0.0, avgX;
    for(int i=0; i<array_size;i++)
        sumX += x[i];
    avgX = sumX/array_size;
    return avgX;
}
```

EXERCISE 4

Q4.1 Why should a C++ program be broken into the main function and a set of other functions ?

Q4.2 What is the difference between the main function and other functions ?

Q4.3 What are library functions and user defined functions ?

Q4.4 What is the purpose of a function ?

Q4.5 What does `#include <iostream>` association do ?

Q4.6 Read and then show the output from each of the following program. If you are not sure whether your answer is correct or not, run the program in computer to check the answer.

(a)
```
//Q4_6a
#include <cstdlib>
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ float F(float),G(float),X;
  X=F(2);
  cout<<"X="<<X<<", G(5)="
      <<G(G(5));
  system("PAUSE");
  return 0;
}
float F(float X)
{
  return(2*pow(X,2)+1);
}
float G(float X)
{
   return (X+2);
}
```

(b)

```
#include <cstdlib>
#include <iosteam>
using namespace std;
int main()
{
   float AVERAGE(int,float[]);
   float K[20];
   for (int i=0; i<5; i++)
       K[i]=i;
   cout << "mean="
         << AVERAGE(5,K)<<endl;
   system("PAUSE");
   return 0;
}
float AVERAGE(int N, float X[])
{  float sum=0;
   for (int i=0; i<N; i++)
       sum += X[i];
   return (sum/N);
}
```

(c)

```
#include <cstdlib>
#include <iosteam>
#define N  3
#define M  5
using namespace std;
float MAX=0.0, MIN=1.0E+10;       // Declare global variable MAX and MIN
void  MAXMIN( float [ ][M], int, int );
float MAXVAL( float [ ][M], int, int );
float MINVAL( float [ ][M], int, int);

int  main()
{
    float X[N][M];
    for (int i=0; i<N; i++) {
       for (int j=0; j<M; j++)
          X[i][j]=0.0;
    }

    for (int i=0; i<N; i++) {
       for (int j=0; j<M; j++)
          X[i][j]=(i+1) + 2*(j+1);
    }
    MAXMIN(X, N, M);
    cout<< "max=" << MAX<< ", min=" <<MIN<<endl;
  system("PAUSE");
  return 0;
}

void  MAXMIN(float X[][M], int RSIZE, int CSIZE)
{
```

```
        MAX=MAXVAL(X, RSIZE, CSIZE);
        MIN=MINVAL(X, RSIZE, CSIZE);
      }

float  MAXVAL(float X[][M], int RSIZE, int CSIZE)
   {
        float MAX=0.0;
        for (int i=0; i<RSIZE; i++){
           for (int j=0; j<CSIZE; j++)
              if (MAX < X[i][j])  MAX=X[i][j];
        }
        return MAX;
   }

float  MINVAL(float X[][M], int RSIZE, int CSIZE)
     {
        float MIN=0.0;
        for (int i=0; i<RSIZE; i++) {
           for (int j=0; j<CSIZE; j++)
              if (MIN > X[i][j])  MIN=X[i][j];
        }
           return MIN;
     }
```

(d)

```
#include <cstdlib>
#include <iostream>
#define M 2
using namespace std;

float MEAN_R1, MEAN_R2;

int  main()
{
 float average(float [ ][M], int, int);
 void sub1(float [ ][M], int);
 float  X[2][2] = { { 1.0, 2.0},  {3.0,4.0}};
 cout<<"average=" <<average(X, 2,2)<<endl;
 sub1(X, 2);
 system("PAUSE");
 return 0;
}
```

```
float  average(float X[ ][2], int RSIZE, int CSIZE)
{
   float sum=0.0;
   for (int i=0; i<RSIZE; i++) {
         for (int j=0; j<CSIZE; j++)
            sum += X[i][j];
   }
   return (sum/(CSIZE*CSIZE));
}
void sub1(float X[ ][M],  int CSIZE)
{
    float SUM_R1=0.0, SUM_R2=0;
    int j=0;
    while (j<CSIZE) {
      SUM_R1 += X[0][j];
      SUM_R2 += X[1][j];
      j++;
    }
    MEAN_R1=SUM_R1/CSIZE;
    MEAN_R2=SUM_R2/CSIZE;
    cout<< "mean_R1="<< MEAN_R1<< ", mean_R2="
        << MEAN_R2 <<endl;
 }
```

Ans: (a) x=9, G(5)=9 (b) mean=2
(c) max=4, min=3 (d) average=2.5,
mean_R1=1.5, mean_R2=3.5

PROGRAMMING

Always Plan Ahead

To write a C++ program, it is essential to first draw up a program design plan (flow chart or pseudo code) which shows the structure of the program and the various levels of details.

Q4.7 Write a C++ function to compute the value of the formulae defined by

$$f(x) = \begin{cases} 0 & x < -10 \\ 2x + 20 & -10 \le x < 0 \\ 20 & 0 \le x < 20 \\ 30 - 0.5x & 20 \le x < 60 \\ 0 & x \ge 60 \end{cases}$$

Test your function by calling it in a main function using x = -5, 10, 40, 100. Print the results using formatted output: x = ****.** f(x) = **.**

Q4.8 Write a function MYEXP to compute e^x using the following series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots.. = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Continue using terms until the absolute value of a term is less than 1.0D-8. Test your function by calling it in the main function using x = 1, 2 respectively. Use at least 9 digits of precision for the function and variables. Print the results using formatted output.

x = *.**, MYEXP(X) = ***.********

Q4.9 The following is an algorithm for finding a root of the equation x=g(x) where g is a function of x.

Input x_o, Tol and Max_iter.
Do for i=1 to Max_iter
  Set $x=g(x_o)$
  If ($|x-x_0|<Tol$) then
    output x; (& 'procedure completed successfully')
    stop
  else
    set $x_0=x$
Output ('method failed after Max_iter iterations, Max_iter=', Max_iter)

Using the above algorithm, write a complete C++ program to find a solution accurate to within 10^{-5} for the equation

$$x = \frac{1}{3}\left(x^2 + 2 - e^x\right)$$

Hint. Choose x_o=0.5, Tol=10^{-5}, Max_iter=100. **Ans**: $x \approx 0.25753$).

Program design:

(1) Define g(x) using a function;
(2) Find the root of $x=g(x)$ using a function with header

float ITER (X0, TOL, Max_iter)
where X0, TOL, Max_iter: input to the function

(3) Design a main function which reads X0, Tol and Max_iter
calls ITER to find a root, and
prints the result or otherwise an error message.

CHAPTER 5

C++ ARRAY PROCESSING

An **array** is a group of storage locations that have the same name.

- Individual members of an array are called **elements** and are identified by using the common name followed by a number of **subscripts** in parentheses.
- Each element can store one data and thus an array can store a group of data.

5.1 One-Dimensional Arrays

A one-dimensional array can be visualized as either one column or one row of spaces for storing data. Each space can store one data and is referenced with the array name followed by a subscript. The storage locations and associated names for a 1-D real array *A* of 6 elements are shown as follows.

A[6]

1.0	2.0	0.5	3.0		
A[0]	*A[1]*	*A[2]*	*A[3]*	*A[4]*	*A[5]*

Declaration

Whenever we create an array, we need to specify its name, type and size by a declaration statement with the form as follows so that the compiler can allocate sufficient storage units for storing a group of data.

```
Type array_name[Array_size];
```

eg.

```
int  a[8], b[20];
```

Notes:

(1) *Array_size* refers to the number of elements of the array. It must be an integer constant or an integer constant expression.

(2) The index of the array starts from 0. For example, "`char b[4]`" represents a character array with 4 elements : b[0], b[1],b[2] and b[3]. We usually call an array of characters as a string. In this case, b[4] is a string of length 4 with the last character NULL '\0'.

Initialization

An array can be assigned initial value in the declaration, eg.

```
static int a[3]={1,2,3}; char b[3]= "me";
```

Remarks

- In C++, only static and external arrays can be initialized.
- If all elements are to be set to a same initial value, we can simplify the statement. For example,

```
static int a[100]={12*100}
```

sets all the 100 elements to the initial value 12.

Input/Output

- To read certain specific elements, reference the elements directly by index.

  eg.

```
cin>> A(1)>> A(11);
```

- To read part of an array, use an implied *for* loop to identify the elements to be read.

  eg.

```
for(int i=1; i<N; i++)
      cin>> A(i);
```

Notes: * N can be an integer constant, integer variable or expression
* Methods for printing values from arrays are the same as for reading.

5.2. Multi-Dimensional Arrays

A 2-D array can be visualized as a group of columns (or a table) as illustrated. The storage locations and associated names for a 2-D array with 4 rows and 5 columns are shown as follows:

A[4][5]

1	0	2	5	6	*Row 0*
2	1	3	4	-1	⋮
-2	1	-9	8	9	⋮
1	0	0	2	3	*Row 3*
Col. 0	1	2	3	4	

Unlike in 1-D arrays, elements in 2-D arrays must be referenced with two subscripts.

- The first subscript references the row
- The second subscript references the column.

eg. A[2][3] refers to the data value in row 2 and column 3. In the above example A[2][3]=8.

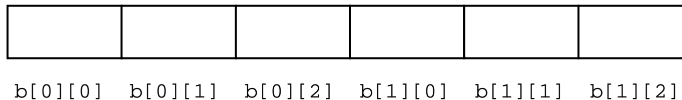
Declaration

In the same was as for 1-D arrays, the sizes (for both subscripts) of 2-D arrays can be specified by a declaration statement of the form

```
array_name[Number of rows][Number of columns]
```

eg. 1: `int b[2][3];`

declares an integer array with 2 rows and 3 columns. The Array elements are arranged in memory row by row, i.e.



b[0][0] b[0][1] b[0][2] b[1][0] b[1][1] b[1][2]

Initialization

As for one dimensional arrasy, in C++, static and external multi-dimensional arrays also can be initialized in the array declaration.

eg. 1: `static int a1[2][3]={1, 2, 3, 4, 5};`

declares an integer array with 2 rows (the first index 0 and 1) and 3 columns (the second index changes from 0 to 2). The 5 initial values are assigned to row one then row two. Obviously the last element in row 2 is not assigned any initial value.

eg. 2: `static float b1[3][4]={{1,2},{4},{10,11,12,13}};`

declares a real array with 3 rows and 4 columns. The first 2 elements in row 1 are assigned 1,2; the first element in row 2 is assigned 4; the 4 elements in row 3 are assigned 10, 11, 12 and 13 respectively.

Input/Output

In the same way as for 1-D arrays, we can read some specific elements or part of an array using an implied *for* loop.

Example. Give a set of statements to define a 2-D array *A* and to read, using *A*, the values in a matrix of *N* (< 10) rows and *M* (< 4) columns row by row (row 1 first, then row 2...).

Sol.

```
int  N=5, M=2;
float A[10][4];
for (int i=0; i<N; i++)
{
     for (int j=0; j<M; j++)
```

```
            cin>> A[i][j];
}
```

Multi-Dimensional Arrays

C++ allows multi-dimensional arrays. We can easily visualize a 3-D array as a cube. Elements in 3-D arrays are referenced with three subscripts. Most applications do not use arrays with more than three dimensions.

5.3 Array Operations

For C++, arrays can only operated on element basis. An array element can be used anywhere that a scalar variable can be used. In exactly the same way as a scalar variable, it identifies a unique location in the memory to which a value can be assigned or input, and whose value may be used in an expression or output list, etc. The great advantage is that by altering the value of the array subscript it can refer to a different location. Thus the use of array variables within a loop therefore greatly increases the power and flexibility of a program. This can be seen from the following for loop which enables 100 data to be input and stored for subsequent analysis in a way which is not otherwise possible.

```
for (int i =1; i<=100; i++)
     cin >> a(i);
```

In C++ and most other programming language, this is the only way that arrays can be used in most types of operations.

Example 1. The Fibonacci series is defined by

$F_1=1$,
$F_2=1$,
$F_n=F_{n-1}+F_{n-2}$, $n>2$.

Calculate and print the first 40 numbers of the Fibonacci series, 5 numbers per line.

Solution.

Algorithm

$F_0=1$,
$F_1=1$,
for n=2 to 39
    $F_n=F_{n-1}+F_{n-2}$
for i=0 to 39
    If (i!=0 & i%5=0) go to next printing line
    Print *Fi*

C++ Program for generating Fibonacci series

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
void main()
{
   ofstream Coutfile("cch5fib",ios::out);
   int i;
   static int f[40]={1,1};
   for (i=2;i<=39; i++)
      f[i]=f[i-2]+f[i-1];
   for (i=0; i<=39; i++)
   {
      if(i!=0 && i%5==0) Coutfile<<"\n";
      Coutfile<<setw(10)<<setiosflags(ios::left)<<f[i];
   }
}
```

Output Results

```
1         1         2         3         5
8         13        21        34        55
89        144       233       377       610
987       1597      2584      4181      6765
10946     17711     28657     46368     75025
121393    196418    317811    514229    832040
1346269   2178309   3524578   5702887   9227465
14930352  24157817  39088169  63245986  102334155
```

Example 2 Given $A = \begin{bmatrix} 5 & 7 \\ 8 & 3 \\ 7 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 12 & 3 & 6 \\ 4 & 2 & 7 \end{bmatrix}$, calculate C=AB, and print the result on the screen.

Solution

```
#include <cstdlib>
#include <iostream>
#include <iomanip>
using namespace std;
int  main()
{
```

```
    int A[3][2]={{5,7},{8,3},{7,4}};
    int B[2][3]={{12,3,6},{4,2,7}};
    int C[3][3];
    int i, j, k;
    // Calculate C=AB
    for(i=0;  i<3;  i++)
    {   for(j=0; j<3;j++)
        {  C[i][j]=0;
           for(k=0; k<2; k++)
              C[i][j]=C[i][j]+A[i][k]*B[k][j];
        }
    }
    for(i=0;  i<3;  i++)
    {
        for(j=0; j<3;j++)
           cout<<C[i][j]<< " ";
        cout<<endl;
    }
    system("PAUSE");
    return 0;
}
```

5.4 Passing Arrays to Functions

In C++, arrays are passed to functions by reference. The function call passes the address of the first element of the array to the function via the array name used as an argument of the function. The size of the array is passed to the function via another argument of the function. When array elements are modified in the called function, it modifies actual elements of the array in their original memory locations.

5.4.1 Defining Functions with Arrays as Arguments

If a function is to receive a **one-dimension array** from the calling function, the function's parameter list must list the array name followed by [ ] as a parameter, the array size can be passed by a separate parameter.

eg Find the minimum in the array xvalue.

Sol

```
int  xmin(int  xvalue[ ], int  xsize)
{
   int MinValue = xvalue[0];
   for (int i = 1;  i  <  xsize;  i++) {
        if (xvalue[i] < MinValue)
                MinValue = xvalue[i];
   }
        return MinValue;
}
```

For **multi-dimensional arrays**, in the function definition, the size of array in each dimension except dimension 1 needs to be specified.

eg. To pass a 2-D array with 10 rows and 4 columns to the function `xmin`, we need the function header to be as follows.

```
int xmin(int xvalue[ ][4], int rowsize)
{
      body of xmin_function
}
```

5.4.2 Calling Functions with Arrays as Arguments

To pass an array to a function, we need to specify the array name without any bracket. For example, if an array *`xvalue`* has been declared as

```
int  xvalue[20];
```

the function call

```
xmin(xvalue, 20);
```

passes the array *`xvalue`* and its size to the function *`xmin`*.

Remarks: Entire arrays are passed to function by reference, but individual elements are passed by value exactly as simple variables.

EXERCISE 5

Q5.1 How is an array specification written?

Q5.2 Write declarations for suitable arrays to store the following sets of data

(a) three matrices of 10 rows and 5 colummns.
(b) a vector with 100 elements

Q5.3 Show the output from each set of statements.

(a)

```
   ......
int  LIST[8];
for(int k=0; k<4; k++)
 LIST[3 -k]=k;
for(int k=0; k<2; k++)
 cout<<LIST[k]<< "  ";
```

(b)

```
  ......
float  TIME[50];
for (int j=0; j< 10; j++)
   TIME [j] = (j-1)*0.5;
   for(int j=1;j<10;j=j+4)
   cout<< "TIME ''<<j<< "="
       << TIME[j]<<endl;
```

(c)

```
  ......
 int K [3][3];
for(int i=0; i<3; i++)
{  K[i][0]=5;
   K[i][1]= -5;
   K[i][2]=0;
}
for (int j=0; j<3; j++)
   cout<< K[2][j]<< " ";
```

(d)

```
   ......
float  DIST [10][10];
float   SUM=10.0;
for (int j=0; j< 3; j++)
  { for(int i=0; i<3; i++)
     { SUM=SUM+1.5;
       DIST[i][j]=SUM;
     }

 }
 for(int i=0; i<2; i++)
 {
   for(int j=0; j<2; j++)
     cout<<DIST[i][j] ]<< " ";
   cout <<endl;
 }
```

Ans:

```
(a) ^ 3 ^ 2
(b) Time ^1 = 0
    Time ^5 = 2
    Time ^9 = 4
(c) 5 -5 0
(d) 11.5^16
     13^17.5
```

Q5.4 An array TIME contains 30 integers. Give statements that print one value from every five values, beginning with the fifth value, in the form:

Time (5) contains **** seconds
Time (10) contains **** seconds
Time (30) contains **** seconds

Ans:

```
for (int k=4; k<30; k=k+5)
  Cout <<"Time("<<k+1
       <<") contains"
       <<Time[k]
       << "seconds"<<endl;
```

Q5.5 Give C++ statements to interchange the first and tenth elements, the second and ninth elements, and so on, of the array NUM that contains 10 integer values.

Ans:

```
for (int i=0; i<10; i++)
{
    float ihold=NUM[i];
    NUM[i]=NUM[9-i];
   NUM[9-i]=ihold;
 }
```

Q5.6 Write a complete program that will read 5 integers to an array from keyboard, one data per line. Write the data in the reverse order from which it was read. Test your program with data 1, 5, 10, 20, 9999.

Ans:
```
int  Num[5];
for (int i=0; i<5; i++)
     cin>> Num[i];
for (int i=4; i>=0; i--)
     cout<< Num[i] <<endl;
```

Programming

Q5.7 Given

$$A=\begin{pmatrix}1 & 2\\ 1 & 3\end{pmatrix},\ B=\begin{pmatrix}2 & 0\\ 1 & 1\end{pmatrix},\ x=\begin{pmatrix}1\\ 2\end{pmatrix},\ y=\begin{pmatrix}2\\ 3\end{pmatrix}.$$

Write a C++ program to calculate $C=A\text{-}B$, $D=2*A*B$, $E=A^T$, $z=x.y$.

Q5.8 Write a program which reads and stores 10 real values into a 1-D array , then find the maximum value. Test your program using 1,2, -3, 10, -6, 2,1,3, 5,4,

Q5.9 Write a program which consists of a main function and two functions INPUT_dat and OUTPUT_dat. The main function calls INPUT_dat to read an integer number N (assume N<100) and then two square matrices(N rows by N columns), then calls OUTPUT_dat to print the value N and the matrices row by row using format output. Test your program using

$$A=\begin{bmatrix}1 & 1 & 1\\ 2 & 1 & 1\\ 0 & 0 & 1\end{bmatrix} \qquad B=\begin{bmatrix}2 & 0 & 1\\ 0 & 2 & 0\\ 1 & 2 & 1\end{bmatrix}.$$

Q5.10 A set of data with *N_row* rows and *N_col* columns, represents the elevations at nodes of a grid. Write a complete program to read the data and to locate the peak point (row number, and columns number).

Print the results using formatted output.

```
Number of peak point =   **

No.      Location(row, column)
 1             **,      **
 2             **,      **
```

Test your program with the following data

```
63  23  21  34
43  30  37  32
38  39  36  28
42  48  32  30
40  42  48  49
```

Hint: 1) To be a peak point, the point must be an interior point (no on the edge) and the elevation at the point must be greater than the elevations at its four adjacent points.Eg If (i, j) is a peak point, then the elevation at (i, j) must be greater than those at (i, j-1), (i, j+1), (i-1, j) and (i+1, j).

2) Algorithm. Use a 2-D array (say MAP) to store the elevations at nodes of the grid; a 2-D array PEAK to store the row & column No. of peak points.

Input N_row, N_col
Input Map
Set Count=0
For each interior point Do
If the point is higher than all 4 adjacent points, increment count by 1
and store the row and Column No. of the point into PEAK
Output Count and PEAK.

Q5.11 Given the following *search* function, write a complete C++ program to search some values stored in a 1-D array.

```
int search(int xvalue[],int xsize, int KEY){
for (int i=0; i<xsize; i++){
     if (xvalue[i]== KEY) {
        return i;
     }
     return xsize;
     }
}
```

Q5.12 Given the following *swap* and *sort* functions, write a complete C++ program to sort all values stored in an 1-D array.

```
void swap(int &a,  int &b) {
     int temp = a;
     a = b;
     b = temp;
}
void sort(int  xvalue[], int xsize) {
    for (int i=0; i<xsize-1; ++i) {
       int k=i;
       for (int j = i+1;  j<xsize; ++j) {
          if (xvalue[j]<xvalue[k])
              k = j;
       }
       if (i!= k)
          swap(xvalue[k], xvalue[i]);
    }
}
```

CHAPTER 6

C++ POINTERS

This chapter introduces the basic pointer concepts and their application in C++ programs.

6.1 Declaration and Initialisation of Pointer Variables

A variable physically represents a memory location which contains a value, while a pointer variable contains the memory address of a variable that, in turn, contains a specific value.

(a) Declaration

Pointer variables must be declared by

```
Type   *pointer_variable_name1, *pointer_variable_name2, …  ;
```

eg.
```
int   *nPtr;                 // nPtr is a pointer to int
double   *aPtr, *bptr;       // aPtr and bPtr are pointers to double
```

Remarks

- *Type* refers to `int`, `float`, `double`, `char` and etc.
- Each pointer variable must be preceded by an asterisk (*). It indicates that the variable being declared is a pointer. It is recommended to include the letters *Ptr* in pointer variable names for improving program readability.
- A pointer that always points to the same memory location is called a constant pointer, which is declared by

```
const   type   *constant_Ptr;
```

eg.

```
const float *xPtr;
```

(b) Initialisation

- Pointers must be declared either in the declaration statements or assignment statements.
- A pointer may be initialised to 0 (or NULL) or an address of a variable.

6.2 Pointer Operators

There are two pointer operators: address operator & and dereferencing operator *.

Name	*Operator*	*C++*	*Function*
Address operator	&	`&a`	Return the memory address of the variable `a`.
Dereferencing operator (indirective operator)	*	`*bPtr`	Return the value of variable to which `bPtr` points to.

eg. 1

```
float  x=2.5;        // declares variable x
float  *xPtr;        // declares pointer variable xPtr
xPtr  =  &x;         // assigns  the address of x to xPtr
cout << *Ptr;        // prints the value of the variable  to which  xPtr points to,
                        which is  x and equals 2.5
```

eg.2

```
……
int  y=2, z;
const int *const_Ptr=&y;    // constant pointer must be initialised
*const_Ptr  = 7;            // allowed,   as  *const_Ptr is not constant
const_Ptr  =  &z;           // not allowed, as const_Ptr is constant
```

6.3 Function Pointers – Passing a Function to Another

The name of a function is the starting memory address of the code that performs the function task.

- To pass a function as an actual argument to another function, the corresponding dummy argument of the called function must be a function pointer.
- A function pointer must be declared by the following form

```
Type (*function_name) (type_of_arg1, type_of_arg2,…);
```

eg.

```
#include <iostream>
#include <cmath>
using namespace std;
float f(float(*func)(float),float x);    // use function pointer

int main()
{
    float t, g(float)
    t=f(g, 2.5);    // actual argument function g is passed to the function pointer func
    cout << "g(2.5)=" << t <<endl;
    return 0;
}

float f(float(* func)(float),float x)    // use function pointer
{
    float t = func(x);
    return t;
}

float g(float x)
{
    return x*x;
}
```

6.4 Passing Arguments to Functions by Reference with Pointer

In C++ there are three ways to pass arguments to a function:

- Pass by value
- Pass by reference with reference arguments
- Pass by reference with pointer arguments

(a) Pass by Value

When an actual argument is passed to a function by value, a copy of the actual argument's value is made and passed to the called function. Changes to the argument's value in the called function do not change the original variable's value in the calling function.

Remark:

(1) The one way flow of data (caller $\rightarrow$ called function) prevents the accidental side effects due to errors in the caller function.

(2) The disadvantage is that if large amount of data is to be passed, it takes considerable execution time and memory space.

(b) Pass by Reference with Reference arguments

With pass by reference, by using a reference parameter corresponding to the actual argument, the called function can access the actual argument's value and modify it. To indicate that an actual argument's value is to be passed by reference, the dummy argument corresponding to the actual argument must be a reference parameter of the form

```
Type  &arg_name;
```

The same convention should also be used when listing the parameter's type in the function header.

eg. ....

```
void  squarebyRef( int &);  // int & indicates that the argument is integer
                             // and is to be passed by reference
int  main( )
 {
    int x=5;
    cout<<"x="<<x<<" before calling squarebyRef"<< endl;
    squarebyRef(x);
    cout<<"x="<<x<<" after calling squarebyRef"<< endl;
    return 0;
}

void  squarebyRef(int &xvalue)
{
    xvalue *= xvalue;           // caller's argument value modified.
}
```

Output : x =5 before calling squarebyRef
x =25 after calling squarebyRef

(c) Pass by Reference with Pointer Arguments

- In C++, when calling a function with a variable argument that is expected to be modified, the address of the variable (instead of the value) is passed. This can be accomplished by applying the address operator (&) to the name of the variable in the calling statement:

```
squarebyPtr( &x);      // pass the address of x to the function squarebyPtr
```

Remark:

If the argument to be passed is an array/string, the address operator is not needed as the name of an array /string is the address of the first element of the array,

eg.

```
int arrayA[10];            //  arrayA  ≡  &arrayA[0];
```

- In the called function, the corresponding dummy argument to receive the address must be a pointer, eg.

```
void squarebyPtr(int  *xPtr)
{
    body of function
}
```

- The indirection operator (*) can be used to form a synonym for the name of the variable, which can then be used to access and change the value of the variable at its original memory.

eg. ……

```
void squarebyPtr(int *);  // int*  indicates the integer being passed
                          //       by reference  with  pointer
int main()
{
   int x=5;
   squarebyPtr (&x);
   cout << "x = " << x <<endl;
   return 0;
}

void squarebyPtr(int *xPtr)
{
    *xPtr *= *xPtr;
}
```

Output : x = 25

Remarks:

If a pointer in the called function can be modified to point to other data items but the data to which it points to should not be modified through the pointer, the statement for function declaration, function calling and function definition should take the following forms:

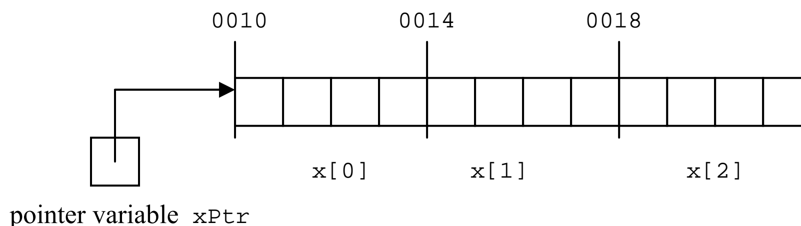
```
:
void  print_char(const char *) ;          // function declaration
:
int main()
{
   const char Univ[ ]="Curtin University" ;  // argument
declaration
   print_char (Univ);
  return 0;
}
// corresponding dummy argument in function definition
void print_char(const char *sPtr)
{
    for(;*sPtr!='\0'; sPtr++ )
       cout << *sPtr;
}
```

6.5 Pointer Arithmetic

Several arithmetic operations can be performed on pointers.

- A pointer can be incremented (++) or decremented (--).
- An integer can be added to (+ or +=) or subtracted from (- or -=) a pointer.

To understand the pointer arithmetic, firstly consider how data are stored in memory.
Let `int x[3]` be an integer array with 3 elements and its first element is at memory location 0010. Assume that a machine with 4–byte integers is used. Then the memory location of each of the elements is shown below.



- Suppose that pointer `xPtr` has been initialized to point to `x[0]` by the following statement

  `int *xPtr = x;` (or `int *xPtr = &x[0]`)

  then the value of `xPtr` is 0010.

- When a pointer is incremented/decremented by an integer *n*, the pointer is incremented/decremented by that integer times the size of the block of each data (4 bytes in the above example).

  Thus, for the above example ,

  if xPtr = 0010 (points to x[0]),
  then xPtr +=2 set xPtr = 0018 (0010+2*4) which points to x[2], and

  if xPtr = 0018 (points to x[2]),
  then xPtr -=1 set xPtr backward to 0014 (points to x[1]).

6.6 Pointers and Arrays

Pointers can be used in array operations involving array subscripting. To demonstrate this, consider the following declarations:

```
float x[6];        // create a 6-element float array x
float *xPtr;       // create a float pointer xPtr
```

As the array name is a constant pointer to the first element of the array, we can set xPtr to the address of x[0] by

```
xPtr = x;        or equivalently        xPtr = &x[0];
```

The following list shows some relationships between the pointer xPtr and elements of the array x[6] for the above example.

```
x[i]     equivalent to     *(xPtr+i)    and   *(x+i)
x[i]     equivalent to     xPtr[i]
&x[i]    equivalent to     xPtr + i
```

Remark: For clarity, it is recommended to use array notation instead of pointer notation when manipulating arrays.

6.7 Dynamic Memory Allocation to Arrays

In previous sections, the shape of an array is given explicitly in the declaration by constants or constant expressions such as

```
int  n=200;
int x[n];           // declare 1-D array of size 200
int b[n][n];        // declare 2-D array of size 200-by-200
```

In these cases, a fixed amount of memory space will be allocated to the array once the array is declared. This is not efficient in terms of the use of memory. Thus, C++ provides two operators `new` and `delete` to allow programmers

- to allocate proper amount of memory space to arrays via the use of the `new` operator;
- to release the memory space whenever the memory space (array) is no longer needed via the use of the `delete` operator.

Declaration of arrays to be dynamically allocated memory space

An array can be allocated space dynamically via the use of a pointer and should be declared by

```
type  *array_Name;
```

eg.

```
integer  *array_A;
```

Note: In the above declaration, no memory space is allocated to array `array_A`. Thus, before the array `array_A` can be used, it must be allocated space via the `new` operator.

Allocation of spaces using the `new` operator

The operator `new` is used to allocate space to an array via the statement of the form

```
Array_name  =  new type[arraySize];
```

eg.
```
n=200;
array_A = new int[n][4];   // allocate memory for an array A of size 201-by-5
```

Note:

Once the above codes are executed, the computer will check whether there is sufficient unused memory space for creating the 2-D integer array with (n+1) rows and 5 columns or not. If yes, the array will be created and the `new` operator will return a pointer of integer type. If no sufficient space is available, the `new` operator will return a NULL pointer value. Thus, the proper codes for using a new operator are as follows:

```
If ((array_A=new int[n][4])==NULL)
{
   cout << "Cannot allocate more memory, terminating"<<endl;
   exit(1);
}
```

Release of space by using the `delete` operator

When the array with space dynamically allocated by the preceding `new` is no longer needed, it may be destroyed and the space is released by using the `delete` operator in the statement of the form

```
delete[] array_name;
```

eg .

```
delete[] array_A;
```

Notes:

(1) The operand of delete is the pointer returned by the new operator (`array_A` in this case). When the pointer is an array, the `delete` operand must be preceded by `[]`.

(2) The array name `array_A` is a constant pointer to the first element of the array. This pointer is not deleted, rather the space `array_A` points to is deleted.

Example Read an integer number N and a set of N real numbers into an one-dimensional array. Then calculate the average value.

```
#include <cstdlib>
#include <iostream>

using namespace std;
float  average(float *,int n);
int main()
{
   int n;
   float *x;
   cout<< "Please input number of data values" << endl;
cin >> n;

if((x=new float[n])!=NULL)
{
    cout<< "Allocate  memory is successful" << endl;
    for(int count=0; count < n; count++)
         cin  >> x[count];
    cout << "Average value is" << average(x,n) <<endl;
    delete [] x;
}

            system("PAUSE");
            return EXIT_SUCCESS;
}

float average(float *px, int total_num)
{
    float xsum=0.0;
    for(int i=0;i<total_num;i++)
       xsum += px[i];
    return xsum/total_num;
}
```

EXERCISE 6

Q6.1 Find the error in each of the following statements.

(a) int *number
cout << number <<endl

(b) int *x, z;
x = z

(c) char s[ ] = “This is a character string.”;
for (; *s != ‘\0’ ; s++)

```
        cout << *s;
    (d) float   y = 26.15;
        float  yPtr = &y
        cout << yPtr << endl;
```

Q6.2 Write a float function MinMaxAver to find the maximum, minimum and average values of a series of N data values stored in an one dimensional array Xarray. Then write a main function to read the data values (N and Xarray), to call the function to calculate the maximum, minimum and average values, and to print the results (hint : use pointers to pass arguments between the main function and the MinMaxAver function).

Q6.3 Write a float function MatMult to calculate the product of two matrix $A_{m\times k}$ and $B_{k\times n}$.

Q6.4 The following is an algorithm for finding a root of the equation x=g(x) where g is a function of x .

Input x_o, tol and max_iter.
for i=1 to max_iter
  Set $x=g(x_o)$
   If ($|x-x_0|<tol$) then
     output x; (& 'procedure completed successfully')
     stop
  else
    set $x_0=x$
Output ('method failed after max_iter iterations, max_iter=', max_iter)

Using above algorithm, write a complete C++ program to find a solution accurate to within 10^{-5} for the equation

$$x = \frac{1}{3}\left(x^2 + 2 - e^x\right)$$

Hint. Choose x_o=0.5, *tol*=10^{-5}, *max_iter*=100. **Ans**: $x \approx 0.25753$).

Program design:

(1) Define g(x) using a function.

(2) Find the root of *x=func(x)* using a function with header

float iter (float(*func)(float), float x0, float tol, int max_iter)

where x0, tol and max_iter input to the function.

(3) Design a main function which reads x0, tol and max_iter
calls iter to find a root, and
prints the result or otherwise an error message.

(4) Use a function pointer to pass the function g(x) to the function iter.

CHAPTER 7

C++ FILE OPERATIONS

This chapter shows how to read/write data from/to a file.

- A file is an external source from which data may be obtained, or an external destination to which data may be sent. We can enter data once into a data file, then when the data is needed, we can read it from the data file. We can write data into a data file, other programs can then use the data and it is still available if we decide to print it.
- A file consists of a sequence of bytes.
- Data can be read from or write to a file in two modes – either sequentially or randomly. The first mode is called sequential access, i.e., data are written to a file one after another in order and also must be read sequentially in this order. The second mode is called random access, i.e., data are written and read in any order by using file-position pointers.
- For scientific computing, sequential-access files are usually used, while random-access files are widely used in many applications such as banking system – customers access the details of their accounts by random access not sequential access. In this chapter, we introduce only operations with sequential-access files.

7.1 Include Header Files

In order to perform input, output and file processing, the program must include iostream header file, the fstream (file input/output) and iomanip (format manipulation) header files. These can be done by including the following statements at the top of the program or in the header file of the program

```
#include <iostream>
#include <fstream>
#include <iomanip.h>
```

7.2 Create/Open a Sequential File

Data cannot be transferred to or retrieved from a file until the file is created/opened. A file may be created/opened by creating ifstream, ofstream of fstream objects.

(a) Open a file for output

A file can be created/opened for output by creating an ofstream object using a statement of the form

```
ofstream  Coutfile( filename ,ios:out );
```

Remarks:

- `ofstream` means outputfile stream. The statement creates a handle for a stream to write in a file.
- The file to be used for output should be placed in the directory for which you run the program

Example.

```
ofstream  Coutfile( "E6Q3_out" ,ios:out );
```

Remarks:

(1) The above statement creates a pointer"Coutfile" which points to the file "E6Q3_out";

(2) **The above Coutfile** is the name of the ofstream object created for writing data to the associated file. User can choose to use their own prefer name, instead of Coutfile, for this purpose.

(3) By default, `ofstream` objects are opened for output, so the statement can be written as

```
ofstream Coutfile(filename);
```

(4) Once an `ofstream` object has been created, the following statements can be used to test whether the open operation was successful.

```
if (! Coutfile)
{
   cerr << "file could not be opened" << endl;
   exit(1);
}
```

If the file was not opened successfully due to no permission for reading/writing or no enough disk space etc, an error message will be displayed and the process is returned to the environment from which the program was invoked.

(a) Open a file for input

Similarly, creating an `ifstream` (input file stream) object will open a file for input.

```
ifstream Cinfile(filename,ios::in);
```

or

```
ifstream Cinfile(filename);
```

Example. The statement

```
ifstream Cinfile("E6Q3_in",ios::in);
```

creates an ofstream object Cinfile associated with the file "E6Q3_in" that is opened for input.

fstream Objects

Both output file and input file can be opened by creating a fstream object with different file – open mode.

```
fstream Coutfile(filename,ios::out ); // for output file

fstream Cinfile(filename,ios::in );   // for input file
```

Clearly, the constructor of ofstream takes 2 parameters. One is the file path (file name), and the other is the file open mode. There are several open modes available as detailed below

Open mode	*Description*
ios.out	If the file is a new file, it will be created. If the file is an existing file, then it will be opened and its content will be destroyed. Data can then be written to the file.
Ios.app:	If the file is a new file, data will be written to it. If the file is an existing file, then it is opened and new data are added to it from the end of the file.
Ios.in:	This is for input data. If the file is a new file, then it is created as an empty file. Otherwise it is opened and its content is made available for reading.

7.3 Writing Data to and Reading Data from a Sequential File

Writing Data to File

- Once an ofstream object Coutfile has been created by

```
ofstream Coutfile( "E6Q3_out" );
```

a "line of communication" is established between the program and the file "E6Q3_out" via the object "Coutfile".

- Data stored in variable_1, variable_2, … can then be written to the file "E6Q3_out" by using the stream insertion operator << and the "Coutfile" object associated with the file by

```
Coutfile<<output_variable_1<<output_variable_2;
```

Reading Data from a Sequential File

Suppose that a data file "E6Q3.in" has been opened by the following statement

```
ifstream  Cinfile( "E6Q3_in");
```

Then data can be retrieved sequentially from the file, normally starting from the beginning of the file, by using the stream extraction operator >> and the "Cinfile" object associated with the file.

```
Cinfile >> input_variable_1 >> input_variable_2;
```

Closing Data Files

After completing writing data to a file or reading data from a file, the file should be closed. A file can be closed by calling the close function. For example, for the files E6Q3_in and E6Q3_out opened, we can close them by the following statements

```
Cinfile.close( );
Coutfile.close( );
```

Example 1. Read an integer number N and a set of N data from a sequential file "E1Q1.in" and write it to a new file "E1Q1_out" (3 values per line).

Sol

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
using namespace std;
int main ( )
{
    ifstream Cinfile("E1Q1.in");
    if (!Cinfile)
   {
```

```
        cerr <<”Input file could not be opened” << endl;
        exit(1);
    }
     ofstream Coutfile(“E1Q1.out”);
     if (!Coutfile)
    {
        cerr <<”Output file could not be opened” << endl;
        exit(1);
    }
    int n, i;
    float  x[100];
    Cinfile >> n;
    for (i=1; i<=n; i++)
       Cinfile >> x[i];
    for (i=1;  i<=n; i+=3)
       Coutfile<<x[i]<<“ ”<<x[i+1]<<“ ”<< x[i+2]<< endl;
    Cinfile.close();
     Coutfile.close():
}
```

Example 2. Write a program to read the character string stored in the file cch5_in and then print the character string.

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
void main()
{
    ifstream Cinfile("cch5_in");
    char ch;
    while (!Cinfile.eof())
    {
        Cinfile.get(ch);
        cout<<ch;
    }
        Cinfile.close();
}
```

Remarks

- The function `eof()` returns a nonzero value if the end of file is reached. So the while loop in the program will loop until the end of the file is reached , and hence the whole character string will be read and printed.

- Cinfile is an object of the ifstream class. This class has a member function called get() that can read one character each time, and so we can use it to read the character string via a while loop.

7.4 Application: Calculation of the Dominant Eigen Value of Matrices

In many cases, we require only one eigenvalue or eigenvector of the matrix A. For example, to analyse the convergence of the iterative methods for solving $A\mathbf{x} = \mathbf{b}$, we only need to calculate the eigenvalue with maximum modulus of the matrix A.

In this section, we introduce the Power Method, which is an iterative technique for finding the dominant eigen-solution $(\lambda_1, \mathbf{x}_1)$ of a matrix, where λ_1 is the eigenvalue with maximum magnitude and $\mathbf{x}_1$ is the associated eigenvector, i.e.

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \text{ ...... } \geq |\lambda_n| .$$

It is also assumed that the eigenvectors of A are linearly independent and that they are normalised so that their largest element is unity.

Numerical Algorithm of the Power Method

Let A be an $n \times n$ matrix. To find λ_1 and $\mathbf{x}_1$,

(1) Choose an $n \times 1$ column vector $\mathbf{z}^{(0)}$ whose largest element is unity;

(2) Perform iterations $\begin{cases} \mathbf{y}^{(k)} = A\mathbf{z}^{(k-1)} \\ \mathbf{z}^{(k)} = \dfrac{\mathbf{y}^{(k)}}{\mu_k} \end{cases}$ for $k = 1, 2, ...$

until convergence is achieved, where μ_k is the element of $\mathbf{y}(k)$ with largest modulus.

Remark: As $k \to \infty$, $\begin{cases} \mu_k \to \lambda_1 \\ \mathbf{z}^{(k)} \to \mathbf{x}_1 \end{cases}$

where λ_1 is the dominant eigenvalue, and $\mathbf{x}_1$ is the associated eigenvector with the element of largest modulus being one.

Analysis of the Power Method

In the following we prove

$\begin{cases} \mu_k \to \lambda_1 \\ \mathbf{z}^{(k)} \to \mathbf{x}_1 \end{cases}$ as $k \to \infty$.

Proof

It has been assumed that the normalised eigenvectors of A, $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, are linearly independent, so

$$\mathbf{z}^{(0)} = \sum_i^N \alpha_i \mathbf{x}_i$$

From $\mathbf{y}^{(k)} = A\mathbf{z}^{(k-1)}$,

$$\mathbf{y}^{(1)} = A\mathbf{z}^{(0)} = \sum_i \alpha_i A\mathbf{x}_i = \sum_i \alpha_i \lambda_i \mathbf{x}_i, \qquad \mathbf{z}^{(1)} = \frac{\mathbf{y}^{(1)}}{\mu_1} = \frac{\sum \alpha_i \lambda_i \mathbf{x}_i}{\mu_1}.$$

$$\mathbf{y}^{(2)} = A\mathbf{z}^{(1)} = \frac{\sum \alpha_i \lambda_i A\mathbf{x}_i}{\mu_1} = \frac{\sum \alpha_i \lambda_i^2 \mathbf{x}_i}{\mu_1}, \qquad \mathbf{z}^{(2)} = \frac{\mathbf{y}^{(2)}}{\mu_2} = \frac{\sum \alpha_i \lambda_i^2 \mathbf{x}_i}{\mu_1 \mu_2}.$$

In general $$\mathbf{z}^{(k)} = \frac{\sum \alpha_i \lambda_i^k \mathbf{x}_i}{\mu_1 \mu_2 \ldots \mu_k} = \frac{\lambda_1^k \left[\alpha_1 \mathbf{x}_1 + \alpha_2 (\lambda_2 / \lambda_1)^k \mathbf{x}_2 + \ \ldots \ + \alpha_n (\lambda_n / \lambda_1)^k \mathbf{x}_n\right]}{\mu_1 \mu_2 \ \ldots \ \mu_k}.$$

As $|\lambda_i| < |\lambda_1| \quad \forall\, i \neq 1$, $\quad \mathbf{z}^{(k)} \to \dfrac{\alpha_1 \lambda_1^k}{\mu_1 \mu_2 \ldots \mu_k} \mathbf{x}_1$ as $k \to \infty$.

Further as both $\mathbf{z}^{(k)}$ and $\mathbf{x}_1$ are normalised vector with largest element unity, we have from above that

$$\mathbf{z}^{(k)} \ \to \ \mathbf{x}_1 \ , \quad \text{as} \quad k \to \infty. \tag{*1}$$

$$\frac{\alpha_1 \lambda_1^k}{\mu_1 \mu_2 \ldots \mu_k} \ \to \ 1 \ , \quad \text{as} \quad k \to \infty. \tag{*2}$$

Equation (*2) implies that

$$\begin{cases} \mu_1 \mu_2 \ldots \mu_k & = \alpha_1 \lambda_1^k \\ \mu_1 \mu_2 \ldots \mu_k \mu_{k+1} & = \alpha_1 \lambda_1^{k+1} \end{cases} \qquad \therefore \quad \mu_{k+1} = \lambda_1 \quad \text{as } k \to \infty. \tag{*3}$$

Convergence Test

We usually stop computation and let $\lambda_1 = \mu_k$, $\quad \mathbf{x}_1 = \mathbf{z}^{(k)}$ if

1^0 The size of λ_1 converge, i.e.,

$$\frac{|\mu_k - \mu_{k-1}|}{|\mu_{k-1}|} < \text{Tol}. \tag{*4}$$

2^0 The element with maximum modules occurs at the same position of the eigenvector at any two consecutive iterations.

Example Write a program to find the dominant eigenvalue and its associated eigenvector using the power method. Hence find the dominant eigen-solution of the matrix

$$A = \begin{bmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{bmatrix}$$

with $\mathbf{z}^{(0)} = [0,1,0]$

Requirement . Read input data from a pre-created data file "power_in" ;

Solution.

Based on the formulae for the power method, the following algorithm can be developed

Algorithm:

```
Set   Niter = 0
      NewMu = 0
Do While (Niter < MaxNit)
      Set  Niter = Niter+1
           OldMu = NewMu
      Call  FindYmult (N, A, Z, Y)            –  y(k) = Az(k – 1)
      Call  FindMu (N, Y, NewMu, NewI)        –  Find μk from y(k)
      Call FindZ (Y, NewMu, Z)                –  z(k) = y(k)/μk
      If ( |NewMu – OldMu| ≤ Tol * |OldMu|  and  NewI = OldI  ) Then
           Set  Ierr = 0
           Return
      EndIf
  EndDo
  Ierr = 1
  Return
```

The algorithm can then be implemented in computer using C++. The program written based on the above algorithm is as follows.

```
/*
  Program for calculating dominant eigen value and associate
             eigen vector using the Power Method       */
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <stdlib.h>
void powereig(int ,float [][10],float [],float ,int );
void findymult(int ,float [][10],float [],float[]);
```

```
void findmu(int ,float []);
void findz(float [],float [],int );
//declare global variables
int err;
float newmu;
//
int main ()
{
  int n, max_iter, i, j;
  float a[10][10],z[10],tol;
//
//open data file "power_in" for input
  ifstream Cinfile("power_in");
  if(!Cinfile)
  {
    cerr<<"file could not be opened"<<endl;
    exit(1);
  }
//
//read data from the file
  Cinfile >>n>>max_iter>>tol;
  for (i=0;i<=n-1;i++)
    for (j=0;j<=n-1;j++)
       Cinfile>>a[i][j];
    ;
  for (i=0;i<=n-1;i++)
       Cinfile>>z[i];
//
/* call the function powereig to find the eigen
       value with largest magnitude            */
  powereig(n,a,z,tol,max_iter);
//
//print results
  if(err==0)
    {
      cout<<"The dominant eigen value =  "
      cout<<newmu<<endl<<" \n";
      cout<<"The associate eigen vector is: ";
      cout<<" [";
      for (i=0; i<=n-1; i++)
          cout <<"  "<<z[i]<<"  ";
      cout<<"]"<<endl;
    }
  else
    cout<<"not converge";
  return 0;
```

```
}
//
int newi;
voidpowereig(int n,float a[][10],float z[],float tol,int max_iter)
{
  int i,oldi;
  float y[10],oldmu;
  newmu=0;
  newi=0;
  err=1;
//
  for ( i=1;i<=max_iter;i++)
  {
    oldmu=newmu;
    oldi=newi;
    findymult(n,a,z,y);
    findmu(n,y);
    findz(y,z,n);
    if(fabs(newmu-oldmu)<=tol*fabs(oldmu)&&
(newi==oldi))
    {       err=0;
            break;
    }
/* for printing results in each iteration
     cout<<"Iteration "<<i<<endl;
     cout<<"Estimate of the dominant eigen value= "<<newmu<<endl;
     cout<<"The associate eigen vector is ";
     cout<<"   [";
     for (int k=0; k<=n-1; k++)
          cout <<z[k];
          cout<<"]"<<endl<<"\n";
*/
  }
}
//
void findymult(int n,float a[][10],float z[],float y[])
{
  int i,j;
  for(i=0;i<=n-1;i++)
  {
    y[i]=0;
    for (j=0;j<=n-1;j++)
       y[i]+=a[i][j]*z[j];

  }
}
//
```

```
void findmu(int n,float y[])
{
  int i;
  newi=0;
  newmu=y[0];
  for (i=1;i<=n-1;i++)
  {
    if(abs(y[i])>abs(newmu))
    {
      newmu=y[i];
      newi=i;
    }
  }
}
void findz(float y[],float z[],int n)
{
  int i;
  for(i=0;i<=n-1;i++)
     z[i]=y[i]/newmu;
}
```

Input Data stored in "power_in"

```
3 100 0.0001
-2  1    0
 1 -2    1
 0  1   -2
 0  1    0
```

Output results

```
The dominant eigen value = -3.414201
The associate eigen vector is: [-0.707106 1.000000 -0.707106].
```

EXERCISE 7

Q7.1 Rewrite the program for Q6.2 so that data values are read from a data file "E7Q1.in".

Q7.2 Rewrite the program for Q6.3 so that data values are read from a data file "E7Q2.in" and results are written to a new data file "E7Q2.out".

CHAPTER 8

C++ Classes and Object Oriented Programming

8.1 Introduction

Fortran and C are action oriented programming languages while C++ is an object-oriented programming language. A C++ program typically consists of declarations, classes, a main function and optionally plus various other functions. Class is the main feature, which distinguishes C++ from C and Fortran.

A class can contain data (data members) as well as functions (member functions) that manipulate the data and provide services to clients. Once a class is defined, it can be used to create objects with each performing a specific task as defined by the class. Fr example, for the banking system, we can create a bank-account class which include member functions to make a deposit, make a withdrawal and to enquiry balance. Then, we can create, from the bank account class, an object for a customer, providing the customer with the banking service, obviously packing software as classes make it possible for other future software systems to reuse the classes.

In this chapter, we describe

- how to define a class with data members and member functions,
- how to call member functions to perform some tasks.

8.2 Defining a Class

A C++ class usually consists of a number of data members and member functions and is defined by the following structure:

```
class class_name
{
    access specifier:
            member function1
            member function2
                   ⋮
    access specifier:
           data member1
           data member2
                ⋮
};
```

Remarks

- The access specifier includes `public`, `private` and `proteted`, and is used to indicate whether the data members and functions are "available to the public" .
- Data members are used to describe the characteristics of each individual object of the class, namely different object could have different characteristics described by different values of the data members. The member functions can be used to manipulate the characteristics (attributes) of the object.

Example: The following figure shows the definition of a `Bank_account` class which includes member functions to make a deposit, make a withdrawal and inquire the current balance.

```
1.   // ch8_1.cpp
2.   #include <iostream>
3.   using namespace std;
4.   class  Bank_account
5.   {
6.       public :
7.           float deposit(float amount)
8.          {
9.               balance += amount;
10.              return balance;
11.         }
12.         float withdrawal(float amount)
13.         {
14.               balance -= amount;
15.               return balance;
16.         }
17.         float current_balance()
18.         {
19.               return balance;
20.         }
21.     private :
22.         int accountNumber;
23.         float balance;
24.   };
```

Fig. 8.1 Definition of a `Bank_account` class

Notes

(1) The class definition begins with the keyword class followed by the class name Bank_account (line 4). By convention, the name of a class begins with a capital letter to distinguish it from object names. The class' body is enclosed in a pair of braces , { and } , as in line 5 and 24. The class definition ends with a semicolon (line 24).

(2) Data members and member functions in a class may be declared as

public (accessible externally),
protected or private (not accessible externally).

The keyword *public* (line 6) indicates that the member functions appearing after this access specifier are "available to the public" , namely it can be called by other functions in the program and by member functions of other classes. On the other hand, variables and functions declared after the keyword *private* (line 21) are accessible only to member functions in the class.

(3) Lines 7-11 define a member function "deposit"; lines 12-16 define another member function "withdrawal", lines 17-20 define a member function "current_balance".

(4) Lines 21-23 are declarations for two data members. Every object of the Bank_account class contains one copy of each of the class's data members. For example, if there are two bank_account objects, then each object will have its own account number and balance.

(5) The structures in C ad C++ (created by the struct statement) can be defined by classes with only public data members. For example, in the simulation of particle motion, one needs to know the particle diameter, particle material density and particle stiffness. To describe these properties, a class "particle" can be defined as follows

```
class particle
{
  public:
        float diameter;
        float density
        float stiffness
};
```

which is equivalent to the following structure

```
struct particle
{
        float diameter;
        float density
        float stiffness
};
```

8.3 Calling Member Functions

Creating Objects

Once a class is defined, it can be used to create objects to perform certain tasks. For example, from the `Bank_account` class, we can create an object to service a bank customer. An object can be created by a declaration statement (similar to type declaration of variables) as follows:

```
class_name  object_name;
```

eg

```
Bank_account  a, b, c;
```

creates three Bank_account objects associated respectively to the bank accounts of customers a, b and c.

Initializing Objects

Once an object of a class is created, a constructor call is made implicitly to initialize the object before it is used in the program.

A constructor is a special member function that has the same name as the class, and is called automatically when an object is created. The following shows the definition of a constructor for the `Bank_account` class, which assigns the constructor's parameter value (accountBalance) to the data member `balance`.

```
Bank_account(float accountBalance)
{
     balance=accountBalance;
}
```

Note:

- If a class does not explicitly include a constructor, the compiler provides a default constructor, namely a constructor with no parameters.

Calling Member Functions

To call a member function of the class so as to change the characteristics of an object (such as to deposit to change the balance of the account), we need to reference the object name and the member function name using a statement of the following function.

```
object_name.member_function_name(actual argument)
```

For example in the `Bank_account` example, the `Bank_account` object **`a`** represents the bank account of customer **`a`**. If the customer is to deposit $2000, then one would need to call the deposit function of the `Bank_account` class, which can be furnished by using the following statement:

```
a.deposit(2000);
```

Example.

The following figure shows the complete C++ codes for defining the `Bank_account` class (with a constructor) and calling the member functions to deposit $2000, withdraw $5000 and enquire the balance, assuming that the current balance before banking is $10,000.

```
1  // ch8_1.cpp
2  #include <iostream.h>
3  #include <cstdlib>    //your compiler may use diff name
4  using namespace std;  //your compiler may use diff name
5  class  Bank_account
5  {
7     public :
8           Bank_account(float accountBalance)
9         {
10                balance=accountBalance;
11        }
12
13          float deposit(float amount)
14         {
15              balance += amount;
16              return balance;
17         }
18         float withdrawal(float amount)
19         {
20               balance -= amount;
21               return balance;
22         }
23         float current_balance()
24         {
25               return balance;
26         }
27     private :
28          int accountNumber;
29          float balance;
30 };
31
32  /*   Main function    */
33  int main()
34  {
35     Bank_account a(10000); //constructor
36     a.deposit(2000);
37     cout<<" balance after deposit="<<a.current_balance();
38     a.withdrawal(5000);
39      cout<<"\n";
40     cout<<"balance after withdrawal="<<a.current_balance();
41     system("PAUSE"); //may be deleted
42     return 0;
43    }
```

Fig. 8.2 A simple example: Bank_account project

Remarks

- Line 5-30 defines the Bank_account class with a constructor;

- Line 35 creates a bank account object with initial account balance 10000;
- Line 36 calls the deposit function to deposit 2000;
- Line 37 calls the balance function to check the new account balance after deposit and then prints the new account balance;
- Line 38 calls the withdrawal function to withdraw 5000;
- Line 39 calls the balance function to check the new account balance after withdrawal and then prints the new account balance.

To check whether the program as shown in Fig. 8.2 working properly, we run the program and obtained the following results which are as we expected.

```
Current balance after deposit =12000
Current balance after withdrawal =7000
```

8.4 Improve Reusability of Classes

Here we discuss two methods for improving the reusability of C++ classes .

Placing a Class in a Separate File for Reusability

Normally in building a C++ class, reusable source codes (such as classes) are saved in a header file that has a .h filename extension.

In any program that uses the class defined, one just need to use the "# include" preprocessor directives to include header files that contain the classes. For example, the above `Bank_account` class definition can be saved into a separate file, say bank_account.h, and the main program only needs to include a statement

```
#include  "bank_account.h"
```

By this, programmers only need to design C++ codes specific to the problem to solve and can utilize a huge amount of classes available at the time of software development.

Separating Interface from Implementation

The interface of a class gives information on what services (functions) the class's object can use and how to request the services (functions). More specifically, a class's interface can be specified by a class's definition that lists only the member functions names, return type and the type of each of the parameters.

Obviously, in constructing a class, the programmer must design the source codes for the member functions to provide the expected services. However, once a class is created, one only needs to know the interface in order to use it. It is not necessary to know how the class carries out the services. Therefore, to improve the usability of classes, member functions usually are defined outside the class definition, so that their implementation details can be hidden from the clients. This ensures the simplicity and efficiency of software development, as programmers can utilize a huge amount of classes available without knowing how they work, and thus can write codes independent of the class implementation.

Example

As an example, the bank account program in Fig 8.2 can be separated into two parts and saved separately into two different files bank_account.h and bank_account.cpp.

bank_account.h

This header file (as shown in Figure 8.3) contains the class's interface with function prototype. A function prototype is a declaration of a function that gives function's name, its return type and the data type of each of its parameters. The header file also specifies the class's private data members so that the compiler knows how much memory to reserve for each object of the class.

```
// bank_account.h

class Bank_account  //class interface
{
    public:
        Bank_account(float accountBalance)
         {
              balance=accountBalance;
         }
         float deposit(float amount );
         float withdrawal( float amount );
         float current_balance();
    private:
         int accountNumber;
         float balance;
};
```

Figure 8.3 Bank_account Class Definition (Interface)

bank_account.cpp

This C++ file (as shown in Figure 8.4) defines the bank_account member functions which are declared in the header file of the class.

```
// bank_account.cpp  // member funs definition

#include <iostream.h>
#include  "bank_account.h"

float Bank_account::deposit(float amount)
{
     balance += amount;
     return balance;
}
float Bank_account::withdrawal(float amount)
{
     balance -= amount;
     return balance;
}
float Bank_account::current_balance( )
{
     return balance;
}
```

Figure 8.4 Bank_account Class Member Function definitions

Remarks

When a member function of a class is defined outside the body of the class, it is necessary to indicate which class the function belongs to by adding "`class_name::`" in front of the function name such as

```
float Bank_account::deposit( float amount)
```

Testing

To check whether the program as shown in Figs. 8.3 and 8.4 working properly, we create a project named bank_account.prj containing three files: bank_account_main.cpp (containing the codes as shown in Fig 8.5), bank_account.cpp and bank_account.h in the same directory, compile all cpp files which are automatically linked together, and then run the executable program. This yields the following output on the screen, that is correct.

```
Current balance after deposit =12000
Current balance after withdrawal =7000
```

```
// bank_account_main.cpp

#include <cstdlib>
#include <iostream>
#include "bank_account.h"
using namespace std;
int main()
{
    Bank_account a(10000); // initializes the accountBalance
    a.deposit(2000);
    cout<<"balance after deposit="<<a.current_balance();
    a.withdrawal(5000);
    cout<< "balance after withdrawal="<<a.current_balance();
    system("PAUSE");
    return 0;
}
```

Fig.8.5 The Main Function for the Bank_account Project

EXERCISE 8

Q8.1 (Invoice class) Create a class called invoice that a book shop may use to represent an invoice for a book sold at the shop. An invoice should include three information as data members, book name, a quantity of the item being perchased and a price per item. The class should have a constructor that initializes the data members and provides a set function for each data member. In addition, provide a member function named getTotalInvoice that calculate the invoice amount, then returns the amount as an float value. If the quantity is not positive, it should be set to zero. If the price per item is not positive, it should be set to zero. Write a test program that demonstrates class invoices capabilities.

Q8.2 Modify the class in Fig. 8.2 such that for the case of withdrawal, the debit amount does not exceed the account's balance. If it does, the balance should be left unchanged and the function should print a message indicating "debit amount exceeded account balance".

Part II Fortran 95

Fortran (Formulae translation) is one of the first computer languages which was originally developed in 1954. Since then, several versions of Fortran have been developed including F66, F77, F95 and etc. F95 is the latest version of the world's oldest and most widely used scientific computing language.

Fortran has been the dominant computer language for engineering and scientific applications in academic circles and many other areas. It always emphasises on efficiency and ease of use. The latest version of Fortran has accommodated many useful features from other programming languages such as pointers and dynamic memory allocation. In addition, Fortran 95 provides two special features including whole array processing and kind type parameters to simplify matrix calculation and to make it possible to write programs portable to most computers. Today, F95 is till one of the dominant computer languages for scientific and engineering applications that require complex mathematical computations.

In this part, you will learn how to use F95 programming language to write F95 program for complex mathematical computations.

CHAPTER 9

F95 ARITHMETIC COMPUTATIONS

Arithmetic operations (adding, subtracting, multiplying and dividing) are the most fundamental operations performed by computers. To be able to write programs for these operations, we need to know how to store data values in computers, how to implement computations, how to input data values to computers and how to print the computed results. Thus in this chapter, we describe

- Methods of storing data with F95 - using constants and variables
- Assignment statements for arithmetic computations
- Simple input/output statements - introduce data values into computers and print results
- Construction of a complete F95 program - layout of F95

9.1 Constants and Variables

Numbers can be introduced into a program by either direct use with constants or indirect use with variables.

Eg. To calculate the area of a circle of radius R, we can use the following program

```
Read(*,*) R
Area=3.14159*R**2
Print *, Area
```

where	*3.14159*	*is used as a constant*
	R	*is a variable used for storing the value of radius*
	Area	*is a variable used for storing the computed result*

When the 1st statement is executed, the computer will wait for the entry of a value from keyboard. The value, once entered, will be assigned to R and execution continues to next statement

Constants

- Constants are numbers used directly in Fortran statements, such as 3.14159, 2, -2.5 etc. Constants may contain plus or minus signs and decimal points, but they may not contain commas.
- In Fortran, there are 6 intrinsic types of constants:

Integer: Integer numbers are whole numbers with or without sign, for examples : 5, -5

Integer constants cannot contain decimal points and commas, eg, 5. 12,000 are not valid integers.

Integer constants are always held exactly in the computer memory

Real: A real number is a number consisting of an integer part and a fractional part.

Real constants can be represented in two forms

Decimal form : -12.3, 0.0, etc.

Exponential form : 10.3e8, -5.2e-10 representing 10.3×10^{8} and -5.2×10^{-10}

Exponent must be integer, eg. 3.0e5.6 is invalid.

- Real numbers are stored in computers in exponential form, eg. -0.135782123e-5

-	- 0 5	1 3 5 7 8 2 1 2 3

- There are limitations on the magnitude and precision of values that can be stored in a computer. All limitations on values depend on the specific computer. Most computers have 7 effective digits for real variables. For such a computer, 23 in the number of above example will be cut off.

Double Precision (used in f77 and is not recommended for new programs)

Complex
Logical
Character } to be described in detail in Chapter 12.

Fortran 95 also includes the capability for programmers to create their own data types to supplement the intrinsic types. For more details, see reference 1.

Variables

A variable represents a memory location that is assigned a name. The memory location can be used to store a value. Once we need the value stored, we reference it with the variable name assigned to the memory location. We can also store a new value in the memory location and in this case, the old value is destroyed and lost.

To store different types of values, we need to use different types of variables. There are 6 intrinsic types of variables, namely, integer, real, double precision, complex, logical, character. In the following, we describe how to name a variable and how to declare the type of a variable in computer programs.

Name A F95 variable name must obey the rules which apply to all F95 names, namely

- It **must begin with a letter (a-z, A-Z).**
- It may be optionally followed by up to 30 more characters.
- It may only contain the letter a-z and A-Z, the digits 0-9 and the underscore _.

Example: 2X, V.2 and X$ are all not valid F95 variable names.

Note: * Upper and lower case letters are treated as identical.
* In F77, underscore is not acceptable and a name can only contain a maximum of 6 characters.

***Type*:** All variables must be declared in the program by a statement of the form

TYPE : : name1,name2,...

where TYPE specifies the data type for which memory space is to be reserved.

eg.

`REAL::I,N,K` declares *i, n*, and *x* as real variables

`INTEGER::K,S,D` declares *k,s* and *d* as integer variables

Notes

- In F77, variable declarations do not contain a double colon.
- If you omit to declare a variable, it will not normally lead to a syntax error . However the variable will be implicitly declared by the following implicit rules:

Default Implicit type rule (**I-N** rule):
By default, any variables starting with character *I* to *N* are integer and all other real. If you are not happy with this rule, you can change it using the following statement

Declared Implicit type rule (this rule can be used to change the **I-N** rule):
eg

IMPLICIT INTEGER (P-R), REAL(M, N)

declares that: All variables beginning with P to R are integer,

All variables beginning with M and N are real,

The I-N rule remains in effect for variables beginning with letters that do not appear in implicit statements.

Declared implicit type statements only affect the variables beginning with a letter in the letter list of the type statement.

The implicit rules usually cause many problems. Thus it is **strongly recommended not to use implicit rules in new programs**. F95 provides a means to avoid using implicit rules by instructing the compiler that all variables must be declared before use. This can be achieved by including the statement

```
IMPLICIT NONE
```

9.2 Assignment Statements

There are only two ways in which a variable can be given a value during the execution of a program - by assignment or by a read statement. We will discuss the assignment statement here.

- Arithmetic computations(+, - *, /) can be implemented in Fortran by using assignment statements.
- General form: Variable_name = expression
- Execution: once an assignment statement is executed, the following two processes occur in the computer

  1) first, calculate the value of the expression

  2) then assign the value to the variable on LHS.

eg.

X=2.0	assign 2.0 to x
Y=X+2.5	evaluate (X+2.5) to yield 4.5, then the value 4.5 is assigned to Y
Y=Y+1	Y will be assigned a value equal to its current value plus 1. Thus Y will become 5.5.

Notes: Only a single variable can appear on the LHS, Eg x+1=y is not allowed.

To understand how to perform arithmetic computing by using assignment statements, we need to know how to translate mathematical formulae to arithmetic expressions and how to evaluate an expression.

9.2.1 Writing Arithmetic Expressions

Expression: An expression is a combination of constants, variables, intrinsic functions, operators and parentheses which can be evaluated to give a single value, eg.

2+x (2+x+sin(y))/2.0

where *x* and *y* denote variables which have been assigned values previously.

a) Intrinsic Functions

Scientific computing usually requires many simple operations such as calculating the sine of an angle. As these operations are so common, they are built in the computer and we can use them directly in the program.The following is a list of some common functions

Function Name	Function Type	Definition
ABS(X)	*Real*	$\lvert X \rvert$
SQRT(X)	*Real*	$\sqrt{X}$
EXP(X)	*Real*	e^x
SIN(X)	*Real*	*Sine of X*
COS(X)	*Real*	*Cosine of X*
ASIN(X)	*Real*	*Arcsin X*
ACOS(X)	*Real*	*Arccos X*
TAN(X)	*Real*	*Tangent of X*
ATAN(X)	*Real*	*Arctangent of X*
LOG(GX)	*Same as GX*	*log GX*

where X represents a real value

An intrinsic function can be referenced in an expression using the following form :

```
Function-Name (argument, ...)
```

eg.

```
Y=SORT(b**2-4.0*a*c)+1.0 ;    z=sin(x)+2*cos(x);   c=exp(2.5)
```

Notes: 1) The argument can be a constant, a variable or an expression, eg. SQRT(3.0), SQRT(x), SQRT(2+x).

2) Some functions require a particular type of input and return a particular type of value, which can be found from most Fortran books

b) Arithmetic Operators

The basic arithmetic calculations can be expressed by using the following operators

Operation	Operator	Algebraic form	Fortran
Addition	+	a+b	a+b
Subtraction	-	a-b	a-b
Multiplication	*	$a \times b$	a*b
Division	/	a/b	a/b
Exponentiation	**	a^3	a**3

Note that No two arithmetic operators may be adjacent (consecutive)

eg. : SUM*(-7) is ok, but SUM*-7 is invalid.

9.2.2 Evaluating Arithmetic Expressions

An expression can be evaluated to yield a single value. In order to translate a mathematical formula correctly to a fortran expression to yield an expected value, we need to know the following points (a) to (d).

a) Priorities of Operations

Because several operations can be combined in one arithmetic expression, it is important to know the priorities of the operations (the order in which the operations are performed). For example, consider

```
Y=2.0*3.0**2
```

If the exponentiation is performed first, we obtain y=18.0; while if multiplication is performed first, we obtain y=36.0.

Fortran assigns the same priorities to operators as does mathematics, as shown in the following table.

Operations in brackets ( )	- inner most,	highest priority
Intrinsic functions	- L to R	↓
Exponentiation **	- R to L	
Multiplication/division */	- L to R	
Addition, subtraction + -	- L to R	lowest

Within the same level of priority, evaluation will proceed from left to right except in the case of exponentiation where the calculation starts from right to left,

eg.1, a**b**c = $(a)^{(b^c)}$

eg.2 For a=1.0, b=2.0, c=0.5

```
X= ( -B + (B**2 - 4.0*A*C)**2)/(2.0*A)
```
⇒ X=1.0

b) Mixed-Mode Operations

When an arithmetic operation is performed using two real numbers, its intermediate result is a real value. If not all the operands are of the same type, we have the so-called mix-mode operation. The mix-mode operation between an integer and real number yields a real number. In summary,

Real	--	Real	⇒	Real
Integer	--	Integer	⇒	Integer
Real	--	Integer	⇒	Real

c) **Truncation Error:**

When a computer stores a real number in an integer variable, it ignores the fractional portion and stores only the whole number portion of the real number, this lost is called truncation. In Fortran, truncation errors usually arise from the following two sources:

Integer_variable = expression

Once the assignment statement is executed, the expression is first evaluated. If the value of the expression is real, then the real value is to be stored into the integer variable. As an integer variable can only store the whole number part, the fraction portion of the real value is ignored.

eg. For *K*=3.14*2 , if *K* is an integer variable then *K* will store a value 6 (not 6.28) after the statement is executed

Integer Division

As the intermediate result of integer division is integer, the fraction portion of the quotient is discarded.

eg. In computer programs, 3/4 yields 0 not 0.75; 1/2 yields 0 and 6/5 yields 1.

For *l*=2, *n*=3, *j*=4, *l*/n**j* yields 0 but j**l*/*n* yields 2, assuming that *l, n, j* are all integer variables.

d) Magnitude Limitations, Under Flow and Over Flow

Every computer has limitations on the maximum & minimum magnitudes of values it can store. This information usually can be obtained from the reference manual of the computer.

eg. VAX	Integer:	Maximum magnitude 2147483647
	Real:	Minimum magnitude 10.0e-38, maximum magnitude 10.0e+38

If the magnitude of a value obtained in calculation is smaller than the minimum magnitude which the computer can recognize, an error called underflow error occurs. If the magnitude of a value obtained is larger than the maximum value, overflow error occurs. If these errors occur, firstly check the algorithm for other mistakes.

9.3. Simple Input/Output Statements

9.3.1. List-directed Input

A program can read data value by using the following statement

READ*, *variable_list* ⇔ READ(*,*) *variable_list*

eg

```
READ*, A,B
```

- When this statement is executed, the computer first waits for the entry of two values from the default input unit such as keyboard.
- Once two values are entered, the first value is assigned to A and the second to B.

Notes:

- The variables in the *variable-list* must be separated by comma,
- The input data should be separated by space or comma, if the variable is integer, the corresponding input data should also be integer,
- Each read statement will read as many lines as needed to find new values for the variables.

  eg. READ *, X, Y, II

  if the input data is { 1.5, 2.5,
3,

  then the values stored in variables are X=1.5, Y=2.5, II=3

- Each read statement begins reading data from a new line

  eg.. READ *, A
READ *, B

  If input data values are all in one line, i.e, 2.5, 2.6
then A=2.5, B=0.0 (as the 1st read statement reads from the 1st data line, the 2nd read statement reads from a new data line- line 2)

9.3.2 List-directed Output

A program can produce output by using the following statement

PRINT *, [‘literal_1’,] expression_1 [, ...] ⇔ WRITE (*,*) expression_1, ...

where square brackets [ ] are not part of the statement, but denote an optional item.

eg. PRINT *, 'X=', X, 'Y=', Y, '(X+Y)/2=', AVERG

Notes: * Each expression should be separated by a comma.
* Each print statement begins to print from a new line.

9.4 Initial Values and Named Constants

Initial Values

In addition to read statements, there is one other method of giving a value to a variable, namely to provide an initial value for a variable as part of the declaration of the variable. This

is achieved quite simply by following the name of the variable by an equal sign and the initial value:

```
REAL ::       a=0.0, b=1.5, c,d,e=1e-6
INTEGER ::  max=100
```

These initial values will be assigned to the variables by the fortran processor before the execution of the program commences, thus avoiding the need, when the program is executed, either to execute a series of initial assignments or to read an initial set of values.

Note: In F77, initial values are assigned by means of a separate DATA statement

```
DATA   list of variable names /list of constants/
```

eg

```
DATA  X1, X2, X3, X4/2.0,3*0.0/
```

- The Data statements follow all type statements but precede any executable statements.
- 3*0.0   represents   0.0, 0.0, 0.0.

Named Constants

Frequently, a program will use certain constant values in many places and there is clearly no intention for these values to be altered. Fortran 95 allows us a convenient method of dealing with these situations by defining what is called named constant by use of the parameter attribute in a declare statement:

```
REAL, PARAMETER ::        Pi=3.14159, PI_by_2=PI/2.0
INTEGER, PARAMETER ::   MAX_Iter=100
```

- Once a named constant is defined, it is not permitted to attempt to change its value at a subsequent point in the program. The only way that its value can be changed is by modifying the declaration statement accordingly and recompiling the program.
- The parameter statements must be placed after all type statements but precede any executable statements.

Note: In F77, named constants are defined by a parameter statement.

9.5 Construction of a Complete F95 Program

The task of writing a program to solve a particular problem can be broken down into three basic steps

(3) specify the problem clearly

(4) analyse the problem, break it down into fundamental elements and then draw up a program design plan (use flow chart or pseudocode to present the plan)

(3) code the program according to the plan developed at step 2.

(4) Test the program exhaustively and repeat steps 2 and 3 as necessary until the program works correctly in all situations that you can envisage. *The fourth step is often the most difficult of all.*

A complete program starts with a PROGRAM statement followed by a number of specification statements (non-executable statements such as TYPE statements) and executable statements and ends with a END statement. The overall structure of a F95 program is shown below:

```
PROGRAM    program_name
IMPLICIT NONE
        non-executable statements
        executable statements
END PROGRAM   program_name
```

The Program Statement:

```
PROGRAM   program-name
```

Every main program unit must start with a PROGRAM statement.

The END Statement

```
END PROGRAM   program_name
```

alternative forms: END PROGRAM
END

Every Fortran program must end with an end statement, which tells the compiler to stop translating statements.

The STOP Statement

```
STOP
```

In some cases, we need such an executable statement to tell the computer to terminate execution of the program. This statement can appear anywhere in the executable portion of the program.

Layout of F95 Programs - free format

(F77 uses fixed format which is not recommended for new programs)

A fortran 95 program is typed line by line.

- A line may contain a maximum of 132 characters;
- A line may contain more than one statement, in which case, a semicolon separates each pair of successive statements.
- A statement may have a maximum of 39 continuation lines. A trailing ampersand (&) indicates that the statement is continued on the next line; if it occurs in a character context, then the first non-blank character of the next line must also be an ampersand, and the character string continues from immediately after the ampersand.
- Comment lines are identified by having an exclamation mark as their first non-blank character.
- Blank characters are significant in some compilers and thus can be used to separate names and constants from other names and constants.
- A statement label, if required, consists of up to five digits representing a number in the range 1 to 9999 which precedes the statement and is separated by at least one blank.

Example 9.5-1_ A fortran 95 progarm for calculating the average of two values.

```
PROGRAM    average
IMPLICIT NONE
        ! This program calculates the average of two values
        ! Variable declaration
                REAL :: n1, n2, average_value
        ! Input data values and calculate average
                READ *, n1,n2
                average_value=0.5*(n1+n2)
        ! Print the result
                print *, 'the value of n1=' , n1, 'the value of n2=', n2,  &
                          'the average value average_value=', average_value
END PROGRAM average
```

SUMMARY

In this chapter, we learn how to define constants and variables in F95. We discussed how to perform arithmetic operations using assignment statements. Some of the considerations that are unique to computer computations were also discussed: mixed-mode operations, truncation errors, magnitude limitations, underflow and overflow. Statements for reading information

from the terminal and for printing answers were also covered. The structure of a complete F95 program is also described with an example.

F95 Syntax Introduced in Chapter 9:

Initial Statement	**PROGRAM** *rogram_name*
Variable declaration	**REAL**: : *list of variable names*
	INTEGER:: list of variable names
Initial value specification	*type* :: *name=initial_value,…*
Name constant declaration	*type*, **PARAMETER** :: *name=constant_value, …*
Assignment statement	*variable_name=expression*
List-directed input	**READ *, *variable_list*** or **READ**(*,*) *variable_list*
List-directed output	**PRINT * ,** ['*literal*',] *expression***,…**
Arithmetic Operators	+, -, *, / , **
Intrinsic functions	ABS, SQRT, EXP, SIN, COS, ASIN, TAN, ATAN, LOG
End statement	**END PROGRAM** *program_name*
Stop statement	**STOP**

Other Key Points

Truncation errors caused by assigning real values to integer variables and by integer division
Order of F95 statements see section 9.5

EXERCISE 9

Q9.1. What is the difference between an integer and a real number ?

Q9.2 List two advantages of a real variable over an integer variable ?

Q9.3. Write each of the following real constants in exponential form (+0.**** E+**).

12.0, $0.126*10^{-13}$, 3.08, $6.023*10^{23}$, 18900000, -41800

(Ans: 0.1200E+2, 0.1260E-13, 0.3080E+1, ...)

Q9.4. Which of the following are not valid symbolic names of Fortran 90 variables? Why?

JOHNY, 2JOH, N/4, A.B, X_C, A15BB, VELOCITY

(Ans: 2JOH, N/4, A.B)

Q9.5 What is implicit declaration? How can it be prevented?

Q9.6 What is the general form of an assignment statement ?

Q9.7 What are Fortran's five arithmetic operators? what are their respectively priorities?
Q9.8. Convert the following formulae into Fortran assignment statements.

(a) $y = \dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$, (b) $z = \dfrac{6\sin(x+y)}{e^{3+a}}$, (c) $z = \sin^{-1}\left(\dfrac{y}{\sqrt{x^2 + y^2}}\right)$

(Ans: (a) y=(-b + SQRT (b**2 - 4*a*c)) / (2*a); ....)

Q9.9 What will be printed out by the following program (work it out by hand).

```
PROGRAM  Q9_9
IMPLICIT NONE
    real :: a, b, p, q, r
    integer :: x, y, z
    a=2.5;  b=4.0;  p=a+b;  x=a+b;  q=a*b;  y=a*b;  r=p/q;  z=x/y
        print *,  p, q, r, x, y, z
END PROGRAM Q9_9
```

Q9.10 How to assigned an initial value to a variable in a TYPE statement? Give two examples.

Q9.11. How to declare a name constant? Give two examples.

Q9.12 What is the maximum number of characters that may occur in one line of F95 program ?

Q9.13 What is the maximum number of lines that a F95 statement may be spread over?

Q9.14 What is the maximum number of F95 statements that may appear on a single line ? How are they separated ?

Programming Exercises

Program Debugging - *Some guidelines (for questions Q9.15 – Q9.20)*

If a program is not working correctly, you should consider taking some of the following steps to isolate the errors.

(4) Check the input potion of your program
- Check the syntax of the read statement.
- Check whether correct values have been assigned to variables in the READ statement. For this purpose, you need to add a PRINT statement immediately after the READ statement so that from the computer output , you can see whether the values you want to give the variables are being correctly assigned to the variables or not. A common mistake is to enter the data values in the wrong order.

(5) Check the assignment statement
- Double check the placement of parentheses. Be sure that you always have the same number of left parentheses as right parentheses.
- Review each variable name on the right-hand side of the assignment statement to be sure you have spelled it exactly as previously used.
- Make sure all variables on the right-hand side of the assignment statement have been previously assigned a value.
- Be sure that arguments of functions are in the correct order and correct data type. For example, trigonometric functions use angles in radians, not in degree.

(6) Check the Output
- Check the syntax of the output statement
- Do you have the correct variable names listed ?

Q9.15 Run the programs of Q9.9 in computer. If the printed results are different from your solutions worked out by hand, find the mistake you made in your hand calculation and think about why.
To perform computation in computers, you need to

1) Create a F95 file (*Q9_15*. f95):
   Save the program and quit to the command mode:
2) Compile the Program Q9_15.f95 to obtain an executable file.
3) Run the program followed by data values required by the read statements (if there is any) in the program.

Remarks: Once a read statement in the program is executed, the computer will wait for the entry of data values from the input device such as keyboard before executing the next statement. Hence, after you run, you need to enter data values for the variables listed in the first READ statement, then press the ENTER key. Then enter data values for the variables listed in the next READ statement, and so on.

Q9.16 The following simple program contains a number of errors. Identify these errors and then produce a corrected version.

```
PROGRAM  Q9_16
IMPLICIT NONE
real : number
! this program contains a number of errors &
          is not a good example of F95  !
print *, “type a number”
read *, “number”
print  “thank you &
            your number was”  number
END PROGRAM  Q9_16
```

Run the program on your computer to check that it does indeed work. If it still does not work, then keep correcting it until it does.

Q9.17 Enter the following program exactly as shown

```
PROGRAM  Q9_17
    ! This program contains three major errors and two examples of bad
programming style
    print *, please type a number
    read *  number
    print *, "the number you typed was",numbr
END PROGRAM Q9_17
```

The program contains three errors, only two of which will probably be detected by the compiler. There are also two additional mistakes in the program which, although not errors, are very poor programming practice, can you find all five. Now compile the original program, correct only those error detected by the compiler. Then run it again typing in the value 268 when requested. Was the answer that was printed correct? If not, why not? How could you improve the program so that the compiler found more of the errors?

Q2.18 Write a program that expects three numbers (two real and one integer) to be entered, but only uses one read statement, and then prints them out so that you can check that they have been input correctly. When typing in the numbers at the keyboard, try (a) typing them all on one line separated by commas or space; (b) typing them one on each separate line.

Q2.19 Write and run a program which will read 6 numbers and find their sum. Test the program with several sets of data.

Q2.20 The following program is intended to swap the values of var_1 and var_2:

```
PROGRAM  swap
IMPLICIT NONE
    real :: var_1=22.2, var_2=66.6
    ! Exchange values
        var_2=var_1
        var_1=var_2
    ! Print the new values in var_1 and var_2
        print *, var_1, var_2
END PROGRAM swap
```

The program contains an error, however, and will not print the correct values. Find the error and correct it so that it works properly.

CHAPTER 10

F95 CONTROL STRUCTURES

So far our programs have been made up of a few simple statements executed one after another. This kind of structure is called sequence structure. Obviously, in order to write useful programs, we need to be able to

* Execute some statements many times - looping or iteration (need a repetition structure).
* Choose between alternative statements based upon a condition - selection (need a selection structure).

10.1 Logical Expressions and Logical Variables

All forms of the Selection and Repetition structures use a condition to determine which path to take in the structure. In Fortran, a condition can be expressed by a logical expression. A logical expression is analogous to an arithmetic expression but is always evaluated to a value either true or false, instead of a number. The simplest forms of logical expression are those expressing the relation between two numerical values, namely the relational expression. In general, a condition can be expressed by a composite logical expression which is formed by combining relational expressions, logical constants and logical variables using logical operators.

a) Relational Expression (R.E.)

A relational expression compares the values of two arithmetic expressions using a relational operator, namely

Arithmetic_expression_1 `Relational_operator` Arithmetic_expression_2

eg. a > b+1 represents condition a > b+1

- List of Relational Operators:

Form 1	Form 2	
<	.LT.	(less than)
<=	.LE.	(less than or equal to)
>	.GT.	
>=	.GE.	
= =	.EQ.	
/=	.NE.	

Each of the six operators has two possible forms. Early versions of Fortran require form 2. F95 accepts both forms but we recommend that you use the form based on mathematical symbols in your programs for clarity.

- The relational expression can be used to describe simple conditions such as $a > b + 1$. If the condition is true, the expression yields a value .TRUE. , otherwise a value .FALSE.

eg A >= 3.5 yields a value .TRUE. if A=4

b) Logical Constants and Logical Variables

Logical Constants:

It has been established that evaluating a logical expression will yield a logical value either true or false. These two values are called logical constants and are written in the forms as follows

.TRUE. and .FALSE.

Logical Variables:

We can declare logical variables to store logical values (.true. or .false.). Logical variables can be declared in a program using the following statement

LOGICAL :: var_1, var_2 .....

c) Logical Operators and Logical Calculations

By combining the relational expressions together using the logical operators (.NOT. , .AND. , .OR. , etc), we can form a composite logical expression to describe a complicated condition, such as 0 < x+y < 2

- **Definitions of Logical Operators**

Operator	*Format*	*Value*	
.AND.	A .AND. B	.TRUE.	only if both A and B are .TRUE.
.OR.	A .OR. B	.TRUE.	if A or B or both of them are .TRUE.
.NOT.	.NOT. A		changes the value of A to the opposite value
.EQV.	A.EQV.B	.TRUE.	only if both A and B have the same logical value

where A and B can be a logical constant, a logical variable or a relational expression.

- **Priorities of logical Operations**

Type	*Operator*	*Execution order*	
Bracket	( )	1	Highest priority
Arithmetic Cal.		2	
Relational Cal.		3	
	.NOT.	4	
Logical Cal.	AND.	5	↓
	.OR.	6	
	.EQV.	7	Lowest

Example 10.1.1

For A=3.5, B=5.0, D=1.0 and C=2.5, evaluate

(A .GE. 0.0) .AND. ((A+C) .GT. (B+D)) .OR. .NOT. .TRUE. (Ans, F)

Note: The parentheses shown in the above example are not strictly necessary because the relational operators have a higher priority than logical operators, but to human eyes the inclusion of parentheses makes the true meaning of the expression much clear.

10.2 Selection Control

In practice, most problems require us to choose between alternative courses of action, depending upon circumstances that are not determined until the program is executed. The selection structure is used to choose different paths through our program. It is most commonly described in terms of a BLOCK IF construct or a CASE construct. In the cases in which the alternatives are mutually exclusive and the order in which they are expressed is unimportant, we usually use the CASE construct or otherwise we use the BLOCK IF construct.

10.2.1 The BLOCK IF Construct

```
block_name: IF (condition 1) THEN
                Statement_group_1 (SG1)
            ELSEIF (condition 2) THEN
                Statement_group_2 (SG2)
            ......

            ELSE
                Else block
            ENDIF block_name
```

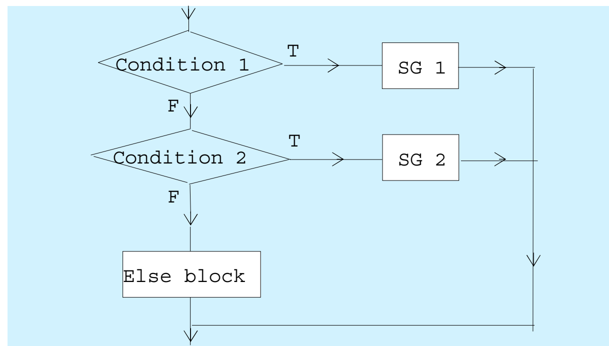


Fig 10.1 Fow chat of the bloack if construct (see page 118 for the notation)

- The inclusion of *block_name* (any name obeying the rules for F95 name) is optional.
- A block if construct is always introduced by a block IF statement and terminated by an ENDIF statement.
- There may be any number of ELSEIF statements, each followed by a block of statements or there may be none.
- There may be one ELSE statement followed by a block of statements or there may be none.

  eg: For two way selection, we can have

  IF-THEN-ELSE Statement

```
IF (condition) THEN
        then block (SG1)
ELSE
        else block
ENDIF
```

  IF-THEN Statement

```
IF (condition) THEN
        then block (SG1)
ENDIF
```

  If the then block has only one statement, we can use

```
IF( condition) executable_statement
```

- For a better appearance and readability, usually we indent the statement groups a few spaces to the right.
- The block IF construct can appear anywhere among the executable statements.

Execution Order:

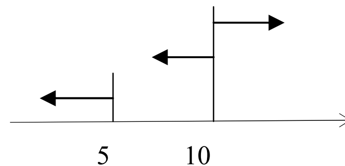
1) Evaluate Condition 1(C.1), if .TRUE. statement group 1 (SG1) is executed, then exit
   .FALSE. go to the statement with C. 2

2) Evaluate C.2, if .TRUE. SG2 is executed, then exit
   .FALSE. go to the statement with next condition

3) If none of the conditions are .TRUE. then the ELSE Block (if there is any) will be executed.

Note: *Statement_group_k* can be executed **only if** *condition k* is true and all other previous conditions are false.

Example 10.2.2. Calculate $Y=\begin{cases} 2+x & \text{if } x \le 5 \\ 4.5+0.5x & \text{if } 5 < x \le 10 \\ 19.5-x & \text{if } x > 10 \end{cases}$




```
IF (X <= 5.0)  THEN
         Y=2.0+X
ELSEIF (X <=10)
         Y=4.5+0.5*X
ELSE
         Y=19.5-X
ENDIF
```

← *(If this statement is executed, it means that the previous condition $x \le 5$ is not true, i.e. 5<x is true. Thus we only need to test the condition $x \le 10$)*

10.2.3 The CASE Construct

In addition to the block if construct which caters for the ordered choice situation, F95 provides another form of selection, known as the CASE construct, to deal with the alternative situation in which the various alternatives are mutually exclusive and the order in which they are expressed is unimportant. Its overall structure is shown as follows.

```
case_block_name:    SELECT CASE (case_expression)
```

```
CASE (case_selector_1)
    block_1 of F95 statements
CASE (case_selector_2)
    block_2

CASE DEFAULT
    block_D
END SELECT case_block_name
```

- The inclusion of case_*block_name* is optional.
- A case construct is always introduced by a SELECT CASE statement and terminated by an END SELECT statement.
- There may be any number of CASE statements, each followed by a block of statements.
- There may be one CASE DEFAULT statement followed by a block of statements or there may be none.
- The *case_expression* is either an integer expression, a character expression or a logical expression; real expressions are prohibited.
- The case_selector can take one of four forms:

case_value	
low_valu :	(range: from the *low_value* to infinite)
: *high_value*	(range: from negative infinite to the *high_value*)
low_value : *high_value*	(range: from the *low_value* to the *high_value)*

  + it can also be the combination of above four forms;
  + only the first form is permitted for logical values
- When the SELECT CASE statement is encountered, the value of *case_expression* is evaluated.
  + If this value matches the *case_value* or the range specified by a case selector (say *case_selector_k*), then the block immediately after this case selector (*block_k*) will be executed and then exit from the construct.
  + If the value of the *case_expression* does not match any case values and ranges specified by the case statements, then the block following the CASE DEFAULT statement will be executed; if there is no CASE DEFAULT statement then an exit is made from the case construct without any codes being executed.

Example 10.2.3

Write a program to read the coefficients of a quadratic equation $ax^2+bx+c=0$ and print its real roots.

Analysis: The program will use $x = \frac{1}{2a}\left(-b \pm \sqrt{b^2 - 4ac}\right)$.

$$\text{If } b^2 - 4ac \begin{cases} \geq 0, & \exists \text{ two real distinct roots} \\ = 0, & \exists \text{ two coincident roots} \\ < 0, & \text{no real root} \end{cases}$$

As real arithmetic is only an approximation, we should never compare two real numbers for equality. This is because two numbers which are mathematically equal will often differ very slightly if they have been calculated in a different way. We avoid this difficulty by comparing the difference between two real numbers with a very small positive number ε. Thus we can require the cases as follows

$$b^2 - 4ac \begin{cases} > \varepsilon, & \exists \textit{ two real distinct roots} \\ \in [-\varepsilon, \varepsilon], & \exists \textit{ two coincident roots} \\ < -\varepsilon, & \textit{no real root} \end{cases} \quad \textit{or} \quad int\left(\frac{b^2 - 4ac}{\varepsilon}\right) \begin{cases} > 0, & \exists \textit{ two real distinct roots} \\ = 0, & \exists \textit{ two coincident roots} \\ < 0, & \textit{no real root} \end{cases}$$

Thus, we can have the following **structure plan** and the corresponding **program**

```
step1, read a, b and c
step2, calculate d=b^2-4ac
step3, calculate selector [int(d/ε)]
step4, select case on  selector
       selector>0
           calculate and print two roots
       seletor =0
           calculate and print a single root
       selector <0
           print "no real root"
```

```
PROGRAM  example10_2_3
IMPLICIT NONE
!
!  A program to solve a quadratic equations
!               ax^2+bx+c=0
!
!  Input:   the values of  a, b and  c
!
!  Output: the roots of the equation
!  Constant declaration
   REAL, PARAMETER :: epsilon=1.0E-6
```

```
!
!  Variable declarations
   REAL :: a,b,c,d,sqrt_d,x1,x2
   INTEGER :: selector
!
!   Read Coef
    PRINT *, "please enter the coef. a,b and c"
     READ *, a,b,c
!
!   Calculation
     d=b**2-4.0*a*c
   selector=d/epsilon
!
!  Calculate and print roots, if any
      SELECT CASE (selector)
      CASE(1:)
          ! Two roots
            sqrt_d=sqrt(d)
            x1=(-b+sqrt_d)/(a+a)
            x2=(-b-sqrt_d)/(a+a)
            print *, "two roots:", x1, "and",x2
      CASE (0)
          ! One root
            x1=-b/(a+a)
            print *, "one root:",x1
      CASE (: -1)
          ! No real root
            print *, "no real root"
      END SELECT
END PROGRAM  example10_2_3
```

Exercise.   Rewrite the program using the Block If construct.

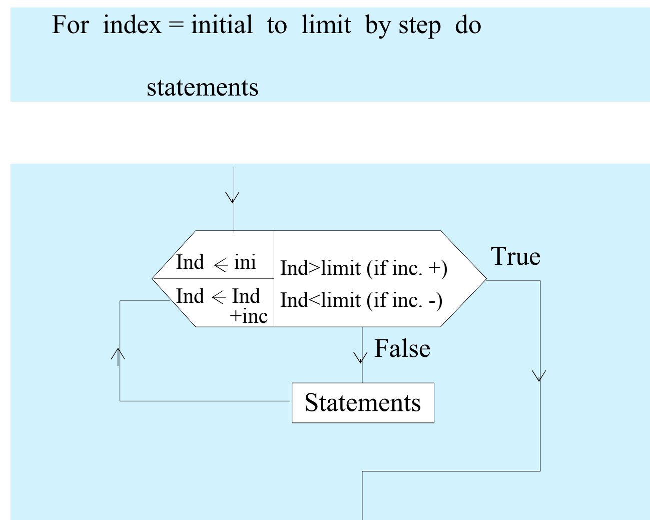
10.3 Loop Control

- This section describes the F95 repetition structures that allow us to repeat certain parts of our program.
- There are two basic repetition structures: a count-controlled DO loop and a conditional DO loop.
- If the number of iterations is known or can be predetermined, we usually use a count-controlled DO loop, otherwise a conditional DO loop.

10.3.1 Count-controlled DO Loop (Do Until the number of count equal to certain value)

If the number of iterations is known or can be predetermined, we can use a count-controlled DO Loop for the repetitive computing.

(a) **Pseudocode**: **and Flow Chart**



(b) **Fortarn Statement**

block_name: DO *index=initial, limit, increment*
Statements
END DO *block_name*

- The DO variable *index* must be an integer variable ;
- The parameter (*initial, limit, increment*) must be integer expressions
- The increment can be either positive or negative, but it cannot be zero; if the increment=1, it can be omitted.
- The inclusion of *block_name* is optional. However, in the situation of nested loops, we recommend to name each DO block to improve the readability of the program.

(c)**Execution of a DO loop**

1) Evaluate the values of *initial, limit and increment*,

2) Index $\leftarrow$ initial

3) IF Index > limit (for positive increment +) exit from the loop
< limit (for negative increment -) exit from the loop
otherwise, execute the statements between DO and END DO and go to next step

4) Index $\leftarrow$ Index +Increment

5) Go to step 3).

Notes:

- Number of loops $= \text{INT}\left(\frac{limit - initial}{increment}\right) + 1$
- The value of index should not be modified by other statements during the execution of the loop.

Example 10.3.1: Write a F95 program segment to calculate $\sum_{i=1}^{50} i = 1 + 2 + \ldots + 50$ using do loop.

```
sum=0
DO  count=1, 50
        sum=sum+count
END DO
```

d) NESTED DO LOOP

A DO loop may be nested within other Do loop. If loops are nested, they must use **different indexes** or loop counters. When one loop is nested within another, the inside loop is completely executed each pass through the other loop.

Example 10.3.2: Write a F95 program segment to compute the factorials of 4 integers.

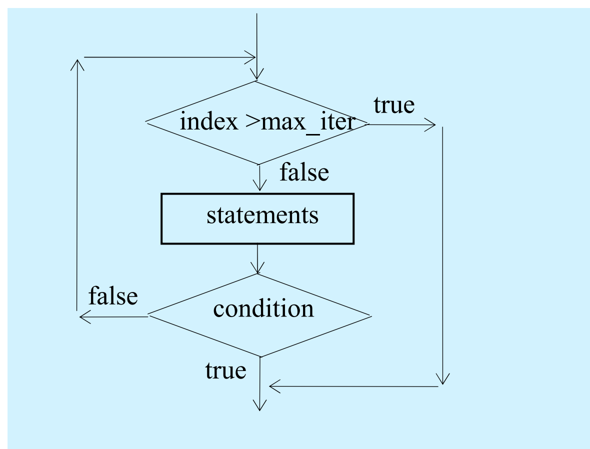
```
do_block_1:     DO i=1,4
                    read *, n
                    IF (n < 0) THEN
                        print *, 'invalid n=', n
                    ELSE
                        nfact=1
                        IF(n > 1) THEN
  do_block_2                DO  k=1, n
                                nfact=nfact*k
                            END DO  do_block_2
                        ENDIF
                        print *, 'n=', n, 'n!=', nfact
                    ENDIF
                END DO  do_block_1
```

10.3.2 Conditional DO Loop (Do until certain condition true)

A condition DO loop identifies a series of steps (statements) that are to be repeated until a certain condition is true or the number of iteration exceeds a predefined value *max_iter*.

(a) **Pseudocode and Flow Chart**

For index from 1 to *max_iter* DO
....
exit if (condition)



(b) Fortran Statement

```
block_name:  DO index=1,max_iter
                  ......
             IF(condition) EXIT block_name
                  .......
             END DO  block_name
```

- The inclusion of *block_name* is optional.
- The EXIT statement causes a transfer of control to the statement immediately following the END DO statement. F95 has another statement, namely CYCLE statement that causes the transfer of control to the start of the loop.
- The inclusion of *index*=1*, max_iter* is optional. However, without such an inclusion, we have the risk that if the condition for exit is never true, the loop will become what is

known as an **infinite loop** and will continue executing until the program is terminated by some external means such as switching off the computer. To avoid possible infinite loop, we recommend that such inclusion should be used to limit the number of iterations to a predefined value *max_iter* unless you are 100% certain that the condition for exit can be met.

- At exit if the index

$$\begin{cases} \le \max_iter, & \text{the condition for exit was met} \\ = \max_iter+1, & \text{the condition for exit was not met (loop ended as number of iterations>max_iter)} \end{cases}$$

If the condition for exit was not met, check the program and algorithm for possible errors.

Example 10.3.3: Write a F95 program to calculate the average value of a set of data. Assume that a zero data value indicates that all data have been read.

```
PROGRAM  EXAMPLE10_3_3
IMPLICIT NONE
      REAL :: x, sum,average_value
      INTEGER :: count
      sum=0.0
      count=0
      read *, x
      DO
          sum=sum+x
          count=count+1
          read *, x
          IF(x= =0) EXIT
      END DO
      average_value=sum/real(count)
      print *, 'average=', average_value
END PROGRAM EXAMPLE10_3_3
```

Ex. 1. Given a sequence of data 2, 1, 4, 0, work out the problem by hand following the computation order of the program (what are the values of SUM and COUNT in the repetition cycles 1, 2, and 3).

2. If the input data values contain 0 and are in the range of -1000 to 1000, modify the program such that it can still be used to calculate the average value. (eg, use 10000.0 to indicate the end of data values)

10.3.3 Other Forms of Loops

There are several alternative forms of the DO loop available, but whose use we do not recommend in new programs.

(a)

```
DO WHILE (condition)
  ......
END DO
```

(b)

```
    DO  n    index=initial,final,increment
    .......
n   CONTINUE
```

10.4 Application: Newton's Method for Soling Nonlinear Equations

The N-R method is one of the most powerful and well-known numerical methods for solving nonlinear equations of the form $f(x) = 0$.

The method starts with an initial estimate x_0 of the solution and refines the approximation step by step.

Graphically, the solution of $f(x) = 0$ is the intersection of $y = f(x)$ with the x-axis (the s as shown). To get an estimate of s from the point $[x_0, f(x_0)]$ on the curve $y = f(x)$, we draw a straight line tangent and find its intersection with the x-axis, x_1, and use this as the new approximation of s. By repeating this process, we can gradually approach s.

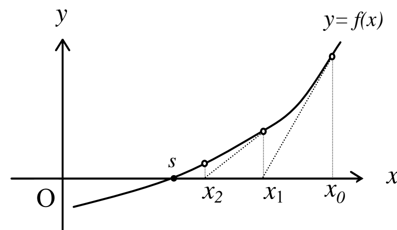
The equation of the tangent line through point $[x_o, f(x_o)]$ is $\dfrac{y - f(x_0)}{x - x_0} = f'(x_0)$

As at $y = 0$, $x = x_1$, we can thus obtain x_1 by letting $y = 0$ and solve the above equation for x. So

$$y = 0 \quad \Rightarrow \quad x = x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Denote x_1 by x_{n+1} and x_0 by x_n, we can rewrite the above formula as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \text{for} \quad n=0,1,2,\ldots$$



Based on the above formula, the following algorithm can be used to find the solution of nonlinear equations

Algorithm

To find a solution to $f(x) = 0$ given an initial approximation x0.

*Input x*0, *Tol and MaxNit.*
Set i =1
While (*i* < *MaxNit*) *do*
    Set x = *x*0 – *f*(*x*0)/ *f'*(*x*0)
    If (|x–x0| < Tol) then
        output x; (*procedure completed successfully*)
        stop
    else
        set *i* = *i*+1
        *x*0 = *x*
Output ('*method failed after MaxNit iterations, MaxNit* =', *MaxNit*)

Example. Consider the nonlinear equation

$$e^x - 3x^2 = 0$$

Write a F95 program to find the roots of the equation accurate to within 10^{-6} in the intervals [0,1] and [3,5] using Newton's method.

```
Program Eqsolver_by_NewtonMethod
!
! Input:
!
!   x0 = initial guess of the solution of f(x)=0
!
!   tol= allowable tolerance:
!      if |x_{n+1} - x_n| <= tol ,stop the iteration process and take x_{n+1} as
!                                  the solution of the equation;
!
!   max_iter = maximum number of iterations:
!      if the number of iterations exceeds max_iter,
!                  stop the process and print "not converge"
!
!  Output
!
!   x = the solution (root) of the nonlinear equation.
!
!declare variables
!
 implicit none
 integer :: i,max_iter
 real    :: x0,tol,x
```

```
!
!input initial guest and control data
!
 print*, 'enter x0, tol, max_iter'
 read(*,*) x0,tol,max_iter
!
!find the solution, print the result or an error message
!
 do i=1,max_iter
   x=x0-(exp(x0)-3*x0**2)/(exp(x0)-6*x0)
   if (ABS(x-x0)<tol) then
        print*,'Number of iteration performed =', i
      print*,'The solution of the eq is x=',x
      stop
   else
      x0=x
   endif
 end do
!
 Print*,’not converge after',max_iter,'iterations'
!
 end program
!
```

!
!-------------- **Input Data & Computation Results** ------------------

```
For the root in [0,1]
   Input Data
        0.5,  1.0E-6,   200
   Results
        Number of iteration performed =  5
        The solution of the eq is x=0.910008

For the root in [3,5]
   Input Data
        4.0,  1.0E-6,   200
   Results
        Number of iteration performed =  5
        The solution of the eq is x=3.733079
```

SUMMARY

In this chapter, we study how to express a mathematical condition using a logical expression, how to write F95 statements to choose alternative courses of action and how to repeat certain part of computation using F95.

F95 Syntax Introduced in Chapter Three

Variable declaration
Block If construct

```
LOGICAL :: list of variable names
IF(logical_expression) THEN
    block_of_code
ELSEIF(logical_expression) THEN
    block_of_code
.......
ELSE
    block_of_code
ENDIF
```

Case Construct

```
case_block_name:   SELECT CASE (case_expression)
                   CASE (case_selector_1)
                        block_1 of F95 statements
                   CASE (case_selector_2)
                        block_2
                   .......
                   CASE DEFAULT
                        block_D
                   END SELECT case_block_name
```

Count-controlled

```
block_name:   DO  index=initial, limit,  increment
                 Statements
              END DO  block_name
```

Conditional DO loop

```
block_name:   DO index=1,max_iter
                 block_of _code
              IF(condition) EXIT
                 block_of _code
              END DO  block_name
```

Relational operators >, >=, <, <=, ==, /=

Logical operators .AND. , .OR. , .NOT. , .EQV. , .NEQV.

EXERCISE 10

Q10.1 What is the difference between a logical operator and a relational operator?

Q10.2 What are the values of the following expressions

(a) 1>2 (b) (1+3) >= 4 (c) 1+3 <= 4 (d) 3>2 .and. 1+2 <3 .or. 4<= 3

Q10.3 What is the purpose of the block if construct ?

Q10.4 Write logical expressions corresponding to the following conditions.

(a) $x+y > 10$ and $x-y \le 0$
(b) $1 < a < 2$ and $1 < b < 2$
(c) $1 < x < 2$ or $x > 5$
(d) Either x or y is zero, but not both.
(e) The distance between two points in the plan having coordinates ($x1$, $y1$) and ($x2$, $y2$) is greater than the distance between ($x1$, $y1$) and ($x3$, $y3$).

(Ans: (a) x+y > 10.0 .AND. x-y <= 0.0) (b) ....)

Q10.5 If A=2.5, B=7.5, C=5.0, D=6.0, L=.TRUE., M=.FALSE., calculate the values of the following logical expressions.

(a) (A+B) < (C+D) .AND. A = = 3.5 ,
(b) (A+B/2.0) /= (C-D) .OR. C = = D ,
(c) .NOT. L .OR. C /= D .AND. M ,
(d) (C/2.0+D) < A .AND. .NOT. .FALSE. .OR. C = =D

(Ans: (a) F, (b) T, (c) F, (d), F)

Q10.6 Write F95 statements that perform the steps in (a) to (d), using the structures indicated.

(a) block IF: If $x > 0.0$, add the value of x to sum and increment count by 1.

(b) block IF: If 5.0 < y <10.0, increment y by 2.0, otherwise increment y by 20.0.

(c) block IF:

$$y = \begin{cases} 0 & x \le 0 \\ 1+x & 0 < x \le 1 \\ 2 & x > 1 \end{cases}$$

(d) count-controlled Do Loop: Calculate $y = \sum_{i=1}^{n} i$, $z = n!$

Q10.7 What values will be printed out from each of the following statements.

(a)
```
READ *, X1, X2, L
READ *, X
READ *, Z, X3, A
PRINT*, L, X, Z, A
```

Data line:
```
0.5, 1.0,
2,
3.0, 4.0,
6.0, 8.0,
9.0
```

(b)
```
NUM=0
DO I=1, 2
    DO K=2, 0, -1
```

```
            NUM=I+K
            PRINT *, NUM
         END DO
      END DO
```

(c)
```
SUM=0
DO
   SUM=SUM+30
   IF(SUM>100.0) EXIT
END DO
PRINT *, SUM
```

Ans: (a) 2, 3.0, 6.0, 9.0
(b) 3, 2, 1, 4, 3, 2
(c) 120.0

Q10.8 What restriction, if any, are there on the case expression in a SELECT CASE statement?
Q10.9 What forms may a case selector takes? are there are any restrictions on any of these forms?

Q10.10 What is the difference between a count-controlled DO Loop and a condition Do loop? when should the count-controlled form be used?

Q10.11 How many times will each of loops controlled by the following DO statements be executed?
(a) DO I=-5,5
(b) DO j=1,12,2
(c) DO k=17,15,-1
(d) DO l=17,15

Q10.12 In Q10.11 (a) and (b), what is the value of the DO variable after termination of the DO loop.

Q10.13 What is an infinite DO loop? How can it be avoided?

Q10.14 What is an EXIT statement used for?

Programming Exercises

***Program Debugging** - Some guidelines (for Q10.15-10.19)*

The most helpful debugging tool is the PRINT statement. Just knowing that your program is working incorrectly does not tell you where to begin looking for errors. If you have the computer print the values of key variables at different points in your program, however, it becomes easier to isolate the parts of program that are not working correctly. The location of these check-points

depend on the program, you need to guess the place which may lead to the error. It is also a good idea to number the check points such as ' Check point 5: x=1.76, y=2.86

(4) If you know that the value of a variable is incorrect, to find the point causing the error, you need to print the intermediate results leading to the final value of the variable.

(5) If you use a IF statement, you could check whether the condition is true or not and be sure it is as you expected.

(6) If you believe that the programming error is within a DO loop, print the values of key variables at each cycle of the loop which will help you to locate the trouble points.

Q10.15 Design the flow chart and a complete Fortran 95 program to calculate and print the maximum, minimum and average values of a series of data. Assume that a data value 10^{20} (1.0E20) indicates that all data have been read. Test your program using the following data: 1.0, 2.8, 9.0, 4.0, 3.2).
(**Hint**. Read and modify the F95 program in example 10.3.3)

Q10.16 Write the pseudocode, flow chart and a complete Fortran 95 program for finding the roots of equation $ax^2+bx+c=0$ where $a \neq 0$ using block IF construct .

Requirements:

* Read the values of *a, b* and *c* from terminal;

* In the case of repeated real roots, print 'Repeated real roots x1= x2=......... ;
In the case of distinct real roots, print 'Distinct real roots x1=......... x2=.........',
In the case of complex roots, print 'Complex roots x1(x2) = ... + (-)......... i.

* Test your program using the following data:(1)a =1, b = 1, c=1.25; (2) a = 1,b =3, c = - 4.

(**Ans**: (a) $-0.5 \pm i$ (b) 1, - 4)

Q10.17 Write a program which will request a number (1 to 6) to be typed at the keyboard and prints out the corresponding words “one”, ”two” etc. If a number outside this range is typed, print “outside the range 1-6”. Write the program using the CASE construct. Can you rewrite the program using the block IF construct?

Q10.18 Write a program to read a integer value n, then calculate $\sum_{i=1}^{n}(2.1 * i)$ and $n!$.

*Q10.19 (optional)
To find a root of a nonlinear equation $f(x)=0$ using fixed-point iteration, we first rewrite the equation as $x=g(x)$. Then, set $x=x_o$ (initial guess) and then perform iteration $x_{i+1} = g(x_i)$ (i = 0, 1, 2, ...) to improve the estimate. If $|x_{i+1} - x_i| < Tol$, we say that the process converges and x_{i+1} is taken as the root. To control the process, we set a limit on the number of iterations (*Max_iter*). If the process does not converge after *Max_iter* iterations, we print an error message and stop computing. The following is an algorithm based on this method

Input x_o, *Tol and Max_iter.*

```
Set i=1
While (i<Max_iter ) do
      Set x=g(xo)
      If ( |x-x0|<Tol ) then
            output x; ( &  'procedure completed successfully' )
            stop
      else
            Set   i=i+1
            set x0=x
      Output ('method failed after Max_iter iterations, Max_iter=', Max_iter)
```

Using above algorithm, write a complete F95 program to find a solution accurate to within 10^{-5} for the equation

$$x^2 - 3x + 2 - e^x = 0$$

(**Hint**. Rewrite the eq. as $x = \frac{1}{3}\left(x^2 + 2 - e^x\right)$ and choose x_o=0.5, *Tol*=10^{-5}, *Max_iter*=100. (**Ans**: $x \approx 0.25753$).

CHAPTER 11

F95 ADDITIONAL DATA TYPES

This chapter introduces the so-called kind type parameter associated with variables and constants and three new intrinsic data types (double precision, complex and character).

- With kind type parameter, we can control the precision of the computation.
- Double precision type was used in early versions of Fortran and is not recommended to be used in new programs
- With complex data type, we can perform calculation involving complex numbers
- With character type, we can read and analyse character data such as names and addresses

11.1 Control of Precision - Kind Type Parameter

In general, the range of values that may be stored in a variable and the precision of real values can vary enormously depending on the computer used and how they are actually represented by the computer being used. For example, a real number can be stored in a computer to about six or seven decimal digits of precision with a single precision real variable or 13-14 decimal digits of precision with a double precision real variable. The precision and exponent range for the same kind of variable are potentially different for every computer. This is a serious hindrance to portability. A program that executes acceptably on one machine may fail on another because of less accuracy or a smaller exponent range.

F95 provides a means to overcome theses problems through the use of a so-called **kind type parameter**. It allows the programmer to specify, in a portable fashion, the degree of precision and the range of values required, so that the compiler can ensure that the most suitable form of data representation is used.

11.1.1 Parameterized variables - concept of kind type parameter

To permit more precise control over the precision and exponent range of values, F95 allows all the intrinsic types other than double precision to have more than one form, known as different kinds, and provides the means for a program to define which kind of variables and constants it wishes to use. The kind of variables and constants can be specified by using a parameter, namely the kind type parameter. When this parameter is not specified explicitly, the kind of the data is set to be default kind. So far, all data used have been of default kind. The kind type parameter value assigned to a default kind is processor - dependent

11.1.2 Determine the required KIND TYPE according to data range and/or precision requirement

(a) For Integer Data

The intrinsic function SELECTED_INT_KIND can be used to provide a suitable kind type for a given range requirement. Thus a reference to

SELECTED_INT_KIND(*r*)

returns a value of the kind type parameter for an integer data type that can represent, at least, all integer value n in the range $-10^r \prec n \prec 10^r$. If it is not possible to represent all the integer values in this range, the function will return a value of -1. If there exit several kind types satisfying the requirement, the one with the smallest exponent range will be return. For example, the following statement

INTEGER, PARAMETER :: *range*=SELECTED_INT_KIND(20)

sets the named constant *range* to the required kind type value of integer variables for storing integer values in the range [-10^{20}, 10^{20}].

(b) For Real Data

The statement

INTEGER, PARAMETER :: *real_kind*=SELECTED_REAL_KIND(*p=p0, r=r0*)

sets the constant *real_kind* to a kind type parameter for a real data type that has at least *p0* decimal digits of accuracy and a decimal exponent range of at least *r0*.

- If no such kind type parameter is available on a particular processor for the range requested, the function will return a value of -1.
- If the precision requested is unsupported, the function will return a value of -2.
- If neither the precision nor the range requested are available, the function will return a value of -3.
- If any of these values (-1,-2,-3) are used as the kind type in a declaration statement, they will cause a compilation error.
- The exponent range argument is optional. If *r=r0* is omitted, the processor will choose the value of *r* .
- "*p=*" and "*r=*" can be omitted, but it is recommended that the full form should be used.

11.1.3 Specify KIND TYPES of constants and variables

(a) Constants

To specify the kind type of a constant, the kind type parameter follows the constant, separated from it by an underscore, except in the case of characters, where the kind type parameter precedes the constant, separated from it by an underscore.

eg.	-124	(default integer)
	628_3	(integer of kind 3)
	-628_*low*	(integer of kind *low*)
	-12.34	(default real)
	402.2E-5	(default real)
	-704.2E-3_3	(real of kind 3)

where *low* is a named constant whose value has been defined by a statement of the form

```
INTEGER, PARAMETER ::  low=1
```

(b) Variables

The kind type parameter associated with a variable is specified by the kind selector in the declaration of the variable,

```
TYPE(KIND=kind_number), ... :: var_1, var_2 ...
```

- TYPE is one of Integer, Real, Complex, Character or Logical. It cannot be Double Precision.
- *kind_number* is either a positive integer constant or a constant integer expression which will be evaluated to a positive value.
- If no kind selector is specified, then the default kind type is used.
- We may omit the KIND=, but we recommend that the full form should always be used to avoid any confusion.

For any variable or constant that is an intrinsic type, the value of its kind type can be found by using the intrinsic function KIND. For example, the statement

```
i=KIND(x)
```

sets *i* to the kind type value of variable *x*.

eg.

```
PROGRAM degree
IMPLICIT NONE
INTEGER ::  i, j, k
INTEGER, PARAMETER :: range=SELECTED_INT_KIND(20)
INTEGER, PARAMETER :: single=SELECTED_REAL_KIND(p=6,r=30)
INTEGER (KIND=range) :: x, y, z
```

```
REAL (KIND=single) :: v
```

This extract defines

- 3 integer variables i, j and k of default kind type;
- integer variables x, y, z that can contain values in the range $-10^{20} \prec n \prec 10^{20}$;
- a real variable v that can contain values in the range $-10^{30} \prec n \prec 10^{30}$ and have at least 6 decimal digits of accuracy.

11.2 Double Precision Data

(it is recommended not to use double precision data in new programs)

With double precision type, we can process data more precisely than we could by using real data. A double-precision variable can store data with more (about twice) digits of precision than a real variable. Thus, using double precision values can increase the precision of our results, but there is a price for such additional precision - the executing time for computation is longer and more memory is required.

(a) Double-precision Constants are written in exponential form, with a *D* in place of *E*.
eg. 0.3485D-15

(b) Double-precision Variables must be declared in the program by either explicit type declaration or implicit type declaration

```
DOUBLE PRECISION    variable-list
```

eg

```
IMPLICIT DOUBLE PRECISION (A- D)
```

! *declares that all variables starting with A/B/C/D are double precision.*

(d) Double precision **Operations**

Double precision —— (Double precision, Real, Integer) → Double precision

(e) Double-precision **intrinsic functions**

- Many of the common intrinsic functions for real numbers can be converted to double-precision functions by preceding the function name with letter *D*, eg. DSQRT, DSIN, DEXP, DLOG, all require double-precision arguments and yield double-precision values.
- Two functions specifically designed for use with double-precision variables are listed below:

DBLE (*x*) : Converts a real argument to a double-precision value,

DPROD (*x1, x2*) : Has two real arguments and returns the double-precision product of the two arguments.

11.3 Complex Data

Fortran includes a special data type for complex variables and complex constants $a+bi$, where $i = \sqrt{-1}$.

(a) A **complex Constant** is specified by two real constants separated by a comma and enclosed in parentheses. The first constant in the parentheses is the real part and the second constant is the imaginary part

eg.

```
A= (3.0,  1.5)
```

assigns a complex value 3.0+1.5i to complex variable A.

(b) **Complex Variables** must be declared in the program using a specification statement.

```
COMPLEX  ::  variable-list
```

or

```
COMPLEX (KIND=kind_number), ... :: var_1, ...
```

(c) **List-directed I/O**:

eg. input 2+3i to A : `READ * , A`

input data should be (2.0, 3.0) *as the complex value must be enclosed in parentheses.*

(d) **Complex Operations**

Complex —— (Complex, Real, Integer) → Complex
Complex —— (Double precision) → is not allowed in standard Fortran.

(e) **Complex Intrinsic Functions**:

- The functions CSQRT, CABS, CSIN, CCOS, CEXP and CLOG are all intrinsic functions with complex arguments and complex values, these function names begin with the letter *C* to emphasize that they are complex functions.
- Some functions specifically designed for use with complex variables are listed below.

REAL (z) : yields the real part of its complex argument.
AIMAG(z) : yields the imaginary part of its complex argument.
CMPLX(a, b) : converts two real arguments a and b into a complex value $a + bi$
CONJG (z) : converts a complex number ($a+bi$) to its conjugate ($a-bi$)

Note:

While C = (2.0, 1.0*) is ok,* C=(a, b) *is not ok.*

When the real and imaginary parts are expressions,we need to use C = CMPLX (a, b)

Example: Solve equation $ax^2 + bx + c = 0$ using complex arithmetic.

```
PROGRAM SOLVER_QUADRATIC_EQ
IMPLICIT NONE
    REAL :: a, b, c
    COMPLEX :: DISCR, ROOT1, ROOT2
    READ*, a, b, c
    DISCR = CMPLX ( b*b - 4.0*a*c, 0.0 )
    ROOT1 = ( -b + CSQRT(DISCR) ) / ( 2.0*a )
    ROOT2 = ( -b - CSQRT(DISCR) ) / ( 2.0*a )
    PRINT *, 'ROOT1=', ROOT1, 'ROOT2=', ROOT2
END PROGRAM SOLVER_QUADRATIC_EQ
```

Exercise. *Run this program in computer using a=1, b=2 and c=5.*

11.4 Character Data

We refer character data as character strings. Like numerical data, we have character constants and character variables. Character constants and variables contain characters A - Z, 0 - 9, blank, + - * / = ( ) , . ' $: and etc.

(a) **Character Constants** are always enclosed in apostrophes. If two consecutive apostrophes are encountered within a character constant, they represent a single apostrophe.

eg. 'Sensor 23' → Sensor 23   ' Newton''s Law' → Newton' Law,

(b) **Character Variables** must always be declared with specification statements. eg.

```
    CHARACTER :: var_1*n1, var_2*n2
    CHARACTER(LEN= n) :: var_1, var_2, var_3 ...
```

- Statement one specifies that variables *var*_1 and *var*_2 are of length *n*1 and *n*2 respectively, namely they can store respectively up to *n*1 and *n*2 characters.
- Statement two specifies that all variables in the list are of length *n*.

(c) **List-directed I/O**:

- Character string to be read must be enclosed in apostrophes;
- A list-directed output statement will print the entire character string of the variables.

eg.
```
CHARACTER :: NAME*8
READ *, NAME                 Input data line:     'P. Hill'
PRINT *, 'NAME: ', NAME      Output data line:    Name: P. Hill
```

(d) **Character Operations**:

Assign values: eg. NAME = 'John'

- If a character constant is shorter in length than the character variable, blanks are added to the right.
- If longer, the excess characters on the right are ignored.

Compare values: A collating sequence lists characters from the lowest to the highest value. A partial collating sequence for ASCII is as follows

^ (blank) " # $ % & ( ) * + , - ./ 0 1 2 ... 9 : ; = ? @ A B ... Z

Thus, comparison of character strings can be made. Comparison is made from left to right, one character at one time, eg.

'A1' < 'A2'   'JOHN' < 'JOHNSTON'   'HILL' < 'WONG'

Application - Sort a list of names into alphabetical order. For example, character variables city1 and city2 contain the names of two cities. Print the names in alphabetical order.

```
CHARACTER (LEN=10) :: CITY1, CITY2

IF (CITY1 < CITY2) THEN
      PRINT *, CITY1, CITY2
ELSE
      PRINT *, CITY2, CITY1
ENDIF
```

Extract substring: eg. If Name = ' Fortran'
then Name (1:1) 'F'
Name (3:3) 'r'
Name (:4) 'Fort'
Name (4:) 'tran'
Name (:) 'Fortran'

Example. A string of 50 characters contains encoded information. Write a loop that counts the number of occurrences of the letter S.

```
 CHARACTER(LEN=50) :: CODE
INTEGER :: COUNT
COUNT=0
DO I=1, 50
      IF (CODE(I : I) == 'S' ) COUNT=COUNT+1
END DO
```

Combine strings : character strings can be combined together by using //,

eg.

```
MO = '06'
DA = '12'
YR = '93'
DATA = MO//DA//YR
NAME = 'John'// ' Wong'
```

Then the character variable DATA will contain 061293, NAME will contain John Wong.

Character Intrinsic Functions

INDEX (*char string 1, cha. string 2*): can be used to check whether string 1 contains string 2.
If No, returns a value *0*
Yes, returns an integer n: the position of the first occurrence of string 2 in string 1.

LEN (*character-string*): returns an integer representing the length of the char string.

CHAR (n) : returns the character at position n of the collating sequence

ICHAR (*character*): returns an integer - the position of the *character* in the collating sequence.

Example: **Name Editing**. Character variables FIRST, MID and LAST contain the 1st, mid and last name of a person. Print the name in format: First name^initial of mid . last name (eg. Peter J. Wong).

```
NAME=FIRST
L=INDEX (NAME, ' ' )
NAME (L : L+3 ) = ' ' // MID(1 : 1) // '.'
NAME (L+4 : )= LAST
PRINT *, NAME
```

11.5 Application: Composite Simpson's 1/3 Rule for Evaluating Integrals

Consider $\int_{x_0}^{x_2} f(x)dx$. Simpson's 1/3 rule for the evaluation of the integral is as follows:

$$\int_{x_0}^{x_2} f(x)dx = \frac{1}{3}h(f_0 + 4f_1 + f_2)$$

Remark:

Graphically, as shown in the figure below, in Simpson's 1/3 rule, the curve of $f(x)$ is approximated by a parabola of $P_2(x)$. The exact value of the integral = area between the x-axis and the curve $y = f(x)$ from x_0 to x_2; while the numerical value of the integral = area between the x-axis and the parabola $y = P_1(x)$ from x_0 to x_2.

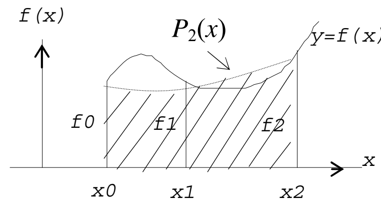
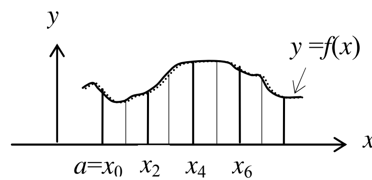


Fig 11.1 Diagram for Simpson's 1/3 rule

The above rule is generally unsuitable for use over large integration intervals, as high-degree polynomials would be required for use over such interval but the coefficients in these polynomials are difficult to obtain. In addition, due to the oscillatory nature of high-degree polynomials, integration using such polynomials may yield inaccurate results. Thus, a piecewise approach to numerical integration that uses the low-degree polynomials is generally applied in practice.

For composite Simpson' 1/3 Rule, we first subdivide the interval [a, b] into n subintervals [x_{i-1}, x_i] (i=1, n) with equal subinterval length h and then use Simpson's 1/3 rule on each consecutive pair of subintervals. As each application of Simpson's 1/3 rule requires two subintervals, n must be an even number, that is $n=2m$, so that the n intervals can be grouped into m pairs $[x_{2(i-1)}, x_{2i}]$ (i=1, m) each with 3 equidistant nodes x_{2i-2}, x_{2i-1} and x_{2i}.



Application of Simpson's 1/3 rule on each pair yields

Composite Simpson's 1/3 Rule:

$$\int_a^b f(x)dx = \sum_{i=1}^{m} \int_{x_{2(i-1)}}^{x_{2i}} f(x)dx = \frac{h}{3}\left(f_0 + 4f_1 + 2f_2 + 4f_3 + \ldots.. + 4f_{n-1} + f_n\right)$$

The following is an algorithm for evaluating integrals using the above rule.

Algorithm for Composite Simpson' 1/3 Rule

```
To approximate the integral  I = ∫_a^b f(x)dx

INPUT endpoints a,b; even positive integer n
     Set h=(b-a)/n
   XI0= f(a)+ f(b)
   XI1= 0 (for summation of f(x_{2i-1}))
   XI2= 0 (for summation of f(x_{2i}) )
For  i=1,..., n-1 do
   Set x = a+ih
   If i is even then
      set XI2=XI2+f(x)
         else
      set XI1=XI1+f(x)
Set XI=h*(XI0+2*XI2+4*XI1)/3.0
OUTPUT (XI)
STOP
```

Example. Write a short Fortran 95 program to calculate $I = \int_0^3 x\sqrt{1+x^2}\,dx$ using the composite Simpson's 1/3 rule. Calculate the integral using n=8. (hint, use a DO loop to calculate the sum). Store real values with at least 15 digits of precision;

```
Program IntegralEval_by_CompSimpson1_3_rule
!
! Input:
!
!   a = the lower bound of the integral;
!
!   b = the upper bound of the integral;
!
!   n = an even positive number : the interval [a, b]
!       will be divided into n subintervals.
!
! Output
!
!   xi= the numerical approximation of the integral value.
```

```
!
!determine the kind type required for variables to store real data
!   with at least 15 digits of precision
!
    implicit none
    integer, parameter::k15=selected_real_kind(p=15)

!declare variables
!
   integer :: i,n
   real(kind=k15):: a, b, h, f,x, xi0, xi1, xi2, xi
   logical :: i_is_even
!
!define a statement function
    f(x)=x*sqrt(1+x**2)
!
! input data
!
   print*, 'enter a, b, n'
   read(*,*) a, b, n
!
!
!calculate the sum of the first and last terms and store it in xi0, then
!  accumulate the sum of the even (odd) terms and store it in xi2 (xi1).
!
     h=(b-a)/n
     xi0=f(a)+f(b)
     xi1=0
      xi2=0
     i_is_even=.false.
     do i=1,n-1
          x=a+i*h
          if (i_is_even) then
               xi2=xi2+f(x)
               i_is_even=.false.
          else
               xi1=xi1+f(x)
               i_is_even=.true.
          endif
      end do
!
     xi=h*(xi0+2*xi2+4*xi1)/3.0
!
! print result
     print*,'The value of the integral is I=', xi
  !
```

```
 end program
!
```

Input Data & Computation Results

```
Input Data
   0,  3,   6
Results
   The value of the integral is I=  10.2063463091
```

EXERCISE 11

Q11.1 How to specify the kind type of a real constant? Give one example.

Q11.2 What does it mean to be of type default real?

Q11.3 How to determine the required kind type of a real variable for storing values with at least 9 digits of precision?

Q11.4 Write a statement to declare an integer variable for storing integer values in the range [-10, 10^8], then write another statement to print the required value of the kind type parameter ?

Q11.5 Write a statement to declare a real variable for storing a real value in the range [-10^{20}, 10^{20}] with at least 12 digits of precision, then write another statement to print the required value of the kind type parameter ?

Q11.6 What happen if an impossible integer range or real precision is requested?

Q11.7. Compute the value stored in CX for CY=1.0+3.0i and CZ = 0.5-1.0i . Assume that CX, CY, CZ are complex variables. (a) CX=CY+CZ (b) CX=CONJG(CZ) (c) CX=REAL(CY) + AIMAG(CZ)
(d) CX=ABS(CY) (e) CX=(5.0, 0.2)
(Ans: (a)1.5+2.0i; (b) 0.5+1.0i; (c) 0.0; (d) $\sqrt{10}$; (e) 5.0+0.2i)

Q11.8. In (a) - (f), give the substring in each reference. Assume that a character string of length 25 called TITLE has been initialised with the statements

```
CHARACTER(LEN=25) :: TITLE
TITLE = 'CONSERVATION OF ENERGY'
```

(a) TITLE (1:25) (b) TITLE(13:16) (c) TITLE (17:) (d) TITLE(:12)
(e) TITLE // 'LAW' (f) TITLE(1 : 7) // 'E' // TITLE(16 :)

(Ans: (b) OF ; (c) ENERGY; (d) CONSERVATION; (f) CONSERVE ENERGY)

Programming

Q11.9 Write a program which calculate $\frac{1}{n!}$ for real values of n increasing in steps of 1.0 starting from 1.0 and continuing until the calculated result is not distinguishable from zero (i.e, until $\frac{1}{n!} == 0$). Run this program using default real kind, and then run it again using each available real kind in turn. What does this exercise tell you?

(Hint, use real kinds corresponding to accuracy requirement 6 digits of precision, 9 digits, ... and etc)

Q11.10 Write a short Fortran 95 program to calculate $G = \int_0^1 x^2 \, e^x dx$ using the composite trapezoidal rule

$$\int_a^b f(x)dx = \frac{\Delta x}{2}[f(a) + f(b) + 2\sum_{i=1}^{n-1} f(a + i\Delta x)] .$$

where $\Delta x = (b - a)/n$, $f(a)$ denotes the value of $f(x)$ at $x=a$, and $f(a+i\,\Delta x)$ is the value of $f(x)$ at $x= a+i\,\Delta x$. Calculate G using n=8. (hint, use a DO loop to calculate the sum). (Ans: 0.72889)

CHAPTER 12

F95 FORMATTED I/O AND FILES

Fortran provides facilities for I/O at two different levels:

(1) List directed I/O: straight forward I/O from/to the default I/O device (The default input and output devices are usually keyboard and screen respectively);

very little control over the source & layout of the input data and output results.

(2) Formatted I/O: specify in the program where to read and write the data (specifying I/O device) & how to interpret the input data and present the output data.

12.1 Formatted Output

To specify the form in which data values are printed and where on the output line they are printed requires formatted output statements. A formatted output statement can take one of the following form:

	PRINT '(*format_specifier*)', *output list*	or		WRITE (*, '(*format_specifier*)') *output list*
	⇳			⇳
	PRINT *k*, *output_list*			WRITE (*, *k*) *output_list*
k	FORMAT (*format_specifier*)	or	*k*	FORMAT (*format_specifier*)

where the *format_specifier* consists of X specification and I, F, E, D and A format codes etc, and is used to tell the computer both the vertical and horizontal spacing to be used when printing the output information.

eg.

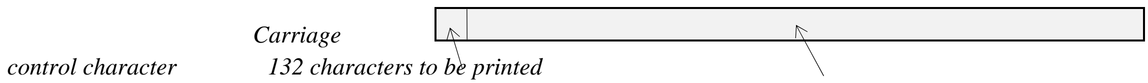
```
        WRITE(*,10) I, X1, X2
10      FORMAT(2X, 'I=', I2, 2X, 'X1=', F9.3, 'X2=', E12.4)
```

The vertical spacing options include: printing on the top of a new page or
using single spacing or
double spacing or
no spacing

Horizontal spacing control include: how many values are to be printed each line
how many digits will be used for each value
how many blanks will be inserted between data

- In printing a line, the computer firstly uses the *format_specifier* to construct each output line internally in memory. This internal memory region, which contains 133 characters, is called a buffer. The buffer is automatically filled with blanks before it is used to construct a line of output.

- The first character of the buffer is called the carriage control character; it determines the vertical spacing for the line. The remaining 132 characters represent the line to be printed.



- Thus, the *format-specifier* must put an appropriate character into column 1 of the buffer to control the vertical spacing, and put the line to be printed into columns 2 - 133 of the buffer using the expected format. The items specified in the format specifier will be put onto the buffer one by one according to the order they appear.

Vertical Spacing Control

We can control the vertical spacing of output, by putting different carriage control character into column 1 of the buffer

Carriage Control Character	Vertical Spacing
1	New page
blank	Single spacing
0	Double spacing
+	No vertical spacing

Literal Specification 'characters string'

- allows us to put characters directly into the buffer. These characters can represent the carriage control character for vertical spacing control or the characters to be printed.

- the characters to be put into the buffer must be enclosed in single quotation marks or apostrophes.

eg

```
        PRINT 4
    4   FORMAT  ('1',  'Test Results' )
```

will construct a buffer `1 Test Results`

will print in a new page `Test Results`

(The 1st character in the buffer is for vertical spacing control)

X Specification (Horizontal spacing control)

Format: *n*X which will insert *n* blanks into the buffer.

eg.
```
    PRINT 4
4   FORMAT (3x, 'Solution : ', 2x, 'y=')
```

will construct buffer | | | Solution y = |

will print on next line: | | Solution y = |

I Format Code (specify the format for printing the values of integer variables)

Format: **I w** (*indicates that the next w positions are to be reserved for an integer value*)

width (*number of positions*) to be assigned in the buffer for printing the value of an integer variable

- If the value to be printed requires more positions than *w*, error occurs with the entire field filling with asterisks *.
- The value is right-justified (no blank to the right of the value) in the specified positions of the buffer.

  eg. If the value of 22 is printed with an I4 specification, the four positions contain two blanks followed by 22.

F Format Code (specify the format for printing the values of real or double-precision variables in decimal form)

Format: **Fw.d** (*indicates that the next w positions are to be reserved for a real value*)

number of positions to the right of the decimal point

total width (number of positions) to be used in printing the value

eg. F11.5 *d=5*

± ****.*****

w=11

eg.
```
    PRINT 5, x, y                                  (assume x=4.2, y=-5)
5   FORMAT ( 1x, 'x=', F5.2, 2x, 'y=', f4.1)   ⇒   | x = | 4. 2 0 | | y = | - 5 . 0 |
```

- If the integer portion of a real value requires more positions than *w*, error occurs with the entire field filling with *
- If the value to be printed has more than *d* decimal positions, only *d* decimal positions are printed.
- The value is right-justified (no blank to the right of the value)

E Format Code (specify the format for printing the values of real variables in exponential form)

This specification is primarily used for very small values, or very large values , or when you are uncertain of the magnitude of a number. If you use a F format that is too small for a value, the output field will be filled with asterisks *. In contrast, a real number will always fit in an E specification field.

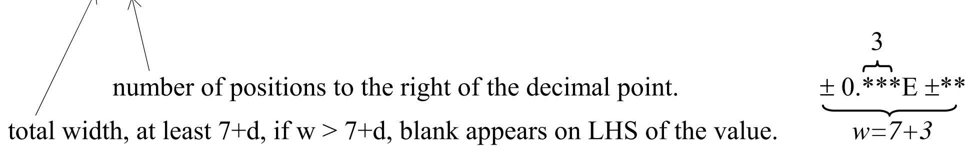
A real value (eg. a real value with 3 decimal places) printed in exponential form appears like

$$S0.{*}{*}{*}ESxx \quad \Leftrightarrow \quad (S0.{*}{*}{*} \times 10^{Sxx})$$

where the symbol S represents the sign of the value or the sign of the exponent. It can be seen that

Total number of positions (10) = Number of decimal positions (3) + 7.

Format: **E w.d** (*indicates that the next w positions are to be reserved for a real value*)



eg   PRINT 15, TIME   (assume that TIME= -0.00125)

15 FORMAT (3x, 'TIME=', E12.4) | | |T I M E =| - 0 . 1 2 5 0 E - 0 2 |

The value is right-justified (no blank to the right of the value).

D Format Code (specify the format for printing the values of double-precision variables in exponential form)

Format: **DW.d** —— The same as E format code except E is replaced by D.

L Format Code (for printing logical variables and logical constants)

Format: **Lw** ← number of positions for printing the value

eg.   write (*, 100) L1, L2, .TRUE.   (assume that L1=.TRUE. L2=.FALSE.)

100 Format (1x, L4, L5, L2) | T| F| T|

A Format Code (for printing character variables)

Format: **A** (print the entire character string)
Aw (w = number of positions for printing the character string)

eg:

```
    CHARACTER :: S*9
    S = 'Australia'
    WRITE (*,10)  S
    WRITE ( *, 20)  S
10  FORMAT (1x, A8)
20  FORMAT (1x, A)
```

A u s t r a l i
A u s t r a l i a

The data is left-justified.

12.2 Formatted Input

To specify the columns to be used in reading data from a data line, we use a formatted READ statement.

	READ k, *variable-list*		READ (*, k) *variable-list*
k	FORMAT (*format-specifier*)	k	FORMAT (*format-specifier*)

- Compared with *format_specifier* for output, no carriage character is needed with READ statement.
- Each read statement begins reading at column 1 of a new data line.
- A *format_specifier* consists of X specification and I, F, E, D and A format codes etc. For different variable types, we should use different format codes.

X Specification

*n*X —— skip *n* positions.

Integer Variables —— use I format code

Iw (a) **w** = the number of positions to be used for the data
(b) blanks in the w positions are usually ignored in F95. However in some compilers, blanks in the w position will be filled with 0, thus data must be right justified in the w positions.

eg.

```
   READ 1, N MEAN
1  Format (I4, 2x, I3)
```

If data line:

									Result
		1	5					2	⇒ N=15, MEAN=2
1	5						2		⇒ N=15, MEAN=2

Notes: As indicated in (b), for the case of the second input date line, in some compilers, the data values will be read as N=1500, MEAN=20.

Real Variables —— use F or E format codes

Fw.d

- **w** = the total number of positions to be used for the data
- if there is a decimal point included in the w positions, the value in the w position will be read as it was entered
- if no decimal point, blanks in the w positions are ignored and **d** digits counted from the last non-zero digit in the w positions will be considered as fractional part. However, in some compilers, blank in the w positions is read as 0 and d digits from the right end of the w positions are considered as fractional part. Thus, it is recommended that data be right justified in the w positions.

```
eg.        Read 1, x, y
         1 Format (F6.2,2x,F5.2)
```

If data line:

2 0 . 3 2 1	| |	5 1 8 6	⇒	x=20.321, y=51.86
3 2 1		1 5 1	⇒	x=3.21, y=1.51

Notes: As indicated in (iii), for the case of the second input date line, in some compilers, the data values will be read as x=3210, y=500.

(b) **Ew.d** (data is entered in an exponential form)

* w = total number of positions to be used for the data
* if a decimal point is included in the w position, its placement will override the value of *d*.
* if no decimal point, one is located according to *d* before storing the value (as in (a) iii).
* E format will accept many forms of input

```
eg      READ 1, x
   1    FORMAT (2x, E9.2)
```

if data line:

	0 . 2 1 3 E - 1 0	⇒	$x=0.213*10^{-10}$
	1 2 3 E - 1 1	⇒	$x=1.23*10^{-11}$

Double Precision

Double precision values may be referred with F or E format code. Another specification is Dw.d, which functions essentially like the E format code.

Complex Variable

In formatted input, a complex value: is read with two real format codes, and no parentheses are needed for the input data. eg.

```
COMPLEX :: C
```

```
      READ 1, C
1     FORMAT (2F6.2)
```

If input data line | 246.2 | 59.3 | then C = 246.2 + 59.3i

Character

In formatted input, use Aw format code, and no apostrophe is needed for the input data.
eg.

```
   CHARACTER  N1*4, N2*4,
   READ  1, N1, N2
 1 FORMAT (A4, 2X, A4)
```

Input data line | JOHN WANG |

12.3 Additional Format Features

Tab Specification

Tn —— Allow you to shift directly to a specified position in the input or output buffer.

eg. 10 format(58x, F7.3) ⇔ 10 format (T59, F7.3)

Slash (/)

Print the current buffer and start a new one. Buffer

Test Results
Time H

eg. PRINT 5

5 FORMAT (3x, 'Test Results'/1x, 'Time H') Print

Test Results
Time H

Repetition

If we have several identical specifications in a row, we can use a constant in front of the specification (or set of specifications).

eg. 10 FORMAT (1x, I3, I3, 1x, F3.1, 1x, F3.1) ⇔ 10 FORMAT (1x, 2I3, 2(1x, F3.1))

Number of Format Codes

(a) There are more format codes than variables on a read or print statement, the computer uses as much of the format codes as it needs and ignores the rest.

(b) There are fewer format codes than variables on an I/O list, eg.

```
      PRINT 10,   TMP, VOL
10    FORMAT(1x, F6.2)
```

In this case, the computer matches variables and format codes until it reaches the end of the FORMAT. Then two events occur:

(1) With a read statement, go to the next data line; with a print statement, print the current buffer and tart a new one

(2) Back up in the format code list until reaching the left parenthesis, and again begin matching the remaining variables to the format codes.

12.4 File Operations

This section shows how to modify the READ and WRITE statements in previous sections to read/write data from/to a file.

- A **file** is an external source from which data may be obtained, or an external destination to which data may be sent. We can enter data once into a data file. When the data is needed, we can read it from the data file. We can write data into a data file, other programs can then use the data and it is still available if we decide to print it.

- A file consists of a sequence of **records**. A record is a unit of input or output.

- Records can be read from or write to a file in two modes - either sequentially or randomly. The first mode is called **sequential access**, i.e., n records are written to a file one after another from record 1, 2, 3 ,... and must be read sequentially in this order. The second mode is called **direct access**. i.e., a file contains n records but these may be written and read in any order by reference to a record number (1 to n) and is specified when the record is read or write.

OPEN Statement - Connecting a File to the Program

Data cannot be transferred to or read from a file until the file is connected to the program. The purpose of file connection is to designate a unit number to be used by the program in referring to the file, and to establish certain properties that affect the way in which data can be transferred to or from the file. A file may be connected to the program by using an OPEN statement:

OPEN (UNIT=*u*,FILE=*name*,STATUS=*stat*,ACCESS=*acc*,FORM=*fm*,RECL=*rl*, &
BLANK=*blnk*,ERR=*s*, IOSTAT=*ios*)

eg.

```
Open (unit=10, file= “E12Q1.dat”)
```

Notes:

(1) The unit number must be specified always, but all other items on the list are optional.

(2) The items may be placed in any order. The phrase 'UNIT=' may be omitted, but if so the unit number must be the first item on the list.

u: INTEGER, unit number

name : CHARACTER, the name of the file to be connected to the unit

stat : 'OLD' (file already exist), 'NEW' (file will be created), 'SCRATCH' (file will be connected to the unit but will be deleted when the unit is closed), 'UNKNOWN' (depend on the defaults at the computer). Def. 'UNKNOWN'

acc: 'SEQUENTIAL' (default) or 'DIRECT'

fm : 'FORMATTED'(for format I/O,default for sequential file),'UNFORMATTED' (default for direct file).

rl: INTEGER, record length (for direct file).

Blnk : Used for formatted files only.'NULL'(all blanks in numeric fields are to be ignored),'ZERO' (blank=0).

s: Label of an executable statement to which control is to be transferred if an error occurs.

ios: INTEGER VARIABLE,which will be positive if an error occurs or zero otherwise.

I/O Data from/to a File & END Option

For Sequential File:

```
READ(Unit number, k)  variable list
```

read data from the file which was associated with Unit_number in an Open statement

- *k* is the label of the FORMAT statement and in list-directed I/O, *k* is replaced by *.
- If we replace unit_number by *, then the I/O device will be the default I/O device.

eg.
```
read(10, file= "E12Q1.dat")
```

```
WRITE(Unit number, k)  expression list
```

store data into the file which was associated with Unit_number in an Open statement)

eg.
```
write(10, file= "E12Q1.out")
```

For Direct File:

```
READ(Unit_number, k, REC=record number) variable list

WRITE(Unit_number, k, REC=record number) expression list
```

END Option:

```
READ(Unit_number, k, END=n) variable list
```

- As long as there is data to read in the data file, read data
- If the last data in the file has already been read,control is passed to the statement with label n.

(c) CLOSE Statement

A file may be disconnected from a unit by a CLOSE statement:

```
CLOSE ( UNIT=u, IOSTAT=ios, ERR=s, STATUS=stat)
```

where *stat* : 'KEEP' (default, file is to be kept after the unit is closed) otherwise 'DELETE'.

(d) REWIND Statement

```
REWIND (UNIT=u, IOSTAT=ios, ERR=s)
```

repositions a sequential file at the first record of the file.

(e) BACKSPACE Statement

```
BACKSPACE (Unit=u, IOSTAT=ios, ERR=s)
```

repositions a sequential file to the last record read.

(f) ENDFILE Statement

```
ENDFILE (UNIT=u, IOSTAT=ios, ERR=s)
```

writes a special record to specify the end of the file.

(g) The INQUIRE Statement

```
   INQUIRE (UNIT=u,IOSTAT=ios,ERR=s,EXIST=e,OPEN=o,NUMBER=n,NAMED=nmd,
+  NAME=fn,ACCESS=acc,SEQUENTIAL=seq,DIRECT=dir,FORM=fm,FORMATTED=fmt,
+  UNFORMATTED=ufm,RECL=rl,NEXTREC=nr,BLANK=blnk)
```

Notes: (a) inquiry can be by unit (UNIT=u) or by name (FILE=name) but not both.
(b) the inquire statement will return information about the file via *ios, s, e, o, ... blnk.*

12.5 Application: Runge-Kutta Methods for Solving Initial Value Problems

In modelling many real world problems, one often needs to solve a differential equation or a set of differential equations subject to certain initial and boundary conditions. However, there are relatively few cases for which an analytical solution can be found. Numerical methods for the solution of differential equations are therefore extremely important.

Consider the first order initial value problem:

$$\begin{cases} \dfrac{dy}{dx} = f(x, y) \\ y(a) = y_0. \end{cases}$$

If an analytical solution, that is a function y(x) satisfying the differential equation and the imposed initial conditions or boundary conditions, cannot be found and we wish to know the relation of y and x for $x \in [a, b]$, then we first divide the interval $[a, b]$ into N subintervals with nodes $a = x_0,\ x_1, \dots \dots,\ x_n = b$, as shown below

x_0		x_1		x_2		...		x_n
y_0	$\rightarrow$	y_1	$\rightarrow$	y_2	$\rightarrow$	...	$\rightarrow$	y_n

Then for each x_i, we compute a corresponding y_i approximating the exact solution $y(x_i)$. The sequence $\left(x_i, y_i\right)\Big|_{i=0}^{N}$ is called the numerical solution to the differential equation.

One of the algorithms in common use for the initial value problems is the Runge-Kutta method of order four, which is as follows.

Fourth order Runge-Kutta Method

$$k_1 = hf(x_n, y_n),$$

$$k_2 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1),$$

$$k_3 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2),$$

$$k_4 = hf(x_n + h, y_n + k_3)$$

$$y_{n+1} = y_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

Remark: Starting from the initial values $(x_0,\ y_0)$, the above formula can be used to generate numerical solutions $(x_i,\ y_i)\Big|_{n=1}^{N}$.

Algorithm

```
Input:   a, b, n, y0
Output: XI, YI – output of the numerical solution (x_i, y_i), i = 1, N.

h = (b–a )/n
 x = a
 y = y0
For i = 1 to n  do
       k1=h*f(x,y)
       k2=h*f(x+h/2,y+k1/2)
       k3=h*f(x+h/2,y+k2/2)
       k4=h*f(x+h,y+k3)
       y=y+(k1+2*K2+2*k3+k4)/6
        x = x+h
       print  x, y
```

Example Write a F95 program to solve the following IVP on the interval [1,2] using step size h=0.1 (i.e divide [1, 2] into 10 subintervals, n=10).

$$y' = \left(\frac{y}{x}\right)^2 + \frac{y}{x}, \quad y(1) = 1.$$

```
Program RK4_Initial_Value_Problem
!
! Input:
!
!   a = the left end of the interval under consideration;
!
!   b = the right end of the interval;
!
!   n = number of subintervals: the interval [a, b]
!       will be divided into n subintervals,
!
!   y0= the value of y(x0).
!
! Output
!
!   [(xi, yi), i=1,n ] = the numerical solution of the IVP
```

```
!
!determine the kind type required for variables to store real data
!   with at least 15 digits of precision
!
    implicit none
    integer, parameter::k15=selected_real_kind(p=15)

!declare variables
!
    integer :: i,n
    real(kind=k15):: a, b, y0, h, f,x,y, k1,k2,k3,k4
!
!define a statement function
    f(x,y)=(y/x)**2+y/x
!
!input data
!
    print*, 'enter a, b, n, y0'
    read(*,*) a, b, n, y0
    h = (b-a)/n
!
!Create a new file for storing numerical results
!
    open (10, file='RK4_out')
!
!Print numerical sols heading and initial data into the file
!
    write(10,10)
        10 format(1x, 'Numerical Solution of the IVP')
    Write(10,11)
        11 format(1x,'      xi          y(xi)      ')
    write(10,20)a,y0
!
!starting from (x0,y0), generate num sols (xn+1,yn+1) step by step
!                        for n=0,1,...
    x = a
    y = y0
    do i=1,n
        k1=h*f(x,y)
        k2=h*f(x+h/2,y+k1/2)
        k3=h*f(x+h/2,y+k2/2)
        k4=h*f(x+h,y+k3)
        y=y+(k1+2*K2+2*k3+k4)/6
        x = x+h
        Write (10, 20) x, y
           20 format (1x, F10.5, 2x, F10.5)
    end do
```

```
    close(10)
!
 End Program!
```

Input Data & Computation Results

Input Data

```
1,  2,   10, 1.0
```

Results

```
Numerical Solution of the IVP
         xi            y(xi)
     1.00000      1.00000
     1.10000      1.21588
     1.20000      1.46755
     1.30000      1.76236
     1.40000      2.10989
     1.50000      2.52290
     1.60000      3.01877
     1.70000      3.62169
     1.80000      4.36640
     1.90000      5.30468
     2.00000      6.51711
```

EXERCISE 12

Q12.1 What will be printed from each of the following statements (a - e). Assume that WORD has been specified with the statements CHARACTER (LEN=6) :: WORD

(a)
```
   WORD='PEOPLE'
   PRINT 1, WORD
 1 FORMAT(3X, A5)
```

(c)
```
I=ICHAR('C')
WORD=CHAR(I)
PRINT *, WORD
```

(e)
```
WORD='TO BE'
K=INDEX(WORD, 'BE')
PRINT *, K
```

(b)
```
WORD='DENSITY'
PRINT *, WORD
```

(d)
```
WORD='CAN''T'
PRINT *, WORD
```

(Ans: (a) ^ ^ PEOPL ; (b) DENSIT; (c) C ; (d) CAN'T ; (e) 4

Q12.2 Show the output of variables

Value of variable	12	-123	-137.5	75.831	-1234.2	-1234.2	$-1.5*10^{-17}$
Format code	I3	I3	F10.2	F8.1	F9.4	E12.4	E12.4
Output							

(Ans: 12, ***, ^ ^ ^ -137.50, ^ ^ ^ ^ 75.8 , ********* , ^-0.1234E+04 ,^ 0.1500E-16 where ^ = blank)

Q12.3 Show the output from the following statements. Use D=3.865D+05, C1=(3.6, -2.46), C2=(-68.5, -714.2)

```
      PRINT *,  'D=', D, 'C1=', C1, 'C2=', C2
      WRITE(*, 1) C1,C2
   1  FORMAT (1x, 4 (F6.1,1x) )
```

(Ans: D=0.3865 D+06 C1=(3.6, -2.46) C2=(-68.5, -714.2) and ^ ^ ^ 3.6 ^ ^ ^ - 2.5 ^ ^ -68.5^ -714.2)

Q12.4 For I=10, J=20, X=5.72, Y=20.3, and the format specification (a)-(d), show the output from the print statement

```
PRINT 10 ,  I, J, X, Y
```

(a) 10 FORMAT(1x, I4, I4, 2x, F6.2, 2x, F6.2)
(b) 10 FORMAT(3x, 2I4, 2(2x, F6.2))
(c) 10 FORMAT('0', 'I=', I2, 'J=', I2, 2x, 'X=', F6.2, 2x, 'Y=', F6.2)
(d) 10 FORMAT(//1x, I4/1x, I4/2x, 2(F6.2, 1x))

(Ans: (a) ^ ^ 10 ^ ^ 20 ^ ^ ^ ^ 5.72 ^ ^ ^ 20.30 - single spacing
(b) ^ ^ ^ ^ 10 ^ ^ 20 ^ ^ ^ ^ 5.72 ^ ^ ^ 20.30 - single spacing
(c) I=10 J=20 ^ ^ x= ^ ^ 5.72 ^ ^ y= ^ 20.30 - double spacing

Q12.5. Show the values that will be stored in variables after reading (where ^ = blank)

Data line	^1234	^1^2^3	1234567	726.89	^^1245E+02	^^112233
Format code	I4	I6	F7.2	F6.1	E10.3	E7.2
Value stored						

Q12.6. Show the values that will be stored in the variables after execution of the following read statements

(a)
```
      READ 15, ID, HI, WIDTH
   15 FORMAT(I5, 2x, 2F5.1)
```
Data line : ^ ^ 456 ^ ^ ^ ^268 ^ .457

(b)
```
      READ(*, 10) ID, Hi, WIDTH
   10 FORMAT (I5, 2x, F4.2, f4.1)
```
Data line : ^ 456 ^ ^12.3 ^ ^ .868

Programming

Q12.7 Write a complete program to read a real value with at least 12 digits of precision from the terminal and compute the sine of the value using the following series

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots\ldots + (-1)^{n-1}\frac{x^{2n-1}}{(2n-1)!}$$

Continue using terms until the absolute value of a term is less than 1.0E-09. Print the computed sinx and the value obtained from the intrinsic function SIN for comparison. Try x=25° and 35° (x in above series and DSIN(x) is in radians). Print results:

x=***.** (degree), sin*x*= **.*****, sin(*x*) =**.*****

Hints

The following is an algorithm (you can also design your own algorithm). To understand how the sinx is calculated by using a condition DO loop, follow the algorithm to work out by hand the values (in terms of x) of the variables **Term** and **Sinx** for n =1, 2 , 3, ...

	ALGORITHM
n=1	
Term=x_r	*Input x (degree)*
Sinx=x_r	*Convert x in degree to x_r in radians*
n=2	*Set Sinx=0*
Term= ?	*n=1*
Sinx= ?	*Term=x_r*
n=3	*While ($\|Term\| > 1.0D-09$) Do*
Term= ?	*Set Sinx=Sinx + Term*
Sinx= ?	*n =n+1*
	*Term = - Term * x_r **2 /((2*n-2) * (2*n-1))*
	Output x, Sinx, sin(x_r)

Note: from the formula given, it can be noted that

$$nth \text{ term} = \text{- previous_term} \times \frac{x^2}{(2n-2)(2n-1)}$$

Q12.8 In a circuit as shown in Fig.1, U=220V, R_0=10 Ω , L_0=0.001H, R_1=100 Ω, C_1=0.0001F, R_2=50 Ω, L_2=0.01H, C_2=0.0002F, w=314.59, write a complete program to calculate the currents I, I_1 and I_2. Read data from a input data file (Q12_8. IN) and store the results in a data file (Q12_8.OUT) using the following formatted:

I =(**.***, **.***)
I1=(**.***, **.***)
I2=(**.***, **.***)

Hint: (1) In the program, you need to use two open statements to associate an input data file(Q12_8.IN) and an output file (Q12_8.OUT) with the program.

(2) Before the program is run, you need to create the input data file (Q12_8.IN) and enter the data values one by one in the order to be read, so that once the read statements in the program are executed, the computer will read the values from the data file.

(3) $Z_0 = R_0 + iwL_0$, $Z_1 = R_1 - i\frac{1}{wC_1}$, $Z_2 = R_2 + iwL_2 - i\frac{1}{wC_2}$

$$Z = Z_0 + Z_{12} = Z_0 + \frac{Z_1 Z_2}{Z_1 + Z_2} , \quad I = \frac{U}{Z} , \quad I_1 = I\frac{Z_{12}}{Z_1} , \quad I_2 = I\frac{Z_{12}}{Z_2}$$

CHAPTER 13

F95 ARRAY PROCESSING

An **array** is a group of storage locations that have the same name.

- Individual members of an array are called **elements** and are distinguished by using the common name followed by a number of **subscripts** in parentheses.
- Each element can store one data and thus an array can store a group of data.

13.1 One-Dimensional Arrays

A one-dimensional array can be visualized as either one column or one row of spaces for storing data, each space can store one data and is referenced with the array name followed by a subscript. The storage locations and associated names for a 1-D real array *A* of 6 elements are shown as follows.

1.0	2.0	0.5	3.0		

A(1:6) *A(1)* *A(2)* *A(3)* *A(4)* *A(5)* *A(6)*

Declaration

Whenever we create an array, we need to specify its name, type and size or range of index, so that the compiler can allocate sufficient storage units for storing a group of data. There are two ways of doing this:

(1) Use a **dimension attribute** TYPE, DIMENSION(*size*) :: *array*_1, *array*_2

eg.

```
REAL, DIMENSION(50)  ::  a, b, c
REAL, DIMENSION(0:50) :: x, y
```

(2) Use an **array specification** TYPE :: *array*_1(*size*), *array*_2(*size*)

eg.

```
REAL :: a(50), b(100), c(-10:50)
```

- The size of arrays can be represented by two ways

n (total number of elements) : the subscript starts from 1 *to n*.

lower_bound : upper_bound : the subscript starts from *lower_bound* to *upper_bound.*

- The two forms of array declaration can be combined in a single statement, in which case the value specified in the dimension attribute applies to all variables which do not have their own array specification:

```
REAL, DIMENSION(-10:20) :: a, b, c(30)
        ! index range for  a and b  : -10, -9, ..., 20;   for c:  1,2, ...,30
```

Input/Output

- To read an entire array, reference the array name directly.
  eg. READ *, A
- To read certain specific elements, reference the elements directly.
  eg. READ *, A(1), A(11)
- To read part of an array, use an implied DO loop to identify the elements to be read.
  eg. READ *, (A(I), I=1, N)

Notes: * N can be an integer constant, integer variable or expression

* Methods for printing values from arrays are the same as for reading.

Example. A set of 50 data has been entered into an input file, one value per line, give a set of statements to read the data into an array TEMP(50).

Sol. We can use one of the following sets

(a)
```
Read *, TEMP
```

(b)
```
Do   I=1, 50
     Read *, TEMP(I)
END DO
```

(c)
```
Read *, (TEMP(I), I=1, 50 )
```

Questions * If TEMP is declared as TEMP(100), then (a) cannot be used. Why ?

* If the form of data in the file is two values per line, (b) cannot be used. Why ?

13.2 Two-Dimensional Arrays

A 2-D array can be visualized as a group of columns (or a table) as illustrated below

2-D array with 4 rows and 5 columns

1	0	2	5	6	*Row 1*
2	1	3	4	-1	*Row 2*
-2	1	-9	8	9	*Row 3*
1	0	0	2	3	*Row 4*
Col. 1	2	3	4	5	

Unlike in 1-D arrays, elements in 2-D arrays must be referenced with two subscripts.

- The first subscript references the row
- The second subscript references the column.

eg. A(3, 4) refers to the data value in row 3 and column 4. In the above example A(3, 4)=8.

Declaration

In the same way as for 1-D arrays, the sizes (for both subscripts) of 2-D arrays can be specified with either a dimension attribute or an array specification.

eg.
```
REAL, DIMENSION(100, 5) :: x, y, z
INTEGER :: count(100, 3), a(100,5)
```

Input/Output

In the same way as for 1-D arrays, we can read the entire array, some specific elements or part of an array using an implied DO loop. It should be addressed that if we use the array name without subscripts in I/O (read *, A) we access the array with the 1st subscript changing fast, 2nd slower, i.e., read column 1, then column 2, ...

Example. Give a set of statements to define a 2-D array *A* and to read, using *A*, the values in a matrix of *N* (< 10) rows and *M* (< 4) columns row by row (row 1 first, then row 2...).

Sol.

(a)

```
Real :: A( 10, 5 )
Read *, N, M
Read *, (( A(I, J),J=1,M), I=1, N  )
```

Remark: In the program above, each cycle of the outer implied Do loop reads one row ; the values in each row are read by the inner implied Do loop.

(b)

```
      Real :: A(20, 10)
      Read *, N, M
      Do   2   I=1, N
            Read *,  ( A(I, J), J=1, M )
  2   Continue
```

Questions

* Can we declare A using
  - i) REAL::A(N, M) - No.
  - ii) REAL::A(5, 4) (assume N=6) - No.
  - iii) REAL:: A(20,20) - OK, why?

* How to modify the program if the data values are to be entered column by column (column 1, then column2, ...)

* With the above programs, can we enter all the data values in one data line ? -OK for (a) but not OK for (b), why?

* Can we enter the data each value per line ? - OK for both (a) and (b), why?

13.3. Multi-Dimensional Arrays

Fortran 95 allows as many as seven dimensions for arrays. We can easily visualize a 3-D array as a cube. Elements in 3-D arrays are referenced with three subscripts. If we use the 3-D array name without subscripts we access the array with the *1st subscript changing fastest*, the *2nd subscript changing second fastest* and the *3rd subscript changing the slowest*. Most applications do not use arrays with more than three dimensions.

13.4 Array Operations

13.4.1 Array Element Operations

An array element can be used anywhere that a scalar variable can be used. In exactly the same way as a scalar variable, it identifies a unique location in the memory to which a value can be assigned or input, and whose value may be used in an expression or output list, etc. The great advantage is that by altering the value of the array subscript it can refer to a different location. Thus the use of array variables within a loop greatly increases the power and flexibility of a program. This can be seen from the following DO loop which enables 100 data to be input and stored for subsequent analysis in a way which is not otherwise possible.

```
DO I=1, 100
      read*, a(i)
END DO
```

In F77 and most other programming languages, this is the only way that arrays can be used in most types of operations. However, F95 enables an array to be treated as a single object in its own right, in much the same way as a scalar object.

13.4.2 Whole Array Operations

Def - Shape of an Array

- **Dimension**: If an array has two (say) subscripts, we say the array is a two dimensional array. For each dimension, there are two bounds(the lower bound and the upper bound) which define the range of values that are permitted for the corresponding subscripts.

- **Rank** - The number of permissible subscripts.

- **Extent of a dimension** - the total number of elements in that dimension.

- **Shape of an Array** - the shape of an array is determined by its rank and the extent of each dimension. Two arrays have the same shape if they have the same rank and the same extent (number of elements) for every dimension.

  eg . If a,b ,c, d and e are declared by

  ```
  REAL :: a(5,5), b(0:4, 0:4), c(5, -1:3), d(10), e(5, 10)
  ```

  then only a, b, and c have the same shape.

Def - Conformable Arrays

- Two arrays are conformable if they have the same shape
- A scalar, including a constant, is conformable with any array
- All intrinsic operations are defined between two conformable objects

Rules for Working with Whole Arrays

Two conformable arrays can appear as operands in an expression or an assignment, and the operation or assignment will be carried out on an element by element basis.

Thus, the following code fragment will result in the arrays a and b having identical values

```
REAL, DIMENSION(50) :: a,b,c,d

a=c*d                ! F95 style

DO I=1,50            ! F77 style
   b(i)=c(i)*d(i)
END DO
```

- It is obvious that the F95 style is much easier to read than the earlier F77 style.

(a) **Assignment**

```
arr_1=0             ! set every element of the array arr_1 to zero.
arr_1=15*arr_2      ! cause every element of the array arr_1 to be assigned a value 15 times
                    ! the corresponding element of the array arr_2,
```

(b) **Initialization**

```
REAl, DIMENSION(50,2) :: a=0.0, b=0.0
INTEGER, DIMENSION(3) :: c=(/ 1, 10, 15 /)     ! set initial value c(1)=1, c(2)=10, c(3)=15
```

(c) **Intrinsic Procedures with arrays as arguments**

Many of the F95 intrinsic procedures accept arrays as actual arguments and will return as their result an array of the same shape as the actual argument in which the procedure has been applied to every element of the array.

Example:

```
arr_1=SIN(arr_2)    ! assign the sine of each element of the array arr_2 to each
                    ! corresponding element of the array arr_1.
```

(d) **Intrinsic Procedures specially deigned for array operations**

Procedure Name	***Result***
MATMUL(matrix_A, matrix_B)	Matrix product of two matrices, or a matrix and a vector.
DOT_PRODUCT(vector_A, vector_B)	Scalar(dot) product of two vectors
TRANSPOSE(matrix_A)	Transpose of the matrix matrix_A
MAXVAL(array)) MAXVAL(array, dim)	Maximum value of all the elements of an array, or of all the elements along a specified dimension of an array
MINVAL(array) MINVAL(array,dim)	Similar to MAXVAL
PRODUCT(array) PRODUCT(array, dim)	Product of all the elements of an array, or of all the elements along a specified dimension of an array
SUM(array) SUM(array,dim)	Similar to PRODUCT

- All above functions are GENERIC functions. A generic function is one whose resultant type(real, integer ...) depends on the type of the argument.

(e) Masked array assignment

F95 allows a finer degree of control over the assignment of one array to another, by use of a mask which determines whether the assignment of a particular element should take place or, alternatively, which of two alternate values should be assigned to each element. This concept is called masked array assignment, and comes into two forms.

(i) WHERE(*mask_expression*) *array_assignment_statement* or

```
WHERE(mask_expression)

        array_assignment_statements

END WHERE
```

where *mask_expression* is a logical expression.

eg. If arr is a real array, then the effect of the statement

```
WHERE(arr > 0) arr=-arr
```

is to change the sign of all the element of the array arr having positive values and to leave those having negative values unchanged. This is because the assignment is performed element by element and only for the element whose value is greater than zero, the logical condition is true and the assignment statement is performed.

(ii)

```
WHERE(mask_exp)
        array_assignment_statements
ELSEWHERE
        array_assignment_statements
END WHERE
```

Example. The following statements

```
WHERE(arr/=0.0)
        arr=1.0/arr
ELSEWHERE
        arr=1.0
END WHERE
```

can be used to replace every non-zero element of the array arr by its reciprocal, and every zero element by 1.0.

(f) **SUB-arrays and array sections**

In F95, we can define a sub-array, consisting of a selection of elements of an array, and then manipulate this sub-array in the same way that a whole array can be manipulated.

Array sections can be extracted from a parent array in a rectangular grid (regular spacing) using subscript triplet notation. A subscript triplet takes the following forms

subscript_1 : Subscript_2 : stride - defines an ordered set of subscripts that start at subscript_1 and end on or before subscript_2 and have a separation of stride between consecutive subscripts.

- If subscript_1 is omitted (: subscript_2 : stride), it starts from the lower index bound;
- If subscript_2 is omitted (subscript_2 : : stride), it ends on or before the upper index bound;
- If stride is omitted (subscript_1 : subscript_2), the stride defaults to the value 1.

Example. If the array arr is declared by

```
REAL, DIMENSION(10) :: arr
```

then

```
arr(1:10)      ! identical to err
arr(3:5)       ! 1-D array containing elements  arr(3), arr(4), arr(5).
arr(:9)        !                                arr(1), arr(2), ...., arr(9).
arr(::4)       !                                arr(1), arr(5), arr(9)
arr(:)         ! identical to arr
```

Aplication example. Calculate the mean value of the first n (n<100) elements of the array x(1:100).

```
mean=sum(x(1:n))/n      ! F95 style

sum=0.0                 ! F77 style
DO i=1,n
   sum=sum+x(i)
END DO
mean=sum/n
```

- Arrays of dimension greater than one can have a section defined by using a subscript triplet for each dimension.

  eg. If the array a has been declared by

  ```
  REAL, DIMENSION(10,10) :: a
  ```

  then

```
a(: , :)        ! identical to a
a(5, :)         ! row 5 of a :  a(5,1), a(5,2), ..., a(5,10)
a(2:5,2)        ! part of column 2 : a(2,2), a(3,2), a(4,2), a(5,2)
a(1:2,3:5)      ! sub-matrix with 2 rows and 3 columns.   row 1:  a(1,3), a(1,4), a(1,5)
                                                          row 2:  a(2,3), a(2,4), a(2,5)
```

13.5 Allocatable Arrays

In previous sections, the shape of array is given explicitly in the declaration by a constant or a constant expression. This means that a fixed amount of space will be allocated to the array once the array is declared. This is not efficient in terms of the use of memory. To overcome the problem, F95 provides another kind of arrays, namely allocatable arrays. With allocatable arrays, users can control the size of the arrays based on the need for the specific problem to be solved.

In the following, we describe how to declare an array as an allocatable array through the use of an allocatable attribute, how to allocate spaces to allocatable arrays and how to release the space of an allocatable array once it is no longer needed.

Declaration of Allocatable Arrays

An allocatable array is declared in a type declaration statement with an allocatable attributed.

eg.

```
Real, Allocatable, Dimension(:,:,:,:):: allocatable_arrayA
```

Remarks:

- The rank of the array is the same as the number of colons. So the above statement declare a rank – 4 array.

- Allocatable arrays are deferred-shape array as the declaration of an allocatable array does not allocate any space for the array and hence the array cannot be used until it is allocated some space in each dimension via the use of an allocate statement.

Allocation of Space

Before an allocatable array can be used, we need to use the allocate statement of the following form to specify the extent in each dimension of the array.

```
Allocate(list of array_specifications, STAT=status_variable)
```

eg.

```
Allocate(arr_1(20), arr2(10:30,0:20), arr3(10,0:20,5))
```

Remarks:

(1) The "STAT=status_variable" element of the ALLOCATE statement enables the processor to report on the success of the allocation process. If the allocation is successful then the status_variable is set to zero, otherwise it will be assigned a positive value.

(2) Once space has been allocated to the array, the allocatable array may be used in the same way as explicit-shape arrays.

(3) Allocatable array can be used as actual array to pass data to procedure but cannot be used as dummy array in procedure.

(4) After the array has been used and is no longer required, the space for the array can be deallocated by a deallocate statement.

Release of Space by a Deallocate Statement

Once an allocatable array is no longer needed, a deallocate statement in the form as below can be used to make the memory space that it was using available for other purposes.

```
Deallocate(list of currently allocated_array, STAT=status_variable)
```

eg.

```
Deallocate(arr_1(20), arr2(10:30,0:20), arr3(10,0:20,5))
```

Remarks

(1) Once an array is deallocated, the values stored in its elements are no longer available.

(2) Exit from a procedure with allocatable arrays will also lead to the lost of values stored in the allocatable array. If the values are to be kept, then the allocatable array should be declared with a save attribute.

```
  Real, Allocate, Dimension(:), save :: A1
```

(3) The greater control provided by allocatable arrays can be used to write programs with more capacity. For example, in writing a program for users to solve linear systems of equations Ax=b, if we define A as a 2-D array with 1000 rows and 1000 columns, then the program is not directly usable if the user want to solve a system with more than 1000 equations. On the other hand, if the user only needs to solve a system with 100 equations, running of the program will unnecessarily use a lot of memory. However, by using allocatable arrays, the problems can be solved. The program can be designed to prompt user to enter the number of equations (say N), then use the N value entered to create an allocatable array with N rows and N columns for the matrix A.

Example. Read an integer number N and a set of N real numbers into an one-dimension allocatable array. Then calculate the average vale.

```
Program EXERCISE
Implicit None
Integer::N,alloc_error,dealloc_error
Real::Average

Real,alllocatable,Dimension(:)::x
        Read(*,*) N
        Allocate(x(N), STAT=alloc_error)
        If(alloc_error /= 0) then
 Print *, "Insufficient space to allocate array when N=",N
            STOP
        end if

Read(*,*) (x(i),i=1,N)
        Average=sum(x(1:N))/N
        Print *, "Average=",Average
        Deallocate(x, STAT=deallc_error)
        If(dealloc_error /= 0) Then
            print  *, "unexpected deallocation error"
            STOP
        end if
    End Program
```

13.6 Application: Solution of Tridiagonal Systems of equations

A square matrix A_T is tridiagonal if its nonzero entries are only on the main diagonal, the sub-diagonal and super-diagonal, i.e $A_T = \left[a_{ij}\right]$ is tridiagonal if $a_{ij} = 0$ for $|i-j| > 1$.

Provided no row interchanges are required for A_T, we can factor A_T as

$$\begin{bmatrix} d_1 & c_1 & & & & \\ a_2 & d_2 & c_2 & & O & \\ & a_3 & d_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & O & & a_{n-1} & d_{n-1} & c_{n-1} \\ & & & & a_n & d_n \end{bmatrix} = \begin{bmatrix} \alpha_1 & & & & & \\ a_2 & \alpha_2 & & & O & \\ & a_3 & \alpha_3 & & & \\ & & \ddots & \ddots & & \\ & O & & a_{n-1} & \alpha_{n-1} & \\ & & & & a_n & \alpha_n \end{bmatrix} \begin{bmatrix} 1 & \beta_1 & & & & \\ & 1 & \beta_2 & & O & \\ & & 1 & \beta_3 & & \\ & & & \ddots & \ddots & \\ & O & & & 1 & \beta_{n-1} \\ & & & & & 1 \end{bmatrix}$$

We can multiply the L and U matrices to obtain a way to compute α_i and β_i recursively. From the above equations, we have

$$\begin{aligned} & d_1 = \alpha_1,\ c_1 = \alpha_1\beta_1 \\ & d_i = a_i\beta_{i-1} + \alpha_i \qquad (i = 2,\ 3, \dots,\ n) \\ & c_i = \alpha_i\beta_i \qquad (i = 2,\ 3, \dots,\ n-1) \end{aligned}$$

which give rise to the following recurrence formulae,

$$\begin{aligned} & \alpha_1 = d_1, \quad \beta_1 = c_1 / \alpha_1 \\ & \left.\begin{aligned} & \alpha_i = d_i - a_i\beta_{i-1} \\ & \beta_i = c_i / \alpha_i \end{aligned}\right\} \qquad (i = 2,\ 3, \dots,\ n-1) \\ & \alpha_n = d_n - a_n\beta_{n-1} \end{aligned}$$

Now consider a tridiagonal system of equations $A_T x = b$, as A_T can be factorized into LU, the system becomes $LUx = b$ and consequently can be solved by the following forward and backward substitutions:

Forward substitution $\quad Ly = b \quad \Rightarrow \quad y_1 = b_1 / \alpha_1$

$$y_i = (b_i - a_i y_{i-1}) / \alpha_i \qquad (i = 2,\ 3, \dots,\ n)$$

Backward substitution $\quad Ux = y \quad \Rightarrow \quad x_n = y_n$

$$x_i = y_i - \beta_i x_{i+1} \qquad (i = n-1,\ n-2, \dots,\ 1)$$

Based on the above formulae, the following algorithm can be developed to solve tridiagonal system of equations.

Algorithm for solving Tridiagonal Systems $A_T x = b$

Input

N = Number of equations.
A(N) = Before entry, must contain the sub-diagonal element of $A_T(a_i)$.
D(N) = Before entry, must contain the main diagonal element of $A_T(d_i)$.
C(N) = Before entry, must contain the super-diagonal element of $A_T(c_i)$.
B(N) = Before entry, must contain the element of the right hand side (b_i).

Output

X(N) = Output. On exit, contain the solution of the tridiagonal system.

Set $\alpha_1 = d_1 \qquad \beta_1 = \dfrac{c_1}{\alpha_1}$

For $i = 2$ to $n-1$

set $\alpha_i = d_i - a_i \beta_{i-1}$

$\beta_i = \dfrac{c_i}{\alpha_i}$

Set $\alpha_n = d_n - a_n \beta_{n-1}$

$y_1 = \dfrac{b_1}{\alpha_1}$

For i= 2 to n do

set $y_i = \dfrac{1}{\alpha_i}\left[b_i - a_i y_{i-1}\right]$

Set $x_n = y_n$

For i= n –1 to 1 by –1 do

set $x_i = y_i - \beta_i x_{i+1}$

Output $\left(x_1, x_2, \ldots, x_n\right)$

Example Write a well-structured F95 program for solving tridiagonal systems $A_T X = b$ using the algorithm given above

$$\begin{bmatrix} 1 & -1 & 0 \\ -2 & 4 & -2 \\ 0 & -1 & 2 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 0 \\ -1 \\ 1.5 \end{Bmatrix}$$

```
PROGRAM TriDiag SysofEqs Solver
   !
   ! This  program  defines and solves the tri-diagonal system Ax=b
   !
   !     |d(1) c(1)                           |  | x(1) |      |b(1)|
   !     |a(2) d(2) c(2)                      |  | x(2) |      |b(2)|
   !     |     a(3) d(3) c(3)                 |  |  ... |      |... |
   !     |                ...                 |  |      |  =   |    |
   !     |              a(n-1) d(n-1) c(n-1)  |  |      |      |    |
   !     |                     a(n)   d(n)    |  | x(n) |      |b(n)|
   !
   !
   !  INPUT:   N        - set to the number of equations;
   !            A(2:n)  - set to the sub-diagonal elements,
   !           D(1:n)   - set to the diagonal elements,
   !           C(1:n-1) - set to the super-diagonal elements,
   !           B(1:n)   - set to the right hand side vector.
   !
   !  OUTPUT:  X        - contains the sol of the tri-diagonal sys.
   !
   !

!************************************************************************
   INTEGER, PARAMETER::REAL K=SELECTED REAL KIND(P=15)
   INTEGER :: I, N

REAL(KIND=REAL K)::A(20),D(20),C(20),B(20),X(20),ALPHA(20),BET(20),Y(20)
!
! define the linear system
   print*,'enter the number of eqs n'
         READ(*,*) N             ! enter number of eqs
   print*,'enter the sub-diagonal elements A(2:n)'
         READ(*,*) A(2:N)        ! enter the sub-diagonal elements
   print*,'enter the diagonal elements D(1:n)'
         READ(*,*) D(1:N)        ! enter diagonal elements
   print*,'enter the super-diagonal elements C(1:n-1)'
         READ(*,*) C(1:N-1)      ! enter the super-diagonal elements
   print*,'enter the right hand side vector b(1:n)'
         READ(*,*) B(1:N)        ! enter the right hand side vector
!
! solve the system of equations
!
   ALPHA(1)=D(1)
   BET(1)=C(1)/ALPHA(1)
   DO I=2,N-1
        ALPHA(I)=D(I)-A(I)*BET(I-1)
        BET(I)=C(I)/ALPHA(I)
   ENDDO
   ALPHA(N)=D(N)-A(N)*BET(N-1)
   Y(1)=B(1)/ALPHA(1)
   DO I=2,N
        Y(I)=(B(I)-A(I)*Y(I-1))/ALPHA(I)
   ENDDO
   X(N)=Y(N)
   DO I=N-1,1,-1
        X(I)=Y(I)-BET(I)*X(I+1)
        ENDDO
     !
     ! Print the results
```

```
!
     WRITE(*,100)
       100 FORMAT(1X,'The solution is')
     WRITE(*,101) (I, X(I),I=1,N)
       101 FORMAT((1X,'X(',I2,')=',F9.5))

END PROGRAM
```

INPUT Data

```
3
-2,   -1,
1,    4,     2,
-1,   -2
0,   -1,     1.5
```

Output

```
The solution obtained is
X(1)=  0.50000
X(2)=  0.50000
X(3)=  1.00000
```

EXERCISE 13

Q13.1 How is an array specification written?

Q13.2 What are the rank, extent, and shape of an array ?

Q13.3 What is meant by the statement that two arrays are conformable?

Q13.4 Write declarations for suitable arrays to store the following sets of data

(a) three matrices of 10 rows and 5 columns; (b) a vector with 100 elements

Q13.5 Show the output from each set of statements.

(a)

```
    INTEGER :: LIST(8)
    DO K=1,4
        LIST(5-K)=K
    END DO
    PRINT 10, (LIST(K),K=1,2)
10  FORMAT(2X,2(I2,2X))
```

(b)

```
      ......
    REAL :: TIME(50)
    DO  J=1, 10
        TIME (J)=REAL (J-1)*0.5
    END DO
    Do J=1,10,4
        PRINT 10,  J, TIME (J)
```

```
10        FORMAT (1X, 'TIME', I2, '=', F4.2)
     END DO
```

(c)

```
     .........
     INTEGER  :: K (3, 3)
     DO  I=1, 3
          K(I, 1)=5
          K(I, 2)=-5
          K(I, 3)=0
     END DO
     PRINT 30,  (K(3, J), J=1,3)
30   FORMAT (1X, I3)
```

d)

```
       .......
     REAL :: DIST (10,10)
     SUM=10.0
     DO  J=1, 3
          DO I=1,3
               SUM=SUM+1.5
               DIST(I,J)=SUM
          END DO
     END  DO
   DO I=1,2
       PRINT 15, (DIST(I,J), J=1,2
       15    ORMAT(1X,2F5.1)
   END DO
```

Ans: (a) ^ ^ 4 ^ ^ ^3
(b) Time ^ 1 = 0.00
Time ^5 = 2.00
Time ^9 = 4.00
(c)^^5
^-5
^^0
(d) ^11.5^16.0
^13.0^17.5

Q13.6. An array TIME contains 30 integers. Give statements that print one value from every five values, beginning with the fifth value, in the form:
Time (5) contains **** seconds
Time (10) contains **** seconds
Time (30) contains **** seconds

Ans:

```
DO  k=5, 30, 5
        Write(*, 2) k, Time(k)
   2    Format (1X, 'Time (', I2, ' ) contains', I4, 'seconds')
END DO
```

Q13.7 Give Fortran statements to interchange the first and tenth elements, the second and ninth elements, and so on, of the array NUM that contains 10 integer values.

Ans:

```
DO   I=1, 5
     IHold=NUM(I)
     NUM(I)=NUM(11-I)
```

```
    NUM(11-I)=IHold
END DO
```

Q13.8 Write a complete program that will read 5 integers to an array from keyboard, one data per line. Write the data in the reverse order from which it was read. Test your program with data 1, 5, 10, 20, 9999.

Ans:
```
Integer :: Num(10)
Read *, ( Num(I), I=1, 5 )
Do  I=5, 1, -1
     Write(*, *) Num(I)
END DO
```

Programming

Q13.9 Given

$$A = \begin{pmatrix} 1 & 2 \\ 1 & 3 \end{pmatrix},\ B = \begin{pmatrix} 2 & 0 \\ 1 & 1 \end{pmatrix},\ \ x = \begin{pmatrix} 1 \\ 2 \end{pmatrix},\ y = \begin{pmatrix} 2 \\ 3 \end{pmatrix}.$$

Write a F95 program, using whole array processing, to calculate C=A-B, D=2*A*B, E=A^T, z=$x.y$.

Q13.10 Write a program which reads and stores 10 real values into a 1-D array, then finds the maximum value using the intrinsic function MAXVAL and then finds the mean value of the last five input values using the intrinsic function SUM applied to an array section. Test your program using 1,2, -3, 10, -6, 2,1,3, 5,4,

Q13.11 Write a program, using masked array assignments, to calculate C=[c_{ij}] where

$$c_{ij} = \begin{cases} 2 * a_{ij} & if\ a_{ij} > 0 \\ 4 * a_{ij} & \text{otherwise} \end{cases}.$$

Test your program using $A = \begin{pmatrix} 2 & 1 & 0 \\ -1 & 3 & 2 \\ 5 & -2 & 6 \end{pmatrix}$.

Q13.12 A set of data with *N_row* rows and *N_col* columns, represents the elevations at nodes of a grid. Write a complete program to read the data and to locate the peak point (row number, and columns number).
Print the results using formatted output.

```
Number of peak point =    **
  No.      Location (row, column)
   1                **,      **
   2                **,      **
```

Test your program with the following data

```
63  23  21  34
43  30  37  32
38  39  36  28
```

42 48 32 30
40 42 48 49

(Hint: 1) To be a peak point, the point must be an interior point (no on the edge) and the elevation at the point must be greater than the elevations at its four adjacent points.Eg If (I, J) is a peak point, then the elevation at (I, J) must be greater than those at (I, J-1), (I, J+1), (I-1, J) and (I+1, J).

2) Algorithm. Use a 2-D array (say MAP) to store the elevations at nodes of the grid;
a 2-D array PEAK to store the row & column No. of peak points.

Input N_row, N_col
Input Map
Set Count=0
For each interior point Do
  If the point is higher than all 4 adjacent points, increment count by 1
    and store the row and Column No. of the point into PEAK
Output Count and PEAK.

Q13.13 (Optinal) .Write a subroutine that sorts a series of (assume *N*) character strings stored in a one dimensional character array into alphabetical order. Test your program using the following six character strings:

White Hill Lee Jones Major Grace

Input the character strings using free-format but output the sorted strings using formatted output, one name per line.

Hint.

- Use a 1-D array to read the character strings.
- Find the name with minimum value and place it first in the list, then scan the other names to find the one with the next minimum value and place it second in the list and so on. The algorithm can be implemented using nested DO Loops. The Inner Loop locates the name with minimum value from the remaining unsorted names. The Outer Loop has (N-1) cycles, and each of the cycles contains an inner loop to locate the name with minimum value and statements to relocate the name.

CHAPTER 14

F95 PROGRAM DESIGN & SUBPROGRAMS

14.1 Top-Down Design: Programs and Subprograms

The easiest way to solve most problems is to break them down into smaller sub-problems and deal with each of these in turn, further subdividing these sub-problems as necessary. Fortran provides two types of procedures, function subprograms and subroutine subprograms, to assist in the solution of such sub-problems. Thus, to simplify program logic, a F95 program is normally designed to consist of a main program, a number of subprograms and modules.

- Execution of the program will start at the beginning of the main program.
- The main program controls the execution order; each subprogram is used to perform some specific action, and modules are introduced to provide controlled access to global data or to create explicit interface between different program units.
- One program unit (main program, function, subroutine or module) needs never be aware of the internal details of any other program unit. The only link between one program unit and a subsidiary program unit is through the interface of the subsidiary program unit. This very important principle means that it is possible to write subprograms totally independent of the main program and of each other. This feature opens up the way for libraries of subprograms: collections of subprograms that can be used by more than one program. It also permits large projects to use more than one programmer; what the programmers need to communicate to each other is the information about the interfaces of their procedures.

The diagram which outlines the structure of a program is called a STRUCTURE CHART.

14.2 Function Subprograms

There are two kinds of functions: intrinsic functions such as SIN and SQRT which are part of Fortran language, and external functions which are defined by the user. A function subprogram separates from the main program unit and can be called by the main program unit to perform certain operations and to return the function value computed via the function name.

Example. Calculate the average value of a series of data.

We use a function AVERAGE to calculate the average of *N* values stored in a 1-D array *X,* use a main program to control the execution order (read *N* & *X*, calculate the average and print the result).

```
! Main Program:
!
PROGRAM E7_2_1
      IMPLICIT  NONE
      INTEGER N
      REAL :: X(100), AVERAGE
!
      READ *, N
      READ *, X(1:N)
      PRINT *,  'AVERAGE = ', AVERAGE(N, X )
END PROGRAM E7_2_1


! Function subprogram (follows the main prog.)
!
REAL FUNCTION  AVERAGE (N, X)
        IMPLICIT  NONE
        INTEGER, INTENT(IN) :: N
        REAL, DIMENSION(100), INTENT(IN) :: X
        REAL :: SUM_V
!
        SUM_V=SUM(X(1:N))
        AVERAGE=SUM_V/REAL(N)
END  FUNCTION AVERAGE
```

Writing a Function Subprogram

- A function subprogram begins with a FUNCTION header of the form

  type FUNCTION *name* (*dummy arguments*)

  followed by any fortran statements, and ends with an **END** statement of the form,

  END FUNCTION *name*

  where ***type*** (integer, real and etc.) is the type of the result of the function, ***dummy arguments*** can be variable names, array names and function names. When the **END** statement is executed, it causes execution of the program to return to the point in the calling procedure at which the function was referenced as though a variable had been

inserted in the code at that point, having as its value the value calculated by the function.

- ***Dummy arguments*** to functions should always be declared with an attribute INTENT(IN). It informs the compiler that the dummy arguments declared may not be changed by any statement in the function.

- A function subprogram must contain a variable having the same name as the function, and this variable must be assigned, or otherwise given, a value to return as the value of the function before an exit is made from the function. This special result variable must have a type. It is permitted either to declare its type as part of the FUNCTION statement, or to declare it by means of a conventional type declaration statement. For example, it is permitted to write

```
FUNCTION AVERAGE(N,X)
    IMPLICIT NONE
    REAL :: AVERAGE
```

- A ***local variable*** such as SUM_V has no existence outside the function. Thus the main program or another procedure could use the variable name for any purpose it wishes with no fear of the two uses of the same name being confused with each other.

- A function can contain reference to other functions.

Calling a Function Subprogram

Function subprograms can be referenced only as operands in expressions. The function name returns one value to the calling program. Functions are referenced in the form:

NAME (*actual arguments*)

- When a function is referenced,
  the 1st dummy argument will be assigned the value of the 1st actual argument
  the 2nd dummy argument will be assigned the value of the 2nd actual argument
  and so on.

- The actual arguments must match the dummy arguments in number, order and type. The argument variable names themselves do not have to match.

14.3 Subroutine Subprograms

Although function subprograms are useful when we need to compute a single value, there are applications in which we would like to use a program unit to return many values. In these instances, a subroutine is required.

Example. Calculate the average and maximum values of a series of data stored in a 1-D array.

```
!  Main Program:
!
PROGRAM E   CH14B
        IMPLICIT NONE
        INTEGER :: N, I
        REAL :: X(100), MAX_X, MEAN
!
        READ *, N
        READ *, ( X(I), I=1,N)
        CALL  AVERAGE (N, X, MEAN, MAX_X)
        PRINT *, MEAN, MAX_X
END PROGRAM E  CH14B
!
! Subroutine (follows the main prog.)
!
SUBROUTINE  AVERAGE (N,  X, MEAN,  MAX_X)
        IMPLICIT  NONE
        INTEGER, INTENT(IN) :: N
        REAL, INTENT(IN) :: X(100)
        REAL, INTENT(OUT) :: MEAN, MAX_X
        MEAN=SUM(X(1:N))/N
        MAX_X=MAXVAL(X(1:N))
END SUBROUTINE AVERAGE
```

Writing a Subroutine

Writing a subroutine is much like writing a function, except that the first line in a subroutine identifies it as a subroutine. The general form of this statement is

SUBROUTINE NAME (*dummy arguments*)

A subroutine differs from a function in the following ways

- A subroutine does not represent a value.
- A subroutine uses the dummy arguments not only for subroutine input but also for output.

  Dummy arguments for subroutine input should always be declared with an attribute INTENT(IN);

  Dummy arguments for subroutine output should be declared with INTENT(OUT);

Dummy arguments for both subroutine input and output should be declared with INTENT(INOUT).

- A subroutine may return one value, many values or no value.

Calling a Subroutine

A subroutine is called with an executable statement whose general form is

CALL subroutine_name (*actual arguments*)

- When a subroutine is called,

  the 1st dummy argument will be assigned the value of the 1st actual argument

  the 2nd dummy argument will be assigned the value of the 2nd actual argument
  and so on.

- The actual arguments must match the dummy arguments in number, order and type. The argument variable names themselves do not have to match.

14.4 Modules

Another form of program unit which does not exist in Fortran 77 is a module. One very important use of modules relates to global accessibility of variables, constants and etc. A module declares/defines a set of variables and constants which can be made *accessible to other program units.*

Defining a Module

A module starts with a MODULE *statement* and ends with an end statement, *as shown below.*

```
MODULE name
      IMPLICIT NONE
      SAVE
          .....
END MODULE name
```

Example.

```
MODULE  global_dat1
      IMPLICIT NONE
      SAVE
      REAL, PARAMETER :: pi=3.1415926, piby2=pi/2.0
       REAL :: x1, x2
END MODULE global_dat1
```

- A module's declaration usually must precede its use

- A wide range of items may be declared within a module and made accessible to other parts of the program in a way which provides an extremely powerful functionality.

Accessing a Module

Any program unit which wishes to access items in a module (say module with name *module_name*) needs to include only a statement of the form

```
USE module_name
```

to make all defined variables and constants available. Note that the USE statement **comes after** the initial statement (PROGRAM, SUBROUTINE or FUNCTION) but before any other statements.

Example.

```
SUBROUTINE sub1
USE global_dat1
IMPLICIT NONE
  ......
    x1 =PI         ! both accessed from module
    print*, x1     ! prints 3.141593 (approximation)
    call sub2      ! x1 now has the value 2.5
    print *, x1    ! prints 2.5
END
!
SUBROUTINE sub2
USE global_dat1
IMPLICIT  NONE
    x1=2.5         ! variable x1 is accessed from module
END
```

14.5 Modules and Explicit Procedure Interfaces

As we have known, the only link between a procedure and a calling program unit is through the interface of the procedure (the names of the procedure, the name and characteristics of each of its dummy arguments). Traditionally, the interface is implicit: calling program unit knows nothing about the procedure and vice versa. F95 provides several means to make the interface of a procedure explicit. One of the ways is to place the procedure in a module. The rules relating to modules specify that

- the interfaces of all the procedures defined within a single module are explicit to each other.

- the interfaces of any procedures made available by the USE association are explicit in the program unit that is using the module

Therefore, in writing a complete program, we recommend to

- group procedures into one or possibly more than one module if necessary

```
MODULE    module_name
       IMPLICIT NONE
CONTAINS
       procedure_1
       procedure_2
       ........
END MODULE module_name
```

Note: a CONTAINS statement must be placed before the first procedure within a module.

- define global data using a different module, or modules.

The following is an example showing the structure of a complete program in which the interfaces of all the procedures are explicit to each other and are also explicit to the main program unit.

```
MODULE my_procedures
       IMPLICIT NONE
CONTAINS
       procedure_1
       procedure_2
       ......
END MODULE my_procedures

PROGRAM  prog_name
       USE  my_procedures
       IMPLICIT NONE
       ......
END PROGRAM prog_name
```

14.6 More about Procedures

(1) Association between Dummy & Actual Arguments

Variables as Dummy Arguments

- The matching actual argument may be a variable, an array element name, a constant, or an expression.

- If the subprogram assigns a new value to the dummy argument variable, the value of the corresponding actual argument will be changed. Thus, if the actual argument is a constant, an assignment to the corresponding dummy argument may change the value of the constant, which will cause severe difficulty later on. Therefore, a constant, or an expression that requires evaluation, must not be used as an actual argument corresponding to a dummy argument that is given a new value during subprogram execution.

Subprogram Names as Dummy Arguments

It is possible to pass the name of a procedure as an argument to another procedure. In this case a new form of declaration is required for the dummy and actual arguments

- The declaration of the type of the function dummy argument should include an EXTERNAL attribute:

```
REAL, EXTERNAL :: dummy_function_name
```

- The corresponding actual argument must also be declared in the calling program unit with either an EXTERNAL attribute or, if it is the name of an intrinsic function, an INTRINSIC attribute:

```
INTEGER, EXTERNAL ::  actual_external_function
REAL, INTRINSIC ::  sin
```

Arrays used in Procedures

- If an array is used as a dummy argument, the corresponding actual argument must be an array and the size of the dummy array should match that of the actual array.

- Any array used in procedures must be declared in the procedure. The size of an array in a procedure can be specified by using constant bounds as we used before. Alternatively, an array used in procedures (either dummy or local array) may be an explicit array whose bounds are integer expressions, the values of which can be determined at the time of entry to the procedure. Such bounds are usually determined by other dummy arguments or information from a module through the USE association.

  Example.

```
SUBROUTINE  example(a,b, lower, upper)
IMPLICIT NONE
INTEGER, INTENT(IN) :: lower, upper
REAL, DIMENSION(lower:upper), INTENT(IN) :: a, b  !dummy argument arrays
INTEGER, DIMENSION(lower:upper, 5) x              ! local array
```

where ***lower*** and ***upper*** are respectively the lower and upper bounds of the corresponding actual arrays.

- A dummy argument of a procedure **that has explicit interface** may be an assumed-shape array.

  An **assumed-shape array** is a dummy argument array whose shape is not known but which assumes the same shape as that of any actual argument that becomes associated with it. The DIMENSION attribute for an assumed-shape array takes the form

  DIMENSION(*assumed_shape_specifier_for_Dim1, assumed_shape_specifier_for_D2*,...

  where each *assumed_shape_specifier* specifies the lower index bound for one dimension of the array and take the form

  lower_bound :

  or if the lower_bound is omitted, it is taken to be 1.

  Example:

```
REAL FUNCTION  example(a,b)
IMPLICIT NONE
INTEGER, DIMENSION(:,: :: a
REAL, DIMENSION (5:,:,:) b    ! lower bound for dimension 1 is 5
```

- Some intrinsic functions

SIZE(array, Dim) - returns the extent of the array for the specified dimension.

LBOUND(array, Dim) - returns the lower bound of the array for the specified dimension.

UBOUND(array, Dim) - returns the upper bound of the array for the specified dimension.

(2) The SAVE Statement

Local variables are those used in a subprogram that are not arguments or global. The values of these local variables are generally lost on exit from a subprogram. A SAVE specification will, however, save the values of local variables, so they will contain the same values as they did at the end of the previous reference.

Examples of SAVE Specifications:

```
REAL, SAVE :: list of real variables to be saved
SAVE   list of variables to be saved
SAVE       ! saves all local objects in the procedure that could be saved.
```

(3) Internal Procedures

There are many cases where a procedure will only be invoked (referenced or called) by one particular program unit, and in this case it is permissible to include that procedure as an integral part of the program unit that will invoke it as an internal procedure.

An internal procedure is a form of subprograms, and must follow all the executable statements of its host program unit, and be separated from them by a CONTAINS statement. Thus if the subroutine *inner* is only used by the subroutine *outer* it may be written as an internal procedure of outer in the following way:

```
SUBROUTINE  outer(a,b,c)
          specification statements
          ......
          executable statements
          ......
CONTAINS
          SUBROUTINE inner(x,y,z)
          .......
          END SUBROUTINE inner
END SUBROUTINE outer
```

An internal procedure is the same as an external procedure with three exceptions:

- The name of an internal procedure is not global - the procedure may only be invoked by the host program unit.
- The name of an internal procedure may not be passed as an actual argument to another procedure
- An internal procedure has access to all the entities (such as variables and constants) of its host except for any which has the same name as a local entity of the internal procedure.

In addition to internal procedures, F95 also includes a much simpler facility called a statement function, which was the only form of internal procedure in F77. This has been totally superseded by the internal subprogram discussed above.

Example: For $f(x)=x^2+e^x+1$, calculate the values of $f(x)$ *at* x=1, 2, 3, 4 and their average.

```
PROGRAM  E7_6_1
       IMPLICIT NONE
       REAL :: F, AVERAGE, X
       F(x)=x**2+exp(x)+1
       AVERAGE=0.25 * ( F(1)+F(2)+F(3)+F(4) )
       PRINT *, F(1), F(2), F(3), F(4), AVERAGE
END   PROGRAM E7_6_1
```

14.7 Application: Solution of Linear Systems of Equations by Permuted LU Methods

The permuted *LU* Method is one of the popular direct methods for solving general linear systems of equations $Ax=b$. The solution process includes the following three basic steps

(a) Factor a permuted A into LU so that the system becomes $LUx=Pb$ where P is a permutation matrix ;
(b) Solve $Ly=Pb$ (by letting $Ux=y$) by a forward substitution process;
(c) Solve $Ux=y$ by a backward substitution process.

Based on the formulation presented in the reference by Wu and Wiwatanapatahee (1987), the following algorithms have been developed for the permuted *LU* factorization to determine *L*, *U* and *P*, and for the substitution process to determine *y* and then the solution *x*.

Algorithm for Permuted LU Factorization

This algorithm uses the Gaussian elimination process with scaled-column pivoting to find the permuted *LU* factorization of *A* (namely, to find *P* and the *LU* factorization of *PA*) where
U is the upper triangular matrix obtained from the elimination process;
L is the lower triangular matrix which is the collection of the multiples m_{ij}

Step 1 Set $s(i) = \underset{1 \le j \le n}{Max} |a_{ij}|$, (*determine the size of each equation*).

Step 2 For $k=1$ to $N-1$ do step 3 to step 6 (*set 1st, ..., $(n-1)$th column below diagonal to zero*)

Step 3 Find the (smaller) $P \ge k$ such that $\left|\dfrac{a_{pk}}{s_p}\right| = \max\limits_{k \le i \le n} \left|\dfrac{a_{ik}}{s_i}\right|$

(*select pivot element for the step*)

Step 4 If $a_{pk} = 0$ then
write '(IERR=1, *A* is singular)' then return.

Step 5 Else
$E_k \leftrightarrow E_p$, $P_k \leftrightarrow P_p$, (row interchange)
(P_k records the order in which the equations are to be processed)

Step 6 For $i = k+1$ to n (do usual Gauss elimination process for the *k*th step)
Set $m_{ik} = \dfrac{a_{ik}}{a_{kk}}$ $(\Rightarrow a(I,k))$
Set $a_{ij} = a_{ij} - m_{ik} a_{kj}$ $(j = k+1, \ ......, \ n)$

Step 7 If $a_{nn} = 0$, return "(IERR=1: *A* is singular)'
Return

Algorithm for Substitutions

For $i = 1$ to n do

$$y_i = b(p_i) - \sum_{j=1}^{i-1} a_{ij} y_j$$

Forward substitution (as $Pb = b$(pi))

For $i = n$ to 1 by -1 do

$$x_i = \frac{1}{a_{ii}} \left(y_i - \sum_{j=i+1}^{n} a_{ij} x_j \right)$$

Backward substitution

End

Example. Based on the above algorithm, write a well structured F95 program (with documentation) for solving a linear system of equations $Ax=b$, using the permuted LU factorisation method. Then use the program to solve the following linear system,

$$\begin{bmatrix} 1.19 & 2.11 & -100 & 1 \\ 14.2 & -0.122 & 12.2 & -1 \\ 0 & 100 & -99.9 & 1 \\ 15.3 & 0.11 & -13.1 & -1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 1.12 \\ 3.44 \\ 2.15 \\ 4.16 \end{Bmatrix}$$

Requirements:

1) Store real values with at least 15 digits of precision;
2) Read input data (A and b) from a pre-created data file and write the solution to another file
3) Use whole array operations whenever possible;
4) Use assumed-shape arrays for dummy arrays in procedures.

```
!
MODULE linear_eqs
  IMPLICIT NONE
  INTEGER, PARAMETER :: real K=SELECTED_REAL_KIND(P=15)
CONTAINS
!
Subroutine LUFACT (A,N,P,IERR)
!*********************************************************************
!  This subroutine can be called to find a permuted LU factorisation
!  of a N*N matrix A (scaled-column pivoting).
!
!  INPUT:  A  -  Before entry, set to matrix A,
!          N  -  Before entry, set to the number of equations
!
!  OUTPUT:  A :     On exit, A contains the L U factorisation (Lii=1).
!           P(N):  On exit, contains the permutation vector.
!                  P(i)=j: indicates that the ith equation of the
!                     new system is the jth eq in the original system
!           Ierr:  Return  1 : if A is singular
```

```
!                          0 :          nonsigular
!  OTHER PARAMETERS:
!           S(i) = size of the ith equation
  !*********************************************************************
*
!
   INTEGER, INTENT(IN)  :: N
   INTEGER, INTENT(OUT) :: P(:),IERR
   REAL(KIND=real k), INTENT(INOUT)  :: A(:,:)
   INTEGER :: PIVOT row, PIVOT(1), HOLD int, I, K
   REAL(KIND=real k) :: S(N), HOLD real, TEMP(N), TOL, mik
   TOL=1.0E-10
   IERR=0
! Step 1: determine the size of each eq and assign initial value to the
!         permutation vector  P
!
   DO  I=1,N
      S(I)=MAXVAL(ABS(A(I,:)))
      IF( S(I) <= TOL ) THEN    ! the system is singular
            IERR=1
            RETURN
      ENDIF
      P(I)=I
   ENDDO
!
! Step 2: Loop over elimination steps 1 to (n-1)
!
  DO  K=1, N-1

! Step 3: determine pivot row (PIVOT row)
!        (find row with largest absolute value of a(i,k)/s(i),i=k,...,n)
!
       PIVOT=MAXLOC(ABS( A(K:n,K)/S(K:n) ))
       PIVOT row=PIVOT(1)+K-1
!
! Step 4 : If A is singular, set IERR=1 then return
!
       IF( ABS( A(PIVOT row,K) ) <= TOL ) THEN
          IERR=1
          RETURN
       ENDIF
!
! Step 5 : Row interchange  (EpivoRT with Ek )
!
       IF(PIVOT row > K) THEN
         HOLD int=P(PIVOT row)       ! record the new order of equations
            P(PIVOT row)=P(K)        ! and the corresponding size
            P(K)=HOLD int
         HOLD real=S(PIVOT row)
            S(PIVOT row)=S(K)
            S(K)=HOLD real
         TEMP=A(PIVOT row,1:N)         ! interchange the rows of A
            A(PIVOT row,1:N)=A(K,1:N)
            A(K,1:N)=TEMP
       ENDIF
! Step 6  : Guassian elimination process (step k)
!
       DO I=K+1,N
            mik=A(I,K)/A(K,K)
```

```
            A(I,K)=mik
            A(I,K+1:N)=A(I,K+1:N)-mik*A(K,K+1:N)
       ENDDO
   ENDDO
!
! Step 7 :  If A(n,n)=0, A singular and thus set IERR=0
!
   IF( ABS(A(N,N)) <= TOL) IERR=1
   RETURN
   !
END SUBROUTINE LUFACT
!
!
SUBROUTINE SUBST(A,N,P,B,X)
! ---------------------------------------------------------------
!
! INPUT:    A,N,P  -         OUTPUT FROM LUFACT,
!           B -              RHS OF THE EQ
! OUTPUT:   X -              SOLUTION OF AX=B
!
!----------------------------------------------------------------
   REAL(KIND=real k), INTENT(IN) :: A(:,:), B(:)
   REAL(KIND=real k), INTENT(OUT) :: X(:)
   INTEGER, INTENT(IN) :: N, P(:)
   INTEGER :: NOE, I, J
   REAL(KIND=real k) :: SUM

! Step 1: Forward substitution
!
   NOE = P(1)
   X(1) = B(NOE)
   DO I=2,N
       SUM =0.0
         DO J=1,I-1
            SUM = SUM + A(I,J)*X(J)
         ENDDO
         NOE = P(I)
         X(I)=B(NOE)-SUM
      ENDDO
!
! Step 2 : Backward substitution
!
      X(N) = X(N)/A(N,N)
      DO I=N-1,1,-1
         SUM=0.0
         DO J=I+1,N
            SUM =SUM+A(I,J)*X(J)
         ENDDO
         X(I) = (X(I)-SUM)/A(I,I)
      ENDDO
      RETURN
END SUBROUTINE SUBST
!
END MODULE linear eqs
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
PROGRAM ASS2P1Q4
   !
   ! This  program  defines  a  linear system of equations Ax=b, then
```

```
    ! solve the system using the module procedures LUFACT and SUBST
    !
  USE linear eqs
  IMPLICIT NONE
  INTEGER :: N, P(20), ERR, I,J
  REAL(KIND=real k) :: A(20,20),B(20),X(20)
! define the linear system
! read values from a pre-created file
  Open(unit=10,file='ass2q4 09.in')
  Open(unit=15,file='ass2q4 09.out')
  READ(10,*)N                          ! enter number of eqs
  print *, N
  READ(10,*)((A(I,J),J=1,N),I=1,N)  ! enter matrix A row by row
  READ(10,*)(B(I),I=1,N)             ! enter the RHS b
  Print*,((A(i,j),j=1,n),i=1,n)
  print *,(b(i),i=1,n)
  CALL LUFACT(A,N,P,ERR)             ! call LUFACT to find the LU
                                     !      factorization of A
  SELECT CASE(ERR)
  CASE(0)
  CALL SUBST(A,N,P,B,X)              ! call SUBST to perform the
                                     !      substitution process

     WRITE(15,100)
      100 FORMAT(1X,'SOLUTION IS')
     WRITE(15,101)  (I, X(I),I=1,N)
      101  FORMAT((1X,'X(',I2,')=',F9.5))
   CASE(1)
     WRITE(15,102)
        102 FORMAT(1X,'A IS SINGULAR')
   END SELECT
END PROGRAM ASS2P1Q4
```

!
Input data stored in the pre-created data file 'ass2q4 09.in'

```
4
1.19,   2.11,    -100,    1,
14.2,   -0.122,  12.2,   -1,
0,       100,     -99.9,   1
15.3,   0.11,    -13.1,  -1,
1.12,   3.44,    2.15,   4.16
```

!
Output

```
The solution obtained is
   X(1)=  0.17683
   X(2)=  0.01269
   X(3)= -0.02065
   X(4)= -1.18261
```

Exercise 14

Q14.1 Why should programs be broken into a main program and a set of procedures ?

Q14.2 What is the difference between a subroutine and a function ?

Q14.3 What is an intrinsic function, external function ?

Q14.4 What is the difference between a dummy argument which is declared with an INTENT(INOUT) attribute and one which is declared as INTENT(OUT)?

Q14.5 What is the purpose of a module ?

Q14.6 What does USE association do ?

Q14.7 Read and then show the output from each of the following programs. If you are not sure whether your answer is correct or not, run the program in computer to check the answer.

(a)

```
PROGRAM Q14_7a
    IMPLICIT NONE
    INTEGER :: F, G, X, A
    F (X)=A*X**2+1
    G(X)=X+2
    A=2
    X=F (2)
    PRINT*,‘X=‘,X,’G(5)=‘,G( G(5) )
END PROGRAM
```

(b)

```
PROGRAM Q14_7b
   IMPLICIT NONE
   INTEGER :: I
   REAL :: AVERAGE
   REAL, DIMENSION(20) :: K
   DO  I=1,5
       K(I)=I
   END DO
   PRINT *, ‘mean=‘,AVERAGE(5, K)
END PROGRAM Q14_7b

REAL FUNCTION AVERAGE(N, X)
  IMPLICIT NONE
    INTEGER, INTENT(IN) :: N
    REAL,DIMENSION(20),INTENT(IN) ::X
    AVERAGE=SUM(X(1:N))/N
END FUNCTION AVERAGE
```

```
(c) MODULE  global_d1
        IMPLICIT NONE
        INTEGER, PARAMETER :: N=5
        INTEGER, PARAMETER :: R_K=SELECTED_REAL_KIND(P=12)
        REAL(KIND=R_K), DIMENSION(N,N) ::X
    END MODULE global_d1
    PROGRAM Q14_7c
        USE global_d1
        IMPLICIT NONE
        INTEGER :: I, J
        REAL :: MAX, MIN
        DO I=1,2
            DO  J=1,2
                X(I,J)=I+2*J
            END DO
        END DO
        CALL  MAXMIN(MAX, MIN)
        PRINT *, 'max=', MAX, 'min=', MIN
    END PROGRAM Q14_7c

    SUBROUTINE  MAXMIN(MAX,MIN)
        USE global_D1
        IMPLICIT NONE
        REAL, INTENT(OUT) :: MAX, MIN
        MAX=MAXVAL(X(1:2,1))
        MIN=MINVAL(X(1,1:2))
    END SUBROUTINE  MAXMIN
(d)   MODULE  my_procedures
          IMPLICIT NONE
      CONTAINS
          SUBROUTINE      sub1(N,X, MEAN_R1, MEAN_R2)
              IMPLICIT NONE
              INTEGER, INTENT(IN) :: N
              REAL, DIMENSION (: , : ), INTENT(IN) :: X
              REAL, INTENT(OUT) :: MEAN_R1, MEAN_R2
              MEAN_R1=SUM(X(1, 1:N))/N
              MEAN_R2=SUM(X(2, 1:N))/N
          END SUBROUTINE sub1

          REAL FUNCTION average(N,X)
              IMPLICIT NONE
              INTEGER, INTENT(IN) :: N
              REAL, DIMENSION(: , :), INTENT(IN) :: X
              average=SUM(X(1:N, 1:N))/(N*N)
          END FUNCTION average
      END MODULE my_procedures

      PROGRAM Q14_7d
          USE    my_procedures
          IMPLICIT NONE
          INTEGER ::  I, J
          REAL :: MEAN_R1, MEAN_R2
          REAL, DIMENSION(10,10) ::  X(1:2, 1:2) = (/ 1.0, 2.0, 3.0,4.0/)
          PRINT *, 'average=', average(2,X)
          CALL SUB1(2,X, MEAN_R1, MEAN_R2)
          PRINT *, 'mean_r1=', MEAN_R1, 'mean_R2=', MEAN_R2
      END PROGRAM Q14_7d
```

Ans: (a) x=9, G(5)=9
(b) mean=3.0
(c) max=4, min=3
(d) average=2.5, mean_R1=2, Mean_R2=3

PROGRAMMING

Always Plan Ahead

To write a program, it is essential to first draw up a program design plan (flow chart or pseudocodes) which shows the structure of the program and the various levels of detail.

Q14.8 Write a function subprogram to compute the value of the function defined by

$$f(x) = \begin{cases} 0 & x < -10 \\ 2x + 20 & -10 \le x < 0 \\ 20 & 0 \le x < 20 \\ 30 - 0.5x & 20 \le x < 60 \\ 0 & x \ge 60 \end{cases}$$

Test your subprogram by calling it in a main program using x = -5, 10, 40, 100. Print the results using formatted output: x = ****.** f(x) = **.**

Q14.9 Write a function subprogram MYEXP to compute e^x using the following series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots.. = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Continue using terms until the absolute value of a term is less than 1.0E-8. Test your function subprogram by calling it in the main program using x = 1, 2 respectively. Use at least 9 digits of precision for the function and variables. Print the results using formatted output. x = *.**, MYEXP(X) = ***.********

Q14.10 Write a program which consists of a main program and two subroutines INPUT_dat and OUTPUT_dat. The main program calls INPUT_dat to read an integer number N (assume N<100) and then two square matrices(N rows by N columns), then calls OUTPUT_dat to print the value N and the matrices row by row using format output. Test your program using

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 2 & 1 \end{bmatrix} .$$

(a) declare arrays used in procedures using constant bounds -see Q14_7b or Q14_7c.
(b) declare arrays used in procedures using assumed-shape arrays - see Q14_7d .

Q14.11The following is an algorithm for finding a root of the equation x=g(x) where g is a function of *x.*

Input x_o, Tol and Max_iter.
Do for I=1 to Max_iter
    Set $x=g(x_o)$
    If ($|x-x_0|<Tol$) then
        output x; (& 'procedure completed successfully')
        stop
    else
        set $x_0=x$
Output ('method failed after Max_iter iterations, Max_iter=', Max_iter)

Using the above algorithm, write a complete F95 program to find a solution accurate to within 10^{-5} for the equation

$$x = \frac{1}{3}\left(x^2 + 2 - e^x\right)$$

Hint. Choose x_o=0.5, *Tol*=10^{-5}, *Max_iter*=100. **Ans**: $x \approx 0.25753$).

Program design:

(1) Define g(x) using a function subprogram;
(2) Find the root of $x=g(x)$ using a subroutine with header

SUBROUTINE ITER (G, X0, TOL, Max_iter, Err, X)

Where G: Input. Suppose the eq to be solved is x=$g(x)$, then before entry G must be specified as the function g. (see subprogram names as dummy arguments)

X0, TOL, Max_iter: input to the subroutine

X: output from the subroutine: solution of the equation x=g(x)

Err : returns a value 0 if the procedure completed successfully
or returns a value 1 if the procedure failed after Max_iter iterations.

(3) Design a main program which reads X0, Tol and Max_iter
calls ITER to find a root, and
prints the result or otherwise an error message.

Chapter 15

F95 Pointers and Dynamic Memory Allocation

In some applications, it is convenient to have a pointer to a variable or an array so as to access the variable/array indirectly. F95 provides this capability by having a new class of variables, namely pointer variables.

The use of pointers provides several benefits, of which the most important in scientific computing is the ability to provide a more flexible and powerful alternative to allocatable arrays. In this chapter, we will demonstrate how to use F95 pointers in program design.

15.1 Basic Concepts

Unlike real and integer variables, a pointer variable does not contain any data, instead it points to a target (a scalar variable or an array) where the data is stored.

Declaration of Pointer Variable

A variable to be used as a pointer must be declared by a type declaration statement with a pointer attribute as follows:

```
type, attribute list, pointers :: list of variable
```

where the ***type*** specifies what type (real, integer, ...) of objects can be pointed to; the ***attribute list*** gives the other attributes (such as dimension etc).

eg.

```
Integer, pointer:: q
```

specifies that the variable ***q*** is a pointer that can point to objects of integer type. It does not specify which integer entity that ***q*** points to.

Declaration of Target Variables

A pointer can be made to point to a target variable. To ensure the efficiency of Fortran codes, any variable to be associated with a pointer must be declared by a type declaration with a **target** attribute, eg.

```
Integer :: n
Integer, target :: x
Integer, Pointer :: p
```

Remarks: The variable p can be set to point to the variable x because the data type match (both are integer) and x has the target attribute. However, p cannot be made to point to n as n does not have the target attribute.

Association between pointers and target variables

When a pointer variable is created, its association status is undefined. A pointer can be associated with a target (a variable/array with a target attribute) by a pointer assignment statement of the form

```
Pointer => target_variable
```

Once a pointer points to a target, its association status is said to be **associated**.

eg 1 (pointer assignment when the target is a variable with a target attribute).

```
Real, pointer :: p1, p2, q
Real, target :: x1, x2
P1 => x1     ! p1 points to x1
P2 => x1     ! p2 also points to x1
P1 +> x2     ! now p1 points to x2 while p2 still points to x1
```

Remarks: In the above, the targets all have the target attribute. However, it is also permitted for pointer assignment to take place between two pointers.

eg 2 (pointer assignment when the target is a pointer)

```
Integer, pointer :: p1, p2, p3
Integer, target :: y
...
P1 => y      ! P1 points to y
P2 => P1     ! P2 now points to y
P1 => P3     ! p1 now has undefined association status
```

Remarks:

(i) The pointer association status of p1, p2 and p3 are initially undefined.

(ii) By P1=>y, the pointer P1 becomes associated with y and its association status becomes defined;

(iii) The statement P2=>P1 does not set P2 to point to P1. Instead, the effect is to set P2 to point to the same target that P1 points to, and hence P2 also points to y.

(iv) In the final statement P1=>P3, the target P3 is a pointer whose association status is undefined, and hence the association status of the pointer P1 also becomes undefined.

Breaking Association between pointers and target variables

In some situations, it is necessary to break the association between certain pointers with their targets. The Nullity statement

```
Nullity (list of pointers)
```

can break the association between the listed pointers and their targets and set the pointer association status to disassociated.

eg

```
integer, target :: n1, n2
integer, pointer :: p1, p2
...
P1 => n1          ! p1 points to n1
P2 => n1          ! p2 points to n1
Nullity (P1)      ! P1 is disassociated while P2 still points to n1
...
P1 => n2          ! P1 now points to n2
...
Nullity (P1, P2)  ! both P1 and P2 are disassociated
```

Checking Pointer Association

In some applications, it is important to know the pointer association status at certain points of the program execution process. This can be done by using the Associated statement.

Eg.

```
Real, pointer :: p, q
Real, target :: x, y
Logical :: Associate1, Associate2
...
P => x
...
Associate1=Associated(P)   ! Associate1 will be assigned 'true'
Associate2=Associated(q)     ! Associate 2 will be assigned
'false'
Print *, Associated (p,x)    ! will print 'true'
Print *, Associated (p,y)    ! will print 'false'
```

Remarks:

(1) Associated (p) returns 'true' if the pointer p is currently associated with a target or otherwise 'false' if it is not.

(2) Associated (p, x) returns 'true' if the pointer p is currently associated with the target x or otherwise 'false'. Note that x must be a variable with a target attribute and p must be not undefined.

15. 2 Using pointers in expressions

When a pointer appears in an expression where a value is expressed, it is treated as if it were the associated object and the value of the object it points to will be used.

Eg 1 (An example of arithmetic expression that involves a pointer)

```
Integer pointer::p,q
Integer, target::x=1,y=2
p=>x                ! p points to x
q=>y                ! q points to y
p=q+1               ! assignment to x(equivalent to x=y+1),so x becomes 3,
                    !     and p still points to x
if(p-1==q) p=>y     ! condition test ( x-1==y), True, and so p now points to y
p=q+1               ! assignment to y (i.e. y=y+1 and so y becomes 3)
```

Remarks:

(i) The first two statements associate p with x and q with y respectively;

(ii) The statement p=q+1 expects a variable name on the left of the assignment operator and an expression on the right. As in q+1, the + operator expects q to have a value, the pointer q is dereferenced and thus the expression becomes y+1 yielding 3. The pointer p on the left side of the assignment operator is also dereferenced to the variable x. Hence, the statement p=q+1 has the effect of x=y+1 which sets the value of x to 3.

(iii) In the statement if(p-1==q) p=>y, both p and q are dereferenced to x and y first and thus the statement is to test whether x-1 is equal to y. As this is true, p=>y is executed resulting in that p also points to y.

(iv) The final statement is equivalent to y=y+1 which sets y to a new value 3.

It should be addressed here that the pointer on the left side of a pointer assignment statement plays a very different role from the pointers in a value-demanding situation, as shown by the following example.

```
Integer pointer:: p,q
```

```
Integer, target::x=10,y=20
p=>x         ! p points to x
q=>y         ! q points to y
p=q          ! equivalent to x=y(set x to the value of y),
             !  & p is unchanged (still points to x).
p=>q         !p now points to y
```

Remarks:

(i) p=q (equivalent to x=y) sets x to have the value 20 and leaves the p unchanged;

(ii) p=>q sets p to point to y and leaves the value of x unchanged.

We now give an example to demonstrate how pointers can be used to improve the program efficiency. Suppose we wish to interchange two character strings of huge size. This conventionally can be implemented by the following statements

```
Character (LEN=1000):: Cstring_1, Cstring_2, Temp
...
Temp=Cstring_1
Cstring_1=Cstring_2
Cstring_2=Temp
```

Remarks: The above program involves three copies of large amount of data and also requires the extra storage space for Temp.

Using pointers, we can achieve the same goal by the following program

```
Character (LEN=1000), target :: Cstring_1, Cstring_2
Character (LEN=1000), pointer :: P1, P2
P1=>Cstring_1          ! P1 points to Cstring_1
P2=>Cstring_2          ! P2 points to Cstring_2
! now work with P1 and P2 instead of Cstring_1 and Cstring_2
...
...
! Interchange pointers so that P1 points to Cstring_2 and P2 points to Cstring_1
P1=>Cstring_2
P2=>Cstring_1
```

Remarks: In the above program with pointers, only two pointers are reset and no large data are copied.

15.3 Pointers and Arrays

One of the most powerful applications of pointers is the use of pointer arrays as a means of dynamically creating memory space for an array when needed, and releasing it when it is no longer needed.

The dimension attribute of a pointer array must take the form of a deferred-shape array in a manner similar to that for an allocatable array. Although pointer arrays are similar to allocatable arrays, they have more capabilities as to be demonstrated in 15.4.

Declaration of pointer arrays

A pointer array can be created by the type declaration of arrays with a pointer attribute.

eg.

```
Real, Dimension(:,:), pointer::x,y
```

declares two pointer arrays, x and y, which can only point to two dimensional real arrays.

Remarks: Similar to the allocatable arrays, the extent of each dimension of the array must be specified by a colon. The total number of colons is equal to the rank of the array.

Association of pointer arrays

Point arrays may be associated with any arrays that have matching type, type parameters (such as kind type) and rank, and also have the target attribute. The extents (index bounds) of the arrays are not matter.

In the following, we give an example

```
Integer, dimension (10,10), target::  a1
Integer, dimension (5,5), target :: a2
Character (LEN=10), Dimension (100), target ::c1
Character(LEN=5), Dimension (100), target :: c2
Integer, Dimension (: , :), Pointer :: p
Character(LEN=10), Dimension (:), Pointer :: q
...
p=>a1        ! Associate p with array a1
p=>a2        ! Associate p with array a2
q=>c1        ! Associate q with array c1
```

It can be observed from the above that

(i) **p** is associated with arrays of different extents at different times, which is allowed as **p** can point to any two-dimension default-integer array.

(ii) **q** is not allowed to point to array **c2** as the type parameter (the character length attribute) does not match although their type and rank are the same.

Remarks: Once an pointer association has been created, the pointer array can be used in place of the target array in an expression as for scalar variables.

Allocating space to pointer arrays

One of the most powerful applications of pointer arrays is the use as a means of dynamically creating memory space for an array when needed, and releasing it when it is no longer needed.
Space can be allocated to a pointer array by a allocate statement of the form

```
Allocate (pointer(dimension specification), STAT=status)
```

where

pointer : is a pointer array having both the dimension and pointer attributes;

dimension specification: is the specification of the extents for each dimension

status: is an integer variable which will be assigned zero if the allocation of space is successful or otherwise a positive number if there is an error.

Remarks: The allocate statement will create an un-name array of the specified type, size and rank, which can be referenced by means of the pointer array.

eg.

```
Integer :: error, m, n
Real,  Dimension(:,:), pointer::A,B
Read(*,*) n,m
Allocate( A(n,n), STAT=error)          ! allocate A
If (error /=0) Then
   Print *, "Allocation Error Occurs"
   Stop
End if
B =>A                                  ! B points to A
...
Allocate( A(m,m), STAT=error)          ! allocate A again
If (error /=0) Then
   Print *, "Allocation Error Occurs"
   Stop
End if
...
```

Remarks

(i) The pointer array **A** is first set to point to a dynamically created un-name real array of size **n** by **n**, and the pointer array **B** is also set to point to the same array;

(ii) Then **A** is set to point to a dynamically created new array of size **m** by **m**, and the association of **A** with the first array (**n** by **n**) is broken. The point array remains pointing to the first **n** by **n** array.

(iii) If the statement **B=>A** were removed, then the space for the first array would become inaccessible to the program. A second execution of the original Allocate statement will create another array instead of associating A with the first array.

Releasing space for arrays created by pointer allocate statement

The space for an array created by a pointer allocate statement can be released by the deallocate statement of the form

```
Deallocate (pointer)     or     Deallocate (pointer, Stat=status)
```

Eg

```
Program Test
Implicit None
Integer :: i, j, n, error
Real, dimension (:, :), Allocatable :: x
Real, dimension (:, :), Pointer :: y
...
Read (*,*)  n
!  Allocate space for the allocatable array x
Allocate (x(n,n), STAT=error)
If (error /= 0) Then
     Print *, " Error in allocating space  for x"
     Stop
End If
Read (*,*)  ((x(i,j), j=1,n), i=1,n)
!
! Allocate space for the point array y
Allocate (y(n,n), STAT=error)
If (error /= 0) Then
     Print *, " Error in allocating space  for y"
     Stop
End If
!
!   Set y to the transpose of x
            Do i=1, n
            Do j=1, n
```

```
      y(j,i)=x(i,j)
End Do
End Do
! Other calculation using y
...
! Deallocate x and y
Deallocate (x, y, STAT=error)
If (error /= 0) Then
    Print *, "error in deallocating space for x and y"
    Stop
End If
! other calculations
...
End Program Test
```

Remarks:

The above program uses an allocatable array **x** to store data of a two-dimension array. Then a pointer array **y** is created and set to the transpose of **x**. When all computations are completed, the space for **x** and **y** is deallocated. The point associate status of y becomes disassociated.

It should also be addressed that a pointer deallocate statement must not be used to deallocate any object or arrays that was not allocated by a pointer allocate statement.

15.4 Pointer Arrays as Argument to Procedures

Allocatable arrays cannot be used as dummy argument of procedures, but pointer arrays can be used as dummy argument. However, the following conditions must be met.

- If a dummy argument is a pointer array, the corresponding actual argument must be a pointer array with the same type and rank.
- The called procedure must have an explicit interface with the calling program unit.

In the example presented previously, the allocation and deallocation of space for pointer arrays have always occurs in the same program unit. This is not necessary. In F95, allocation and deallocation of space can be executed in different program units.

In the following example, a set of data (X) is passed from the main program to a subroutine destroy via a pointer array.

```
Module my_Procedure
      Implicit None
Contains
      Subroutine destroy(N, X, Xmean)
      Implicit None
      Real::Xmean
      Integer::N
      Real, Dimension(:), pointer::X   ! Dummy argument is a pointer array
      Xmean=sum(X(1:N))/N
      Deallocate(X)
End Module

Program Creater
      Implicit  None
      Use my_procedure       !Create explicit interface with subroutine destroy
      Real::Ymean
      Integer::N
      Real, Dimension(:), pointer::Y
      Read(*,*) N
      Allocate(Y(N))             !allocate space to pointer array Y
      Read(*,*) Y(1:N)
      Call destroy(N,Y,Ymean) !Pass data to program via pointer array Y
      Print *, "Ymean =", Ymean
End Program
```

Remarks: Allocation and deallocation of space to pointer arrays can be done in different program units. In the above example, the space for the elements of the actual pointer array Y is allocated in the main program. Through the calling statement, the dummy pointer array X is associated with Y. After using X in the called procedure, the subroutine deallocates it. This also deallocates the actual argument Y.

EXERCISE 15

Q15.1 What are the restrictions on pointers and targets being procedure dummy arguments?

Q15.2 Write a function that has a rank one, real pointer array as a dummy argument. The function should calculate the maximum value in the array, return it as the function value and deallocate the space for the pointer array. Then write a main program to read a data value N, allocate space to an one-dimensional pointer array and read N data value, and then call the function to calculate the maximum value of the N data values.

Q15.3 Rewrite the program for Q14.10 using pointer arrays
(1) Data are passed to procedure by a pointer array
(2) Use the allocate statement with the integer N to specify the size of the matrix.

Part III MATLAB

MATLAB is a high level software package with many built-in functions for mathematical calculation and graphic display of results. In this part, we will introduce some basic operations that will enable you to learn the software and develop your MATLAB programs for solving scientific computing problems.

CHAPTER 16

MATLAB COMPUTATION AND GRAPHICS

16.1 MATLAB GETTING START

To start the MATLAB program in PC, double-click the MATLAB icon. A command window will open with the MATLAB prompt >>.

To solve a problem by MATLAB, you can work directly in the command window by entering the MATLAB commands. Alternatively, you may create a M-file, enter all MATLAB commands in the file via the edit window and save the file, and then run the M-file on the command window to solve the problem. The following 2 sections demonstrate how to solve a simple problem in these two modes.

16.2.1 Work on Command Window

The MATLAB command window is the main window where you type commands directly to perform certain tasks.

Example 1.1 Enter $A = \begin{pmatrix} 1 & 2 \\ 2 & 5 \end{pmatrix}$ and $B = \begin{pmatrix} 2 & 5 \\ 3 & 1 \end{pmatrix}$. Then calculate $C = A + B$.

In the command window at the MATLAB prompt >>, we enter

```
>> A=[1,2;2,5];
>> B=[2,5;3,1];
>> C=A+B
```

which results in

```
C =
        3   7
        5   6
```

16.1.2 Work on Edit Window

The MATLAB editor window is a text editor where you can load, edit, save and also run your MATLAB program (sequences of MATLAB commands).

To open an editor window, choose in the command window

`File -> New -> M-file` for creating a new M-file

`File -> Open -> M-file` for opening an existing M-file

eg. for the example, we create an M-file ex16_1.m and enter the following commands

```
A=[1,2;2,5];
B=[2,5;3,1];
C=A+B
```

and then save the file ex16_1.m.

Once a M-file is created, it can be run to perform the specific task through the editor window or the command window.

(a) To run a M-file in the editor window, simply choose the Debug/Run in the command menu.

(b) To run a M-file in the command window, simply type in the file name in the command line and press enter
eg.

```
>> ex16_1.m
```

will run the ex16_1.m and yield the following result

```
C =
     3   7
     5   6
```

Notes:

If the path of the file you want to run is not listed in the MATLAB search path, you need to add the path to the MATLAB-path list by clicking the menu 'File->Set->Path', clicking the 'Add-Folder' button, browsing/choosing the folder name and finally clicking the save button.

16.1.3 Use of MATLAB Help Window

The MATLAB Help Window gives you access to a great deal of useful information about the MATLAB language and MATLAB computing environment. It also has a number of example programs and tutorials. In addition, the "lookfor" command can help to find relevant functions for your job. The "help" command helps you to know how to use a particular command.

eg

```
>> lookfor  repeat
```

or

```
>> help for
```

16.2 MATLAB ARITHMETIC COMPUTATIONS

Arithmetic operations ($+, -, \times, \div$) are the most fundamental operations performed by computers. To be able to write programs for these operations, we need to know how to store data values in computers, how to implement computations, how to input data values to computers and how to print the computed results. Thus in this chapter, we describe

- Methods for storing data with MATLAB by using constants and variables
- Assignment statements for arithmetic computations
- Simple input/output statements for introducing data values into computers or printing results

.

16.2.1 Constants and Variables

Data are stored in MATLAB by the use of constants and variables.

Constants

- In MATLAB, integer and real numbers are represented by one of the following 2 forms

  Decimal form: -12.3, 0.0, 5 etc.

  Scientific form: -5e8, 0.82e-11 etc (-5×10^{8} and 0.82×10^{11})

- Imaginary numbers use either i or j as a suffix : -3.14i, 3e5i etc. Thus, a complex number 3.5+2i can be written as 3.5+2i or 3.5+2j.

- All numbers are stored internally using long format by the IEEE floating point standard with a finite precision of about 16 significant digits and a finite value range $10^{-308} - 10^{+308}$.

Variables

In MATLAB, a variable represents an $n \times m$ rectangular matrix (array). Each element of the array can store one data value and thus a MATLAB variable can store a group of data.

- Unlike in C++ and F95, MATLAB does not require declaration on the data type and dimension of the variable. When a new variable name is encountered, MATLAB automatically creates the variable with proper data type and an appropriate amount of storage. If the variable already exits, its contents are changed and new storage locations are allocated.

eg. `number_of_values = 60` creates a 1-by-1 matrix named `number_of_values` and stores the value 60 in the single element.

`x_values = [10,20,26]` creates a 1-by-3 matrix with 3 elements respectively storing the value 10, 20 and 26.

- A MATLAB variable name begins with a letter and can optionally followed any number of letters, digits and underscores. MATLAB use only the first N characters of the name. Thus, it is necessary to make each variable name unique in the 1st N characters. To find out the value of N for a particular machine, use the following function

```
N = namelengthmax
N=
     63
```

- MATLAB is case sensitive.

16.2.2 Input and Output of Data

I/O of data from MATLAB command window

In MATLAB, we can deal with vectors and matrices in the same way as scalars.
To input the matrix A and vectors B and C defined below

$$A = \begin{bmatrix} 1 & 2 \\ 6 & 2 \\ 5 & 8 \end{bmatrix}, \quad B = \begin{bmatrix} -1 & 3 & 5 \end{bmatrix}, \quad C = \begin{bmatrix} 1.5 \\ 5.2 \\ 2.6 \end{bmatrix}$$

we type in the MATLAB command window:

```
>>  A = [1 2; 6 2; 5 8]
```

By pressing enter, the above gives

```
A =
    1      2
    6      2
    5      8
```

Similarly, B and C can be defined by

```
>> B = [-1   3   5];

>> C = [1.5;   5.2;   2.6];
```

Remarks:

- Data values for elements of matrices are separated by some spaces.
- At the end of each row, need to put a semicolon to indicate the end of the row
- At the end of a statement, press <enter> to check the result of executing the statement, or otherwise add a semicolon ";" before pressing enter to indicate that the results are not to be shown.
- The format of output can be controlled by using a "format" command.

  eg.

```
>> x=1/3
x =
   0.3333
>> format long
>> x
x =
   0.33333333333333
>> format long e
>> x
x =
    3.333333333333333e-001
```

Input and Output of data from/to files

MATLAB can handle two types of files.

- Binary format mat_files "***.mat" can preserve the values of more than one variable, but cannot be shared with other programming environment except for the MATLAB environment.
- ASCII data files "***.dat" can be shared with other programming environments but preserve the values of only one variable.

I/O of data from/to mat files

```
>> save xyz x y z       % store the values of x, y, z into the file "xyz.mat"
>> load xyz x y         % read the values of x, y, z from the file "xyz.mat"
```

I/O of data from/to ASCII files

- To store data into an ASCII dat-file (in the current directory), make the file name the same as the name of the variable storing the data and add '/ascii' at the end of the same statement, namely

```
>> save x.dat x /ascii
```

Remarks

Only the value of one variable can be saved.
Non-numeric data cannot be handled by using a dat-file.

- To read the data from the dat-file in MATLAB, just type the (lower case) filename ***.dat after load, namely

```
>> load x.dat
```

I/O data from keyboard

The 'fprintf()' function (fprintf= "file print formatted") can be used to control the format of printout. The fprintf function uses one argument to give formatting instructions followed by a list of values to be printed, as shown below.

```
fprintf('formatstring', variables_to_be_printed)
```

The format string controls where and how the values in variables are to the printed. The following is some format strings

`%wg` : set a total width of w for printing the general real number;
`%w.nf` : set floating point format-total width w with n digits for fractional part;
`%ws` : set total width of w for character string output.

eg.

`fprintf('%5g',10)`		□□□10
`fprintf('%10.4f',523.456)`	⇒	□□523.4560
`fprintf('%10s', 'unix')`		□□□□□□unix

We can input a value or a string from the command line with the input() function.

```
yval=input('Enter a number: ');
name=input('Enter your name: ', 's');
```

eg.

```
>> yval=input('Enter a number: ');

Enter a number: 2

>> yval

yval =  2

>> name=input('Enter your name: ', 's');

Enter your name: SIAM
```

```
>> name

name = SIAM
```

16.2.3 Assignment Statement

Assignment statements are used to perform arithmetic computations and then assign the computed results to variables. The general form of an assignment statement is

```
variable_name = expression
```

Once an assignment statement is executed, the following two processes occur.

(1) Firstly, calculate the value of the expression

(2) then assign the value to the variable on the left hand side.

eg. `x = 2.0` assign 2.0 to x
`x=x+2.6` evaluate x+2.6 to yield 4.6, then the value 4.6 is assigned to x.

Notes: To understand how to correctly perform arithmetic computing by using assignment statements, we need to know how to translate mathematical formulae to arithmetic expressions and how to evaluate an expression.

Writing Arithmetic Expression

An arithmetic expression is a combination of constants, variables, intrinsic functions, operators and parentheses which can be evaluated to give a single value.

Intrinsic Functions

- MATLAB provides a large number of standard elementary mathematical functions such as abs(x), sqrt(x), exp(x) and sin(x). Most of these functions accept complex arguments. To get a list of the elementary functions, type

```
>> help elfun
```

- MATLAB also provides many advanced mathematical and matrix functions such as Bessel and Gamma functions. To get a list of advanced functions, type

```
>> help specfun

>> help elemat
```

- Elementary functions like `sqrt` and `sin` are part of the MATLAB core, while other functions like Bessel are implemented in M-files.

Operators

The basic arithmetic calculation can be performed by using the following operators in MATLAB.

Operations	Operators
Addition	+
Subtraction	−
Multiplication	*
Division	/
Power	^
Complex conjugate transpose	'

To evaluate arithmetic expressions, MATLAB assigns the same priorities to operators as does mathematics.

16.3 MATLAB CONTROL STRUCTURES

16.3.1 Logical Expressions

Logical expressions are used to describe mathematical conditions. In general, a logical expression consists of relational expressions, logical constants, logical variables and logical operators.

Relational expression

- A relational expressions compare the values of two arithmetic expressions using a relational operator, namely

```
Arithmetic_expression1 relational_operator arithmetic_expression2
```

eg.

```
t > 2+x
```

List of relational operators

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal (check equality for two scalars.
~=	Not equal

`isequal(A,B)`	Yields 1(true) if matrix A equals matrix B
`isempty(A)`	Yields 1(true) if matrix A is empty.

- A relational expression can be used to describe simple conditions such as a>b+1. If the condition is true, the expression yields a value 1 (true) or otherwise 0(false).

Logical operators and logical expressions

By combining the relational expressions together using logical operators (`&,|,~`), we can form a logic condition to describe a complicated condition.

- **Definition of logical operators**

Name	**Operators**	**MATLAB codes**	**Values**
AND	`&`	`a&b`	1 (true) if both `a` and `b` are true.
OR	`|`	`a|b`	1 (true) if at least one of the `a,b` is true.
Not	`~`	`~a`	1 (true) if `a` is 0 (false) or vice versa.

- **Priority of logical operations**

Type	**Operator**	**Execution order**
Bracket	`()`	1
Arithmetic calculation	`+,-,*,/,^,'`	2

Relational calculation	`<,<=,>,>=`
	`==,~=,`
	`isequal(A,B)`
	`isempty(A)`
		3

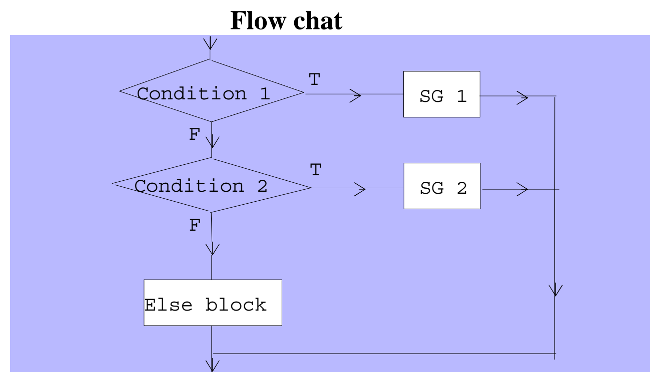
Logical calculation	`~`	4
	`&`	5
	`|`	6

16.3.2 Selection Control

In practice, most problem require us to choose between alternative courses of action, depending upon circumstances that are not determined until the program is executed. MATLAB provides two selection structures for choosing alternative courses of action including the if-elseif-end structure and the switch-case structure.

if-elseif-end structure

Flow chat



MATLAB code:

```
f   condition_1
      block_1
elseif  condition_2
      block_2
:
else
    else_block
end
```

eg.

```
A=input('Enter the first number: ');
B=input('Enter the second number: ');

if A > B
   'The first one is greater.'
elseif A < B
   'The first one is smaller.'
elseif A == B
   'Both are equal.'
else
   error('Unexpected situation')
end
```

switch-case structure

MATLAB code:

```
switch (case_expression)
   case  case_selector_1
       block_1 of statements

   case  case_selecter_2
       block_2 of statements
       :
   otherwise  block_D of statement
end
```

eg. Program for arithmetic calculation

```
A=input('Enter number a: ');
ch=input('Enter (+,-,*,/): ', 's');
B=input('Enter number b: ');


switch ch

    case '+'

       M = A+B;

    case '-'

       M = A-B;

    case '*'

       M = A*B;

    case '/'

       M = A/B;

    Otherwise

       error('This is impossible')

end
```

16.3.3 Loop Control

MATLAB provides two repetition structures, including `for` loop and `while` loop for repeating certain part of the program if certain condition is true.

For loop

If the number of iteration is known or can be predetermined, we can use a counter-controlled `for` loop for the repetitive counting.
The general form of MATLAB code for the for loop is :

```
for  index = i_0 :  increment : i_last
       statements
end
```

Remark:

If the increment is 1, it can be omitted.

eg.

```
for i=1:m
     for  j=1:n
        A(i,j)=1/(i+j+5);
     end
end
```

While loop

A while loop repeats a group of statements while certain condition is true.

The general form is

```
while  condition
    statements
end
```

eg.

The following codes solve the equation $x - \frac{\pi}{2} - \frac{1}{3}\sin x = 0$ by using the fixed point iteration method $x_{i+1} = \frac{\pi}{2} + \frac{1}{3}\sin x_i = 0$ from the initial guess $x_0 = 0$, and stop when $|x_{i+1} - x_i| < Tol.$.

```
Tol=0.00000001;
```

```
x0=0.0;
x=pi/2+(1/3)*sin(x0);
while (abs(x-x0) > Tol)
   x0=x;
   x=pi/2+(1/3)*sin(x0);
end
```

Continue and break statements

The continue statement passes control to the next iteration of the while loop.

16.4 Matlab MATRIX AND ARRAY CALCULATION

MATLAB provides facilities for matrix calculation at two different levels including linear algebra matrix operations and array element-by-element operations.

16.4.1 Linear Algebra Matrix Operations

(a) The following table lists the basic linear algebra matrix operations

Operations	*Operators*	*MATLAB code*	*Linear algebra form*
Transpose	'	A'	A^T
Addition/Subtraction	+(-)	A+B(A-B)	A+B(A-B)
Matrix multiplication	*	A*B	AB
Matrix left division	\	X=A\b	X=INV(A*b (solution of AX=b)
Matrix right division	/	X=B/A	X=B*INV(A) (solution of XA=B)

(b) MATLAB also provide various functions specially for linear algebra matrix operations eg.

[L,U]=LU(A) ⇒ factors A to L and U

INV(A) ⇒ calculates an inverse of A

X=A\b ⇒ solves of Ax=b

`Eig(A)`	⇒	returns the eigenvalues of A
`[X,D]=eig(A)`	⇒	Diagonal elements of D are eigenvalues of A and columns of X are the corresponding eigenvectors such that AX=XD.
`[y,i]=max(X)`	⇒	y=maximum, i=the index of the maximum value in X

16.4.2 Array Operation on Element-by-Element Basis

Array operation refers to element-by-element arithmetic operation instead of the usual linear algebra matrix operation. Preceding an operator with a period `'.'` indicates an array or element-by-element operation. The following lists some basic array element-by-element operations.

Array multiplication and division

For `X=[1,2]; Y=[4,5]`

```
Z=X.*Y
```

results in

`Z=[4, 10]`

Element by element power

For `X=[1,2]; Y=[4,5]`

```
Z=X.^Y
```

yields

`Z=[1,32]`

Matrix function

`exp(A)` and `sqrt(A)` are computed on element-by-element based.

Subscript triplet notation and matrix generation

```
subscript_b : stride: subscript_e
```

defines an ordered set of subscripts that starts at `subscript_b` and end on or before `subscript_e` and have a separation of `stride` between consecutive subscripts.

eg. `0:2:8` defines an order set `0 2 4 6 8`
`-0.5:0.25:0.5` defines an order set `-0.5 -0.25 0 0.25 0.5`

For `x = 0.0:0.2:0.6;`

```
y = exp(-x).*sin(x);
```

```
[x y]
```

defines a matrix with vector `x` as column 1 and vector `y` as column 2.

```
0      0
0.2    0.1627
0.4    0.2610
0.6    0.3099
```

Subscripting/Subarray

Let `A` be a 10×10 matrix, then

`A(1:5, 7:10)` is a 5-by-4 submatrix of A from the first 5 rows and the last 4 columns,

`A(:,3)` is the 3rd column of A.

16.5 M-FILES : SCRIPTS AND FUNCTIONS

MATLAB is usually used in a command-driven mode. When a single-line command is entered, MATLAB processes it immediately. MATLAB is also capable of executing sequences of commands stored in files. Disk files that contain MATLAB statements are called M-files because they have a file type of ".m".

There are two kinds of M files

- Script files which contain a long sequence of MATLAB commands (program)
- Function files which define new functions that solve user-specific problems.

Both types of M-files are ASCII text files and can be created using an editor or word processor.

16.5.1 Script Files

When a script is invoked, MATLAB simply executes the commands in the file. The statements in the script file operate globally. Scripts are useful for solving problems that require long sequences of commands.

16.5.2 Function Files

If the first line of an M-file contains the keyword “function”, the file is a function file. A function differs from a script in that arguments may be passed, and that variables inside the file are local only and do not operate globally. Function files create new MATLAB functions for solving specific problems using MATLAB language.

Function Definition

There are three forms of function definitions based on the number of outputs to be returned by the function

- **No output**

```
function func_name(arg1,…,argN)
```

or

```
function []=func_name(arg1,…, argN)
```

eg

```
function printmessage()
% Print the messeage “Thank you for using our services”
fprintf('%20s','Thanks for using our services');
```

- **Single-value output**

```
function  output=func_name(arg1,…,argN)
```

eg.

```
function y=mean(X)
% Average value, for vectors, mean(X) returns the mean value.
% for matrices, mean(X) is a row vector containing the mean values
%   of each column
[m,n]=size(X);
if (m==1)
    m=n;
end
y=sum(X)/m;
```

- **Multiple-value output**

```
function  [output1,…,outputM]=func_name(arg1,…,argN)
```

eg.

```
function [x,y]=polar(theta, r)
x=r*cos(theta);
y=r*sin(theta);
```

Remarks

(1) When a M-function file is invoked for the first time during a MATLAB session, it is complied and placed into memory. It is then available for subsequent use with recompilation. It remains in memory for the duration of the session.

(2) The `what` command shows the list of M-files in the current directory on your disk.

(3) User can put all his/her m-files in a folder within the MATLAB folder as his/her own library of M-files. MATLAB responses to them in the normal way.

16.6 GRAPHICS

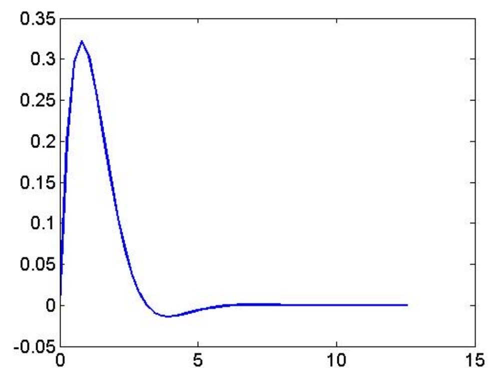
16.6.1 2-D Plots

(a) Line x-y plot

If `x` and `y` are vectors of the same length, the command plot(x,y) draws an x-y plot of the elements of `x` versus the elements of `y`.
For example, the following codes

```
x = 0:pi/12:4*pi;
y=sin(x).*exp(-x);
plot(x,y)
```

produce a 2-D plot of a scalar-valued function $y = \sin(x)e^{-x}$ versus `x`, as shown on the right.



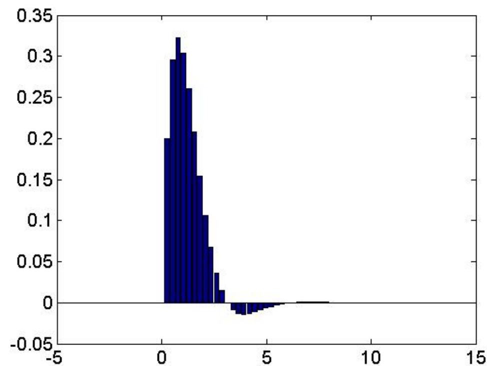
Remarks

(1) The statement in line 1 "`x = 0:pi/12:4*pi;`" creates a sequence of data values starting form o and ending at 4*pi with increment pi/12, and then stores the values in x.
(2) The statement in lines 2 generates a vector y from x via array element-element calculation.
(3) In `plot(x,y)`, if `x` is a vector, `y` is a matrix and the number of element in each column of `y` is the same as the number of element in `x`, then `plot(x,y)` plots columns of `y` versus `x`, using a different line type for each.
(4) If `x` and `y` are both matrices of the same size, `plot(x,y)` plots the columns of `x` versus the corresponding columns of `y`.
(5) The color of the lines can be specified by using the color codes : r (red), g (green), b (blue), w (white).

For example,

```
plot(x,y,'r')        % use a red line
plot(x,y,'+g')       % use green line and + marks
```

(b) Bar plot



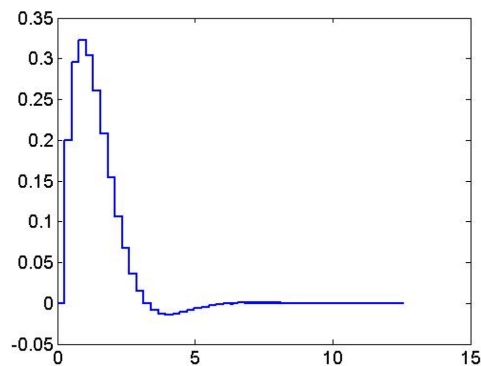
Example, the following codes

```
x = 0:pi/12:4*pi;
y = sin(x).*exp(-x);
bar(x, y);
```

produce a vertical bar chart of a scalar-valued function $y = \sin(x)e^{-x}$ versus x, as shown on the right.

Remarks: `bar(x, y)` creates a vertical bar chart `y` versus `x`.

(c) Stairstep plot



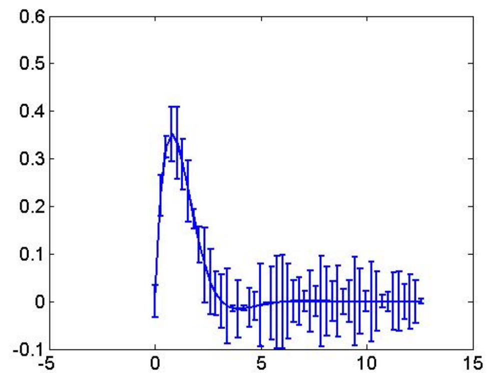
Example. The following codes

```
x = 0:pi/12:4*pi;
y = sin(x).*exp(-x);
stairs(x,y);
```

produce a stair-step plot of y versus x, as shown on the right.

Remarks: `stairs(x,y)` creates a stair-step plot of y versus x.

(d) Errorbar plot



Example, the following codes

```
x=0:pi/12:4*pi;
y= sin(x).*exp(-x)
ey=erf(y);
e= rand(size(y))/10;
errorbar(x,ey,e);
```

plot the error bars to show the confidence level of data or the deviation along a curve, as shown on the right.

Remarks :

`erf()`, `rand()`, `size()` and `errorbar()` are built-in functions:

`erf()` represents an error function defined by

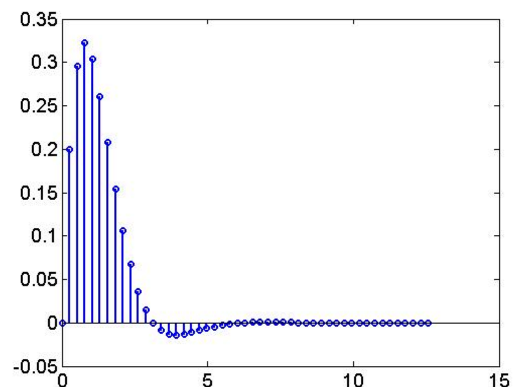
$$erf(y) = \int_0^y e^{-t^2} dt \ .$$

`size()` returns the numbers of rows/columns of a 1-D/2-D/3-D array.

`rand()` generates a vector consisting of uniformly distributed random numbers.

`errorbar()` shows the confidence level of data or the deviation along a curve. The command `errorbar(x,ey,e)` plots `ey` versus `x` with symmetric error bars `2*e(i)` long.

(e) **Stem plot**



Example, the following codes

```
x = 0:pi/12:4*pi;
y = sin(x).*exp(-x);
stem(x,y)
```

produce discrete-sequence data plot of a scalar-valued function $y = \sin(x)e^{-x}$ versus `x`, as shown on the right.

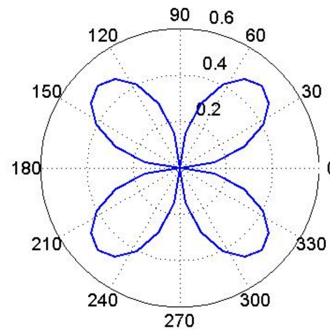
Remarks: `stem()` plots discrete sequence data `y` versus `x`.

(f) **Polar plot**

Example, the following codes

```
theta = 0:pi/20:2*pi;
r = sin(theta).*exp(-theta);
polar(theta,abs(r));
```

plot a graph `r=r(θ)` in polar coordinates in a Cartesian plane with polar grid, as shown on the right.



Remarks

(1) The statement in lines 2 generates a vector `r` from `theta` via array element-element calculation.

(2) `polar()`plots `r=r(θ)`in polar coordinates in a Cartesian plane with polar grid.

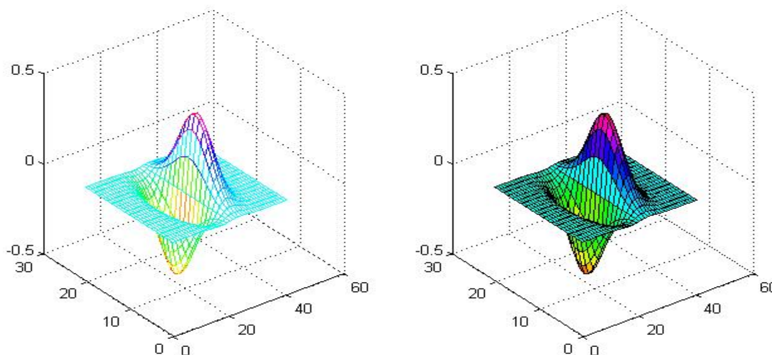
16.6.2 3-D Plots

(a) Mesh plot and surface plot

The following codes

```
xi = -4:.2:4;
yi = -2:.2:2;
[x,y] = meshgrid(xi,yi);
f = x.*exp(-x.^2 -y.^2);
subplot(1,2,1);
mesh(f);
colormap(hsv)
subplot(1,2,2);
surf(f);
```

generate two figures below including a mesh type graph and a 3-D shaded surface of f(x,y), respectively.



Remarks:

(1) The statement in line 1 "`x = -4:0.2:4`" creates a sequence of data values starting form -4 and ending at 4 with increment 0.2, and then stores the values in `x`.

(2) The statement in line 2 "`y = -2:0.2:2`" creates a sequence of data values starting form -2 and ending at 2 with increment 0.2, and then stores the values in `y`.

(3) meshgrid() in line 3 generates grid points for plotting a mesh-type graph.

(4) `mesh()` plots a mesh type graph of `f(x,y)`.

(5) `subplot()` divides the current figure into rectangular panes.
In the statement "`subplot(nrow, ncol, fops)`", the first two arguments set the figure to consist of `nrow*ncol` subfigures. The last argument specifies the location of each subfigure. In this example, there are two subfigures. One is plotted on the left column following the command "`subplot(1, 2, 1)` as shown in line 5" and another is plotted on the right column following the command "`subplot(1, 2, 2)`" as shown in line 8.

(6) `colormap()` is a built-in function for `setting` a color map which is a m-by-3 matrix of real numbers between 0.0 and 1.0. Each row is a RGB vector that defines one color.

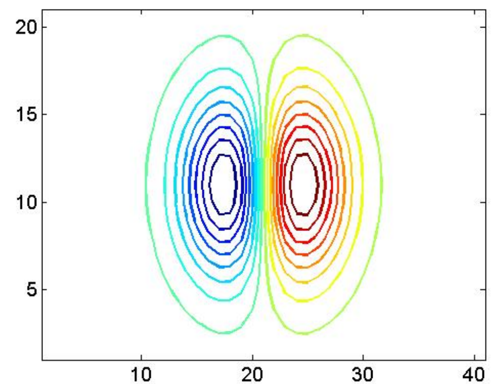
(7) `surf(f)` creates a 3-D shaded surface of f(x,y).

(b) Contour plot

Example, the following codes

```
xi = -4:0.2:4;
yi = -2:0.2:2;
[x,y] = meshgrid(xi,yi);
z = x.*exp(-x.^2 -y.^2);
contour(z,16);
```

generate a 2-D contour plot of a scalar-valued function $z = xe^{(-x^2)(-y^2)}$ on grid point (x,y) generated from points (xi,yi), as shown on the right.



Remarks:

(1) meshgrid() in line 3 generates grid points for plotting a mesh-type graph.

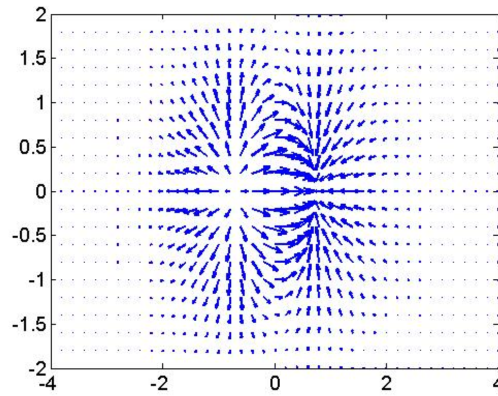
(2) `contour(arg1,arg2)` gives a 2-D contour plot of a scalar-valued function of two variables. In line 5, the statement "`contour(z,16);`" creates 16 contour lines of a scalar-valued function `z`.

(c) Vector plot

Example, the following coeds

```
xi = -4:0.2:4;
yi = -2:0.2:2;
[x,y] = meshgrid(xi,yi);
z = x.*exp(-x.^2 -y.^2);
[px,py] = gradient(z,.2,.2);
quiver(xi,yi,px,py,2);
```

plot a mesh-type graph of gradient vectors on grid points (xi, yi), as shown on the right.



Remarks:

(1) `gradient(arg1,arg2,arg3)` produces numerical gradient on each grid point.

(2) `quiver(arg1,arg2,arg3,arg4,arg5)` plots gradient vectors. In this example, "`quiver(xi,yi,px,py,2)`" plots gradient vector [px, py] corresponding to vector [xi, yi] with the scale of size 2. The last argument is optional, if it is omitted, `quiver` plots gradient vector with the scale of size 1.

REFERENCES

Amos Gilat. MATLAB: An Introduction with Applications 2nd Edition. John Wiley & Sons. 2004. ISBN 0471694207.

Bar-David, T. *Object-Oriented Design for C++*. Englewood Cliffs, NJ: Prentice ll, 1993.

Brain.L. Daku. Learn MATLAB FAST. 2005. ISBN:0471274690.

Cargill, T. *C++ Programming Style*. Reading, MA: Addison-Wesley, 1993.

Chapman, Stephen J. Matlab Programming for Engineers. Pws Pub Co, Boston, Massachusetts, U.S.A. ISBN:0534951511.

Chivers**,** Ian**,** Sleightholme, Jane. *Introducing Fortran 95*. Springer. 2000, XVII, 480 p.
ISBN: 1-85233-276-X

David G. Stork, Elad Yom-Tov. *Computer Manual in MATLAB to Accompany Pattern* Classification. Wiley-Interscience. 2004. ISBN: 0471429775.

Deitel, H.M. and Deitel, P.J. *C++ How to program*. Pearson Prentice Hall, 2005.

Ed Akin. *Object-Oriented Programming via Fortran 90/95*. Cambridge University Press. 2003.

Etter D.M. *Fortran 77 with Numerical Methods for Engineers and Scientists*. The Benjamin/Cummings Publishing Company, Inc. 1992.

Ellis, T.M.R., Phillips, I.R. and Lahey, T.M. *Fortran Programming 90*, Addison-Wesley. 1994.

Gerald Ratzer and Joseph Vybihal (Kendall/Hunt). *Fortran, C and Algorithms*--Programs are at http://www.cs.mcgill.ca/~ratzer/progs.html.

Gooch, T. "*Obscure C++*." Inside Microsoft Visual C++ Vol.6 No. 11, November 1995, 13-15.

Hahn, Brian D. Essential Matlab for Scientist and Engineers. Elsevier Science & Technology. ISBN: 0750652403.

Hunt Brian R., Ronald L. Lipsman, Jonathan M. Rosenberg , John E. Osborn , and Garrett J. Stuck. *A Guide to MATLAB for Beginners and Experienced Users.* Cambridge, UK and New

International Standard: Programming Languages—C++. ISO/IEC 14882:1998. New York, NY: American National Standards Institute, 1998.

Josuttis, N. *The C++ Standard Library: A Tutorial and Reference*. Boston, MA: Addison-Wesley, 1999.

Kruse, R.L. and A.J. Ryba. *Data Structures and Program Design in C++*. Upper Saddle River, NJ: Prentice Hall, 1999.

Langer, A. and K. Kreft. *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Reading, MA: Addison-Wesley, 2000.

LAPACK95 Users' Guide. SIAM,

Lemmon, David R., Schafer, Joseph L. *Developing Statistical Software in Fortran 95*. Springer. 2005, 328 p., Softcover ISBN: 0-387-23817-4

Marc. E. Herniter. Programming in MATLAB. Nelson Engineering. 2000.

Matsche, J.J. "Object-Oriented Programming in Standard C." *Object Magazine* Vol. 2, No. 5, January/February 1993, 71-74.

Michael Metcalf and John Reid *Fortran 90/95 Explained*. Oxford University Press. 2006.

Musser, D.R., G.J. Derge and A. Saini. *STL tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Second Edition. Reading, MA: Addison-Wesley, 2001.

Plauger, P.J. *The Standard C Library*. Englewood Cliffs, NJ : Prentice Hall, 1992.

R. A. Vowels. *Introduction to Fortran 90/95, Algorithms, and Structured Programming*. 1997

Rudra Pratap. Getting started wuth MATLAB7: A quick introduction for Sciences and Engineers. Oxford University Press, USA. 2005. ISBN:0195179374.

Sedgwick, R. *Bundle of Algorithms in C++, Parts 1-5: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms* (Third Edition). Reading, MA: Addison-Wesley, 2002.

Sessions, R. *Class Construction in C and C++: Object-Oriented Programming*. Englewood Cliffs, NJ: Prentice Hall, 1992.

Stepanov, A. and M. Lee. "The Standard Template Library." 31 October 1995 www.cs.rpi.edu/~musser/doc.ps.

Stephen Chapman. *Fortran 90/95 for Scientists and Engineers*. McGraw-Hill. 1998.

Steve Morgan and Lawrie Schonfelder. *Programming in Fortran 90/95*-electronic version of the book McGraw-Hill.

Stroustrup, B. "What is Object-Oriented Programming?" *IEEE Software* Vol. 5, No. 3, May 1988, 10-20.

Stroustrup, B. *The C++ Programming Language*. Special Third Edition. Reading, MA: Addison-Wesley, 2000.

Thomson-Engineering Staff. Essential of MATLAB Programming. 2005. ISBN: 0495073008. York:: Cambridge University Press. 2002. ISBN: 052100859X.

INDEX

Arrays dimension attribute, 165
A format code, 153
Actual arguments, 185, 189, 210
Algorithm, 5
  decomposition, 5
  flow chart, 5
  pseudocode, 5
  stepwise refinement, 5
  structured algorithm, 5
  top to down design, 5
Algorithm design
  decomposition, 6
  stepwise refinement, 6
Allocatable arrays, 210
  allocation of space, 173
  declaration, 173
  release of space, 174
Allocate, 173, 208
Allocation of spaces, 75
Argument, 14
Arithmetic computation, 101
Arithmetic computations, 8
Arithmetic expression, 12
Arithmetic Expression, 13
Arithmetic expressions, 104
Arithmetic logic unit, 3
Arithmetic operator, 14
Arithmetic operators, 105
Array element operations, 168
Arrays, 58, 190
  1-Dim, 58, 165
  2-Dim, 59, 167
  array specification, 165
  declaration, 58
  dimension attribute, 191
  multi-Dim, 61
  operations, 61
Assays
  passing arrays to functions, 63
Assignment statement
  addition assignment, 12
  subtraction assignment, 12
Assignment statements, 12, 104
Assumed shape array, 191
  assumed_shape_specifier, 191
  intrinsic functions, 191
Backspace statement, 158
Boolean data, 10
C++, 2
C++ program, 20
  blank line, 21
  classes, 22
  comment line, 21
  function, 22
  main function, 22
  `namespace std`, 21
  statement, 22
  top to down design, 45
Calling a subroutine, 187
Case construct, 120
Central processing unit, 3
Character data
  intrinsic functions, 143
  operations, 142
  variables, 141
Character constant, 10
  \', 10
  \n, 10
Character data, 10
Classes, 89
  data members, 89
  defining a class, 89
  member functions, 91
Close, 158
Collating sequence, 142
Compile, 7
Complex data
  intrinsic functions, 140
  operations, 140
  variables, 140
Computer, 3
computer system, 3, 7
Conditional DO loop
  F95 statement, 126

flow chart, 126
Pseudocode, 126
Constants, 9, 101
Constructor, 92
Control structures, 27
Count controlled DO loop
F95 statement, 124
flow chart, 124
pseudocode, 124
D format code, 152
Data members, 89
Deallocate, 174, 211
Decomposition, 5, 6
Delete operator, 75
Dimension attribute, 191
Do while loop, 37, 128
double, 10
Dummy arguments, 184, 190, 210
Dummy drguments, 189
Dynamic memory allocation, 74, 202
E format code, 152, 154
Elseif statements, 119
Endfile statement, 158
Executable statements, 110
execute program, 7
Exit statement, 126
Explicit interfaces, 188, 190
External memory, 3
F format code, 154
F format code, 151
F95 program
comment lines, 111
layout, 111
File operations
direct access, 156
sequential access, 156
`write data to file`, 157
File operations, 156
File operations
open, 156
File operations
`read data from file`, 157
File operations
end option:, 158
Files, 78
create a file, 78
reading data, 81
writing data, 80
float, 10
Flowchart, 5
For loop, 34
execution, 35
general form, 34
nested *for* loop, 35
Format, 149, 153
Format output, 19
`fixed`, 19
`left`, 19
`right`, 19
`scientific`, 19
`setprecision`, 19
`setw(n)`, 19
stream manipolators, 19
Format specifier, 149, 153
Formatted input
character, 155
real, 154
Formatted input, 153
Formatted input, 153
Formatted input
complex, 154
Formatted output
format codes X,I,F,E,D,A, 149
Formatted output, 149
Formatted output
buffer, 150
Formatted output
carriage control character, 150
Fortran 95, 100
Function, 22, 45, 183
calling a function, 48
defining a function, 46
function body, 22
function declaration, 48
function header, 22
library function, 45
return 0, 22
user defined function, 45
Function overloading, 50
Function pointers, 69

Global data, 189
Global variables, 51
Header file, 45
I format code, 151, 153
If statement, 30
If statements, 29, 118
Implicit none, 104
Include, 19
  cmath, 45
  `cstdlib`, 50
  `iomanip`, 19
  `iostream`, 21
Infinite loop, 127
Initial values, 20, 108
Input device, 3
Inquire tatement, 158
`int`, 9
Integer, 102
Intent(in), 186
Intent(inout), 186
Intent(out), 186
Internal procedures, 191
Intrinsic functions, 105
Kind type parameter
  data range, 137
  precision requirement, 137
Kind type parameter, 136
L format code, 152
Library functions, 13
List directed input, 107, 149
List directed output, 108, 149
Literal specification, 150
Literal Specification, 150
Local variable, 185
Local variables, 51
Logical calculation, 28
Logical calculations, 117
Logical constants, 28, 117
Logical errors., 7
Logical expressions, 116
Logical operators, 28, 117
Logical variables, 117
Logical Variables:, 28
long double, 10
long int, 9
Loop control, 33
Member functions, 91
Members functions, 90
Memory, 3, 4
  external, 4
  internal, 4
Mixed mode operations, 106
Mixed-mode operation, 15
Modules
  accessing a module, 188
  defining a module, 187
Named constant, 20
Named constants, 109
`New` operator, 75
Non-executable statements, 110
Object, 89
Object oriented programming, 89
Open, 79, 156
Output device, 3
Over flow error, 17, 107
Pointer arithmetic, 73
Pointer arrays
  allocating space to pointer arrays, 208, 209
  declaration, 207
Pointer operators, 69
Pointers, 68, 202
  association with variables, 203, 204
  declaration, 68, 202
  initialisation, 68
  passing arguments to functions, 70
Pointers and Arrays, 73
Pointers arrays, 207
Print, 108, 149
priorities of operations, 15
Private, 91
Procedures, 190
Processor, 3
Program, 3
Project, 51, 52
Public, 91
Read, 107, 153, 166, 167
`READ`, 157
Real, 102
Real data, 9
Recursive function, 49

Relational expression, 27, 116
Relational operator, 116
Relational operators, 27
Release of space, 75
Repetition structures, 5
Rewind statement, 158
Run-time errors, 7
Save, 191
Scope, 51
Selected_int_kind, 137
Selected_real_kind, 137
Selection control, 29, 118
Sequence structures, 5
short int, 9
Statement function, 192
Static global functions, 51
Static global variables, 51
Stepwise refinement, 5
Stop, 110
Stream input, 17
Stream output, 18
Subroutine, 185
Switch statement, 31
Syntax error, 7
Tab specification, 155
Top to down design, 183
Truncation error, 17, 107
  sources, 17
Under flow error, 17, 107
unsigned double, 10
unsigned float, 10
unsigned long double, 10
unsigned long int, 9
unsigned short int, 9
Variable, 10
  declaration, 11
  name, 11
  type, 11
Variables, 102
While loop, 36
Whole array operations
  conformable arrays, 169
  shape of an array, 169
Whole array operations
  intrinsic procedures, 170
Whole array operations
  masked array assignment, 171
Write, 149, 157
X specification, 151, 153