UNIVERSITY OF CROSS RIVER STATE

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL ELECTRONICS ENGINEERING

COURSE TITLE: ADVANCED COMPUTER PROGRAMMING AND STATISTICS

3.0 What is Programming

A computer program is a series of instructions given to the computer at a particular time to solve a problem. It is usually written in a particular programming language, and named according to the language used in writing it.

3.1 Programming Language

The Programming Language is the language through which we can pass instructions to user and the system. A Computer Language (Programming Language) is composed of a set of characters, words and rules that can be used to write a computer program. Examples of some of the programming languages are BASIC, FORTRAN, COBOL, C, C++ etc. that the user and the computer must understand it.

Apart from Programming Language, a Command Language (CL) helps the user to communicate with and obtain services from the system in the following ways:

Identifies the user to the system

Asks for program to be run and

Tells the computer where the data may be found.

Programming languages can be classified according to their levels as follows:

Machine Language

Symbolic Language

High Level Language

3.1.1 Machine Language

Every computer has its own machine language. Instructions to the computer are given in machine language since this is the only language computer actually understands. The machine language varies from machine to machine which is a function of the hardware of the computer. Programs written in machine language are only good for that particular machine.

The machine language is a low level interface, natural to the hardware of the computer. The instructions are usually represented by binary numbers, that is, the communication symbols or sequence of instructions consist of 0's and 1's otherwise known as the binary digit (BIT). Since the machine language depends on the particular computer model, it is said to be machine dependent. The early days of computer programming was almost exclusively machine language programming. The difficult nature of writing in machine language and the machine dependent nature which is very involved and detailed led to the development of symbolic languages and later higher level languages.

3.1.2 Symbolic Languages

Arising from the machine language difficulties, is the development of a higher level up the ladder (hierarchy) of programming languages called assembly languages. Here the sequence of 0's and 1's are replaced with symbols, so that instructions can be given in symbolic codes called mnemonics. The symbolic language is a middle level language between the machine language and high level languages. The language adopts mnemonic codes to represent machine instructions. The uses of mnemonic codes imply that mnemonic names are used for operators, addresses and instructions. Ordinarily, the computer does not understand these codes until they are translated to machine language using the appropriate translator. Symbolic language is largely machine dependent just like machine language. An example of symbolic language is the Assembly language. We use Assembler to translate Assembly language to machine language for the understanding of the computer.

We can define *Assembler* as a machine language instruction or program that translates symbolic instructions such as Assembly language instructions into machine language instruction. The difficulty in understanding, the dependence of the language on the machine model and peculiarity of the mnemonics codes posed a serious disadvantage to the programmer. This necessitated the introduction of high-level languages because the

programmer must be concerned with various details and complex sequence of instructions to be able to write symbolic language.

3.1.3 High Level Languages

High level language introduces the use of "almost English" language in writing computer programs which are also known as problem oriented languages. Examples are FORTRAN, COBOL, PASCAL, C, etc.

BASIC - Beginners All Purpose Symbolic Instruction Code

COBOL - Common Business Oriented Language

FORTRAN - Formula Translation

PASCAL - Named after French mathematician Blaise Pascal

The language use English like phrases to represent computer instructions (codes) in a way that resemble programmers thinking process using English like expressions. The language is procedure oriented rather than machine oriented. Programs written in this type of language can, with possible minor modifications, run on different machines as long as there is appropriate translator/compiler. A *translator* is a machine language program that translates source codes written in either symbolic or high level language to machine language program known as the *object language*. They can also be referred to as compiler languages and are almost machine independent and very easy to use. If the source program (source code) is a high level language program such as FORTRAN, PASCAL, etc., the translator that produces the object code in machine understandable form is called a Compiler. A *compiler* is a special program which translates programs written in high-level language into machine language.

In more detail, the compiler (e.g. FORTRAN compiler, PASCAL compiler, etc.) reads the source code and generates the machine code (object code), and saves it unto disk file. When the program is to be executed, the computer simply loads the object code into the memory and executes it. An interpreter, a kind of translator such as BASIC, is required to translate the source program to machine understandable form. Interpreter reads the source program directly, line by line, starting from the beginning of the source program each time it is to be executed and generates machine codes necessary to carry out the instructions as they proceed through the program, without saving it onto disk. If the source program is a low level language (Assembly language), the translator that produces the object code in machine

understandable form is the *Assembler*. It must be noted that original program is called the *source program*, and its translation is called the *object program*. It is also important to remember that a high/low level language program is first translated into machine language instructions before the computer can actually executes them as shown in figure 1.1 below.

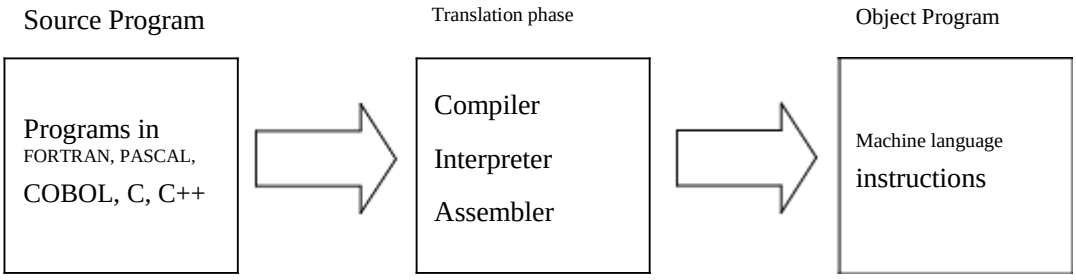


Fig. 1.1: Illustration of Translators

Although compiler languages are less efficient than machine or assembly languages, they do relieve the programmers from the burden of keeping careful details.

3.2 Program Process and Errors

There are general violations of the rule of programming or pitfalls of computer arithmetic that are referred to as *Errors* or *Program Errors*. The arithmetic errors could be conversion error or round-off error. Program could also exhibit three types of errors namely; syntax errors, run-time errors and logical errors. During program execution, errors detected will be listed, corrected by the programmer and program re-run carried out again.

Conversion Error: numbers are mostly stored in their binary form. A decimal number 0.5 is represented by the binary number 0.1 in such computers. Again, a number 1/10 is represented by 0.0001100110011……. which has no finite binary representation. Depending on the computer word length, errors will be noticed if the arithmetic operations become bigger.

Round-off Error: this occurs when a specific significant digits are used, for example, 1/3 which supposed to be 0.3333333….. could be stored as 0.333.

Syntax Error: errors due to misspelled (spelling errors) statements or instructions or wrong use or non use of punctuation marks (such as commas, colon etc.) where necessary. When

such errors are detected, the programmer should look at the program for possible spelling errors or omission of punctuation marks.

Run-time Error: this error occurs when the translator come across wrong mathematical operation during program execution. This could occur, as an example, when a number is divided by zero.

Logical Error: program logic is always the cause of the error when the sequences of programming steps are not executed properly. Logical error may not have impact with the smooth translation process of the program but may give wrong result. There could be successful interpretation or compilation of a program that actually has a logical error. The logical errors are mostly noticed upon execution of the program.

3.2.1 Processing Error Steps

The following steps are very necessary in dealing with errors in program compilation and execution. They are:

Program writing and correction requires patience. The programmer should carefully through the program for possible correction of any error.

Use appropriate manuals and reference text books for the programming language employed for possible assistance.

Double check for misspelled statements or omission of punctuation marks

Check for proper use of data types

Make sure variables meant for counting purposes are properly initialized

Self Assessment Exercise

1. What is a computer program?
2. Differentiate between an interpreter and compiler
3. What type of programming error that is not easily noticed during compilation and why?

4. State why High level language is very easy to use compared with the earlier ones

3.3 Properties of a Good Program

A good programming practice is needed to write a good program. Program development has to follow problem definition, analysis, modelling, algorithm/flowchart design, coding and documentation. The following properties are required for developing a good program.

Program Correctness: a good program must be able to solve the intended problem with relevant results. The output (result) must be readily available for testing with assumed or calculated results using real or dummy data.

Documentation: every module or procedure must be preceded with comments on brief explanation of the module in the program. These make programs easy to read and understand by other users that may want to modify or improve on the program. Complete documentation of the whole program is also necessary to give details of the input, output, processing tasks and manual guide.

Robustness and Scalability: programs that can survive various unexpected events are said to be robust and those that can easily be upgraded are scalable. They are sometimes called safe or defensive programs because of the way they are written, the choice of variable names, surviving incorrect data etc.

User Interface: a good look or design of the medium of interaction of the user and the program must be well taken care so as to have a good user interface. This is the part of the program that performs the dialog aspect of the program with the user and must be easy and friendly to use.

Program Style: the programming language rules are to be adapted in writing programs. You should not do what is not to be done with the programming language in question.

Use of Tools and Library Functions: programming languages have special tools and libraries that can assist in developing computer programs. Some editors are also used to enhance the development of programs.

3.4 Structure of a Program

Programs are written in one or more units. The main unit is called the *Main Program* whine the other optional units are subroutines which could be either subprograms or functions.

The main unit is usually of the form:

i. Program Name

ii. TYPE declaration statements

iii. Statements Performing Arithmetic and other Operations

iv. STOP

v. END PROGRAM

As indicated in item one above, each program has a name by which it is distinguished. The next segment starts with Type declaration statements. Example of these type statements are REAL and INTEGER statements amongst others. The following segment is the statements performing arithmetic and other operations where the processes to executing the instructions are stated. There must be an END statement or END of program statement and STOP statement preceding END statement in case of FORTRAN language. The END statement is used to inform computer of the physical end of the program. Hence, there could be only one end statement in a program which normally appears at the last statement of a program.

3.5 Example of a Simple Program

We write a program which compute and output the sum and product of three numbers. In solving the program, the language chosen is FORTRAN 90/95 to compute the sum and product of three numbers. The three numbers are identified by letters A, B, and C with the variables SUM and PRODUCT as summation and the products of the three numbers respectively. First, the writer must know how to sum and find product of the three numbers numerically.

Note that letters A, B and C are simply variable locations for three numbers because the actual values are not known. It is also used because computer stores its data in variable locations in the memory and retrieved during computations.

The simplified FORTRAN 90/95 program for the summation and the product of three numbers are as follows:

```
Program SUMPRODUCT                          Line 1
IMPLICIT NONE                               Line 2
INTEGER :: A, B, C, SUM, PRODUCT            Line 3
READ (*, *) A, B, C                         Line 4
SUM=A+B+C                                   Line 5
PRODUCT=A*B*C                               Line 6
WRITE (*, *) SUM, PRODUCT                   Line 7
STOP                                        Line 8
END PROGRAM SUMPRODUCT                      Line 9
```

The above program is explained as follows:

Line 1: this indicates the program name

Line 2: the IMPLICIT NONE statement is supported by FORTRAN 90/95 as standard. It is used to show all variables are explicitly typed and declared. This statement should come before any other declarations or executable statements in the program.

Line 3: a declaration statement for A, B, C, SUM and PRODUCT as Integers numbers. The letters A, B and C contain the actual numbers to be read by the computer and the output of the summation of the numbers and product stored in SUM and PRODUCT respectively.

Line 4: the READ statement causes data to be read from appropriate device to the memory locations of variables A, B and C.

Line 5 and 6: the two lines indicate the process to be carried out or executed by the program. The computation of the sum and product of the three numbers are carried out with the resultant solution stored in SUM and PRODUCT.

Line 7: this is the output statement that generates the results in the form of the sum and product of the three numbers. The (*, *) in the statement indicate that the output can be generated from appropriate device e.g. Printer.

Line 8: the STOP statement acts like a purse to the program which indicates the end of the program.

Line 9: the END statement tells the compiler that it has come to the end of the program.

3.0 Debugging Code

This unit looks at some of the debugging features available in some compilers. You'll learn how to set breakpoints in your code to stop execution at any given point, how to watch the value of a variable change, and how to control the number of times a loop can execute before stopping. All these can help you determine just what is going on inside code.

3.1 Understanding Types of Errors

The errors caused by a computer program (regardless of the language in which the program is written) can be categorized into three major groups: design-time errors, runtime errors, and logic errors.

3.1.1 Correcting Design-Time Errors

Design-time errors, which are the easiest type of error to find and fix, occur when you write a piece of code that does not conform to the rules of the language in which you're writing. They are easy to find because some compilers tells you not only where they are but also what part of the line it doesn't understand. Design-time errors, also called syntax errors, occur when the compiler cannot recognize one or more statements that you have written. Some design-time errors are simply typographical errors (e.g., a mistyped keyword). Others are the result of missing items (e.g., undeclared or untyped variables).

A program with as few as one design-time error cannot be compiled and run; you must locate and correct the error before you can continue.

3.1.2 Understanding Runtime Errors

Runtime errors are harder to locate than design-time errors because compiler does not give you any help in finding an error until it occurs in a program. Runtime errors occur when a program attempts something illegal, such as accessing data that does not exist or a resource to which it does not have the proper permissions. These types of errors can cause a program to crash, or hang, unless they are handled properly.

Runtime errors are much more difficult to find and fix than design-time errors. Runtime errors can take on dozens of different shapes and forms. Here are some examples:

Attempting to open a file that doesn't exist

Trying to log in to a server with an incorrect username or password

Trying to access a folder for which you have insufficient rights

Accessing an Internet URL that no longer exists

Dividing a number by zero

Entering character data where a number is expected (and vice versa)

As you can see, runtime errors can occur because of an unexpected state of the computer or network on which the program is running, or they can occur simply because the user has supplied the wrong information (e.g., an invalid password, a bad filename). You can therefore write a program that runs fine on your own computer and all the computers in your test environment but fails on a customer site because of the state of that customer's computing resources.

3.1.3 Understanding Logic Errors

The third type of error, the logic error, is often the most difficult type to locate because it might not appear as a problem in the program at all. If a program has a logic error, the output or operation of the program is simply not exactly as you intended it. The problem could be as simple as an incorrect calculation or having a menu option enabled when you wanted it disabled. Quite often, logic errors are discovered by users after the application has been deployed.

Logic errors also occur at runtime, so they are often difficult to track down. A logic error occurs when a program does not do what the developer intended it to do. For example, you might provide the code to add a customer to a customer list, but when the end user runs the program and adds a new customer, the customer is not there. The error might lie in the code that adds the customer to the database; or perhaps the customer is indeed being added, but the

grid that lists all the customers is not being refreshed after the add-customer code, so it merely appears that the customer was not added.

3.2 Desk Checking Logic

Debugging an application should actually start when you are designing the program flow. After you write the algorithm for a procedure, you should walk through it using a pencil and paper to make sure it works as you intended. This process is called desk checking. To desk check an algorithm, you supply sample values and walk through the algorithm, performing the calculations.

To see how this works, let us use the algorithm example:

1. Declare variables for Wage, HoursWorked, GrossPay, WithholdingAmount, and NetPay.
2. Declare a constant for WithholdingPercent and set it equal to 20%.
3. Input Wage.
4. Input HoursWorked.
5. Set GrossPay Wage * HoursWorked.
6. Set WithholdingAmount GrossPay * WithholdingPercent.
7. Set NetPay GrossPay WithholdingAmount.
8. Output NetPay.

To desk check this algorithm; you perform each step, using sample values for the variables. You document the expected value for each variable after the step completes. You test expected input values and unexpected input values to try to find potential problems in the algorithm. For example, what if the user entered a negative number for hours worked?

3.3 Setting Breakpoints

When trying to debug a large program, you might find that you want to debug only a section of code; that is, you might want your code to run up to a certain point and then stop. This is where breakpoints come in handy: They cause execution of code to stop anywhere a breakpoint is set. You can set breakpoints anywhere in code, and the code will run up to that point and stop. Note that execution of the code stops before the line on which a breakpoint is set.

You can set breakpoints when you write code, and you can also set them at runtime by switching to the code and setting a breakpoint at the desired location. You cannot set a

breakpoint while a program is actually executing a section of code, such as the code in a loop, but you can when the program is idle and waiting for user input.

When the development environment encounters a breakpoint, execution of the code halts, and the program is considered to be in break mode. While the program is in break mode, a lot of debugging features are available. In fact, a lot of debugging features are available to you only while a program is in break mode.

3.4 Stepping Through Code

When debugging an application, it is helpful to be able to step through the code line by line. You can use the followings:

Step Into: This includes stepping into any function or procedure that the code calls and working through it line by line.

Step Over: This works in a similar way to Step Into, but it enables you to pass straight over the procedures and functions; they still execute, but all in one go. You then move straight on to the next line in the block of code that called the procedure.

Step Out: This allows you to jump to the end of the procedure or function that you are currently in and to move to the line of code after the line that called the procedure or function. This is handy when you step into a long procedure and want to get out of it. The rest of the code in the procedure is still executed, but you do not step through it.

Self Assessment Exercise

1. Compare design-time error, runtime error, and logic error. Give an example of each.
2. A logic error is also called a syntax error. True or False?
3. Which type of error results in a hang or an exception during execution?
   (a) design-time error
   (b) logic error
   (c) runtime error
   (d) syntax error
4. Logic errors are often discovered by end users. True or False?
5. Desk checking is performed after you write code. True or False?
6. You can set a hit counter on a breakpoint to cause a program to pause after a loop executes a specific number of times. True or False?

3.0 Describing a Program, Using Algorithms and Flowcharts

When you write a program, that program must perform some task. It might be a real-world business task, such as managing payroll or processing orders. Or it might be a program that provides entertainment, such as a game. In either case, the program will perform a large number of tasks, and for each of them, you must provide a very specific set of instructions.

Before you begin to write code, you need a clear understanding of the tasks the program must perform: the requirements and the steps it should take to perform them. The steps a program must take to perform a task is known as the program's flow. You can design a program's flow by using algorithms and flowcharts.

3.1 Writing Algorithms Using Pseudocode

An algorithm is a step-by-step description of a procedure. Writing an algorithm allows you to think about the logic of a program without worrying about the syntax of the programming language.

You can think of an algorithm as being like a recipe:

1. Put 1 tablespoon of oil in a frying pan.
2. Heat the oil on the stove at low temperature.
3. Break three eggs into a bowl.
4. Beat the eggs with a whisk.
5. Add cheese.
6. Add seasonings.
7. Add the egg mixture to the frying pan.
8. Cook on both sides.

You write an algorithm by using pseudocode. As the word suggests, pseudocode is like code but different. In fact, pseudocode is a description of the program flow, written in the language you speak. When you write pseudocode, you usually keep the sentences short and to the point.

You clearly define each step that must be taken to perform the task. For example, suppose you need to write a program that accepts a person's hourly wage and calculates the person's pay. For the sake of this example, let's assume a withholding rate of 20%. The algorithm for this procedure is as follows:

1. Declare variables for Wage, HoursWorked, GrossPay, WithholdingAmount, and NetPay.
2. Declare a constant for WithholdingPercent and set it equal to 20%.
3. Input Wage.
4. Input HoursWorked.
5. Set GrossPay Wage * HoursWorked.
6. Set WithholdingAmount GrossPay * WithholdingPercent.
7. Set NetPay GrossPay WithholdingAmount.
8. Output NetPay.

There are different ways you could write this pseudocode. For example, you could write step 5 as "Multiply Wage by HoursWorked and store the result in GrossPay." However, the point of writing an algorithm is to describe the program's logic, not to worry about language.

Self Assessment Exercise

1. Compare an algorithm with a flowchart
2. Compare pseudocode and code

3.2 Using Flowcharts

A flowchart is a graphical representation of a program's logic. You can draw a flowchart on paper or using a flowcharting tool such as Microsoft Visio. There are even flowcharting objects available in Microsoft Office.

For example

Example 1: Finding the sum of two numbers.

– Variables:

A: First Number

B: Second Number

C: Sum (A+B)

– Algorithm:

Step 1 – Start

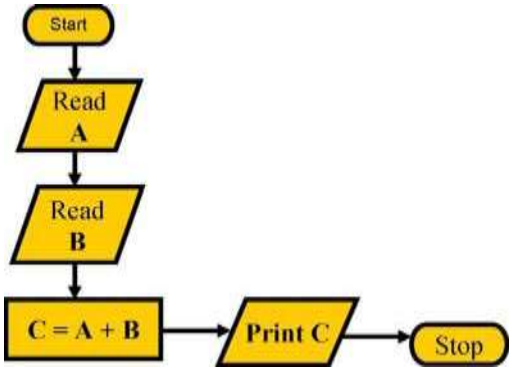
Step 2 – Input A

Step 3 – Input B

Step 4 –Calculate C = A +

B Step 5 – Output C

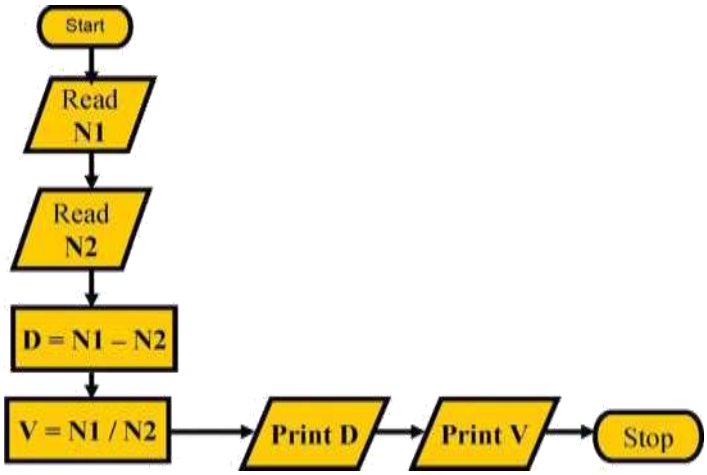
Step 6 – Stop



Example 2: Find the difference and the division of two numbers and display theresults.

– Variables: --Algorithm:

- N1: First number * Step 1: Start
- N2: Second number * Step 2: Input N1
- D : Difference * Step 3: Input N2
- •V : Division * Step 4: D = N1 –N2
- Step 5: V = N1 / N2
- Step 6: Output D
- Step 7: Output V
- Step 8: Stop



When you draw a flowchart, you should use industry-standard shapes to represent each step in the process. You usually draw the flow from top to bottom or from left to right. Arrows connect the shapes to define the flow. The oval is a terminator and marks the beginning and end of the flow. The parallelograms are used to represent input and output. The rectangles are used to represent a process.

Self Assessment Exercise

1. Which of the following is a step-by-step description of a process written in pseudocode?
    (a) algorithm
    (b) flowchart
    (c) control of flow
2. Which of the following is a graphical depiction of a program's flow?
    (a) algorithm
    (b) control of flow
    (c) flowchart
    (d) UML diagram

4.0 Conclusion

Flowcharts and algorithm is to show graphical representation and step-by-step method of solving problems.

3.0 Number system

Binary numbers can be changed to decimal (base 10, the kind most people use) in a simple way. The value of the bit (binary 1 or 0) on the right side of the number is 1. Every other bit has a value two times the value of the bit to its right. Add the values of every bit that is a 1 together to get the decimal number. {The 4 bit code has weight associated with it from left i.e. 1, 2, 4, 8, 16, 32, 64, ... Thus the code 1101 converted into decimal number will be 13.
For example, the binary number 101 is 5 in decimal. The bit on the right is 1 and has a value of 1. The middle bit has a value of 2 (1 times 2), but it is a 0, so it is not added. The bit on the left is 1 and has a value of 4 (2 times 2). The bits that are 1's have values of 1 and 4. 1 + 4 = 5.

3.1 Computers

All computers use binary at the lowest level. Most computer storage, like compact discs and DVDs, use binary to represent large files. With computers, 8 binary bits together is called a byte. The size of files is commonly measured in kilobytes or megabytes (sometimes in gigabytes). A kilobyte is 1,024 bytes. A megabyte is 1,024 kilobytes, or 1,048,576 bytes, while a gigabyte is 1,024 megabytes.

3.2 Binary numeral system

The binary numeral system is a way to write numbers using only two digits: 0 and 1. These are used in computers as a series of "off" and "on" switches. In binary, each digit represents twice as much as the next digit to the right. Here is a list of some numbers that can be made from these digits (zero is represented by a single "0"):

Decimal	Binary	Explanation
1	1	1
2	10	2+0
3	11	2+1
4	100	4+0+0
5	101	4+0+1
6	110	4+2+0
7	111	4+2+1
8	1000	8+0+0+0
9	1001	8+0+0+1
10	1010	8+0+2+0
11	1011	8+0+2+1
12	1100	8+4+0+0
13	1101	8+4+0+1
14	1110	8+4+2+0
15	1111	8+4+2+1
16	10000	16+0+0+0+0

Binary is a numbering system that is a series of 1s and 0s meaning (to the computers) on and off. It is base 2 and our number system (decimal) is base 10. Binary was invented by many people but is credited to Gottfried Leibniz, a German mathematician. The idea of binary was created in the 1600s. Binary has been used in nearly everything electronic; from calculators to supercomputers.

Self Assessment Exercise

1. What is the maximum decimal number you can represent in 4 bits?

3.0 Framing Methods

A common method of combining a sequence of bytes to form one unit, usually several such units are stringed one after the other.

Examples of framing:

Variable-size records used in FORTRAN unformatted files

Packets used in TCP/IP and other communication protocols

A possible structure of one such unit may be:

Signature	Count	Data bytes	Integrity check

SIGNATURE: some byte combination that identifies the unit type (by some convention agreed on between the users).

COUNT: an integer (whose size is agreed on by convention), the value of COUNT is the number of data bytes to follow. With the COUNT field units can have variable length, that flexibility is important in many applications. DATA BYTES: the 'payload'

CHECK: Some error checking/correcting code associated with the data when the data is written, the code can be rechecked whenever the unit is accessed to validate data integrity.

3.1 Caching

Whenever we have available two data storage methods, it usually happens that one of them will be slower and cheaper so we can buy more storage, and the other faster and more costly, for example:

Local files vs. files accessed through the web

Local filesystem vs. network filesystem (NFS)

Local files/partitions vs. main memory (RAM)

Dynamic memory vs. static memory

Caching promotes efficiency when two conditions are met:

We have "locality of reference", i.e. subsequent data references are usually made to "close by" items.

Reading a "block" of consecutive data is relatively quick.

For example, caching of read operations (from a slow medium) may be implemented by the following guidelines:

Whenever a read operation from the slower media is needed we read a data block containing the required item.

The data block is kept on the faster media, if there is no available place, older blocks are discarded.

When a data item is needed the faster storage is first checked to see if it contains the required item.

If the required item is on the faster storage it is read from it and used, otherwise it is read from the slower media (see #1).

Write operations (to a slow medium) may also be cached, but this is a bit more complicated. Caching is an effective technique that is used intensively in all computing systems. As caches may be shared (among processes etc), the time it takes to perform a task becomes non-deterministic, other users of the cache, doing their own work may or may not discard your data, and so affect computation time.

A high-performance Fortran programmer must take into consideration the size of the various caches (memory, I/O) when writing programs. A simple advice is to keep often used data in arrays, instead of reading it again and again from a file.

3.2 Buffering

Data transfers between hardware and software, or between different pieces of the software, are many times "blocked" and not simply "byte-by-byte", for example:

commands written at the operating system prompt, are interpreted and executed only after you press the "return" key. When a "return" (or a pre-defined "control sequence") is sent from the keyboard, the accumulated string is made available to the running program.

Disk controllers transfer data in "chunks" that are multiples of a sector (512 bytes). I/O is therefore "buffered", sometimes few times on different levels to satisfy this condition.

Buffering is done utilizing an array (the buffer), large enough to hold a chunk of data of the required size, and a few procedures that are used to put and get data from the buffer.

3.3 Logical Layering

Well, the best example would be Fortran itself. Poor “virtual programmers” need a simpler and portable interface to our computer. The solution is to define a 'high-level language' e.g. Fortran/C/... and have on each machine a program (compiler) that translates programs written in Fortran/C/... to machine language. At the price of some CPU time we get a portable and easy to use interface to every computer.

Examples for 'logical layering' are innumerable, they can be found in the following interfaces:

Interactive user / Operating system / Different kinds of terminals

User program / Operating system / Different disks and tape drives

File names / File system / Data blocks on disks and tape drives

Graphic package / Different graphic devices

Word-processor / Different printers

In a PC: DOS functions / ROM-BIOS interrupts

In TCP/IP: DNS names/IP addresses/Ethernet hardware addresses

The 'logical interface' is implemented by a 'translator' that can translate between the two languages spoken on the two sides of the interface.

3.4 Centralized resource management

When resources such as the pool of logical unit numbers, access to some data file, etc are shared by several routines or processes, the natural way to eliminate access collisions is to have a centralized management.

3.5 Object-Oriented Programming

The idea behind OOP is (approximately) that sometimes it is more natural to partition a program to LOGICAL ENTITIES communicating by passing messages instead of FUNCTIONAL UNITS as in the modular paradigm.

A good example is a modern visualization program (e.g. SGI Explorer, or the popular AVS), in which you choose the required data processing modules from a menu, connect them together with some mouse clicks, and then tell the input module the name of your data file, and the whole thing starts producing pictures.

The natural conceptual model for this kind of magic is to view the program as a set of logical entities: the input reader module that reads the 3-dimensional data, and passes it to a module that may take a 2D section, then the graphic module translates the floating-point numbers into a colour picture where the colours correspond to the data values.

The logical entities used in OOP are a sweeping generalization of the data structure called 'record' in Pascal and 'structure' in C: an ordered collection of simpler data structures, not necessarily of the same type, that can be accessed individually by a name that consists of two parts, the structure name and the name of the 'field'.

The required generalization is made by allowing a "record" to also include procedures (usually called 'member functions') performing functions specific to the record type, for example:

Allocating memory for the internal variables and initializing the variables (constructor)

Maintaining the variables's values according to a pre-defined algorithm coded into the 'member procedures'.

Deallocating the memory storage when the "record" is no longer necessary (destructor)

The new data structure is suitable for implementing the abstract concept of an 'object'. In short, objects are data structures with associated procedures, which are used to maintain the data structure values in a consistent way.

Object-oriented programming can be directly expressed with suitable programming languages (e.g. Objective C, C++) that support 'objects'. Fortran 90 supports a few OOP concepts, and the rest can be done "by hand" if the programmer is disciplined enough.

3.6 Lookup Tables

Lookup tables are a simple programming trick to speed a special kind of calculations. If you have to compute again and again some function or expression and the possible values the argument(s) can take are relatively few you can do the computing once and put the results in an array. To retrieve results you just reference the array.

Self Assessment Exercise

1. Differentiate between caching and buffering

3.0 Brief History of Fortran Language

The programming language Fortran was originally designed for the solution of problems involving numerical computation. The development of Fortran dates back to the 1950s, the first Fortran system being released in 1957, for the IBM 704.

In the early 1960s, as other manufacturers released Fortran compilers for their own computer systems, the need to control and standardize Fortran became apparent. A standards committee was established in 1962, and the first Fortran standard was published in 1966.

Unfortunately, the 1966 Standard did not give a clear, precise definition of Fortran. In the 1970s a new standard was formulated to overcome the problems of Fortran 66 and incorporate several new features. In 1978, the new standard, Fortran 77, was published.

The standards preceding Fortran 90 attempted mainly to standardise existing extensions and practices. Fortran 90, however, is much more an attempt to develop the language, introducing new features using experience from other languages. A further revision of Fortran 90 brought in new features not existing in Fortran 90, and this led to Fortran 95.

3.1 Language Evolution

Fortran 90 is a superset of Fortran 77, and so all standard Fortran 77 programs should run. To prevent the language growing progressively larger, however, as new revisions are produced, the standards committee has adopted a policy of removing obsolete features. The following major new features are included in Fortran 90/95:

Array processing

Dynamic memory allocation, including dynamic arrays

Modules

Procedures:

- Optional/Keyword Parameters
- Internal Procedures
- Recursive Procedures

Pointers

Other new features include:

Free format source code

Specifications/IMPLICIT NONE

Parameterised data types

Derived types

Operator overloading

CASE statement

EXIT and CYCLE

Many new intrinsic functions

New I/O features

The new features allow the writing of more readable compact code, resulting in more understandable modular programs with increased functionality. Numerical portability is provided through selected precision, programming errors are reduced by the use of explicit interfaces to sub-programs, and memory is conserved by dynamic memory allocation. Additionally, data parallel capability is provided through the array processing features, which makes Fortran 90/95 a more efficient language on the new generation of high performance computers.

3.2 Coding Convention

The coding convention followed throughout the student notes is:

All keywords and intrinsic function names are in capitals; everything else is in lower case.

The bodies of program units are indented by two columns, as are INTERFACE blocks, DO-loops, IF-blocks, CASE-blocks, etc.

The name of a program, subroutine, or function is always included on its END statement.

In USE statements, the ONLY clause is used to document explicitly all entities which are actually accessed from that module.

In CALL statements and function references, argument keywords are always used for optional arguments.

Self Assessment Exercise

1. Where is Fortran derive its name?
2. What year is the Fortran Language originated?
3. Give different versions of Fortran Language and their major features

3.3 Sources and Types

3.3.1 Source Form

Fortran 90/95 supports two forms of source code; the old Fortran 77 source code form (now called fixed form), and the new free form. Using free source form, columns are no longer reserved and so Fortran statements can now appear anywhere on a source line. The source line may contain up to 132 characters.

The character set now includes both upper and lower case letters and the underscore. A good convention is that the words which are not in your control are in upper case and names which you invent yourselves, such as variable names, are in lower case.

Identifier names can consist of between 1 and 31 alphanumeric characters (including the underscore), the only restriction being that the first must be a letter. Remember that the use of sensible names for variables helps readability.

Fortran 90/95 introduces new symbols, including the exclamation mark, the ampersand, and the semicolon, and the alternative form of relational operators. These are discussed in the following paragraphs.

The exclamation mark introduces a comment. A comment can start anywhere on a source line and thus can be placed alongside the relevant code. The rest of the line after the ! is ignored by the compiler. For example:

```
REAL :: length1 ! Length at start in mm (room temperature)
REAL :: length2 ! Length at end in mm (after cooling)
```

The ampersand character, &, means `continued on the next line'. Usually you will arrange the line break to be in a sensible place (like between two terms of a complicated expression), and then all that is needed is the & at the end of all lines except the last. If you split a string, though, you also need an ampersand at the start of the continuation line.

```
a = b + &
c + d + e

WRITE (*,'NATIONAL OPEN UNIVERSITY
OF& & NIGERIA')
```

The semicolon is used as a statement separator, allowing multiple statements to appear on one line.

```
a = 2; b = 7; c = 3
```

Alternative forms of the relational operators are now provided:

.LT.	or	<
.LE.	or	<=
.EQ.	or	==
.NE.	or	/=
.GT.	or	>
.GE.	or	>=

3.3.2 Program and Subprogram Names

All programs and subprogram have names. A name can consist of up to 31 characters (letters, digits, or underscore), starting with a letter.

```
PROGRAM test
...
...
END PROGRAM test
```

where test is the name of the program. The same syntax applies for other program elements, such as FUNCTION or MODULE.

3.3.3 Specifications

Fortran 90/95 allows an extended form of declaration, in which all the attributes of a particular entity may be declared together. The general form of the declaration statement is:

```
type [ [, attribute ] ... :: ] entity list
```

where type represents one of the following:

```
INTEGER [([KIND=]kind-value)]
REAL [([KIND=]kind-value)]
COMPLEX [([KIND=]kind-value)]
CHARACTER [(actual-parameter-list)]
LOGICAL [([KIND=]kind-value)]
TYPE (type-name)
```

The attribute is one of the following:

PARAMETER, PUBLIC, PRIVATE, POINTER, TARGET, ALLOCATABLE, DIMENSION(extent-list), INTENT(inout), OPTIONAL, SAVE, EXTERNAL, INTRINSIC For example, it is now possible to initialize variables when they are declared, so there is no need for a separate DATA statement:

```
REAL :: a=2.61828, b=3.14159 ! two real variables declared and assigned initial values
INTEGER, PARAMETER :: n = 100, m = 1000 ! two integer constants declared and assigned values
CHARACTER (LEN=8) :: ch ! character string of length 8 declared
INTEGER, DIMENSION(-3:5,7) :: ia ! integer array declared with negative lower bound
INTEGER, DIMENSION(-3:5,7) :: ia, ib, ic(5,5) ! integer array declared using default dimension
```

3.3.4 Strong Typing

For backward compatibility, the implicit typing of integers and reals by the first character is carried over, but the IMPLICIT statement has been extended to include the parameter NONE. It is recommended that the statement

```
IMPLICIT NONE
```

be included in all program units. This switch off implicit typing and so all variables must be declared. This helps to catch errors at compile time when they are easier to correct. The IMPLICIT NONE statement may be preceded within a program unit only by USE and FORMAT statements.

3.4 Procedures and Modules

3.4.1 Program Units

Fortran 90/95 consists of the main program unit and external procedures as in Fortran 77, and additionally introduces internal procedures and modules and module procedures. A program must contain exactly one main program unit and any number of other program units (modules or external procedures).

A module exists to make some or all of the entities declared within it accessible to more than one program unit. A subprogram which is contained within a module is called a module procedure. A subprogram which is placed inside a module procedure, an *external procedure*, or a main program is called an *internal procedure*.

The following diagram illustrates the nesting of subprograms in program units:

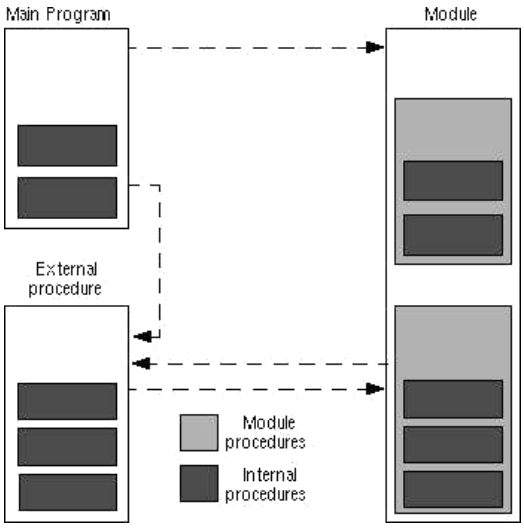


Fig. 2.1: Program Units

The form of the program units and procedures is summarised below.

Main program:

[PROGRAM program_name]
[specification-statements]
[executable-statements]
[CONTAINS
internal procedures]
END [PROGRAM [program_name]]

Module:

MODULE module_name
[specification-statements]
[executable-statements]
[CONTAINS
module procedures]
END [MODULE [module_name]]

External procedures:

[RECURSIVE] SUBROUITNE subroutine_name(dummy-argument-list)

[specification-statements]
[executable-statements]
[CONTAINS
internal procedures]
END [SUBROUTINE [subroutine-name]]

or

[type] [RECURSIVE] FUNCTION function_name &
(dummy-argument-list) [RESULT(result_name)]
[specification-statements]
[executable-statements]
[CONTAINS
internal procedures]
END [FUNCTION [function-name]]

Module procedures have exactly the same form as external procedures except that the word SUBROUTINE or FUNCTION must be present on the END statement.
Internal procedures also must have the word SUBROUTINE or FUNCTION present on the END statement:

[RECURSIVE] SUBROUTINE subroutine_name(dummy-argument-list)
[specification-statements]
[executable-statements]
END SUBROUTINE [subroutine_name]

[type] [RECURSIVE] FUNCTION function_name &
(dummy-argument-list) [RESULT (result_name)]
[specification-statements]
[executable-statements]
END FUNCTION [function_name]

3.4.2 Procedures

Procedures may be subroutines or functions. Self-contained sub-tasks should be written as procedures. A function returns a single value and does not usually alter the values of its arguments, whereas a subroutine can perform a more complicated task and return several results through its arguments.

Fortran 77 contained only external procedures, whereas in Fortran 90/95, structurally, procedures may be:

Internal - inside a program unit.
External - self contained (and not necessarily written in Fortran).
Module - contained within a module.

An Interface block is used to define the procedure argument details, and must always be used for external procedures.

3.4.3 Modules

A major new Fortran 90/95 feature is a new type of program unit called the module. The module is very powerful in communicating data between subprograms and in organising the overall architecture of a large program.

The module is important for both sharing data and sharing procedures (known as module procedures) between program units. Modules also provide a means of global access to entities such as derived type definitions and associated operators. A program may include several different modules, but they must all have different names. The form of a module is:

```
MODULE module-name
[specification-statements]
[executable-statements]
[CONTAINS
module-procedures]
END [MODULE [module-name]]
```

3.5 Fortran 90/95 Compilers and Compilation Procedures

The following compilers can be used to compile and execute a Fortran 90/95

programs. Lathey's Fortran 90 compiler

Microsoft Fortran PowerStation 4.0 professional

NAGware Fortran 90 compiler

Microsoft Developer Studio (MS-DEV) Tool

To code, compile and execute a Fortran 90/95 program using MS-DEV, the following steps are carried out:

Start MS-DEV if available on your computer

Start the editor and type in your program

- Click on File menu
- Select New File

- Click on BUILD menu
- Click on Compile
- Click on BUILD
- Click on Execute when you have compile successfully

The Fortran source codes are translated into machine language during compilation. If errors are detected, they are corrected by the programmer and re-compiled. Successful compilation and build will generate object code with the extension (.EXE). With the executable code (EXE) created, the program can be executed.

A very good first program to write, compile and execute, is one that writes a message to the screen. A program in Fortran 95 to do this might look like this:

```
! Writes a welcoming message onto the user's
screen PROGRAM hello
   IMPLICIT NONE WRITE(*,*)
   'hello world' END
PROGRAM hello
```

You can see that programs contain words that are special to Fortran (i.e. PROGRAM, IMPLICIT, WRITE, END) and some that are chosen by the programmer (in the above program there are two of these, the program name hello, and the output text hello world).

We have chosen to indicate Fortran 95 commands using CAPITAL letters and words chosen by the programmer using lower case letters. You do not have to do this, though it can make the program easier to read.

Note that text following a '!' is treated as a comment. A comment does not form part of the program proper, i.e. the text is ignored by the computer. It is good programming practice to include a comment at the start of your program explaining what the program is intended to do.

Comments are a very important part of a program even though they are not used by the computer to compile the program. They are crucial for you and other programmers to understand how you have organised your programs, and how they run. You should get in the habit of including comments in your programs now.

Note that Fortran statements must usually appear on a single line, and not be broken across lines. Also, Fortran only allows each line to be a maximum of 132 characters long. If you wish to write a statement longer than this, or you simply want to break a statement over multiple lines, you should use the & symbol as explained earlier.

A useful way to make sure that your program is readable and easy to understand is to indent sections of the code

Self Assessment Exercise

1. Introduce yourself to any Fortran 90/95 Compiler environment of your choice
2. As an exercise, learners should code, compile and run the above program with the output ‘Introduction to Fortran Programming’.
3. What do you understand by the terms: internal and external procedures

3.0 Data types and simple calculations

Obviously, one of the most common tasks of a computer program is to perform calculations. To do this, the programmer must define variables. In mathematics, variables are usually single characters, such as x or t or i etc.

When programming, however, you have the flexibility to give your variables names which reflect their meaning. So, you could use x_coordinate or time or radius or counter. You should not use Fortran 95 reserved words (words that are Fortran 95 commands such as "program", "end" etc.) and you cannot leave spaces (which is why "x_coordinate" has an underscore; "x coordinate" could not be used).

Variables can be of the following type:

REAL (a number with a decimal point, such as 3.142, 1.0, -0.002)

INTEGER (a whole number, without a decimal point, such as 0, 5654, -26472)

COMPLEX (a complex number (whose real and imaginary parts are REAL, not INTEGER))

CHARACTER (a string of characters)

LOGICAL (can take the values .TRUE. or .FALSE. only)

The user has to declare all variables that will be used in the program at the beginning of the program. This allows the computer to allocate sufficient memory for each variable and also to know what operations it can expect each variable to be used for. For example, the computer cannot do mathematical operations on a character string. Note that in the list above, 1.0 was listed as a REAL variable because it has a decimal point (even though the numerical value of the variable is an integer). REAL variables can hold integer values, but not vice versa.

We will focus on the REAL and INTEGER variables as these are most commonly used. Variables are defined as REAL or INTEGER immediately after the IMPLICIT NONE command, thus,

```
PROGRAM reals_and_integers
 IMPLICIT NONE
 REAL :: x, radius, sum, error=1.0e-5
 REAL, PARAMETER :: pi = 3.14159265
 INTEGER :: i, counter=0
 WRITE(6,*) pi,error,counter
END PROGRAM reals_and_integers
```

Note that the initial values of some variables have been specified at the same time as defining them as REAL or INTEGER. In addition, one of these variables, pi has the extra word PARAMETER in its definition. This tells the compiler that you in fact want this "variable" never to vary. The code will not run any differently, but the compiler will warn you if you attempt to vary pi later on that you set it to be a constant. You should use it in the declaration of variables that you want really to be constants, so that you get this extra error checking from the compiler.

Note also the format used for numbers multiplied by some power of 10. The quantity "error" is equal to 1.0×10-5, or 0.00001

Finally, look at the WRITE statement. The values of the variables will be printed to the screen so you will see 3.1415927 1.0E-5 0

The two data types INTEGER and REAL may be used in numeric expressions, but care must be used to ensure that the types of the variables match, or that the types are converted as you want them. The rule is that an operation (e.g. addition, division ...) between two variables of the same type (e.g. both REAL or both INTEGER) will always return a result of that same type. When an operation takes place between a REAL and an INTEGER, the INTEGER is first converted to a REAL equivalent, and the final result will also be REAL.

Now, let us do some simple calculations. The Fortran 90/95 commands are +,-,*,/ for addition, subtraction, multiplication and divide, respectively. The command ** is used to raise to a power, thus 2**4 is 2 raised to the power of 4 and is equal to 16.

Precedence

What do you think the result of the expression 1+2/3+4 should be? If the operations take place from left to right in order, then 1 and 2 would be added to give 3, this would then be divided by 3 to give 1, which would be added to 4 to give 5. In fact, the answer is 5 2/3. This is because division takes precedence over addition and is evaluated first. The order of precedence is:

Operator	**Precedence**
**	High
*,/	Medium
+,-	Low

Within the same level of precedence, the operations proceed left to right, except a series of ** operations, which takes place right to left.You can change the order of evaluation of the operations using brackets, so (1+2)/(3+4) gives the result 3/7. Consider the following program.

```
PROGRAM calculations
 IMPLICIT none
 REAL :: a, b, c
 INTEGER :: i, j, k

 i = 1  ! assigns the integer value 1 to the integer variable i
 j = 2  ! assigns the integer value 2 to the integer variable j
 j = i+j          ! assigns the value of i + j to the integer variable j
 k = i/j          ! assigns the integer value i/j (=0) to the integer variable k
 a = 1.0; b=2.0          ! assigns values to the real variables a and b.
 c = 2.0+a/b  ! note that a/b is executed first. A value 2.5 is assigned to c.
 a = a/2.0*b  ! Careful, this is (a/2.0)*b not a/(2.0*b). a retains the value 1.0.
 b = i/j          ! two integers will truncate to integer before assigning to real variable.
```

```
WRITE(*,*) i,j,k
WRITE(*,*) a,b,c

END PROGRAM calculations
```

Note the semi-colon which allows you to put several commands on the one line.
Note also that the equals sign (=) means "assign the value of the expression on the right hand side to the variable on the left" and not quite the same as the mathematical equality symbol. You can't, for example, write x**2 + 2*x + 1 = 0 in Fortran and expect it to solve the equation. Calculations like j=i+j look a little strange at first. The computer calculates the right-hand side of the equal sign and whatever value is obtained is assigned the variable on the left of the equals sign.

Self Assessment Exercise

1. Using a piece of paper, work through the above program and WRITE DOWN the values that YOU think will be written to the screen by the WRITE(*,*) statements. Copy this program into the Fortran editor, compile and run. Were you correct?
2. Which of the following numbers are integers (in a Fortran sense)?
   (a) -34 (b) 1.0 (c) 0 (d) 1.0E+01 (e) 54635
3. If i and j are defined as INTEGER, which of the following Fortran calculations will yield an integer value?
   (a) i*j (b) 2.0*i*j (c) i/j (d) i**2 (e) (i/4.0 + j)

4. If i and j are defined as INTEGER, which of the following Fortran calculations will yield a real value?
   (a) i**0.5 (b) i/j (c) i-j+1 (d) i+j+0.5 (e) i**j

5. What is the value of the calculation (1/2)*(3.0+5.0)?
   (a) 4.0 (b) 4 (c) 0 (d) 0.0

Practical Exercise 1: Using Integer and Real

Make sure you have read section 3.0 before attempting the following exercise.

Consider the following program.

```
 PROGRAM using_integers_and_reals
   IMPLICIT none
   REAL :: a, b, c, d, e, f, result, error
   INTEGER :: i, j, k, l, m
   i = 1; j = 2
   a = 3.0; b = 2.0
   result = (a+b) * (a/b - 3.0)
   j = j + 1
   k = I / j
   l = j + j / 2
   m = I ** j
   c = a / 2.0 * b
   d = I / j
   error = 10**(-3)
   e = REAL(i)/REAL(j)
   f = 3.0e-4 * b - 2.0e-4 * a
   WRITE (*,*) 'result = ', result
   WRITE (*,*) 'j = ', j
   WRITE (*,*) 'k = ', k
   WRITE (*,*) 'l = ', l
   WRITE (*,*) 'm = ', m
   WRITE (*,*) 'c = ', c
   WRITE (*,*) 'd = ', d
   WRITE (*,*) 'error = ', error
   WRITE (*,*) 'e = ', e
   WRITE (*,*) 'f = ', f
 END PROGRAM using_integers_and_reals
```

Your task, in this exercise, is not to write a program, but to interpret an existing program. On a piece of paper, write down the values of all the variables that will be printed on the screen.

Once you have done this, copy this program, compile and run it. How many answers did you get correct? Give yourself a mark out of 10!

Suppose you write a program to convert centimetres into inches. You will arrange the program to allow you to enter the number of centimetres at the keyboard every time the program is run. This is achieved using the READ statement. The following program will convert centimetres to inches.

```
! This program converts centimetres to
inches PROGRAM cm_to_ins
 IMPLICIT  none
 REAL :: cm, ins
 WRITE(6,*) 'Please enter the number of
 centimetres' WRITE(6,*) 'to be converted to
 inches' READ(5,*) cm
 ins = cm/2.54
 WRITE(6,*) cm,' cm = ',ins,'
inches' END PROGRAM cm_to_ins
```

In the expression WRITE(6,*), the number 6 is predefined in Fortran to mean "write to the screen", and in READ(5,*) the 5 means "read from the keyboard". You can set up different numbers to refer to files on the computer. Look at the following version of the centimetres-to-inches conversion program.

```
! This program converts centimetres to
inches PROGRAM cm_to_ins
 IMPLICIT  none
 REAL :: cm, ins
 OPEN(10, file='cm_to_ins.out')
 WRITE(6,*)' Please enter the number of
 centimetres' WRITE(6,*)' to be converted to
 inches' READ(5,*)cm
 ins = cm/2.54
 WRITE(10,*)cm,' cm = ',ins,'
 inches' CLOSE(10)
END PROGRAM cm_to_ins
```

Look at the final WRITE statement. The only change is that "6" has been replaced by "10". This allows the information to be written to a file rather than a screen. But the name of the file must be defined. This is achieved using the OPEN command.

If you run this version of the conversion program, no output will appear on the screen. The output has been written to the file called cm_to_ins.out. You can check this by using text editor to view cm_to_ins.out from the screen. You will see (for example),

50.0 cm = 19.68504 inches

Any physicist should know that it is not justified to present this result to 7 significant figures! The computer does not know this however. You can modify, or FORMAT, the output and you will see how this can be achieved in the subsequent section.

Practical Exercise 2: Fahrenheit to Celsius

Write a Fortran 95 program to convert a temperature in fahrenheit to its equivalent value in celsius. Your program should READ the fahrenheit value from the terminal and output the celsius value to the screen. Make sure you include appropriate comments.

The temperature in celsius is equal to (5/9) × (F-32) where F is the temperature in fahrenheit.

DOES YOUR PROGRAM WORK?

Test your program by converting the following temperatures in Fahrenheit to Celsius.

82.0

212.0

Check the output and see if the answers are

27.8

100.0

If you get the wrong answers, debug, compile and rerun.

3.2 The Input Statement

The Input statement reads one or more values from an input device, and stores them into variables specified by the programmer. In the process of data input, the data read are stored by the program in the main memory.

General form of READ statement syntax is:

READ (*,*) input list

Example:

```
PROGRAM StudentGPA
IMPLICIT NONE
REAL :: gpa
INTEGER :: Matric
CHARACTER (Len = 25) :: Name
READ (*,*) Matric, Name, gpa
STOP
END PROGRAM
```

In the example, data to be supplied to the computer are Integer type Matric number; 25 Character length Name; Real type gpa. The order of the values to be supplied must also match the order of the variables i.e. the input list: Matric, Name and gpa.

3.3 The Output Statement

The output statement causes the processed data to be sent to appropriate output device. Examples of such devices are screen (computer monitor) or console and printer. Sometimes PRINT statement is used instead of WRITE statement to exclusively print to the screen. The syntax for the general form of WRITE statement is:

```
WRITE (*, *) output list
```

Example:

```
PROGRAM StudentGPA
IMPLICIT NONE
REAL :: gpa
INTEGER :: Matric
CHARACTER (Len = 25) :: Name
READ (*,*) Matric, Name, gpa
WRITE (*,*) Matric, Name, gpa
STOP
END PROGRAM
```

3.3.1 PRINT statements

The Print statement is mainly used to generate output to the screen or console or computer monitor. They are also used as users prompts for a response in the cause of executing a program.

The general form of PRINT syntax is:

PRINT *, output list

Example:

```
PRINT *, A, B, C
PRINT *, Matric, Name, gpa
PRINT *, 'Enter Yes (Y) or No (N)'
```

3.4 Formatted Input/Output Statements

When the type of information and the location are needed in the input/output list, then the FORMAT statement is used to accompany the READ and the WRITE statements. The general form is:

READ (unit, format label) input list

Or

WRITE (unit, format label) output list

Example:

```
READ (5,10) a, b, c, d
10 FORMAT (4F5.2)
```

Or

```
WRITE (6, 20) sum, mean
20 FORMAT (1H, 2F10.2)
```

The READ statement in the example is linked with the FORMAT statement labelled 10 (i.e. format label). The format descriptor "4F5.2" means that 4 correspond to the number of variable to be read; The F specified that these variables are floating points or real numbers; The number 5.2 indicates allocated spaces including the decimal part that each real numbers (a, b, c, d) will occupy. In this case, the total size is 5 including the decimal point and the decimal place is 2. Suppose a = 6.1, then the descriptor will make a = 006.1 and so are the other variables.

In a nut shell, 4F5.2 is described as follows:

4 - values to be read

F - for floating point or real number

5 - size of the value including the decimal point

2 - number of decimal places

20 - format statement label

1H - takes a new header line before the output

2 - number of output list i.e. sum and mean

F - float or real number specification

10 - size or number of columns of the output

2 - decimal places

3.4.1 FORMAT Field Descriptors

The following are the field descriptors that are used along with the FORMAT statements:

I - Integers

F - Real (Floating Point)

E - Exponential

G - Large or too small Real or Exponential

A - Alphanumeric

X - Skipping fields

H - Hollerith, skipping lines and pages

/ -Slash for printing blank lines

T - indicates starting columns to read data The

syntax for using the field descriptors for Integer is:

Integer:nIw

Real:nFw.d

Exponential:nEw.d

Alphanumeric:Aw

Skipping Field: wX

Slash: no of slash indicate number of blank lines

where n = optional number of variables; w = width or size of columns to be occupied by each variable type; d = number of decimal places.

3.5 Using Files: Open and Close Statements

Fortran statements used to control the disk file for input and output include OPEN, CLOSE, READ, WRITE, REWIND, BACKSPACE commands.

3.5.1 The OPEN Statement

The form is:

OPEN (open list separated by commas)

Where open list comprises of some clauses specifying the unit number, the file name and information on how to access the file. Examples are as follows:

To open a file for Input:

OPEN (UNIT = 8, FILE = 'datain.dat', STATUS = 'OLD', ACTION = 'READ')

The unit 8 specifies the I/O unit number associated with the file 'datain.dat'. The Status='old' specifies that the file already exist. The action means read only.

To open file for Output:

OPEN(UNIT = 10, FILE = 'datout.dat',STATUS = 'NEW', ACTION='WRITE')

In case of avoiding overwriting an existing file, STATUS = 'REPLACE' is used.

To open a scratch file:

OPEN (UNIT = 12, STATUS = 'SCRATCH', IOSTAT = ierror)

A temporary file is created that is automatically deleted when the file is closed or at program termination. The IOSTAT returns error code in variable ierror if there is error. It is also used for other OPEN methods.

3.5.2 The CLOSE Statements

The Close statement closes an opened file and releases the I/O unit number associated with it. The format is

CLOSE (close list)

Even if CLOSE command is not specified in a program, file closed automatically when the program terminates. The close list should specify the I/O number and other associated clauses.

3.5.3 Data and Parameter Statement

The DATA statement within the program is used to initialize variables at compile time.
The format is

```
DATA variable list/constants/
```

Example

```
PROGRAM Sumval
IMPLICIT NONE
INTEGER :: a, b , c, sum
DATA a, b, c/4, 4, 9/
SUM = a + b + c
WRITE (*, *) sum
END Sumval
```

The DATA statement can be written as DATA a, b, c/2 * 4, 9/. The 2 * 4 indicates that there are 2 sets of figure 4 in data list.

3.5.4 File Positioning

The order of accessing file records can be changed using the BACKSPACE and REWIND statements. Ordinarily, Fortran reads file records from the first to the last sequentially. The BACKSPACE statement moves back one record each time it is called. REWIND statement restarts the file reading at the beginning each time it is encountered. The general form of these two statements are:

```
BACKSPACE (UNIT = unit no)
REWIND (UNIT = unit no)
```

The unit no is the number associated with the file to be read or accessed.

Self Assessment Exercise

1. What is the field descriptor to format large real numbers
2. Which clause in the OPEN statement that reports error code if error is encountered.

3.0 Control Statements

At the initiation of a control statement control is immediately transferred to another part of the program to execute some tasks, and thereafter continue in its sequential execution to the end of the program unless otherwise stated by another control statement. Some of these control statements and program loops include:

Block IF

ELSE and ELSE IF

The CASE

WHILE Loop/DO Loop

3.1 The Block IF Statement

IF ... END IF conditional statements are used for conditional execution of lines of code. The format is

IF (logical expression) THEN

    Statements in block

END IF

There are various forms of this construct, but the simplest is as shown in the following example.

    IF (i == 2) sum = 0.0

The statement is only executed if the expression in brackets is .TRUE.. In the example, this means that sum = 0.0 only if the value of i is equal to 2. If i is not equal to 2, no action is taken. Note the use of "==" and not "=" in the condition. "==" is called a relational operator. The full list of relational operators is below:

Command	**Meaning**
a < b	true if a is less than b
a <= b	true if a is less than or equal to b
a > b	true if a is greater than b
a >= b	true if a is greater than or equal to b
a == b	true if a is equal to b
a /= b	true if a is not equal to b

There are also logical operators of which .OR. and .AND. are the most useful. Their use is easily understood, for example,

```
IF (i == 2 .OR. i <= 0) sum = 0.0
```

will set sum = 0.0 when i is negative, 0 or 2.

Suppose you want to execute, not one, but several statements if a certain condition is satisfied. You may use, for example,

```
IF (i == 1) THEN
   sum = 0.0
   i = i + 1
END IF
```

3.2 The ELSE and ELSE IF

This is useful when more than one statement is true in order to execute one set of statement. The format is

```
IF (logical expression1) THEN
        Statement1 block
ELSE IF (logical expression2) THEN
        Statement2 block
ELSE
        Statement3 block
END IF
```

If logical expression1 is true the program executes the Statement1 block and skips to the next statement after ENDIF. Otherwise, the program checks the Statement2 block if logical expression2 is true else Statement3 is executed.

You could make the IF statement even more powerful by using the ELSE ELSEIF construct. For example

```
IF (i < 0 ) THEN
   sign = -1
ELSE IF (i > 0) THEN
   sign = 1
ELSE
   sign = 0
END IF
```

The statements in the first part are only executed if the first conditional expression is true. Statements in the second part are only executed if the first conditional statement is false and the second conditional expression is true. Otherwise, the third set of statements is executed. This example, produces a value of sign of -1, 1 or 0 depending on whether i is negative, positive or zero, respectively.

3.3 Named Block IF

The general form of the IF construct is:

```
[name:] IF (logical expression) THEN
        [block]
[ELSE IF (logical expression) THEN [name]
        [block]
[ELSE [name]
        [block]
END IF [name]
```

Notice there is one minor extension, which is that the IF construct may be named. The ELSE or ELSE IF parts may optionally be named, but, if either is, then the IF and END IF statements must also be named (with the same name). For example

```
select: IF (i < 0) THEN
```

```
CALL negative
ELSE IF (i==0) THEN select
CALL zero
ELSE select
CALL positive
END IF select
```

For long or nested code this can improve readability.

Self Assessment Exercise

4. What control structure is suitable for executing one set of statement if more than one statement is true

3.4 CASE Construct

Repeated IF ... THEN ... ELSE constructs can be replaced by a CASE construct. The general form of the CASE construct is:

```
[name:] SELECT CASE (expression)
[Case (selector)[name]
block]
.
.
END SELECT [name]
```

The expression can be of type INTEGER, LOGICAL, or CHARACTER, and the selectors must not overlap. If a valid selector is found, the corresponding statements are executed and control then passes to the END SELECT. If no valid selector is found, execution continues with the first statement after END SELECT.

For example

```
SELECT CASE (day) ! sunday = 0, monday = 1, etc
CASE (0)
extrashift = .TRUE.
CALL weekend
CASE (6)
extrashift = .FALSE.
CALL weekend
```

```
CASE DEFAULT
extrashift = .FALSE.
CALL weekday
END SELECT
```

The CASE DEFAULT clause is optional and covers all other possible values of the expression not included in the other selectors. It need not necessarily come at the end. A colon may be used to specify a range, as in:

```
CASE ('a':'h','o':'z')
```

which will test for letters in the ranges a to h and o to z.

3.5 WHILE Loop/DO Loop

A WHILE loop is a block of statements that are repeated indefinitely as long as the control clause is still valid or satisfied.

The general form of the DO loop is:

```
[name:] DO [control clause]
block
END DO [name]
```

The END DO statement should be used to terminate a DO loop. This makes programs much more readable than using a labelled CONTINUE statement, and, as it applies to one loop only, avoids the possible confusion caused by nested DO loops terminating on the same CONTINUE statement.

For example

```
DO i = 1,n
DO j = 1,m
    a(i,j) = i + j
END DO
END DO
```

Notice that there is no need for the statement label at all. It is often desirable to execute the same type of calculation many times. One way of doing this is through the controlled DO loop.

The most common form of the DO loop is shown in the example below:

```
 DO i=1,9
```

...

... END

DO

The action of the DO loop in this example is as follows:

Command	**Action**	**Value of i**
DO i=1,9	i is set equal to 1. Is i > 9? No	i=1
...	statements are executed	i=1
END DO	i has 1 added to it	i=2
DO i=1,9	Is i > 9? No	i=2
...	statements are executed	i=2
END DO	i has 1 added to it	i=3
DO i=1,9	Is i > 9? No	i=3
...	etc. etc	...
...	etc. etc	...
...	statements are executed	i=9
END DO	i has 1 added to it	i=10
DO i=1,9	Is i > 9? Yes Go to first statement after END DO	i=10

In the above example, i takes on the values 1,2,3,4,5,6,7,8,9 and the statements in the loop are executed for each value of i. When the loop has finished, the value of i in this example is 10, and the program continues running after the END DO statement.

Consider the following simple program which finds the product of a set of numbers entered at the keyboard.

```
! This program finds the product of a set
! of numbers entered at the
keyboard. PROGRAM
calculate_product IMPLICIT none
 REAL :: value, product=1.0
 INTEGER :: i, n
 WRITE(6,*) ' This program evaluates the
 product ' WRITE(6,*) ' of n real numbers '
 WRITE(6,*) ' '
 WRITE(6,*) ' Enter the value of n'
```

```
READ(5,*)n
DO i=1,n
  WRITE(6,*) ' Enter value number',i
  READ(5,*) value
  product = value*product
  WRITE(6,*) ' i= ',i,' The product is',product
 END DO
END PROGRAM calculate_product
```

In the DO loop, the value of i starts at 1 and increases in steps of 1 to a maximum value of n. After this, DO loop control moves to the first statement after the END DO (in this case the END statement).

Copy, compile and run the program and check that you understand the values of i and product printed to the screen.

A DO-loop can calculate a triangular number as follows:

```
PROGRAM triangular
  IMPLICIT NONE
  INTEGER :: tri=0, n, i
  WRITE(6,*) "Enter n to calculate the nth triangular number:"
  READ(5,*) n
  DO i=1,n
  tri=tri+i
  END DO
  WRITE(6,*) n,"-th triangular number is ",tri
END PROGRAM triangular
```

By default a DO-loop will always increase its index by one every time through the loop. If you want to change this, or to make it decrease, you give a third number, which is the increment (or decrement) like this:

```
DO i=1,9,2
   ...
```

```
    ...
  END DO
```

This is identical to the previous example except that the DO statement is DO i=1,9,2 rather than DO i=1,9. The presence of the "2" simply means that i increases in increments of 2, and so i=1,3,5,7,9 in this case.

Practical Exercise: Finding an average

Write a Fortran 90/95 program which calculates the average of a set of n numbers entered at the keyboard. Your program should READ the value of n from the terminal, execute a DO loop to determine the average and output the average value to the screen. Make sure you include appropriate comments.

DOES YOUR PROGRAM WORK?

Test your program by averaging the following 5 numbers

8.0

9.0

0.5

-1.0

2.8

Check the answer. The answer is

3.86

3.5.1 Named DO and END DO

The DO and END DO may be named.

For example

```
rows: DO i = 1,n
cols: DO j = 1,m
      a(1,j) = i + j
END DO cols
END DO rows
```

One point to note is that the loop variable must be an integer and it must be a simple variable, not an array element.

The DO loop has three possible control clauses: an iteration control clause (as in example above); a WHILE control clause (described below); an empty control clause ("EXIT and CYCLE")

3.5.2 DO WHILE

A DO construct may be headed with a DO WHILE statement:

The form is

```
DO WHILE (logical expression)
   body of loop
END DO
```

The body of the loop will contain some means of escape, usually by modifying some variable involved in the test in the DO WHILE line.

For example

```
DO WHILE (diff > tol)
..
.
diff = ABS(old - new)
..
.
END DO
```

Note that the same effect can be achieved using the DO loop with an EXIT statement which is described below.

3.5.3 EXIT and CYCLE

The EXIT statement permits a quick and easy exit from a loop before the END DO is reached. The CYCLE statement is used to skip the rest of the loop and start again at the top with the test-for-completion and the next increment value.

Thus, EXIT transfers control to the statement following the END DO, whereas CYCLE transfers control to a notional dummy statement immediately preceding the END DO.

These two statements allow us to simplify the DO statement even further to the `do forever' loop:

```
DO
.
..
IF ( ... ) EXIT
.
..
END DO
```

Notice that this form can have the same effect as a DO WHILE loop.

By default the CYCLE statement applies to the inner loop if the loops are nested, but, as the DO loop may be named, the CYCLE statement may cycle more than one level. Similarly, the EXIT statement can specify the name of the loop from which the exit should be taken, if loops are nested, the default being the innermost loop.

```
outer: DO i = 1,n
middle: DO j = 1,m
inner: DO k = 1,l
.
..
IF (a(i,j,k)<0) EXIT outer ! Leave loops
IF (j==5) CYCLE middle ! Omit j==5 and set j=6
IF (i==5) CYCLE ! Skip rest of inner loop, and
. ! go to next iteration of
. ! inner loop
.
END DO inner
END DO middle
END DO outer
```

Self Assessment Exercise

1. Write an ELSE and ELSE IF construct for Examination grading system (A = 70 and above; B = 60 to 69; C = 50 to 59; D = 45 to 49; E = 40 to 44; F = below 40)

6.0 Tutor Marked Assignment

1. Write a program to quadratic equation using the ELSE IF statement to differentiate the roots
2. Use DO loop to find the sum of 100 natural numbers.

Module 2: FORTRAN Programming Language

Unit 4: Arrays and Subscripted Variables

3.0 Subscripted Variables (Arrays)

Suppose you want to do a calculation on a quantity x, not once, but for 100 different values of x. obviously you would not read these numbers at the keyboard, it would take forever! You would arrange for the 100 values of x to be read in from a file by the program. But what do you call these values of x? Do you define variables in your program as x1, x2, x3, ... x100? Suppose you had 1000 values of x instead of 100. Clearly defining variables in this way is clumsy. We get around this problem by the use of ARRAYS. Arrays allow multiple values to be stored with one variable name. In the case above, the array variable would just be x. This array variable has an index that allows each element of the array to be accessed individually. So the first value of x is x(1), the second value is x(2), the 100th value is x(100) and the ith value is x(i).

The computer needs to know how many variables an array variable is to contain. In the above example, array variable x is to contain 100 values. This is achieved through the DIMENSION statement. Consider the first example which demonstrates a 5 element real array (indices from 1 to 5).

```
PROGRAM array1
 IMPLICIT none
 REAL, DIMENSION(1:5) :: a
 INTEGER :: i
```

```
DO i = 1,5
a(i) = i * i
END DO
DO i = 1,5
  WRITE(6,*) a(i)
END DO
END PROGRAM array1
```

The numbers written to the screen are:

1.

4.

9.

16.

25.

so that a(1)=1.0, a(2)=4.0, ..., a(5)=25.0.

3.1 Representation of Arrays

Arrays are represented in one or two-dimensional array and more. Example of two-dimensional array of numbers is a matrix.

Example of One-dimensional array format is

S(1), S(2), S(3), …., S(N)

Where S is the array name and 1, 2, 3, .., N are the subscripts. S(1), S(2), .. S(N) are the elements of the array.

Example of Two-dimensional array format is

```
A(1,1)    A(1,2) ……   A(1,N)
A(2,1)    A(2,2) …….  A(2,N)
:         :           :
A(M,1)    A(M,2) ……   A(M,N)
```

In A(2,1), 2 indicates the row and 1 indicates the column. The above example is M by N two-dimensional array A(M,N) where m and n are integer number. A three-dimensional array is represented by A(X,Y,Z). The fortran limit is 7-dimensions.

3.2 Array Terminology and Specifications

Fortran permits an array to have up to seven subscripts, each of which relates to one dimension of the array. The dimensions of an array may be specified using either a dimension attribute or an array specification. By default the array dimensions start at 1, but a different range of values may be specified by providing a lower bound and an upper bound. For example,

```
REAL, DIMENSION(50) :: w
REAL, DIMENSION(5:54) :: x
REAL y(50)
REAL z(11:60)
```

Here, w, x, y and z are all arrays containing 50 elements.

The rank of an array is the number of dimensions. Thus, a scalar has rank 0, a vector has rank 1 and a matrix has rank 2. The extent refers to a particular dimension, and is the number of elements in that dimension. The shape of an array is a vector consisting of the extent of each dimension. The size of an array is the total number of elements which make up the array. This may be zero. Two arrays are said to be conformable if they have the same shape. All arrays are conformable with a scalar.

The general form of an array specification is as follows:

```
type [[,DIMENSION (extent-list)] [,attribute]... ::] entity list
```

Here, type can be any intrinsic type or a derived type (so long as the derived type definition is accessible to the program unit declaring the array). DIMENSION is optional and defines default dimensions in the extent-list, these can alternatively by defined in the entity list. The extent-list gives the array dimensions as:

integer constants

integer expressions using dummy arguments or constants

':' to show the array is allocatable or assumed shape

As before, attribute can be any one of the following

PARAMETER, PUBLIC, PRIVATE, POINTER, TARGET, ALLOCATABLE, DIMENSION(extent-list), INTENT(inout), OPTIONAL, SAVE, EXTERNAL, INTRINSIC

Finally, the entity list is a list of array names with optional dimensions and initial values.

The following examples show the form of the declaration of several kinds of arrays, some of which are new to Fortran 90/95:

Initialisation of one-dimensional arrays containing 3 elements:

```
INTEGER, DIMENSION(3) :: ia=(/1,2,3/), ib=(/(i,i=1,3)/)
```

Declaration of automatic array logb. Here loga is a dummy array argument, and SIZE is an intrinsic function which returns a scalar default integer corresponding to the size of the array loga:

```
LOGICAL, DIMENSION(SIZE(loga)) :: logb
```

Declaration of dynamic (allocatable) arrays a and b. The dimensions would be defined in a subsequent ALLOCATE statement:

```
REAL, DIMENSION (:,:), ALLOCATABLE :: a,b
```

Declaration of assumed shape arrays a and b. The dimensions would be taken from the actual calling routine:

```
REAL, DIMENSION(:,:,:) :: a,b
```

3.3 Array Operations

In Fortran 77 it was not possible to work with whole arrays, instead each element of an array had to be operated on separately, often requiring the use of nested DO-loops. When dealing with large arrays, such operations could be very time consuming and furthermore the required code was very difficult to read and interpret. An important new feature in Fortran 90/95 is the ability to perform whole array operations, enabling an array to be treated as a single object and removing the need for complicated, unreadable DO-loops.

In order for whole array operations to be performed, the arrays concerned must be conformable. Remember, that for two arrays to be conformable they must have the same shape, and any array is conformable with a scalar. Operations between two conformable arrays are carried out on an element by element basis, and all intrinsic operations are defined between two such arrays.

For example, if a and b are both 2x3 arrays

$$a = \begin{bmatrix} 3 & 4 & 8 \\ 5 & 6 & 6 \end{bmatrix}, \; b = \begin{bmatrix} 5 & 2 & 1 \\ 3 & 3 & 1 \end{bmatrix}$$

the result of addition is

$$a + b = \begin{bmatrix} 8 & 6 & 9 \\ 8 & 9 & 7 \end{bmatrix}$$

and of multiplication is

a x b =

If one of the operands is a scalar, then the scalar is broadcast into an array which is conformable with the other operand. Thus, the result of adding 5 to b is

$$b + 5 = \begin{bmatrix} 5 & 2 & 1 \\ 3 & 3 & 1 \end{bmatrix} + \begin{bmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \end{bmatrix} = \begin{bmatrix} 10 & 7 & 6 \\ 8 & 8 & 6 \end{bmatrix}$$

Such broadcasting of scalars is useful when initialising arrays and scaling arrays.

An important concept regarding array-valued assignment is that the right hand side evaluation is computed before any assignment takes place. This is of relevance when an array appears in both the left and right hand side of an assignment. If this were not the case, then elements in the right hand side array may be affected before the operation was complete.

The advantage of whole array processing can best be seen by comparing examples of Fortran 77 and Fortran 90/95 code:

Consider three one-dimensional arrays all of the same length. Assign all the elements of a to be zero, then perform the assignment a(i) = a(i)/3.1 + b(i)*SQRT(c(i)) for all i. Fortran 77 Solution

```
REAL a(20), b(20), c(20)
...
DO 10 i=1,20
a(i)=0
10 CONTINUE
...
DO 20 i=1,20
a(i)=a(i)/3.1 + b(i)*SQRT(c(i))
20 CONTINUE
```

Fortran 90/95 Solution

```
REAL, DIMENSION(20) :: a, b, c
...
a=0
...
a=a/3.1+b*SQRT(c)
```

Note, the intrinsic function SQRT operates on each element of the array c.

Consider three two-dimensional arrays of the same shape. Multiply two of the arrays element by element and assign the result to the third array.

Fortran 77 Solution

```
REAL a(5, 5), b(5, 5), c(5, 5)
...
DO 20 i = 1, 5
DO 10 j = 1, 5
c(j, i) = a(j, i) * b(j, i)
10 CONTINUE
20 CONTINUE
```

Fortran 90/95 Solution

```
REAL, DIMENSION (5, 5) :: a, b, c
...
c = a * b
```

Consider a three-dimensional array. Find the maximum value less than 1000 in this array. In Fortran 77 this requires triple DO loop and IF statements, whereas the Fortran 90/95 code is:

```
REAL, DIMENSION(10,10,10) :: a
amax=MAXVAL(a,MASK=(a<1000))
```

Find the average value greater than 3000 in an array.

In Fortran 77 this requires DO loops and IF statements, whereas Fortran 90/95 code is:

```
av=SUM(a,MASK=(a>3000))/COUNT(MASK=(a>3000))
```

Note in the last two examples the use of the following array intrinsic functions:

MAXVAL - returns the maximum array element value.

SUM - returns the sum of the array elements.

COUNT - returns the number of true array elements.

3.4 Other Fortran 90/95 Array Commands

Fortran 90/95 has some useful array commands. For example, suppose you want to initialise an array by making every element equal to 0.0 (a common task). You could use the following command, a = 0.0 or if you wish to put the first 5 elements of the array to zero, i.e. a(1)=0.0, a(2)=0.0, ..., a(5)=0.0, you could put a(1:5) = 0.0

A large number of mathematical operations can be performed on each array element by referring to the whole array, so if a, b and c are all REAL arrays of the same size and shape you can write things like

```
a = b**2 ! each element of b is squared and assigned to the elements of
```

```
a = b/c ! each element of b is divided by corresponding elements in c
        ! and assigned to same elements in a
a = exp(b) ! each element of b is exponentiated and assigned to
           the ! corresponding elements in a
```

The array manipulation powers of Fortran 90/95 are rather powerful. See more examples.

A matrix is a two-dimensional array. In Fortran 90/95, a two-dimensional array can be declared in a similar way to a one-dimensional array. For example, the variable b below is a "3 by 4" array.

```
INTEGER, DIMENSION(1:3,1:4) :: b
```

You can think of these as arrays consisting of three rows of four columns each:

b(1,1)	b(1,2)	b(1,3)	b(1,4)
b(2,1)	b(2,2)	b(2,3)	b(2,4)
b(3,1)	b(3,2)	b(3,3)	b(3,4)

3.4.1 Allocatable Arrays

A useful feature of Fortran 90/95 is that you can declare an array of a given dimension, but with a size that depends on something that happens in the code. These arrays are called ALLOCATABLE arrays. Arrays are allocated with the ALLOCATE statement and then deallocated with the DEALLOCATE statement. An example follows:

```
! This program allocates an array with a size based on user input and
! generates that many random numbers
PROGRAM allocate_example
IMPLICIT NONE
 REAL, DIMENSION(:), ALLOCATABLE :: a
 INTEGER :: i
 WRITE(6,*) 'How many random numbers do you want to calculate ?'
 READ(5,*) i
 ALLOCATE(a(i))
 CALL RANDOM_NUMBER(a)
 WRITE(6,*) a
 DEALLOCATE(a)
END PROGRAM allocate_example
```

Self Assessment Exercise

Make sure you have read this unit before attempting the following exercise.

1. Write a program that calculates the determinant of a 3 by 3 matrix.

Your program should READ in the matrix from a file using the OPEN and READ commands. Therefore, start by writing a Fortran program which will read the elements of the matrix

1.0 3.0 2.0

5.0 1.0 5.0

7.0 6.0 5.0

from a data file. You can create the data file yourself using text editor. Just enter the numbers and save the file. Make use of processing the whole array by name, and Fortran will do the right thing. Reading in and writing a matrix is not one of these.

The only solution to doing this in one line is to use the following implied do-loop technique to read in the numbers.

```
READ(10,*) ((a(i,j),j=1,3),i=1,3)
```

This specifies explicitly which order the 9 elements get read in.

Now calculate the determinant of this matrix and print the result out to the screen.

2. Define a new 3 by 3 matrix. Let the elements of the new matrix each equal 2 times the elements of the original matrix. Calculate the determinant of the new matrix. What do you notice?

3.0 Functions and Subroutine Programs

SUBROUTINEs and FUNCTIONs are used to aid program development and flexibility by providing additional structure in the program layout and allowing encapsulation of significant sections of code, which can then be easily reused many times. SUBROUTINEs and FUNCTIONs can be particularly useful in large and complex programs. The main difference between a SUBROUTINE and a FUNCTION is that a FUNCTION can only return a single value, whereas a SUBROUTINE can do more complicated things, like alter the value of many variables.

The examples below are designed to highlight particular features of the use of FUNCTIONs and SUBROUTINEs. The first example shows how a FUNCTION command can be used to calculate the quadratic x_2-3x+4 for values of x ranging from 0.0 to 2.0 at intervals of 0.1, and also, using the same FUNCTION, to approximately calculate the derivative at the points x.

```
PROGRAM quadratic
```

```
IMPLICIT none
REAL :: x, eps=1.0e-5
INTEGER :: i
DO i=0,20
x=REAL(i)*0.1
  WRITE(6,'(3f12.5)') x, quad(x), (quad(x+eps)-quad(x-eps))/(2*eps)
END DO
```

```
CONTAINS

! And here is the function
 REAL FUNCTION
 quad(xx) IMPLICIT none
  REAL :: xx
  quad = xx**2 - 3.0*xx + 4.0
 END FUNCTION quad
END PROGRAM quadratic
```

Notice the following points

The FUNCTION subprogram is contained within the main PROGRAM, following a CONTAINS statement.

The FUNCTION subprogram mirrors the syntax of a main program in having its own IMPLICIT NONE statement, and its own variable declarations.

The FUNCTION command is completed with END FUNCTION.

The FUNCTION is called several times from the main function with different arguments. Whatever argument the function is called with gets associated with xx inside the FUNCTION. xx is a local copy of the argument inside the FUNCTION and is only accessible inside the FUNCTION.

In the main program, the value returned by the function is associated with quad(...)

In the FUNCTION, the value is associated with the function quad by the command quad = ...

The FUNCTION contains no WRITE statements and does not refer to variables from the PROGRAM. It only uses the arguments passed to it.

Now let us see how a similar program can be written using a SUBROUTINE. It evaluates the quadratic function and its exact derivatives at various values of x.

```
PROGRAM quadratic
 IMPLICIT none
 REAL :: x, value, dvalue
 INTEGER :: i
 DO i=0,20
 x=REAL(i)*0.1
```

```
    CALL quad(x,value,dvalue)
   WRITE(6,'(3f12.5)') x, value, dvalue
   END DO
  CONTAINS

  ! And here is the subroutine
   SUBROUTINE quad(xx,val,dval)
   IMPLICIT none
     REAL, INTENT(IN) :: xx
     REAL, INTENT(OUT) :: val,
     dval val = xx**2 - 3.0*xx + 4.0
     dval = 2*xx - 3.0
   END SUBROUTINE quad
  END PROGRAM quadratic
```

Notice the following points

The SUBROUTINE subprogram is contained within the man PROGRAM, following a CONTAINS statement.

In the main program, the SUBROUTINE is called using the CALL command.

The SUBROUTINE is completed with END SUBROUTINE.

The SUBROUTINE subprogram mirrors the syntax of a main program in having its own IMPLICIT NONE statement, and its own variable declarations.

The variable x in the main program and the results of the calculation, value and dvalue, are arguments passed to and from the SUBROUTINE subprogram.

All arguments must have declarations matching in their data type between the SUBROUTINE and calling program.

You should use the INTENT(IN) declaration to indicate arguments passed TO the subroutine and INTENT(OUT) for arguments passed FROM the subroutine to the calling program.

If you wish to use a subprogram to calculate a SINGLE quantity, you should consider using a FUNCTION. If you wish to calculate more than a single quantity, you must use a SUBROUTINE.

It is important to understand that functions and subroutines can be called many times from the same program, or from other subprograms. The arguments from the calling program are associated with the "dummy" arguments inside the SUBROUTINE or FUNCTION each time it is called.

Self Assessment Exercise

Make sure you have read section 3.0 before attempting the following exercise.

You may have been told that the chance, or probability, of two or more people in a class of 30 having a birthday on the same day is surprisingly high. In fact, the chance of this happening is about 70%.

In this exercise, your objective is to calculate the probability that two or more people in a class of c people have a birthday on the same day.

This probability is given by the following

equation. P = 1 - n!/(n - c)! n^c

where n is the number of days in a year and c is the number of people in a class.

Clearly, you need to be able to calculate the factorial of a number. This is an ideal use of the FUNCTION command, although you could use a SUBROUTINE if you preferred. A plan of solving it might be

> Write a FUNCTION which calculates the factorial of a given number m
>
> Call your FUNCTION with arguments n and n-c to work out the factorials you need, and hence work out the expression above.

You will almost certainly run into problems trying to generate numbers as big as the factorials you need in this exercise. Check out for KIND to overcome these problems.

DOES YOUR PROGRAM WORK?

What is the probability that at least two students in the Level 1 group have birthdays on the same day?

Practical Example: Using random numbers

Random number simulations are used widely in physics and in a range of other disciplines. The Fortran 90/95 language possesses intrinsic subroutines RANDOM_NUMBER and RANDOM_SEED which produce a pseudo-random number uniformly distributed between 0 and 1.

First, look at the program below which simply generates and prints 5 random numbers.

```
PROGRAM print_random
 IMPLICIT none
 REAL :: r
 INTEGER :: i
 CALL RANDOM_SEED
 DO i=1,5
  CALL RANDOM_NUMBER(r)
  WRITE(6,'(f9.7)') r
 END DO
END PROGRAM print_random
```

The 5 random numbers are printed to the screen (example below)

0.0505590

0.8959959

0.5748507

0.6094229

0.9629579

What happens if the program is run again? A different set of random numbers are produced. This is because CALL RANDOM_SEED initiates with a different starting seed value each time the program is run. Note that the RANDOM_SEED subroutine is called just once.

Now delete the CALL RANDOM_SEED command. What happens if the program is run repeatedly? The same set of random numbers is produced. This is why CALL RANDOM_NUMBER is called a "pseudo-random" number generator. Starting with the same seed, it will always produce the same set of random numbers. Change the seed, and the sequence of random numbers will change.

Self Assessment Exercise

Make sure you have read “Using Random Numbers” above before attempting the following exercise.

In this exercise, your objective is, first, to test the quality of the Fortran 90/95 random number generator and then, second, to produce a random number generator that produces a normal distribution. First, produce a large number, n say, of random numbers using the RANDOM_NUMBER intrinsic subroutine as described above.

Now, count the number of random numbers falling in the range 0.0000000-0.0999999, those between 0.1000000-0.1999999, and so on.

Print the values to a file. The distribution should be uniform? Is it?

Second, generate a new random number by taking 12 random numbers generated using the RANDOM_NUMBER intrinsic subroutine, adding them together, and subtracting 6. In other words,

R = r1 + r2 + ... + r12 - 6

Generate a large number of random numbers R and perform same process. What does the distribution look like?

3.1 Categories of Subprograms

There are essentially four categories of subprograms namely:

Built-in library functions

Arithmetic statement functions

User defined functions

Subroutines

Arithmetic Function Statement: this is a single executable statement of the

form Function name (arguments) = arithmetic expression

Example: G(X) = X*X – 5.0*X + 2.0

Arithmetic functions always come before the first non-executable statement in a main program of Fortran.

User-defined Function: is a complete block of statements usually located after the END statement in a program. This is unlike arithmetic statement which is just one statement. The format is

FUNCTION NAME (list of arguments separated by commas)

:

Declaration

:

Execution statements

RETURN

END function name

OR

```
TYPE FUNCTION NAME (list of arguments separated by commas)
    :
    Name = expression
    :
RETURN
END function name
```

Where TYPE is any of the following types (INTEGER, REAL, COMPLEX, etc.)

Example

```
FUNCTION add(x, y, z)
Z = x + y
RETURN
END add
```

The function is invoked from the main program by using the arithmetic statement as follows:

sum = add(x, y, z), where the addition of x and y is stored in variable sum.

3.2 The COMMON Statement

This statement is used to share common storage location in the main program and subprograms. The format is

```
COMMON list
```

It must appear at the beginning of the main program, subroutine and function subprograms. Example

```
PROGRAM area
  COMMON i, j, k, l
  READ (*,*) i, k
  j = i * k
CALL areaplus
WRITE (*,*) l
STOP
END

SUBROUTINE areaplus
COMMON i, m, n, q
Q = SQRT (I + m **n)
```

```
RETURN
END
```

Self Assessment Exercise

1. What is the difference between a subroutine and a function

4.0 Conclusion

Subprograms in program developments has a lot of advantages such as defined program modules; re-usable statements; logical clarity; easy to debug and tested amongst others. Problems of complex programs are alleviated by the use of the subprograms and they are complete and independent.

5.0 Summary

Subprograms fall into two basic categories: functions and subroutines. The similarities and differences were examined and the returning of a single value by FUNCTION seems the main significant difference.

6.0 Tutor Marked Assignment

1. Write a subroutine subprogram with arguments and return statements that find the sum, average and mean of three set of numbers.
2. Introduce COMMON statement to solve question 1 above.

CIT 736 - Computer Programming (2 Units)

Module 3 PASCAL Programming Language

Unit 1 Elements of Pascal Language

Unit 2 Data Input and Output

Unit 3 Control structures

Unit 4 Arrays

Unit 5 Sub-Programs

Module 3: PASCAL Programming Language

Unit 1: Elements of PASCAL Language

3.0 Pascal Programming Basics

In a program, you must always obey the rules of the language, in this case, the Pascal language. A natural language has its own grammar rules, spelling and sentence construction. The Pascal programming language is a high level language that has its own syntax rules and grammar rules. A very simple program is shown below:

```
Program Example1;
(* Program to print Hello World*)
Begin
  Write ('Hello World. Prepare to learn PASCAL!!');
  Readln; {comment}
End.
```

The program is written only to display the message: 'Hello World. Prepare to learn PASCAL!!'. This is simply shown on the screen. So, to display any message on the screen, you should use "write" (or "writeln"). When using these two terms, any text within the brackets and the inverted commas '(' ')', is displayed on the screen. However, if a variable is

used instead of a text, without using the inverted commas, the CPU will display the stored variable in the memory, on the screen. The “readln” statement is used to 'stop' the program and wait until the user presses a key. If the 'readln' statement is missing in this program, then the message is displayed on the screen without giving any chance for the user to read it.

A program in Pascal starts with the reserved word 'Program' (although it is not explicitly required) and ends with 'End', followed by a full stop (this is required). A full-stop is never used within the program, except when dealing with records. A program can be written in a text file, save the text file as filename.pas and open it with Turbo Pascal. The .pas extension is required.

After declaring all the variables which are required to be used later in the program, the main program always starts with the reserved word 'Begin'. Without this word, the compiler will display a diagnostic (error message). Another important thing which must be noticed is the semi-colon (;). The semicolon is used after each statement in the program, except those that you will learn later. The messages in between the braces ({ }) or (* *) are called comments or in-line documentation. Comments within the braces are not read or compiled by the compiler/ interpreter.

Later in the course, you will also learn how to control input and output exceptions - unexpected runtime errors. One last thing on errors is this: there are 2 major error types which are - Runtime Errors and Compilation Errors. Runtime errors are those which occur unexpectedly during the execution of the program, whereas a Compilation error is one which is detected during the compilation process.

For example

```
Sum := Num1 + Num2;
```

The result of the above statement is the addition of the values stored in variables 'Num1' and 'Num2'. The important thing that you should know is that one cannot make the same statement as follows:

```
Num1 + Num2 := Sum;
```

This is another syntax error. It is the fact that transfer of information is from left to right and not from right to left. So, mind not to make this error. The ':=' is called the assignment statement, and should be discussed later.

3.1 Character Set and Identifiers

Pascal uses the letters A to Z (both upper and lowercase), digits 0 to 9, and special symbols in the building blocks of Pascal to form program elements.

The special symbols are listed below:

+	.	<	(	(*
-	:	<=	)	*)
*	;	>	[	(.
/	,	>=	]	.)
:=	‘	<>	{	@
=	^	--	}	

An *identifier* is a name such as constant, variable or procedure given to some program element. They are comprised of sequence of letters or digits with the first character being a letter. Examples of identifiers are B1, Product, Figure12 etc.

There are standard identifiers in Pascal with predefined meanings. They are:

abs	dispose	input	Ord	read	sin	trune
arctan	eof	integer	output	readln	sqr	unpack
Boolean	eoln	in	Pack	real	sqrt	write
char	exp	maxint	page	reset	succ	writeln
chr	false	new	Pred	rewrite	text	
cos	get	odd	Put	round	true	

Standard identifiers are used for their predefined purposes except otherwise declared or redefined by programming procedures. For example *sqrt* is to find the square root of a value. The meanings of the identifiers are in the compiler package help menu and the functions will be treated in the subsequent course units.

3.2 Numbers and Strings

Numbers are bunch of numerical figures which can include signs, decimal point and exponent or scale factor. Commas and blank spaces are not included in numbers but it can be preceded by plus (+) or negative (-) signs. Numbers without any sign is assumed positive and they have limits in size with respect to the type of number, the word size of computer and the compiler. Example: integer is between (-32768 to 32767), word - 0 to 65535.

Numbers can be subdivided into two main categories:

Integer Numbers: an integer number has no decimal point. Examples are 67, -234, 900 etc. **Real Numbers**: a real number must have a decimal point and the decimal point must appear between two digits. Examples are 0.01, 11.94, -5.6 etc.

In some cases, exponent can be included to shift the location of the decimal points. Example is $1.23 \times 10_2$ written in exponential form as 1.23E+2 or 0.123E+3. The latter is shifted by one decimal place which reflected on the exponent by making it +3. **Note**: a comma and spaces are not allowed within a number.

A *string* is a sequence of characters (i.e. letters, digits and special characters) enclosed by apostrophe.

Examples are

‘OPEN UNIVERSITY’

’10 PASCAL STREET’ etc.

Soon, you should learn how to input text by the user using 'string variables'. The following program is written showing an example of a string variable, prompting the user to input his name, whatsoever:

```
Program Example2;
Var name, surname: String;

Begin
        write('Enter your name:');
         readln(name);
         write('Enter your surname:');
         readln(surname);
         writeln;{new line}
         writeln('Your full name is: ',name,' ',surname);
         Readln;
End.
```

If we take a look at this program, we notice a new variable type: 'String'. Both the name and surname variables are of type string. When the program is run and prompts the user to input his name, the name which is keyed in by the user goes directly to its place in the memory called 'name'. Same occurs to surname. You must note that the variables 'name' and 'surname' are not reserved words, but are used by the programmer as his variables. The *writeln* is used to move on for a new line. In this case, a line is created. The next message displays the full name of the user using the above format. If a string variable is required to be displayed on

screen, it should be put in between inverted commas and commas if it is concatenated with another message. Example:

writeln('Your name is: ',name);

or:

writeln('Your name is:',name,'. Your surname is ',surname,'.');

3.3 Variable, Constant and Assignment Statement

A *variable* is an identifier whose value can change during program execution. Every variable must be individually defined before it can be used in a program. The syntax for variable is:

VAR *name* : type OR

VAR *name1, name2, ... nameN* : type

Apart from variables, there are also items in the program which are referred to as constants. Unlike variables, *constants* keep their value or string unchanged for the whole program. Here is a program, not so much different from the previous one:

```
Program Example3;
Var
   surname: String;

Const {the reserved word 'const' is used to initialize
   constants} name = 'Victor';

Begin
        write('Enter your surname:');
        readln(surname);
        writeln;
        writeln;
        writeln('Your full name is: ',name,' ',surname);
        readln;
End.
```

In the above program, the constant 'name' is assigned to as 'Victor' and is of type string. However, in other cases, you might have used integer constants (whole numbers), i.e.:

```
Const
  age = 15;
```

The constant 'age' is a value that could be used whenever it is required in a program.

Example:

age2 := age + 15;

The above example shows an addition of the value of the variable 'age' with the value 15. The value of the constant 'age' remains 15, but the value of the variable 'age2' becomes 30. The last example will take us to the assignment statement that is used for addition.

The ***assignment statement*** is used to assign entity or value to a variable: text if it is a string variable or a numeric value if it is an integer variable. The syntax is:

Variable := data item

Examples are:

name := 'victor';

age := 15; {also: "age:='15';" BUT in this case, 'age' is a string variable}

3.4 Data Types

Pascal supports different data types such as the simple data types, structured data types and pointer data types. The data types are associated with identifiers to show the type of data to be associated with.

The data types are summarized below:

Simple Data Type

- Standard data types
  - Integer
  - Real
  - Char
  - Boolean
- User-defined data types
  - Enumerated
  - Sub range

Structured Data

Type o Arrays

o Records

o Files o

Sets

Pointer Data Type

- A pointer has the POINTER attribute, and may point to (be an alias of) another data object of the same type, which has the TARGET attribute, or an area of dynamically allocated memory. At this level, emphasis will be on the simple data type.

3.5 Reserved Words

Reserved words in Pascal have their standard and predefine meaning in writing programs. Some of them are:

Array	Begin	Case	Const	End	File
Function	Goto	Not	Or	And	Until
If	Do	Else	For	Then	Type
Clrscr	GotoXy	Textbackground	Textcolor	Readkey	
Delay	Halt	Uses	etc.		

For example, to include a library in your program, one should use the reserved word 'uses', because it is used to call a library of functions and procedures. Here is the declaration part of an incomplete program:

```
Program Example4;

Uses Crt; {This make use of the crt library}

Var PD, Dname, Cmodel : String;
   CostPD, TCostPD, Distance : Real; {real is a decimal (described later)}
Begin
        -----
        -----
End.
```

The 'Crt' (short for cathode-ray tube) library has a wide range of functions and procedures that you will use so frequently. Some of them are listed below.

Reserved Word	**Description**
Clrscr	Clears screen
Gotoxy(int,int)	Takes the cursor to the pre-defined position
Textbackground(word/int)	Background colour

Textcolor(word/int)	Colour of text
Readkey	Reads a key; Could be assigned to a variable
Delay(int)	Waits for the included time(milliseconds)
Halt(parameter)	Program terminates

Examples of program segments:

Clrscr: (clear screen)

```
writeln('When you press enter, the screen would be cleared!');
readln;
clrscr;
```

Gotoxy(int,int): (Go to position x and y);

```
gotoxy(10,10);
Writeln('The position is 10 pixels from the left of the screen, and ten pixels');
Writeln('from the top of the screen.');
readln;
```

Textbackground(word/int): (Background colour);

```
Textbackground(red); {word - red}
Writeln('Note the difference');
Textbackground(5); {integer - 5}
ClrScr;
Writeln('Note the difference'); Readln;
```

Textcolor(word/int): (Text colour);

```
Textcolor(red); {word - red}
Writeln('Text colour');
Textcolor(5); {integer - 5}
Writeln('Text colour'); Readln;
```

Readkey: (Reads a key-press);

```
Writeln('Press ANY key!!!');
Readkey;
```

Delay(int): (Waits for some time);

```
Writeln('1');
Delay(1000);{1000 milliseconds}
Writeln('2');
Delay(1000);
Writeln('3');
Readln;
```

Halt(int): (Program terminates with an exit code);

```
writeln('Press enter and the program terminates!);
Readln;
Halt(0);
```

Note that instructions following 'halt' are not executed since the program terminates when halt is encountered.

3.6 Standard Functions and Operator Precedence

Below are some of the standard functions in Pascal. Most of the functions are supplied with parameters. Example is the use of x parameter in the functions below:

ABS(x) - Computes the absolute value of x

COS(x) - Computes the cosine of x (x in radians)

EXP(x) - Computes exponential of x

LN(x) - Computes natural logarithm of x where x > 0

ROUND(x) - rounds the value x to the nearest integer

SIN(x) - computes the sine of x (x in radians)

SQRT(x) - computes square root of x

SQR(x) - computes the square of x

EOLN - detects end-of-line

EOF - detects end-of-file

Example: SQRT(25) = 5

Operator Precedence: Operators generally have associated hierarchy that determines the order of precedence for evaluating an expression. The operator hierarchy in Pascal is as follows:

1. NOT
2. * / DIV MOD AND
3. + - OR
4. = <> < <= > >= IN

Relational Operators

=	Equal to	<>	Not Equal to	<	Less than
<=	Less than or equal to	>	Greater than	>=	Greater or equal to

Logical Operators

OR - expression will be true if either operand is true or both

AND - expression will be true only if both are true

NOT - a prefix that negates a Boolean expression

The operators are used in expressions. Expression is collection of operands (i.e. numbers, constants, variables etc.) joined together by certain operators to for algebraic phrase that represents a value. Expressions are evaluated by taking the operators with the highest priority before the lower ones. Where operators are of the same priority, the expression is evaluated from left to right with the exception of expression in parenthesis which are always evaluated first before non-parenthesis expressions.

Example: if a = 8, b = 4 and c = 2, then the expression (a+b)/c = 6 because a+b is evaluated first.

The two categories of expression in Pascal are numerical and Boolean expressions.
Numerical results to numerical value whereas Boolean results into either true or false.

Example: Numerical expression a + b = 12; Boolean expression b = c is false

Self Assessment Exercise

1. Differentiate between a string and an integer variable using a Pascal expression
2. What do you understand by the term identifier
3. Give three categories of data types in Pascal
4. Code examples 1 through 4 above, run and execute them

3.0 Input and Output Routines

Pascal supports all standard input and output routines, plus the extensions listed below.

Routine	Description
append	Opens a file for writing at its end.
close	Closes a file.
filesize	Returns the current size of a file.
flush	Writes the output buffer for the specified Pascal file into the associated operating system file.
getfile	Returns a pointer to the C standard I/O descriptor associated with the specified Pascal file.
linelimit	Terminates program execution after a specified number of lines has been written into a text file.
message	Writes the specified information to stderr.
open	Associates an external file with a file variable.
read and readln	Read in boolean, integer and floating-point variables, fixed- and variable-length strings, enumerated types, and pointers.
remove	Removes the specified file.

reset and rewrite	Accepts an optional second argument.
seek	Resets the current position of a file for random access I/O.
tell	Returns the current position of a file.
write and writeln	Outputs boolean integer and floating-point variables, fixed- and variable-length strings, enumerated types, and pointers; output expressions in octal or hexadecimal; allows negative field widths.

3.1 Input of Data

The following statements are used for input of data.

Read statement

Readln statement

The readln statement reads a new line of data whereas read statement continue on the same line.

For example, consider the following program;

```
program input;
 var
  c : int;
begin
readln(c);
read(c);
end.
```

readln() and read() place user input into the assigned variable. In this case, if the user entered a number then it would be stored in the variable c. readln() create a new line after the input is entered while read() would cause the cursor to remain on the same line instead.

Both readln and read can be useful when programming in the Windows environment. If a compiled application is executed outside of the command prompt, the application window would close immediately after performing its task. The following code can be used to keep the window open after finishing the programmed task:

```
program RemainOpen;
 begin
  writeln('Hello, world!');
  readln;
end.
```

As you notice, readln statement will keep the window open and the parentheses are not required when no parameters is needed.

3.2 Output of Results

The following statements are used to output results:

Write statement

Writeln statement

When writeln is used, the subsequent write statement will begin a new line of output. For now, we will use console output. Here is an example of how output is done:

```
program output;
begin
writeln(1);
write('1')
end.
```

writeln() displays whatever is in the parentheses and prints a "new line character", making newly displayed characters start on the next line. The ‘;’ symbol is just to separate the two statements.

write() displays whatever is in the parentheses without printing a "new line character".

Do note that characters and strings must be placed within single quotation marks. You may have noticed that write and writeln work for more than one type. You may also wonder how write(1) and write('1') could do the same thing yet, 1 and '1' be represented in completely different ways. The answer is that compiler always knows the data type and thus how it is formatted.

Self Assessment Exercise

1. Show the difference between the followings
    a. Read and Readln
    b. Write and Writeln
2. Give five input and output data routine extensions

3.3 Formatted Output

Formatting output stream in Pascal make the results more legible and this is done by altering the field widths associated with the data type and also adding certain format features within the write and writeln statements.

For example

Write ('The sum of A and B is', sum : 3);

If the sum of A and B is 12, there will be a blank space preceding the number 12 since width 3 is declared in the write statement.

In case of output with decimal places: the number after the first colon indicates the total field width followed by another colon and immediately with a number to show the number of decimal places. For example:

Writelin ('The Product is', product : 10 : 2);

If the product is 25689.345, then the result will be 25689.35 preceded by two blank spaces to justify the field width of 10.

Module 3: PASCAL Programming Language

Unit 3: Control Structures

3.0 Program Control

A computer program is normally executed starting from the first instruction, and continues in that order of sequence until branching instruction is encountered. At that point, the control is immediately transferred to another part of the program to execute some tasks, and thereafter continue in the sequential execution to the end of the program except another branching instruction is encountered. Some of the control statements and program loops include:

IF statement

Repeat-Until loop

For loop

WHILE-DO loop

CASE-OF statement

Now, it is time to learn the most important rules of programming: the IF statements - decision making, for loops and the repeat-until loop. Almost, these three general programming constructs are common in every programming language and you have to make sure that when you have finished reading this unit, make sure that you have practiced them enough before continuing with learning Pascal because they are of outmost importance.

3.1 The IF Statements

The “IF statement” executes the proceeding statement(s) conditionally. This means that if an action comes to be true, then the statement(s) after the IF statement are executed, else these statements are skipped. It works like this:

If this happens (action), then do this (reaction, if action is true).

OR:

If this happens (action), then do this (reaction, if action is true), else do this (reaction, if action is false).

In Pascal, the “IF statement” should be written as follows:

If conditional expression then code ... ; {if one action}

OR:

If conditional expression then Begin instructions ... End; {if more than one action is required}

Note that you should not use an assignment statement in the “IF” construct, otherwise the compiler will raise a syntax error. i.e.:

Wrong:

If x := 20 then x := x + 1; {the underlined character must be excluded}

Correct:

If x = 20 then x := x + 1; {only an equal sign is used for comparison}

3.1.1 Conditional Statements: If..Then..Else

The simplest control structure is the if..then..else statement. The following program segment shows the use of the if.. then.. else statement.

```
if var1 = 0 then
   writeln('var1 is 0!') (*No semicolon before an 'Else' keyword*)
else if var1 = 1 then
begin
   writeln('var1 is 1!');
   (*More code*)
end (*No semicolon before an 'Else' keyword*)
else if var1 = 2 then
begin
   writeln('var1 is 2!');
```

```
   (*More code*)
end (*No semicolon before an 'Else' keyword*)
else
begin
   writeln('var1 is not 0, 1, or 2!');
   (*More code*)
end; (*Semicolon used to indicate the end of the If-then-else*)
```

In a normal if statement, the 'reaction' cannot be performed if the condition is not true. But in an if..then..else statement, there is at least one set of statements to be performed. Let's take a look at the example below:

```
writeln('Who has discovered the land of
America?'); Readln(ans);
If (ans = 'Christopher Colombus')
        then score := score + 1                    {if this is false,}
ELSE
        writeln('sorry, you"ve got it wrong!');        {then this is true}
```

Note that if the 'else' term is included with an if statement, then there should be no semi-colon before the 'else' term; just as seen in the above example.

3.1.2 Nested If statements

An example of nested if statements is shown below.

```
If Sel = '3' then
 Begin
  ClrScr;
  Write('Are you sure?(Y/N)');
  YN := Readkey;
  If YN = 'y' then HALT; {Nested if statement}
  If YN = 'n' then Goto 1;{Another Nested if statement}
 End;
```

A nested if statement, is in the form:

```
If (this happens) then       {if 1}
  If (this happens) then   {if 2}
    (do this) etc...
  Else (do this)          {if 2}
 Else (do this) etc...      {if 1}
```

A nested IF statement is an IF statement within another IF statement, as shown above.

3.2 The Repeat-Until Loop

This loop is used to repeat the execution of a set of instructions for at least one time. It is repeated until the conditional expression is obeyed. The following example shows the model of the "repeat-until" loop:

```
Repeat
        ..(code)
        ..(code)
        ..(code)
Until conditional statement;
```

Here's an example:

```
Uses Crt;
Var YN : String;

Begin
 Writeln('Y(YES) or N(NO)?');
 Repeat {repeat the code for at least one time}
  YN := Readkey ;
  If YN = 'y' then Halt; {Halt - exit}
  If YN = 'n' then Writeln('Why not? Exiting...');
 Delay(1800); { wait a second plus 800 milliseconds }
 Until (YN = 'y') OR (YN = 'n');
End.
```

3.3 The For Loop

The FOR loop is a sort of repeat-until loop. The FOR loop, repeats a set of instructions for a number of times. The FOR loop is in the form:

- If used for only one action:

```
for {variable}* := {original value} to/downto {final value}
   do {code...(for one action)}
```

- If used for more than one action:

```
for {variable}* := {original value} to/downto {final value} do
   Begin {code...}
   {code...}
End;
```

Generally, this variable is called the “loop counter”. Now, an example of the FOR loop is shown below.

```
Program ForLoop;
Uses Crt;

Var Counter : Integer; {loop counter declared as integer}

Begin
 For Counter := 1 to 7 do {it's easy and fast!}
 writeln('for loop');
 Readln;
End.
```

3.3.1 Nested for loops

A nested for loop is similar to that of the nested if statements. A nested for loop is in the form:

```
for {loop counter} := {original value} to {final value} do {Begin-if required}
      {code if any..begin should be included (i.e more than one action)}
 for {loop counter} := {original value} to {final value} do {Begin-if required}
      {code..if more than one action, include begin in the second for loop}
 {End; - if begin is included in the second for loop)}
```

{code if any..begin should be included in the first for loop} {End; - if begin is included in the first for loop)}

The nested for loop is rarely used and it may cause problems.

Self Assessment Exercise

1. Do the IF statement for K = 1 if A – B < 100, and sets K = 2 otherwise
2. Write a FOR statement to count from 1 to 100
3. What is the significant difference between FOR and Repeat-Until loop

3.4 While-Do Loop

This type of loop is executed while the condition is true. It is different from the 'Repeat-Until' loop since the loop might not be executed for at least one time. The code works like this:

While <condition is true> do the following:

instruction 1;
instruction 2;
instruction 3;
etc...

End; {If while-do loop starts with a begin statement}

Example Program on the While-Do loop:

```
Program WhileDo;
Uses Crt;
Var Ch : Char;
Begin
 Writeln('Press "q" to exit...');
 Ch := Readkey;
 While Ch <> 'q' do
  Begin
   Writeln('I told you press "q" to exit!!');
```

```
  Ch := Readkey;
 End;
End.
```

3.5 The CASE-OF Statement

So far, you have learned how to use an “IF statement”. But in some cases the 'case statement' is preferred to the IF statement because it reduces some unnecessary code but the same meaning is retained. The case statement is very similar to the IF statement, except in that it does not accept literal conditional expressions (i.e.: strings) but surprisingly enough, it allows single character conditional expressions. Here is how it works:

```
Case {variable of type: integer or character ONLY} of
      {input statement- within inverted commas if of type char} : {code..}
      {input statement- within inverted commas if of type char} : {code..}
   ...
End; {End Case}
```

Now you should note the difference and the intelligent use of the case statement over the IF statement.

The Program is written using the IF statement:

```
Program CaseExample1;
Uses Crt;
Label Return; {used respectively with the
        goto statement; beware of it}

Var SEL : Integer;
   YN : Char;

Begin
 Return: Clrscr;
 Writeln('[1].PLAY GAME');
 WRITELN('[2].LOAD GAME');
```

```
WRITELN('[3].MULTIPLAYER');
WRITELN('[4].EXIT GAME');
Writeln('note: Do note press anything except');
Writeln('numbers; otherwise an error occurs!');
Readln(SEL);
If SEL = 1 then
 Begin
  Writeln('Are you able to create a game');
  Writeln('of yourself using pascal??');
  Delay(2000);
  Goto Return;
 End;
If SEL = 2 then
 Begin
  Writeln('Ahhh... no saved games');
  Delay(2000);
  Goto Return;
 End;
If SEL = 3 then
 Begin
  Writeln('networking or 2 players?');
  Delay(2000);
  Goto Return;
 End;
If SEL = 4 then
 Begin
 Writeln('Exit?');
 YN := Readkey;
 If YN = 'y' then
 Begin
    Writeln('Noooooooooooo...');
    Delay(1000);
    Halt; {EXIT PROGRAM}
   End;
```

```
  If YN = 'n' then
 Goto Return;
 End;
End.
```

Now, the next program is written using the case statement and the output is almost the same.

```
Program CaseExample2;
Uses Crt;
Label Return; {use of the goto statement is not recommended..avoid
it} Var SEL : Integer;
    YN : Char;

Begin Return:Clrscr;
 Writeln('[1].PLAY GAME');
 WRITELN('[2].LOAD GAME');
 WRITELN('[3].MULTIPLAYER');
 WRITELN('[4].EXIT GAME');
 Writeln('note: Do note press anything except');
 Writeln('numbers; otherwise an error occurs!');
 Readln(SEL);
 Case SEL of
  1 : Begin
      Writeln('Are you able to create');
      Writeln('a game of yourself using pascal??');
      Delay(2000);
      Goto Return;
     End;
  2 : Begin
      Writeln('Ahhh... no saved games');
      Delay(2000);
      Goto Return;
     End;
```

```
 3 : Begin
     Writeln('networking or 2 players?');
     Delay(2000);
     Goto Return;
    End;
 4 : Begin
     Writeln('Exit?');
     YN := Readkey;
     Case YN of {a sort of a nested case statement}
      'y' : Begin
            Writeln('Noooooooooooo...');
            Delay(1000);
            Halt;
           End;
      'n' : Goto Return;
     End;{End Case2}
    End;{Close Conditional Expression 4}
End; {End Case1}
End.
```

3.5.1 The Case-Else Statement

Again this is similar to the if..then..else statement. Study the program below to learn how to use the 'else' term following the 'case statement':

```
Program CaseExample3;
Uses Crt;
Label Return; { avoid it }
Var YN : Char;

Begin
 Return:ClrScr;
 Writeln('Exiting?');
 YN := Readkey;
```

```
Case YN of
 'y' : Halt;
 'n' : Begin
       Writeln('What are you going to do here, anyway?');
       Delay(2000);
       Halt;
      End;
 Else
 Begin
  Writeln('Either press "y" for yes');
  Writeln('or "n" for no.. please try again..');
  Delay(3500);
  ClrScr;
  Goto Return;
 End;
End; {CASE}
End. {PROGRAM}
```

Self Assessment Exercise

1. What is the significant difference between CASE and IF statement
2. Write the syntax for CASE-OF statement

3.0 Arrays

What are Arrays?

An Array is a powerful data structure that stores variable data having the same data type. It is just like a small fixed number of boxes linked together one after the other storing things that are related to each other. An array is said to be a static data structure because, once declared, its original size that is specified by the programmer will remain the same throughout the whole program and cannot be changed.

Up until now, we have used single variables only as a tool to store data. Now we will be using the array data structure and here is how it is declared:

```
Var
myArray : Array[1..20] of Integer;
<arrayName> : Array[n..m] of <Data Type>;
```

An array data structure defines the size of the array and the data type that it will use for storing data. In the above example, the array stores up to 20 integers however I may have used 30 integers or more. This size depends on your program requirements.

Arrays are used just like ordinary variables. They are used to store typed data just like the ordinary variables. You will now learn how to assign data to arrays and read data from arrays. In the example above, 20 integers were declared and they can be accessed as follows:

To assign values to a particular integer of an array, we do it like this:

myArray[5] := 10;

myArray[1] := 25;

<arrayName>[index] := <relevant data>

You just take the array in subject, specify the index of the variable of the array and assign it a value relevant to the data type of the array itself. Reading a value from an array is done as follows:

```
Var
   myVar : Integer;
   myArray : Array[1..5] of Integer;

Begin
 myArray[2] := 25;
myVar := myArray[2];
End.
```

Just like ordinary variables, arrays should be initialized; otherwise scrap data will remain stored in them. If we want to initialise 2 whole 20-sized integer and boolean arrays to 0 and false respectively, we do it like this:

```
Var
   i         : Integer;
   myIntArray : Array[1..20] of Integer;
   myBoolArray : Array[1..20] of Boolean;

Begin
 For i := 1 to 20 do
 Begin
  myIntArray[i] := 0;
 myBoolArray[i] := false;
 End;
End.
```

3.1 Static Size Arrays

Examples of static size array such as lists and are defined in syntax of:

```
type MyArray1 = array[0..100] of integer;
type MyArray2 = array[0..5] of array[0..10] of
 char; {or}
type MyArray2 = array[0..5,0..10] of char;
```

Examples of this would be having a list of different types or records.

```
type my_list_of_names = array[0..7] of string;
var the_list: my_list_of_names;
begin
  the_list[5] := 'John';
the_list[0] := 'Titi';
end;
```

This would create an array of ['Titi','','','','John','',''']. We can access these variables the same way as we have set them:

```
begin
  writeln('Name 0: ', the_list[0]);
writeln('Name 5: ', the_list[5]);
end;
```

The following example shows an array with indices of chars.

```
var
  a: array['A'..'Z'] of integer;
  s: string;
  i: byte;

begin
  readln(s); {reads the string}
  for i := 1 to length(s) do {executes the code for each letter in the
    string} if upcase(s[i]) in['A'..'Z'] then
      inc( a[upcase(s[i])] );
```

```
  {counts the number the times a letter is counted. the case doesn't
 count} writeln('The number of times the letter A appears in the string is
 ',a['A']); {returns 5 if the string is 'Abracadabra'}
end.
```

3.2 Dynamic Arrays

Lists also can contain unlimited number of items, depending on the amount of computer memory. This is achieved by dynamic arrays, which appeared with Delphi (not available with Turbo Pascal). To declare a dynamic array, simply declare an array without bounds:

```
var a: array of <element_type>;
```

Then, define the size at any moment with the function

```
SetLength SetLength( a, <New_Length> );
```

The indices of the array will from 0 to length(a)-1.

You can get the current length with the Length function, as for String type. To go through the array, you can write for example:

```
var
 i: integer;
 a: array of
integer; begin
randomize;
 setlength(a, 10); { a will contain 10 random numbers
 } for i := 0 to length(a)-1 do
   a[i] :=
random(10)+1; end;
```

The memory is freed automatically when the array is not used anymore, but you can free it before by assigning nil value to the variable which is the same as defining a length of 0.

Self Assessment Exercise

1. What do you understand by subscripted and unsubscripted variables
2. Arrays should be initialized. Why?

3.3 User-Defined Data Types

Now that we have used various built-in data types, we have arrived at a point where we want to use our defined data types. Built-in data types are the ones we used lately, such as Integer, Boolean and String. Now we will learn how to specify our own customised data types and this is just how it is done:

```
Type
<myDataType> = <particularDataType>;
```

The "Type" keyword is a reserved Pascal word used to define our own data types. So you start defining your own data types by using this keyword. After defining your own data types, you may start using them just like the other built-in data types as follows:

```
Var
<myVar> : <myDataType>;
```

Below is a new simple data type and note how it will be used in the program below:

```
Type
   nameType = String[50];
   ageType = 0..150; { age range: from 0 to 150 }

Var
   name : nameType;
   age : ageType;

Begin
 Write('Enter your name: ');
 Readln(name);
 Write('Enter your age: ');
 Readln(age);
 Writeln;
 Writeln('Your name:', name);
 Writeln('Your age :', age);
 Readln;
End.
```

In the above example we defined a String[50] and a 0..150 data type. The nameType only stores strings up to 50 characters long and the ageType stores numbers only from 0 to 150.

We can define more complex user-defined data types. Here is an example of more complex user-defined data types:

```
Type
   i             = 1..5;
   myArrayDataType = Array[1..5] of Byte;
   byteFile      = File of Byte; { binary file }

Var
   myArrayVar : myArrayDataType;
   myFile     : byteFile;

Begin
 Writeln('Please enter 5 number from (0..255): ');
 For i := 1 to 5 do
  Readln(myArrayVar[i]);
 Writeln('You have entered the following numbers: ');
 For i := 1 to 5 do
  Writeln('Number ',i,': ',myArrayVar[i]);

 Writeln('Now writing them to file...');
 {store the numbers in a file}
 Assign(myFile, 'example.dat');
 ReWrite(byteFile);
 Write(myFile, myArrayVar[i]);
 Close(myFile);
 Writeln('Done, you may exit..');
 Readln;
End.
```

In the above example I showed you how to incorporate arrays as user-defined data types. Note that you may use user-defined data types more than once.

3.4 2 Dimensional and Multi-Dimensional Arrays

2 Dimensional arrays (2D) and multi-dimensional are arrays which store variables in a second or n_{th} dimension having n x m storage locations. Multi-dimensional arrays including the 2 dimensional arrays are declared by using multiple square brackets placed near each other or using commas with one square bracket as an alternative. Here is how multi-dimensional are declared:

```
my2DArray      : Array[i..j][k..l] of <DataType>;
myMultiDimArray : Array[m..n][o..p][q..r][s..t]... of <DataType>;
```

Let us have the 2 dimensional array defined first. Think of a grid where each box is located by using horizontal and vertical coordinates just in the example below:

```
1       2       3       4       5
2
3                       3,4
4
5               5,3
```

An example of a 5 by 5 2D array illustrated on a grid

Let us declare an array having 3 by 5 dimensions, assign a value to a particular variable in the array and illustrate this on a grid just like the one above:

```
Var
my2DArray : Array[1..3][1..5] of Byte;

Begin
 my2DArray[2][4] := 10;
End.
```

Having the vertical axis as the 1st dimension and the horizontal one as the 2nd dimension, the above example is illustrated as follows:

```
1       2       3       4       5
```

```
2                    10
3
```

Multi-dimensional arrays are rare and are not important. The single and 2D dimensional arrays are the 2 most frequent dimensions.
The following example is a bit more complex example than the previous examples and it also uses a 2 dimensional array to illustrate their use.

```
Uses Crt;

Type
  myRange     = 1..5;
  arrayIntType = Array[myRange] of Integer;
  myFileType = File of arrayIntType;

Var
     i       : myRange;
  myFile : myFileType;
  { the next array is 2 dimensional }
  arrayInt : Array[1..2] of arrayIntType;

Begin Clrscr;
 Randomize;
 For i := 1 to 5 do
  Begin
   arrayInt[1][i] := Random(1000);
  Writeln('rand num: ',arrayInt[1][i]);
  End;

 Assign(myFile, 'test.dat');
 ReWrite(myFile);
 Write(myFile, arrayInt[1]);
 Close(myFile);
```

```
ReSet(myFile);
Read(myFile, arrayInt[2]);
Close(myFile);

For i := 1 to 5 do
 Writeln(i,': ', arrayInt[2][i]);
Readln;
End.
```

Self Assessment Exercise

1. Give examples of Built-in data types
2. Show by example the difference between one and two-dimensional arrays

Module 3: PASCAL Programming Language

Unit 5: Sub-programs

3.0 Modular Programming

Modular programming is a way of dividing a more involved program into areas or modules that perform specific tasks. This module is sometimes referred to as subprogram. The subprogram which is small program can also be referred to as procedure.

Before we begin, let us first clarify the key difference between functions and procedures. A procedure is set of instructions to be executed, with no return value. A function is a procedure with a return value.

The definition of function/procedures is thus:

```
Function Func_Name(params...) : Return_Value;
Procedure Proc_Name(params...);
```

The function/procedure definition is usually followed by the local variables and the body. However, to provide prototypes, simply add a forward keyword behind the definition instead of the local variables and the body. Of course, the whole function must be defined somewhere else in the program. The following example illustrates the use of this:

```
Function Add(A, B : Integer): Integer; Forward;
Function Bad(A, B, C : Integer) : Integer;
Begin
```

```
 Bad := Add(Add(A,B),C);
End;
```

```
Function Add(A, B : Integer) : Integer;
Begin
 Add := A + B;
End;
```

In this example, Add is first defined as a function taking two integer variables and returning an integer, but it is defined as a forward definition (prototype), and thus no statement body is written. Later, we see that Add is defined with a body. Note that the two definitions of Add must be congruent with each other, or the compiler will complain.

From the above example, we can also gather that in Pascal, a function's return value is given by the value of the variable with the function's name (or by the variable named result), when the function returns. As you can see in the Bad function, an undefined variable named "Bad" has been assigned a value. That is the return value for the Bad function. Similarly, in Add, the variable named "Add" has been assigned a value, which is its return value.

3.1 Procedures

Procedures are just like small programs. Sometimes they are called subprograms. They help the programmer to avoid repetitions. A procedure starts off with a “begin” statement and ends up with an “end” statement. It can also have its own variables, which cannot be used with the main-program.

Now have a look at the program which uses a procedure:

```
Program ProcedureExample1;
Uses Crt;

Procedure DrawLine;
{This procedure helps me to avoid the repetition of steps
[1]..[3]} Var Counter : Integer;

Begin
 textcolor(green);
 For Counter := 1 to 10 do
```

```
  Begin {Step [1]}
  write(chr(196)); {Step [2]}
  End; {Step [3]}
End;

Begin
        GotoXy(10,5);
        DrawLine;
        GotoXy(10,6);
        DrawLine;
        GotoXy(10,7);
        DrawLine;
        GotoXy(10,10);
        DrawLine;
        Readkey;
End.
```

When you are required to debug the program, bugs could be much more easier to find out as the program is sliced into smaller chunks. You may run the program and notice a mistake at a certain point and which is located in a particular procedure/function. It would be much more difficult to find a mistake in a program if it would be one whole piece of code.

Self Assessment Exercise

1. Is procedure and function interrelated? How?
2. Which one returns value? Procedures or Functions

3.2 Procedures with Parameters

The use of parameters offers a better approach to the exchange of information between a procedure and its reference point. The manner the information is exchanged depends, however, on the manner in which the parameters are defined and utilized. The new program using parameters from the previous one is as follows:

```
Program ProcedureExample2;
Uses Crt;

Procedure DrawLine(X : Integer; Y : Integer);
{the decleration of the variables in brackets are called parameters or
arguments} Var Counter : Integer; {normally this is called a local variable}
Begin
 GotoXy(X,Y); {here the parameters is used}
 textcolor(green);
 For Counter := 1 to 10 do
  Begin
  write(chr(196));
  End;
End;

Begin
 DrawLine(10,5);
 DrawLine(10,6);
 DrawLine(10,7);
 DrawLine(10,10);
 Readkey;
End.
```

Now, this program includes a procedure which uses parameters. Every time it is called, the parameters can be variable, so that the position of the line could be changed. This time, we have also eliminated the gotoxy statement before every DrawLine statement. The numbers in the brackets of the DrawLine are the parameters which state the position of the line. They also serve as a gotoxy statement.

When you apply parameters to a procedure, variables should be declared on their own, and must be separated by a semi-colon ";". They are put in between the brackets, following the procedure name. The variables (known as the parameters) should be used by the procedure/sub-program only.

The use of parameters in procedure can have value and variable parameters. These are discussed in the subsequent sections.

3.2.1 The Variable Parameter

Parameters of procedures may be variable. In this case, you can pass data and get data through the procedure using a variable parameter. Here is a declaration of a variable parameter:

Procedure <PROCEDURE_NAME(Var Variable_Name : Type);>

Here is an example of how to use a variable parameter and its purpose:

```
Program VAR_PARAM_EXAMPLE;

 Procedure Square(Index : Integer; Var Result : Integer);
 Begin
   Result := Index * Index;
 End;

Var
 Res : Integer;

Begin
 Writeln('The square of 5 is: ');
 Square(5, Res);
 Writeln(Res);
End.
```

3.2.2 Value Parameters

They are very simple to use and are declared simply by including their names and corresponding data types within the procedure header without the prefix “VAR”. It is found in the previous examples on procedures.

3.3 Functions

The second type of sub-program is called a function. The only difference from the procedure is that the function returns a value at the end. Note that a procedure cannot return a value. A

function start and end in a similar way to that of a procedure. If more than one value is required to be returned by a module, you should make use of the variable parameter. A function can have parameters too. If you change the sub-program from procedure to a function, of the previous program, there will be no difference in the output of the program. Just make sure which one is best when you can to implement a module. For example, if you don't need to return any values, a procedure is better. However if a value should be returned after the module is executed, function should be used instead. Example of a program using a function is seen below:

```
Program FunctionExample;
Uses Crt;

Var SizeA, sizeB : Real;
   YN : Char;
   unitS : String[2];

Function PythagorasFunc(A:Real; B:Real) : Real;

{The pythagoras theorem}
Begin
 PythagorasFunc := SQRT(A*A + B*B);
{Output: Assign the procedure name to the value. If you forget to assign the
function to the value, you will get a trash value from the memory} End;


Begin
 Repeat
 Writeln;
 Write ('Enter the size of side A : ');
 Readln(sizeA);
 Write('Enter the size of side B : ');
 Readln(sizeB);
 Repeat
  Write('metres or centimetres? Enter : [m or cm] ');
```

```
 Readln(unitS);
Until (unitS = 'm') or (unitS = 'cm');
Writeln(PythagorasFunc(sizeA,sizeB),' ',unitS);
Writeln;
Write('Repeat? ');
YN := Readkey;
Until (YN in ['N','n']);
End.
```

Self Assessment Exercise

1. Distinguish between value and variable parameters
2. State one reason why modular programming is better.