BSC. COMPUTER SCIENCE AND BSC. IN COMPUTER TECHNOLOGY
YEAR 1 SEMESTER 1

CCS 121: INTRODUCTION TO PROGRAMMING

NOTES

Notations and fundamental concepts

Introduction to computer system

A computer is an electronic device that can perform calculations and analysis at very high speeds. A computer system is composed of two main parts known as hardware and software. Hardware is any physical component that can be seen and touched. Examples of computer hardware are: Keyboard, mouse, central processing unit (CPU), hard disk, visual display unit. Software is a non physical component that enables the computer to perform its task. A computer software is a set of instructions that enables the coordination and execution of computer processes to produce the results to the user. The Two main categories of software: System software and Application software. The system software acts as the interface between the users and the hardware. System software coordinates the allocation of hardware resources during the execution of a computer process. Examples of system software are: Unix, Linux, Microsoft Windows 2007. Application software is a program written to help users solve their problems. Examples of application software are: Microsoft word 2007, Microsoft Excel 2007, Student management system, Accounting system.

Computer Hardware

Computer hardware is divided in three main categories: Input devices, System unit and Output devices.

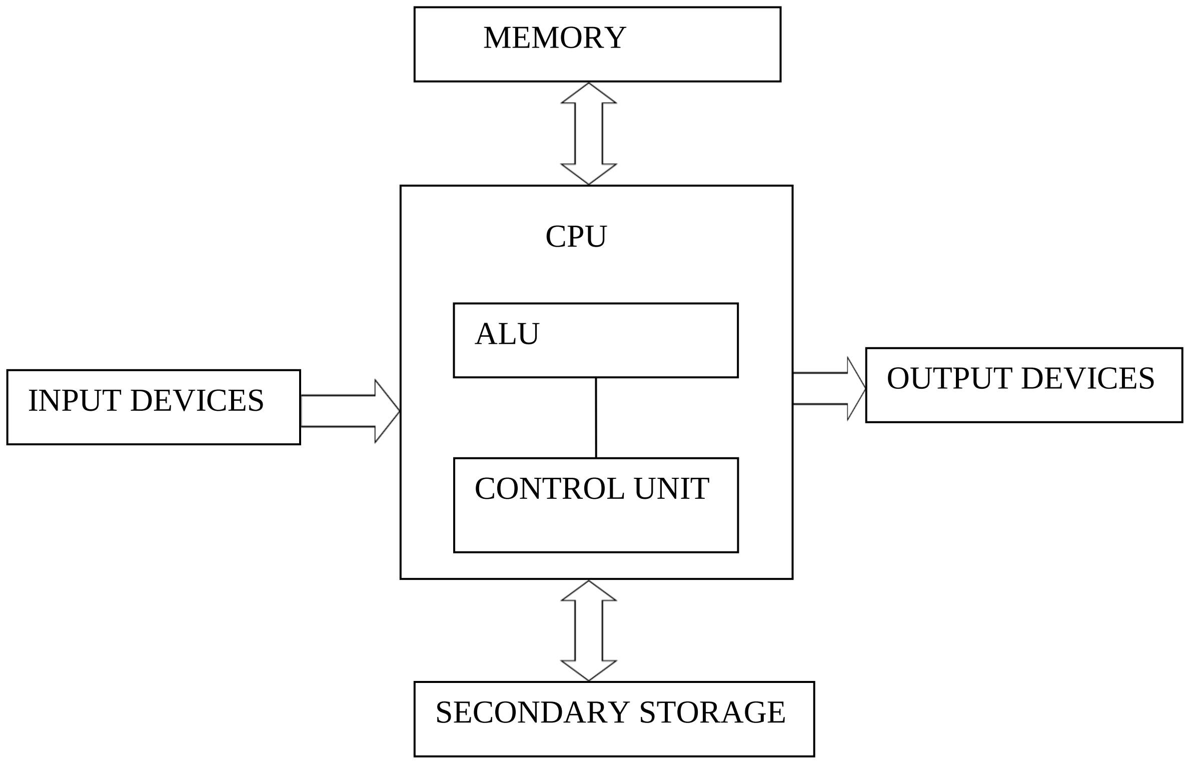
Input and Input Devices

Input is any data or instructions that are used by a computer.

Input devices are hardware used to translate input data - words, sounds, images, and actions that people understand into a form that the system unit can process.

Some commonly used input devices are:

- **The keyboard:** used to key in characters.
- **The mouse:** used to point to or click on a certain area of the screen.
- **The microphone:** used to input sound into the computer.
- **The scanner:** moves across text and images. It converts scanned data into a form the system unit can process.



Output and Output Devices
Output is a processed data or information, and typically takes the form of text, graphics, photos, audio, and/or video.

Output devices are any hardware used to provide or to create output.

They translate information that has been processed by the system unit into a form that humans can understand.

Some commonly used output devices are:

- **Monitors**: Present visual images of text and graphics.
- **Printers:** Present the information on paper.
- **Speakers:** Present voice information

System unit
The main components of the system unit are: Central Processing Unit (CPU), Primary memory and secondary storage.

Central Processing Unit (CPU): It is the part of the computer that carries out the instructions of a computer program. It is the unit that reads and executes program instructions. Hence it is known as the “brain” of the computer. The CPU consists of Arithmetic Logic Unit (ALU) and control unit.

Arithmetic and Logical Unit (ALU): It is the unit where all Arithmetic operations (addition, subtraction etc.) and logical functions such as true or false, male or female are performed. Once data are fed into the main memory from input devices, they are held and transferred as needed to ALU where processing takes place. No process occurs in primary storage. Intermediate generated results in ALU are temporarily placed in memory until needed at later time. Data may move from primary memory to ALU and back again to storage many times before the process is finalized.

Control Unit: It acts as a central nervous system and ensures that the information is stored correctly and the program instructions are followed in proper sequence and the data are selected from the memory as necessary. It also coordinates all the input and output devices of a system.

Memory Unit: It is also known as the primary storage or main memory. It stores data, program instructions, internal results and final output temporarily before it is sent to an appropriate output device. It consists of thousands of cells called storage locations. Any character, either a letter or numerical digit is stored as a string of (0, 1) Binary digits (BITS). These bits are used to store instructions and data by their combinations. There are three well-known types of memory chips: random-access memory (RAM), read-only memory (ROM), and complementary metal-oxide semiconductor (CMOS).

RAM

- Random-access memory (RAM) chips hold the program (sequence of instructions) and data that the CPU is presently processing.
- RAM is temporary or volatile storage because everything in RAM is lost as soon as the microcomputer is turned off. Memory capacity is in bytes.

ROM

- Read-only memory (ROM) chips have programs built into them at the factory.
- ROM chips are not volatile and cannot be changed by the user.
- ROM chips typically contain special instructions for detailed computer operations.

CMOS

- The chip supplies such information as the current date and time, amount of RAM, type of keyboard, mouse, monitor, and disk drives.
- It is powered by a battery and does not lose its contents when the power is off.

The Secondary Storage

Secondary storage provides permanent or non volatile storage. Using secondary storage devices, data and programs can be retained after the computer has been shut off. The capacity of such devices is measured in bytes, kilobytes, megabytes, gigabytes, and terabytes. The main types of secondary storage devices are: hard disks, floppy disks, optical disks (compact disks, digital video disks)

Units of memory: The computer stores a character in the storage cells with binary (0, 1) mechanism. The basic unit of memory is a bit (binary digit – 0, 1). To store a character, a computer requires 8 bits or 1 byte. This is called the word length of the storage unit. Hence the storage capacity of the computer is measured in the number of words it can store and is expressed in terms of bytes. The different units of measurement are

8 Bits = 1 Byte
2^{10} (or) 1024 Bytes = 1 Kilo Byte (KB)
2^{20} (or) 1024 KB = 1 Mega Byte (MB)
2^{30} (or) 1024 MB = 1 Giga Byte (GB)

Computer software

Software is a program or a set of instructions that enables a computer to perform its operations to solve the problems of the users. There two broad categories of software: System software and Application software. Any computer software or program is developed using a programming language. A programming language is the mechanism that enables the interaction between the computer and human being.

System software

System Software is a collection of programs that handle hundreds of technical details with little or no user intervention.

There are four types of programs that makeup system software

- **Operating systems -** coordinate computer resources, provide an interface between users and the computer, and run applications. Famous operating systems: Microsoft Windows (most common), Unix, Linux, and Macintosh
- **Utilities** - perform specific tasks related to managing computer resources. Some Windows Utilities include: Backup, Disk cleanup and Disk Defragmenter
- **Device drivers -** specialized programs that allow particular input or output devices to communicate with the rest of the computer system.
- **Language translators -** convert the programming instructions written by programmers into a language that computers understand and process. Examples include: Compilers and interpreters

Application software

Application software is a program developed to be manipulated by the users to solve their problems. Application programs can be general purpose or special purpose. General purpose programs are used to solve general problems, and the same copy of the program can be bought by anybody who wishes to use it. Example of general purpose program include Microsoft Office 2007. A special purpose program is developed and tailored to solve a given problem for a particular user. Examples of special purpose program are: Student management system, Payroll System, Motor Vehicle registration System.

History and overview of programming languages

Introduction

A programming language is a set of rules that provides a way of telling a computer what operations to perform. A programming language is a set of rules for communicating an algorithm to a computer. It provides a linguistic framework for describing computations. Programming is a process of writing a computer program to solve a given problem. A program is a list of instructions written in a special code, or programming language. The program tells the computer which operations to perform and in what sequence to perform them. Programming languages enable human being to communicate with a computer for the purpose of problem solving.

Categories of programming languages

There are three main categories of programming languages:

- Machine languages.
- Assembly languages.
- High-level languages.

Machine languages and assembly languages are also called low-level languages

Machine languages

- A Machine language program consists of a sequence of zeros and ones.
- Each kind of CPU has its own machine language.

Advantages

- Fast and efficient
- Machine oriented
- No translation required

Disadvantages

- Not portable
- Not programmer friendly

Assembly languages

- Assembly language programs use mnemonics to represent machine instructions
- Each statement in assembly language corresponds to one statement in machine language.
- Assembly language programs have the same advantages and disadvantages as machine language programs.

High-level languages (HLL)

A HLL language program consists of English-like statements that are governed by a strict syntax (rules).

Advantages

- Portable or machine independent
- Programmer-friendly

Disadvantages

- Not as efficient as low-level languages
- Need to be translated

Examples: Pascal, C, C++, Java, FORTRAN, Visual Basic, and Delphi.

Characteristics of high level languages

- Easy to learn
- Easy to find errors
- Machine-Independent
- Availability of Library Functions
- Shorter Programs
- Well-Defined Syntax and Standard
- Source code understandable by any other program

A brief history of programming languages

Low level languages

Those are machine languages and assembly languages, machine-dependent coding systems. They were initially fully binary, and then symbolic. There is one native machine language, and usually one assembly language per processor model.

High level languages

Fortran

Fortran was the first effectively implemented high-level language that introduced variables as we know them now, loops, procedures, statement labels and much more.
The earliest versions of Fortran had many unique features, often awkward, later kept along for compatibility. It is still widely used in engineering applications that require much array manipulation.
The newest version, Fortran 90, has converged toward other popular programming languages.

Algol 60
It was the first to have block structure, recursion, and a formal definition. It is not used now, but it is the ancestor of most contemporary languages. It was the most important innovation in the history of programming languages

Cobol

- Used to be very popular in Business-oriented computations
- Very strict program organization
- Poor control structures
- Elaborate data structures, record type introduced for the first time.

PL/I

- A combination of features believed (at the time) best in Fortran, Algol 60, Cobol.
- The first language designed to be completely general, good for all possible applications
- Actively promoted by IBM
- An interesting feature introduced in PL/I: Event handling.

Basic

- The first in history of language of personal computing.
- The first programming language for many programmers: designed to be easy to learn.
- Very simple, limited, though still general-purpose.
- Present-day versions of Basic are full-fledged languages—not "basic", and not easy to learn any more.

Simula 67

- An extension of Algol 60 designed for simulation of concurrent processes.
- Introduced the central concepts of object orientation: classes and encapsulation.
- Predecessor of Smalltalk and C++.
- Now unused.

Algol 68

- Extremely difficult to implement.
- A very clever formal description, unfortunately hard to understand for most potential users.
- Completely unused.

Pascal

- A conceptually simplified and cleaned-up successor of Algol 60.
- A great language for teaching structured programming.

- An excellent first language to learn: teaches good programming habits.
- Its later extensions (for example, Delphi) are full-fledged systems programming packages, as powerful as any Java kit.

Modula-2
- A better, conceptually uniform successor of Pascal.
- Mechanisms to program concurrency (many processes running in parallel).
- Not used as much as it deserves.
- Its successors, Modula-3 and Oberon, are even more conceptually appealing, practically useful—and almost not used at all. (They lost the popularity contest with C++.)

Ada
- The result of an elaborate, multi-stage design process, and a more successful attempt at generality than PL/I.
- Completely standard
- Ada has been designed to support concurrency in a very neat, systematic way.

C
- The implementation language of Unix.
- A great tool for systems programming and a software development language on personal computers.
- Once fashionable, still in use, but usually superseded by C++.

Lisp
- One of the earliest programming languages.
- Based on the concept of computing by evaluating functions. Very good for symbolic computing.
- For years, the only language for Artificial Intelligence work. (Prolog is 12 years younger.)

Prolog
- A very high-level programming language.
- Declarative, based on a subset of logic, with proofs interpreted as computation.
- Very powerful:
  - Non-deterministic (built-in backtracking).
  - Elaborate, flexible pattern matching.
  - Associative memory.
  - Pattern-directed procedure invocation.

Smalltalk

- It is the purest object-oriented language ever designed (till now), cleaner than Java, much cleaner than C++.
- Comes complete with a graphical interface and an integrated programming environment.

C++

- An object-oriented extension of the imperative language C.
- This is a hybrid design, with object orientation added to a completely different base language.
- Complicated syntax, difficult semantics.
- Very fashionable, very much in demand.

Java

- A neat, cleaned up, sized-down reworking of C++.
- Full object orientation (though not as consistent as Smalltalk)
- Designed for Internet programming, but general-purpose.

Scripting languages

- Text processing:
  - Perl
  - Python
- Web programming
  - JavaScript
  - PHP

The Language Generations

The language generations span many decades, and begin with the development of machine code. Each generation adds new features and capabilities for the programmer to use. Languages are designed to create programs of a particular type, or to deal with particular problems.

Modern languages have led to the development of completely different styles of programming, involving the use of more human-like or natural language and re-usable pieces of code.

First generation of languages (1GL): These were machine languages. Instructions and addresses were numerical values known as bits (0 or 1). These programs were linked to the machine they were developed on.

Second generation languages (2GL): These languages allowed symbolic instructions and addresses. The program was translated by an assembler. Languages of this generation include IBM, BAL, and VAX Macro. These languages were still dependent on the machine they were developed on.

Third generation languages (3GL): The languages allowed the programmer to concentrate on the problem, rather than the machine they were writing for. Other innovations included structured programming and database management systems. 3GL languages include FORTRAN, COBOL, Pascal, Ada, C, and BASIC. All 3GL languages are much easier for humans to understand.

Fourth generation languages (4GL): These are known as non-procedural, they concentrate on what you want to do rather than how you are going to do it. 4GL languages include SQL, Postscript, and relational database orientated languages.

Fifth generation languages (5GL): These languages did not appear until the 1990s, and have primarily been concerned with Artificial Intelligence and Fuzzy Logic. Programs that have been developed in these languages have explored Natural Language (making the computer appear to communicate like a human being). Fifth generation programming allows people to interact with computers without needing any specialized knowledge. People can talk to computers and the voice recognition systems can convert spoken sounds into written words.

Overview of programming paradigms

A Programming paradigm is a model for a class of Programming Languages that share a set of common characteristics and its differences

- A programming paradigm is a paradigmatic style of programming.
- Provides (and determines) the view that the programmer has of the execution of the program.
- Different programming languages advocate different programming paradigms.
- Some languages are designed to support one particular paradigm, while other programming languages support multiple paradigms

Categories of programming paradigms

Imperative (procedural) paradigm

- Describes computation in terms of a program state and statements that change the program state.
- Most computer languages are in the imperative style.
- Key Features: Stored memory, sequencing, selection, iteration, array and pointers.
- Example : FOTRAN, COBOL, Pascal, Algol, BASIC, Ada, C

Advantages

- Low memory utilization
- Relatively efficient
- The most common form of programming in use today.

Disadvantages

- Difficulty of reasoning about programs
- Difficulty of parallelization.
- Tend to be relatively low level.

Object Oriented paradigm

- Based on imperative style with added data + abstraction and encapsulation.
- Revolutionary concept in software development.
- Key Features :Abstraction, Encapsulation, Polymorphism, Inheritance
- Example : Smalltalk , Java, C++, C#, Visual Basic

Advantages

- Conceptual simplicity
- Models computation better
- Increased productivity.

Disadvantages

- Can have a steep learning curve, initially
- Doing I/O can be cumbersome

Functional paradigm

- Functional programming emphasizes the definition of functions.
- Functional programming languages have largely been emphasized in academia rather than in commercial software development.
- Lambda calculus forms the basis of almost all functional programming languages today.
- Key features: No mutable variables, No Iteration, function and expression, recursive.
- Example : Haskell, Miranda, LISP, Scheme

Advantages

- Small and clean syntax
- Better support for reasoning about programs
- They allow functions to be treated as any other data values.
- They support programming at a relatively higher level than the imperative languages

Disadvantages

- Difficulty of doing input-output
- Functional languages use more storage space than their imperative cousins

Logic (Declarative) paradigm

- Use of pattern-directed invocation of procedures from assertions and goals.
- The point of logic programming is to bring the style of mathematical logic to computer programming.
- Logic provides a way to prove whether the question is true or false.
- Key features: No mutable variables, Statements are logical predicates, Every statements are either succeeds or fails, Recursive.
- Example : PROLOG

Advantages

- Good support for reasoning about programs
- Can lead to concise solutions to problems

Disadvantages

- Slow execution
- Limited view of the world: That means the system does not know about facts that are not its predicates and rules of inference.
- Difficulties in understanding and debugging large programs

Structured programming: problem solving techniques, algorithms, pseudocode, flowcharts

Algorithms

- An **algorithm** is procedure consisting of a finite set of unambiguous rules (instructions) which specify a finite sequence of operations that provides the solution to a problem, or to a specific class of problems for any allowable set of input quantities (if there are inputs).
- In other word, an **algorithm** is a step-by-step procedure to solve a given problem

Performing a task on the computer

1. The first step in writing instructions to carry out a task is to determine what the output should be - that is, exactly what the task should produce.
2. The second step is to identify the data, or input, necessary to obtain the output.
3. The last step is to determine how to process the input to obtain the desired output,
   - that is, to determine what formulas or ways of doing things can be used to obtain the output.

There are two forms of representing Algorithm:

1. Pseudocode
2. Flow chart

Pseudocode

- Pseudocode is a type of structured English that is used to specify an algorithm.
- Pseudocode is an informal high-level description of the operating principle of a computer program.
  - It uses the structural conventions of a programming language, but is intended for human reading rather than machine reading.
- Pseudocode cannot be compiled nor executed, and there are no real formatting or syntax rules.

Example of Pseudocode:

```
Ready and open subscriber file
Get a record
Do while more records
If current subscriber subscription count is > 3 then
Output the record
Get another record
end
```

Advantages of Pseudocode

- Reduced complexity.
- Increased flexibility.
- Ease of understanding.

Why is Pseudocode Necessary?

- The programming process is a complicated one.
- You must first understand the program specifications.
- Then you need to organize your thoughts and create the program.
- You must break the main tasks that must be accomplished into smaller ones in order to be able to eventually write fully developed code.
- Writing Pseudocode will save you time later during the construction & testing phase of a program's development.

How to Write Pseudocode Statements

There are six basic computer operations

1. A computer can receive information
    - Read (information from a file)
    - Get (information from the keyboard)
2. A computer can put out information
    - Write (information to a file)
    - Display (information to the screen)
3. A computer can perform arithmetic
    - Use actual mathematical symbols or the words for the symbols
    - Example:
      Add number to total
      Total = total + number
      +, -, *, /
      Calculate, Compute also used

4. A computer can assign a value to a piece of data
    - **There are 3 cases**
        1. i. to give data an initial value
            - o Initialize, Set
        2. ii. to assign a value as a result of some processing
           '='
           *x=5+y
        3. iii. to keep a piece of information for later use
            - o Save, Store

5. A computer can compare two piece of information and select one of two alternative actions

```
IF condition THEN
    some action
ELSE
    alternative action
ENDIF
```

6. A computer can repeat a group of actions

   -
     ```
     WHILE condition (is true)
             some action
     ENDWHILE
     ```

   -
     ```
     FOR a number of times
         some action
     ENDFOR
     ```

Data Dictionaries

- The pseudo code by itself doesn't provide enough information to be able to write program code.
- Data Dictionaries are used to describe the data used in the Pseudo Code.
- The standard data types used in Pseudo Code are Integer, Double, String, Char and Boolean.

Name	**Data Type**	**Description**
Number1	Integer	The first number to be added
Number2	Integer	The first number to be added
total	Integer	The total of number1 and number2 added together

Example 1:

Program Specification:

Write a program that obtains two integer numbers from the user. It will print out the sum of those numbers.

Pseudocode:

```
Prompt the user to enter the first integer
Prompt the user to enter a second integer
Compute the sum of the two user inputs
Display an output prompt that explains the answer as the sum
Display the result
```

Example 2:
Finding average of any three numbers.
We might usually specify the procedure of solving this problem as "add the three numbers and divide by three". Here, Read (or Ask) and Write (or Say) are implied. However in an algorithm, these steps have to be made explicit. Thus a possible algorithm is:

Example 2:

```
Step 1      Start
Step 2      Read values of X, Y, Z
Step 3      S = X + Y + Z
Step 4      A = S / 3
Step 5      Write value of A
Step 6      Stop
```

Example 3:
Finding square and cube.

```
Step 1     Start
Step 2     Read value of N
Step 3     S = N * N
Step 4     C = S * N
Step 5     Write values of S, C
Step 6     Stop
```

Example 4:
Finding biggest of two numbers.

```
Step 1    Start
Step 2    Read A, B
Step 3    If A > B, then BIG = A, otherwise BIG = B
Step 4    Write BIG
Step 5    Stop
```

Example 5:

Calculate pay.

```
Step 1    Start
Step 2    Input hours
Step 3    Input rate
Step 4    pay = hours * rate
Step 5    Print pay
Step 6    End
```

Exercise:

Write a Pseudocode for these problems.

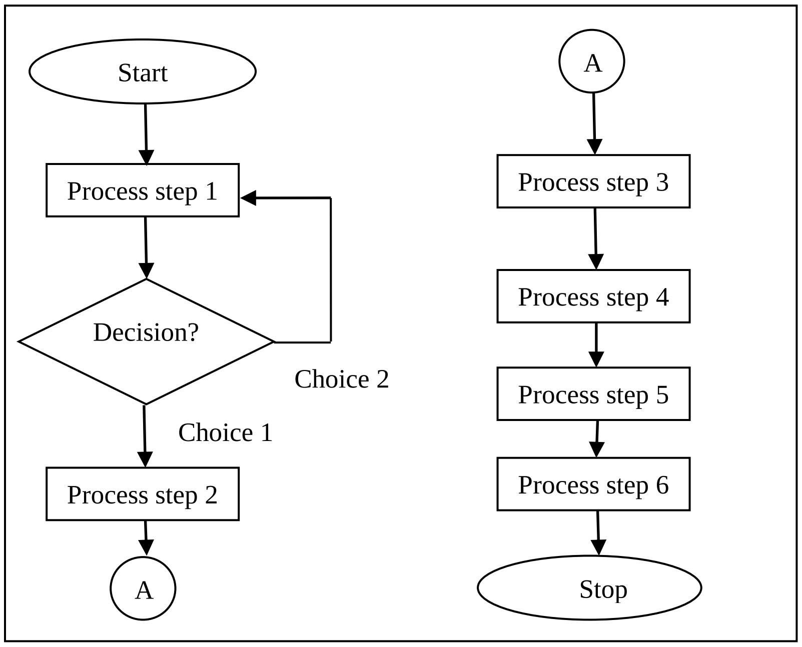
1. S = (A + B + C) / Y
2. Convert from Celsius to Fahrenheit.

   (Multiply by 9, then divide by 5, then add 32)
3. Area of Circle ($\pi r2$)
4. Volume of Sphere ($(\pi r^3)4/3$)
5. Average speed = (Distance traveled)/(Time taken)

Flowcharts

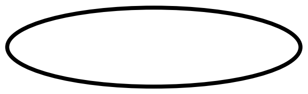
- A **flowchart** is a graphical representation of an algorithm.
- These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems.
- Once the **flowchart** is drawn, it becomes easy to write the program in any high level language.
- A flowchart can therefore be used to:
  - Define and analyze processes
  - Build a step-by-step picture of the process for analysis, discussion, or communication
  - Define, standardize or find areas for improvement in a process

Basic flowchart



Flowchart Symbols

Start and **end** symbols



- Represented as lozenges, ovals or rounded rectangles
- Usually containing the word "Start" or "End", or another phrase signalling the start or end of a process, such as "submit enquiry" or "receive product".

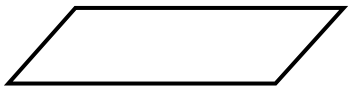
Arrows

- Showing what's called "flow of control" in computer science.
- An arrow coming from one symbol and ending at another symbol.
- Represents that control passes to the symbol the arrow points to.

Processing steps

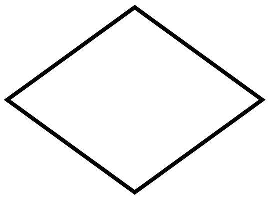
- Represented as rectangles.
- Examples: “Add 1 to X"; "replace identified part"; "save changes" or similar.

Input/Output



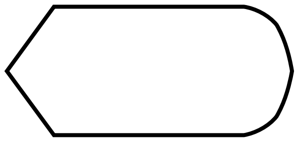
- Represented as a parallelogram.
- Examples: Get X from the user; display X.

Conditional or decision



- Represented as a diamond (rhombus).
- These typically contain a Yes/No question or True/False test.

Display



- Indicates a process flow step where information is displayed to a person (e.g., PC user, machine operator).

Rules for Flowchart

1. Every flow chart has a START symbol and a STOP symbol.
2. The flow of sequence is generally from the top of the page to the bottom of the page. This can vary with loops which need to flow back to an entry point.
3. Use arrow-heads on connectors where flow direction may not be obvious.
4. There is only one flow chart per page.
5. A page should have a page number and a title.
6. A flow chart on one page should not break and jump to another page
7. A flow chart should have no more than around 15 symbols (not including START and STOP).

Advantages of Using Flowcharts

- **Communication**: Flowcharts are better way of communicating the logic of a system to all concerned.

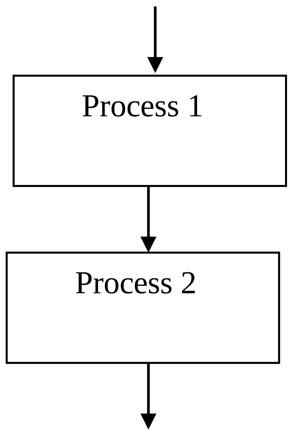
- **Effective analysis**: With the help of flowchart, problem can be analysed in more effective way.
- **Proper documentation**: Program flowcharts serve as a good program documentation, which is needed for various purposes.
- **Efficient Coding**: The flowcharts act as a guide or blueprint during the systems analysis and program development phase.
- **Proper Debugging**: The flowchart helps in debugging process.
- **Efficient Program Maintenance**: The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part.

Basic Control Structures

- Sequence
- Selection
- Loop

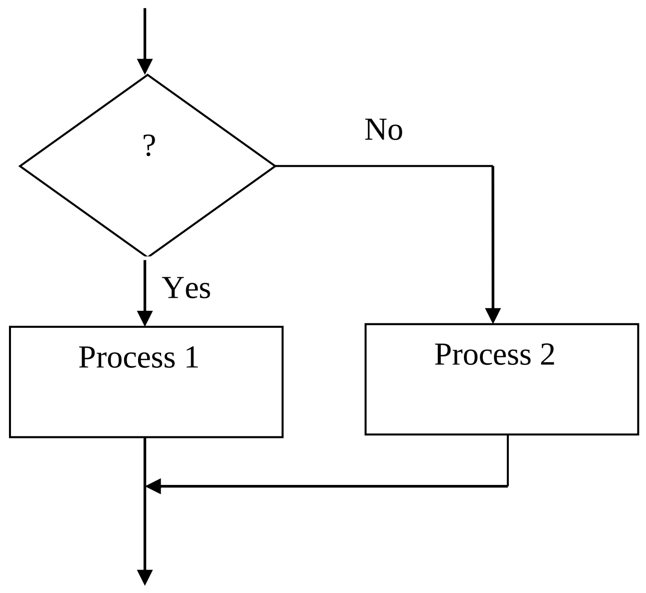
Sequence

- Steps that execute in sequence are represented by symbols that follow each other top to bottom or left to right.
- Top to bottom is the standard.



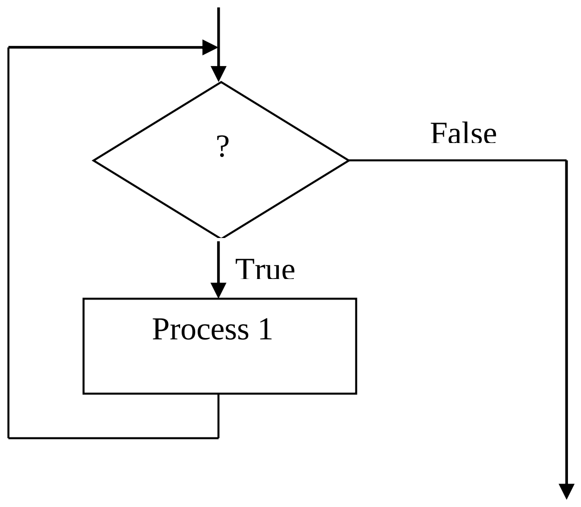
Selection

- Once the condition is evaluated, the control flows into one of two paths.
- Once the conditional execution is finished, the flows rejoin before leaving the structure.



Loop

- Either the processing repeats or the control leaves the structure.
- Notice that the return line joins the entry line before the question.



Example 1:

Algorithm:

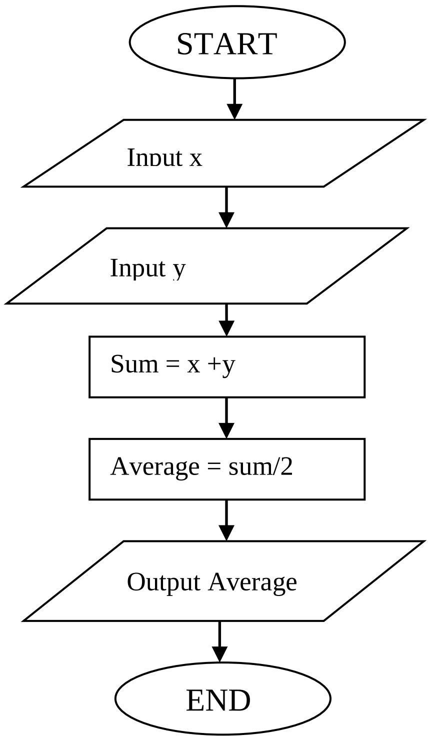
Input: two numbers x and y

Output: the average of x and y

Pseudocode

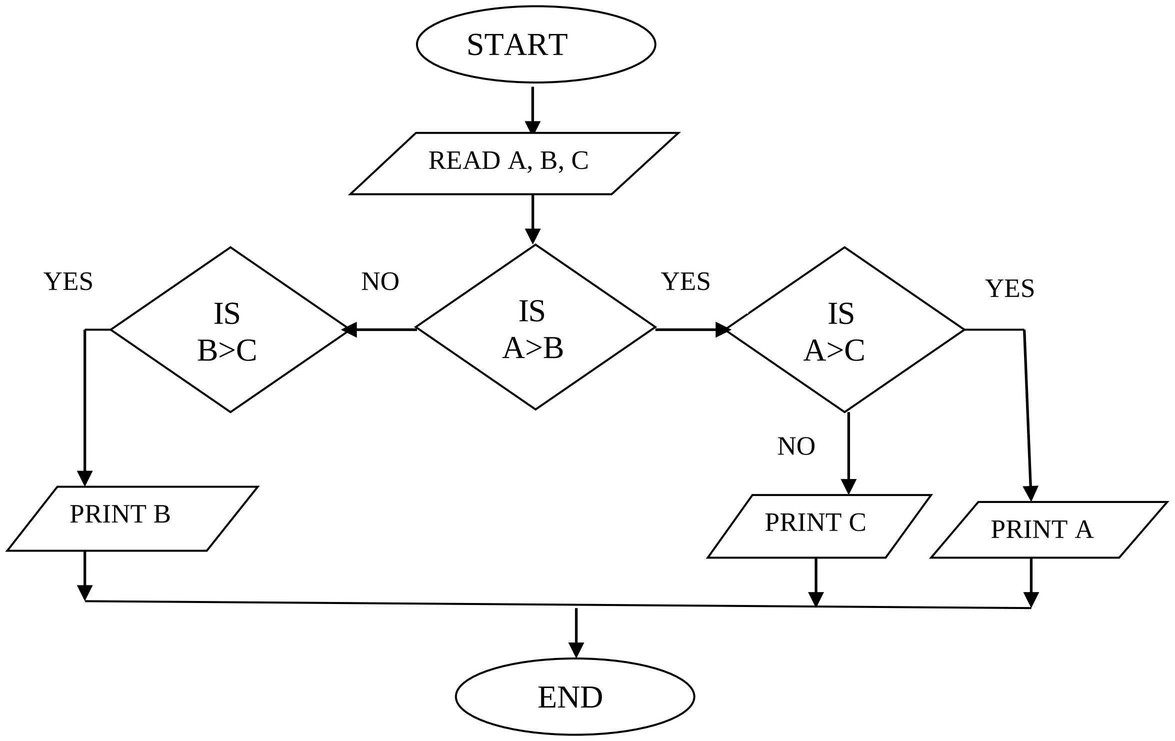
```
Steps:
1. input x
2. input y
3. sum = x + y
4. average = sum /2
5. output average
```

Flowchart



Example 2:

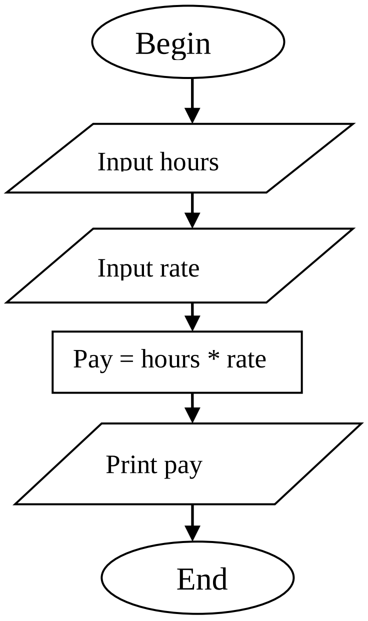
Draw a flowchart to find the largest of three numbers A, B, and C.



Example 3:
Calculate Pay – sequence
Pseudocode

```
Begin
input hours
input rate
pay = hours * rate
print pay
End
```

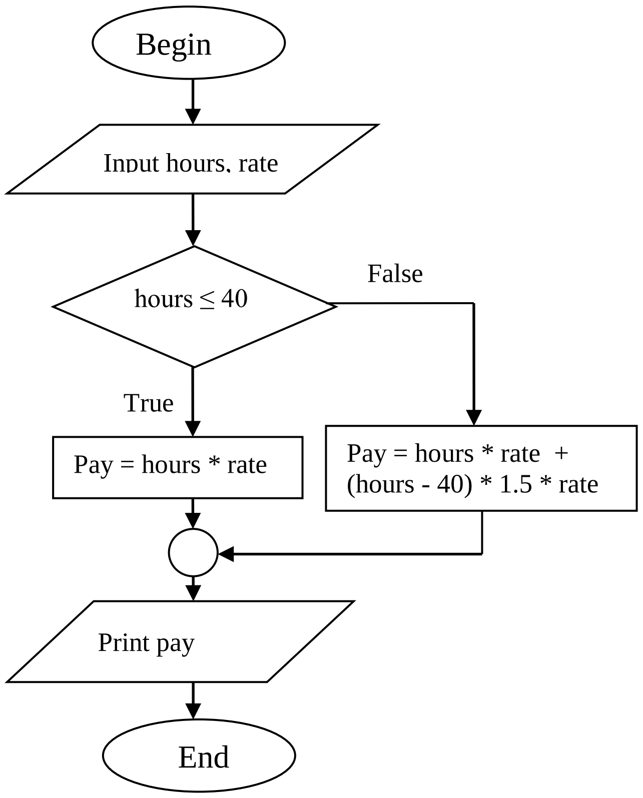
Flowchart



Example 4:
Calculate Pay with Overtime – selection
Pseudocode

```
Begin
input hours, rate
if hours ≤ 40 then
pay = hours * rate
else
pay = 40 * rate + (hours – 40) * rate * 1.5
print pay
End
```

Flowchart



Structure of a PASCAL Program

Turbo Pascal program structure is made up of 4 main sections:

PROGRAM HEADING	Giving the name of the program.
UNITS TO BE USED	A list of all the units required by the program.
DECLARATIONS	Declarations of all objects to be used by the program. These include: **Constants** **Types** **Variables** **Subprograms** (functions and procedures) All these declarations are called GLOBAL declarations.
MAIN PROGRAM	This is where execution starts. It has the following syntax: **Begin** *statements* **End.**

Example of a Simple PASCAL Program

The following example gives an idea of the STRUCTURE of a program in PASCAL.

```
PROGRAM tables;

CONST
TableOf = 3;

VAR
num1, num2, i : INTEGER;
{ This function multiplies two integer numbers and returns their
product.}
FUNCTION ProductOf (num1,num2:INTEGER) : INTEGER;

BEGIN
ProductOf := num1 * num2;
END; {Function ProductOf}

{Main program starts here }
```

```
BEGIN

WRITELN ('Table of ',TableOf);
WRITE ('Enter start:'); {Prompt & input start}
READLN (num1);
WRITE ('Enter end:'); {Prompt & input end}
READLN (num2);
FOR i := num1 TO num2 DO {Use a loop to print out the table}

BEGIN {within the given range}

WRITELN (i,' x ',TableOf,' = ',ProductOf(i,TableOf));
END;
END. {Program tables}
```

Explanations:

1. The program starts with a header which consists of the reserved word **PROGRAM** followed by the name of the program.
2. The next section of the program is the declaration of **CONST**ants. Constant values to be used by the program (e.g. Maximum mark for an exam) should be declared here and be given a meaningful name (eg MaxMark). Thereafter, the program should refer to the constant by its name and not by its value. This has two distinct advantages:
   i) It makes the program more readable
   ii) If the value of the constant is to be changed, one has only to change its declaration rather than change every occurence of the constant within the program.
3. The declaration of **VAR**iables to be used by the program now follows. All variables have to be declared before they can be used. A variable is declared by means of an **IDENTIFIER** (its name) and a **TYPE-SPECIFIER** (to indicate what type of data it will store). In the **VAR** declaration, the line number_of_students, mark, average_mark **: INTEGER**; would serve to declare three variables called **number_of_students**, **mark**, and **average_mark** each being of type **INTEGER**.
4. Next, **subprograms** are declared. In the example, only one subprogram, a **FUNCTION** called **ProductOf**, is declared. This function takes two numbers and returns their product as its result. The program tables.pas from the previous chapter also contains an example of a **PROCEDURE** declaration - a procedure is another type of subprogram.
5. The actual program instructions start after all the declarations. Note that these instructions are enclosed within **BEGIN** and **END** statements. Instructions may be enclosed within nested **BEGIN ... END** statements as shown above. Consequently, it is

good practice to indent the program statements according to their level of nesting. This is allowed because Pascal is a free format language. Note that both upper and lower case characters may be used. No distinction is made between the two. Note also that each statement is terminated by a semi-colon '**;**'. This also holds for **END** in some cases but the **END** statement at the very end of the program *must* be followed by a full stop.

6. Comments can be placed anywhere in the source program and must be enclosed within curly brackets **{** ... **}** or **(*** ... ***)**.

Writing a Pascal program

To write a Pascal program, you must install a Pascal compiler in the hard drives (Hard disk). A Pascal program is written in a code Editor.

Opening a Pascal code Editor

To open a Pascal Programming Language Code Editor, use the following steps:

1. Double Click FreePascal IDE Icon from the Desk top.
2. The code Editor window opens.
3. Click File Menu then Click Save As Menu item
4. Type LAB25_1 under Name text field in the window that appears.
5. Click OK Button to save an empty file.
6. Your Code Editor's name should change to LAB25_1.pas

Write the following simple Pascal program in the code editor the click save menu item from the menubar.

```
Program LAB25_1;
Const
Tax_Rate = 0.2;
UnitPrice = 50.00;

Var
quantity : Integer;
taxAmount, totalAmount,  netAmount : Real;

Begin
Writeln('Enter the quantity of item :');
Readln(quantity);
totalAmount := quantity*UnitPrice;
taxAmount := totalAmount*Tax_Rate;
netAmount := totalAmount – taxAmount;
```

```
Writeln('Tax rate = ', Tax_Rate);
Writeln('Unit price = ', UnitPrice);
Writeln('Quantity = ', quantity);
Writeln('Tax amount = ', taxAmount);
Writeln('Total amount = ', totalAmount);
Writeln('Net amount = ', netAmount);
End.
```

Opening an existing Pascal program

If you want to do modification (editing) on an existing program, open the program as follows:

1. Click on File menu then Click on Open
2. Click on the name of the file (e.g LAB25_1.PAS) from a list of files in the Open a file window.
3. Click on the Open Button.
4. The code Editor for the selected file opens.
5. Do your modification then save the changes.
6. You can then compile and run your program.

Compiling a Pascal Program

Compilation is the process of translating the source code written in high level language (Pascal) into a machine language. During compilation, syntax errors are detected. Syntax errors are errors generated by the compiler if proper syntax (rules) of the programming language is not followed. Any detected syntax error must be debugged to enable a program to be executed. Any program with syntax error cannot be executed.

In order to compile a program:

1. Click Compile menu from editor containing the code to be compiled.
2. Click compile from the dropdown menu
3. If there is no syntax error, a compiling window will appear indicating that compilation is successful. Press any key to return to the code window.
4. You can then run your program.
5. If there are some syntax errors, the compiler will display them on the code window
6. Debug all the detected syntax errors then run your program.

Running (Executing) a Pascal program

Executing a program refers to the process of running a compiled program in order to generate output. During program execution, logical errors e.g run time errors can be detected. Detection of some logical errors requires careful analysis of output data against the input data generating them. For example, a program can be doing addition instead of multiplication.

Do the following in order to execute a Pascal program:

1. Click Run Menu, then click Run from the dropdown menu
2. If the program does not require any input from the keyboard, then the results will be displayed in the output section of the editor.
3. If the program requires some input from the keyboard, then a DOS window will be displayed to enable the user to input the data from the keyboard.
4. Any detected run time errors will be displayed in the output window of the editor.
5. In order to detect some logical errors, you need to analyze the input data and the expected output.

Reserved Words

These are words that have predefined meanings in Pascal that cannot be changed.
These words cannot be used as a name of a variable or an object.
Examples include:

- and
- array
- begin
- case
- const
- div
- do
- downto
- else
- end
- file
- for
- foward
- function
- goto
- if
- in
- label
- mod
- nil
- not
- of
- or
- packed
- procedure
- program
- record
- repeat
- set
- then
- to
- type
- until
- var
- while

Data types

A data type specifies the type of data to be stored in a memory location. A memory location can be declared as a variable or a constant. The memory location of a variable can change during program execution. The content of a constant cannot change during program execution.

The following basic data types are supported in Pascal:

Char

The Char data type is used for individual 8-bit codes, many of which are used to represent printable characters. Each Char requires one byte of memory and can have a value from 0 to 255. The value of a Char variable must be enclosed in single quotes e.g. ‘A’.

Boolean

A variable of the Boolean type can have only two values, TRUE and FALSE. Zero (0) is used to represent FALSE and one (1) is used for TRUE.

Integer

An Integer is a two-byte ordinal data type. The possible range is from -32768 to 32767.

Examples of integer values are: 2, 28, 690, 32700

Longint

A LongInt is a four-byte ordinal data type, making it twice as long as an Integer in terms of bits. The possible range is from -2147483648 to 2147483647.

Examples of LingInt values are: 195720, 8564420, 4378009

Real

A Real is a six-byte *non*-ordinal data type. This means that it *cannot* be used as the loop control variable in a *for-to-do* loop. The range of a positive Real variable is from 2.9 x 10^{-39} to 1.7 x 10^{38}. A Real maintains about 11 significant digits.

Examples of Real values are: 0.15, 6.875, 888431.085

String

A string is used to store a string of zero or more characters. Any value of string type must be enclosed in single quotation marks e.g 'Kenya' .
Examples of String values are: 'Dan Kibet', 'Pascal', 'Date of birth'

Variables and constants

Variables

A variable is a memory location whose value can change during program execution. Any variable in Pascal must be declared before it is used in a program. A variable must have a valid identifier (name) and a data type indicating the type of data to be stored in it.

Types of variables

There are two main types of variables: Local variables and global variables.

Local variables: These are variables that are declared within a procedure or a function. These variables can only be used within the procedure or function they have been declared in.
Global variables: These are variables that are declared within the **var** section of a Pascal program. The scope of these variables is the entire program, meaning that any procedure or function defined in the program can manipulate these variables.

Variable declaration: This is the statement that enables the compiler to allocate a memory space for a given variable. A variable declaration must have a valid identifier (name of variable) and a data type indicating the type of data to be stored in it. The reserved word "**var"** is used to declare a variable.

Rules for naming a variable

1. Name must not be any of the reserved word
2. Name must start with an alphabet letter
3. There should not be any space between the characters of an identifier
4. If the name is composed of two words, use an underscore to separate the words, alternatively, write the two words continuously by capitalizing the first letter of the second word.
5. The name must contain between 1 and 255 characters

General syntax of variable declaration

Var VariableName : Data type; (Each program statement must be terminated by a semi colon ;)

Examples of variable declaration

```
Var marks : Integer;
Var name : String;
Var grade : String;
Var gender : Char;
Var population : LongInt;
Var basic_pay : Real;
Var grossPay : Real;
Var done : Boolean;
Var passed : Boolean;
```

Accessing and assigning values to variables

Accessing a variable refers to locating the variable to enable values to be placed in it or be retrieved from it. A variable is accessed using its name. Assigning a value to a variable refers to the act of placing or storing data or information in the variable. The assignment operator (:=) is used to assign a value to a variable.

Examples of value assignment to variables

```
marks := 85;
name := ‘Benson’;
grade := ‘A’;
gender :=  ‘M’;
population := 42000000;
basic_pay := 28000.00;
grossPay := 45000.50;
done := True;
passed := True;
```

Constants

A constant is a memory location whose value does not change during program execution.
Constant declaration: This is used to identify the name of the constant and the value it is containing. A constant is declared using the reserved word **const** as follows:

General syntax:

```
Const  ConstantName  =  value ;
```

Examples

```
Const TaxRate = 0.2;
Const Birth_Rate = 0.15;
Const Institution_Name = ‘ Maseno University’;
Const Male = ‘M’;
Const Female = ‘F’;
Const MinGrade = ‘B’;
Const MaxPay = 800000.00;
Const Starting = True;
```

Operators in Pascal

An operator is a symbolic representation of an operation in a programming language. Operators are used to manipulate data and variables in a program. There are three main types of operators:

1. Arithmetic operators
2. Comparison operators
3. Logical operators

Arithmetic operators: Are used for mathematical computations to manipulate data.

The following are the arithmetic operators supported in Pascal:

Operator (Symbol)	Operation	Usage
+	Addition: Adds two or more numbers to give the sum.	20 + 15 = 35
-	Subtraction: Subtract two or more numbers to give the difference.	56 – 30 = 26
*	Multiplication: Multiply two or more numbers to give the product.	45 * 2 = 90
/	Real number division: Divides two real numbers to give a real number quotient.	65/2 = 32.5
DIV	Integer division: Divides two integer numbers to give an integer value.	73 DIV 2 = 36
MOD	Remainder (Modulo): Divides two integer numbers to give the remainder.	20 MOD 3 = 2

Comparison operators: Are used to compare two values to determine their equality or inequality. They are used for decision making in program execution. They are very useful in controlling the execution of a program.

Examples of comparison operators are:

Operator	Meaning	Usage
<	Less than: Decision is made whether the left value is less than the right value	4 < 9
>	Greater than: Decision is made whether the left value is greater than the right value	10 > 3
=	Equal to: Decision is made whether the left value is equal to the right value	8 = 8
<=	Less than or equal to: Decision is made whether the left value is less than or equal to the right value	5 <= 5
>=	Greater than or equal to: Decision is made whether the left value is greater than or equal to the right value	12 >=12
< >	Not equal to: Decision is made whether the left value is not equal to the right value	3 < > 7

Logical operators: They are used for logical decision making. Boolean expressions can be combined using the logic operators **NOT**, **AND** and **OR**, sometimes also called **boolean operators**.

Three examples of logical operators are:

operator	Operation	Results
AND	X AND Y	TRUE if and only if both X and Y are TRUE
OR	X OR Y	TRUE if at least one of X and Y is TRUE
NOT	NOT X	The inverse of X

Simple input and output

For a Pascal program to interact with a user for the purpose of problem solving, the program should be able to read(data) from the keyboard, process the data and display the output on the screen.

Reading data from the keyboard
This is done using the read or readln function. These two functions enable data to be read from the keyboard and stored in the variable.
Example
```
Var category : String;
Read(category);  OR Readln(category);
```

Displaying output on the screen
The output is displayed on the screen using Write or Writeln function. These functions can be to display string literals, the content of a variable or a constant. Write function will continue to display the data items on the same line.

Example
```
Var marks : Integer;
Writeln('Enter an integer number: ');
Readln(marks);
Writeln('The marks entered = ',marks);
```

Example
Write a Pascal program that will allow a user to enter the integer values of length and width of a rectangle from the keyboard. The program should compute the area and perimeter of the rectangle and displays the result on the screen.

Solution

```
Program Rectangle;
Var length, width, area, perimeter : Integer;
Begin
        Writeln(‘Enter length : ’);
         Readln(length);
         Writeln(‘Enter width : ’);
         Readln(width);
        Area := length*width;\
        Perimeter = 2*(length + width);
        Writeln(‘The area = ’, area);
        Writeln(‘The perimeter = ’,perimeter);
End.
```

Control structures

Control structures are the statements that control the flow of execution of program statement. It is important for program statements to have proper flow of execution in order to generate the correct output. There are three main categories of control structures supported in Pascal:

1. Sequence structures
2. Selection (Decision) structures
3. Iteration/Repetition/Loop structures

Sequence structures

Sequence control structure allows program statements to be executed sequentially in the order they have been written in the code. For the program to generate the correct results, the program statements must be written in the correct order. If the statements are not ordered properly then the results will be wrong.

Example

```
Program Salary;
Var name : String;
Var salary : Real;
Begin
   Writeln(‘Enter name : ’);
   Readln(name);
   Writeln(‘Enter salary : ’);
   Readln(salary);
   Writeln('The name is:  ',salary);
   Writeln('The salary = ',salary);
End.
```

Selection (Decision) structures

Selection control structure enables some Boolean expression to be evaluated in order for some code statements to be executed if the result of evaluation is true or false. There are three types of selection structures:

1. If…then
2. If…then…else
3. Case…of

If…then structure

This structure allows a statement or a block of statements to be executed if the result of a Boolean expression is true. If the result is false, the statements within the structure are not executed. The program goes and executes the first statement after the structure. If…then structure is also called one-way selection structure.

General Syntax

```
If(Boolean expression) then
 Begin
    Statements;
End;
```

Example 1

```
If (marks<40) then
Writeln('You have failed');
```

Example 2

```
taxRate := 0.2;
  if (income < 10000) then
 (*begin and end; are required if the statements are more than one*)
    begin
      writeln('Eligable for social assistance');
      taxCredit = 100;
    end;
  tax = income * taxRate;
```

Example 3

```
(*A program to divide an integer number by another integer*)
Program Divide;
        Var num1,num2,result : Integer;
Begin
         Writeln('Enter first integer number: ');
           Readln(num1);
           Writeln('Enter second integer number: ');
           Readln(num2);
If(num2>0) then
    Begin
      Result := num1 DIV num2;
     Writeln('The result = ',result);
    End;
End.
```

If…then…else

This control structure allows one block of statement to be executed if the Boolean expression is true, and another block of statements to be executed if the result of the evaluation is false. If…then…else is also called a two-way control structure.

General syntax

```
if (boolean-expression) then
Begin
        statements;
End {No semicolon required}
    Else
Begin
        statements;
End; (*Marks the end of control structure*)
```

Example 1

```
if (age >= 18) then
    writeln('Adult') {No semicolon required for a single statement}
  else
    writeln('Not an adult'); {semicolon marks the end of control structure}
```

Example 2

```
if (income < 10000) then
   begin
     writeln('Eligible for social assistance');
     taxRate := 0.1;
   end
else
   begin
     writeln('Not eligible for social assistance');
     taxRate := 0.2;
   end;
tax = income * taxRate; {statement after the control structure}
```

Example 3

```
(*A program to divide an integer number by another integer*)
Program Divide;
          Var num1,num2,result : Integer;
Begin
           Writeln('Enter first integer number: ');
             Readln(num1);
             Writeln('Enter second integer number: ');
             Readln(num2);
If(num2>0) then
     Begin
        Result := num1 DIV num2;
       Writeln('The result = ',result);
     End {End of if block, no semicolon required}
Else
Begin
Writeln('Division by Zero or less is not allowed');
End; {End of control structure}
End.
```

Nested decision making

This allows more than one decision to be made in a layered sequence. The outer decision must evaluate to true before the inner decision is evaluated.

Example 1

```
if (income <  10000) then
  if (citizen = true) then
    writeln('Eligable for social assistance');
tax := income * TAX_RATE;
```

Example 2

```
if (x > 0) then
     if (y > 0) then
     writeln('x and y greater than zero')
else
writeln('x is greater than zero');
```

Decision-Making with multiple alternatives

If a program is required to make more than three decision alternatives, then it can use multiple if…then OR Else if…then.

Example 1

```
if (x > 0) then
     writeln('X is positive');
if (y > 0) then
     writeln('Y is positive');
if (z > 0) then
     writeln('Z is positive');
```

Example 2

```
if (gpa = 4) then
     letter := 'A';
   if (gpa = 3) then
     letter := 'B';
   if (gpa = 2) then
     letter := 'C';
   if (gpa = 1) then
letter := 'D';
     if (gpa = 0) then
letter := 'F';
```

Example 3

```
if (gpa = 4) then
  letter := 'A';
else if (gpa = 3) then
  letter := 'B';
else if (gpa = 2) then
  letter := 'C';
else if (gpa = 1) then
  letter := 'D'
else if (gpa = 0) then
  letter := 'F'
else
  writeln('GPA must be one of 4, 3, 2, 1 or 0');
```

Case… of decision structure

This structure is used to make multiple decision alternatives. It evaluates the value of an expression and determines if it matches any of the case constants already defined in the structure. If the value of the expression matches the case constant then a block of statements under that constant is executed. If there no match then a default statement is executed. The case constant must be an integer value.

General syntax

```
CASE < expression > OF
const1:
           Begin
                 < statement 1 >;
           End;
const2:
          Begin
                 < statement 2 >;
          End;
...
constn:
          Begin
                  < statement n >;
           End;
ELSE
          Begin
                 < default statement >;
          End; { of case }
```

Example 1

```
Var mark : Integer;
comment : STRING ;
CASE mark OF
10 : Begin
           comment := 'Full Marks';
      End;
9 : Begin
         comment := 'Excellent';
     End;
8 :Begin
           comment := 'Very Good';
    End;
```

```
7 : Begin
      comment := 'Fairly Good';
      End;
6 : Begin
               comment := 'Not Bad';
      End;
5 : Begin
            comment := 'Can do better';
     End;
4 : Begin
             comment := 'Poor';
      End;
3 : Begin
             comment := 'Very Poor';
     End;
2 : Begin
             comment := 'Terrible';
     End;
1 : Begin
             comment := 'Rubbish';
     End;
0 : Begin
            comment := 'Total loss!'
     End;
ELSE
     Begin
              comment := 'Invalid Mark';
     End;
```

Example 2

```
(*A program to compute the area of different objects*)
Program ComputeArea;
             Const PI = 3.142;
              Var area: Real;
              Value,radius,length,width: Integer;
    Begin
        {Generation of menu to chose from}
              Writeln('Type 1 to compute area of a circle');
              Writeln('Type 2 to compute area of a square');
              Writeln('Type 3 to compute area of a rectangle');
        Readln(value);
     CASE value OF
      1:
         Begin
              Writeln(‘Enter the radius’);
               Readln(radius);
               Area := PI*radius*radius;
              Writeln(‘The area of a circle  =  ‘,area);
          End;
      2:
          Begin
               Writeln(‘Enter the length’);
               Readln(length);
              Area := length*length;
              Writeln(‘The area of  a square  =  ‘,area);
           End;
      3:
           Begin
                Writeln(‘Enter the length’);
                 Readln(length);
                Writeln(‘Enter the width’);
                 Readln(width);
                Area := length*width;
                Writeln(‘The area of  a rectangle =  ‘,area);
            End;
     ELSE
           Writeln(‘The object is not defined’);
           End; {End of the CASE OF structure}
End. {End of the program}
```

Iteration/Repetition/Loop structures

A **loop** is a **control structure** which **repeats** one or more statements if the evaluation of a certain condition is met. If the condition is not met then the execution of the statements within the loop is terminated. The program then goes to the statement after the loop. Pascal supports three kinds of looping mechanisms: **FOR…DO, REPEAT…UNTIL** and **WHILE…DO**.

1. The **FOR…Do** loop has two variations - increasing counter and decreasing counter.
2. The **REPEAT…UNTIL** loop tests its termination condition at the end of the loop
3. **WHILE…DO** loop tests its termination condition at the beginning of the loop.

It is very important to determine the kind of loop which best serves a particular situation.

The **REPEAT** loop differs from the **WHILE** loop in that the **REPEAT** loop tests its termination condition at the end of the loop, while the **WHILE…DO** loop tests its termination condition at the beginning of the loop.

FOR…DO loop

```
SYNTAX 1:  FOR <counter> := <start value> TO <end value> DO
                          <statement> ;
SYNTAX 2: FOR <counter> := <start value> DOWNTO <end value> DO
                           <statement> ;
```

Explanation:

- **Counter:** Is the variable whose value is being evaluated
- **Start value:** is the initial value of the counter variable
- **TO:** is a function that increments the value of counter by 1in each iteration
- **DOWNTO:** is a function that decrements the value of counter by 1in each iteration
- **End value:** is the last counter value that determines the loop termination condition
- Counter can be given any valid variable name

Example1

```
Var num : Integer;
FOR num := 8 DOWNTO 1 DO
Begin
      Writeln('The number = ',num);
End;
```

Example2

```
Var i,square : Integer;
FOR i:=1 TO 10 DO
Begin
Square := i*i;
Writeln('The value of  ', i,  'squared = ',square);
End;
```

Example3

```
(*A program to display odd integer numbers*)
Program Factors;
        Var int ,j: Integer;
           Begin
      Writeln('Enter an integer: ') ;
      Readln(int);
        FOR j := ABS(int) DOWNTO 1 DO
          Begin
             IF (j MOD 2 = 1) THEN { if INT is divisible by 2 }
             Write(j,' ,');
          End;
    End;
```

REPEAT…UNTIL loop

- The condition is tested at the end of the loop
- It allows the statements to be executed at least once before the condition is tested
- The statements are executed when the condition is false
- When the condition becomes true, loop execution terminates

SYNTAX

REPEAT <statements> **UNTIL** <condition> ;

Example 1

```
Var num,sun : Integer;
Begin
  Num := 1;
  Sum : 0;
Repeat
Begin
Sum := sum + num;
num := num + 1;
End;
Until num>=10;
```

Example 2

```
Var n,p : Integer;
Begin
  n := 15;
  p : 1;
Repeat
Begin
p := p + n;
n := n - 1;
End;
Until num<=0;
```

Example 3

```
Var n : Integer;
Begin
  n := 1;
Repeat
Begin
Writeln('*');
n := n + 1;
End;
Until num > 5;
```

WHILE…DO loop

- Loop condition is tested at beginning of the loop
- Statements are executed while the condition is true
- When the condition becomes false, execution of the loop statements is terminated
- Since the condition is tested at the beginning of the loop, the statements in the loop may never get executed.

SYNTAX

WHILE <condition> **DO** <statement> ;

Example1

```
var i ,s: integer;
i: = 1;
s := 0;
while (i <= 5) do
  begin
    s := s + j;
    i := i + 1;
  end; (* while *)
```

Example 2

{Starting with any positive integer, generate a sequence such that the successor of a number N in the sequence is N*3+1 if N is odd, and N/2 if N is even. It appears that from whichever INTEGER you start, you will eventually generate a 1.}

```
Var j: Integer;
Writeln('Enter a positive non-zero integer: ');
Readln(j);
While j <> 1 DO
BEGIN
IF ODD(j) THEN
Begin
j := j * 3 + 1
End
ELSE
Begin
j := j DIV 2;
WRITE (j:8);
End;
```

Procedures and functions
A computer program is supposed to solve problems and provide solutions to the user. If the number of problems to be solved are few, only one program code is enough to provide the required solution. As the number of problems increases, it becomes very difficult to include all the solutions in a single unit program code. In order to provide solution to complex problems, the main program is normally divided into subprograms called modules. In programming, each module is responsible for solving one problem, hence making problem solving easier. A module can either be a procedure or a function.

Procedures
A procedure is subprogram that performs a given task but does return a value to the calling program.
There are two types of procedures;

1. Predefined procedures
2. User defined procedures

Predefined Procedures
These are the procedures that have been defined in the Pascal language to perform specific tasks. Examples of predefined procedures are:

- dispose
- get
- new
- pack
- page
- put
- read
- readln
- reset
- rewrite
- unpack
- write
- writeln

User defined procedures
These are the procedures that are defined by a user to perform some specific tasks.

User defined procedure definition (Declaration)
A procedure must be declared in the program in order for it to be recognized by the compiler. Some procedures do not require parameters during declaration, but others require parameters when being declared. **Parameter** is a variable used by a procedure to receive values during the procedure call. The value that is passed to the procedure during call is referred to as an

argument. A procedure is declared in the declaration section of the Pascal program structure. A procedure is declared using the keyword **Procedure** followed by a valid name of the procedure and a semicolon.

User defined procedures without parameters

These procedures do not require parameters during declaration. They also do not require arguments during their call.

General format for declaration

```
procedure nameOfProcedure;
   Begin
         Statements of the procedure ;
   end;  { End of procedure}
```

Example 1

```
Procedure DisplayInformation;
         Begin
           Writeln ('I have been enjoyed learning Pascal language');
           Writeln (' I wish other programming languages were as enjoyable as Pascal');
         End;  (* End of procedure DisplayInformation *)
```

Example 2

```
Procedure AddTwo;
  Var
       x,y, s : Integer;
       Begin
           Writeln(‘Enter first number:’);
            Readln(x);
            Writeln(‘Enter second number:’);
            Readln(y);
           s := x + y;
        Writeln(‘The sum = ’, s);
End;
```

Calling a procedure without parameters

For a procedure to perform its task, it must be called by another program. A procedure can be called by another procedure or by the main program. A procedure is called using the name of the procedure as follows:

ProcedureName;

e.g

```
DisplayInformation; {procedure call without arguments}
AddTwo;
```

Example 3

```
Program DisplayInfo;   {A program to display information using a procedure}
  Procedure DisplayInformation;
        Begin
          Writeln ('I have been enjoyed learning Pascal language');
          Writeln (' I wish other programming languages were as enjoyable as Pascal');
        End;  (* End of procedure DisplayInformation *)
Begin
     DisplayInformation;  {procedure call without arguments}
End.
```

Example 4

```
Program SumTwo;        {A program to add two numbers}
  Procedure AddTwo;
  Var
       x,y, s : Integer;
       Begin
          Writeln('Enter first number:');
           Readln(x);
           Writeln('Enter second number:');
           Readln(y);
          s := x + y;
       Writeln('The sum = ', s);
End;
Begin
     AddTwo; {procedure call without arguments}
End.
```

Declaring a procedure with parameters
General format:

```
Procedure DisplayInformation(p1 : datatype1; P2 : datatype2; pn : Datatypen);
Local variable declaration
Begin
  Statements within the procedure;
End;
```

OR

```
Procedure DisplayInformation(pP, P2, Pn : Datatype);
Local variable declaration
Begin
  Statements within the procedure;
End;
```

Example 5

```
Procedure MultiplyTwo(x, y : Real);
Var p : Real;
Begin
       P := x * y;
       Writeln('The product = ', p);
End;
```

Calling a procedure with parameter

A procedure with parameters is called by passing parameter values or arguments to the procedure.

e.g

```
MultiplyTwo(25, 50);  {procedure call with arguments}
```

Example 6

```
Program Multiplication; {A program to multiply two numbers using a procedure}
Var
       Num1, num2 : Real;
       Procedure MultiplyTwo(x, y : Real);
       Var p : Real;
      Begin
              P := x * y;
             Writeln('The product = ', p);
      End;
```

```
Begin
       Writeln('Enter the first number: ');
        Readln(num1);
        Writeln('Enter the second number: ');
        Readln(num2);
     MultiplyTwo(num1, num2); {procedure call with arguments}
End.
```

Example 7

Write a Pascal program that uses a procedure called Averaging to compute the average of three Real numbers. The values used to call the procedure are entered from the keyboard

```
Program ComputeAverage;
    Var num1, num2, num3 : Real;
    Procedure Averaging(x, y, z : Real);
        Var average : Real;
         Begin
                average:= (x + y + z)/3;
                Writeln('The average = ', average);
       End;
Begin
       Writeln('Enter first number : ');
       Readln(num1);
       Writeln('Enter second number : ');
       Readln(num2);
       Writeln('Enter third number : ');
       Readln(num3);
     Averaging(num1, num2, num3);
End.
```

Methods of calling a procedure or a function

There are two ways that can be used to call a procedure or a function:

1. **Call by value:** A copy of the value is passed to the procedure. The calling program cannot change the original value of the parameter.
2. **Call by reference:** The memory location of the variable is passed to the called procedure. The calling program can change the value of the variable

Functions

A function is subprogram that performs a given task and returns a value to the calling program. There are two types of functions:

1. Predefined functions
2. User defined functions

Predefined Functions

These are the functions that have been defined in the Pascal language to perform specific tasks. Examples of predefined functions are:

- abs
- arctan
- chr
- cos
- eof
- eoln
- exp
- ln
- odd
- ord
- pred
- round
- sin
- sqr
- sqrt
- succ
- trunc

User defined functions

These are the functions that are defined by a user to perform some specific tasks.

User defined function definition (Declaration)

A function must be declared in the program in order for it to be recognized by the compiler. When declaring a function, you must specify the function name, a list of parameters and their data types, and the return type. Parameters are used to store values during the function call. The value that is passed to the function during call is referred to as an **argument.** A function is declared in the declaration section of the Pascal program structure. A function is declared using the keyword **Function** followed by a valid name of the function and a parameter list enclosed in arc brackets followed by a full colon then return data type and a semicolon.

Format for defining a function:

```
   function name (Name of parameter 1 : type of parameter 1;
                       Name of parameter 2 : type of parameter 2;
                                   :                   :
                           Name of parameter n : type of parameter n):  return type;
    begin
           Statements of the function ;
                :                :
        name := expression; (* Return value *)
    end;
```

Example 1

```
   function calculateGrossIncome (grossSales  : real;  costOfGoodsSold : real) : real;
   begin
        calculateGrossIncome := grossSales - costOfGoodsSold;
   end;
```

Example 2

```
Function ComputePerimeter(length : Integer; width : Integer ) : Integer;
Var
Perimeter : Integer;
Begin
      Perimeter := 2 * (length + width);
      ComputePerimeter := perimeter;
End;
```

Example 3

```
Function RegisterName(name : String) : String;
Begin
      RegisterName:= name;
End;
```

Example 4

```
Function Product(x : Real; y : Real; z : Real) : Real;
Begin
       Product := x * y * z;
End;
```

Calling a function
For a function to perform its intended task, it must be called by another program. A function can be called by a procedure or the main program. A function is called by passing parameter values or arguments to the function.

General format
Variable := FunctionName(argument1, argument1,.., argumentn);

Example 5
grossIncome := calculateGrossIncome(23339000, 156000);

Example 6
Prod := Product(6, 90);

Example 7
```
Var p, l, w : Integer;
Writeln('Enter the value of length: ');
Readln(l);
Writeln('Enter the value of width: ');
Readln(w);
P := ComputePerimeter(l, w);
```

Example 8
Write a Pascal program that uses a function called ComputeVolume. The function receives the values of length, width, and height of type Real, the compute the volume and returns a Real value. The function is called by values received from the keyboard. The program should display the value of the computed volume on the screen

Solution
```
Program CubeVolume;
Var
Length, width, height, volume : Real;
Function ComputeVolume(l, w, h : Real) : Real;
Var v : Real;
Begin
        V := l * w * h;
        ComputeVolume := v;
End;
Begin
      Write('Enter lenth:');
```

```
          Readln(length);
          Write(‘Enter width:’);
          Readln(width);
         Write(‘Enter heght:’);
          Readln(height);
        Writeln(‘Length = ’, length , ‘ , ’ , ‘Width = ’, width, ‘ , ’ , ‘Height = ’, height  );
      Volume := ComputeVolume(length, width, height);
        Writeln(‘The volume = ’ , volume);
    End.
```

Example 9

```
{Implementing a function in the main program}
Program ComputeProduct;
Var prod, num1, num2, num3 : Real;
    Function Product(x : Real; y : Real; z : Real) : Real;
    Begin
           Product := x * y * z;
    End;
Begin
      Writeln(‘Enter the first number: ‘);
      Readln(num1);
     Writeln(‘Enter the first number: ‘);
     Readln(num2);
     Writeln(‘Enter the first number: ‘);
     Readln(num3);
     Prod :=  Product(num1, num2, num3);  (* Function call with arguments as variables*)
    Writeln(‘The product = ’, prod);
End.
```

Arrays and Records

Arrays and records are used to store multiple data items.

Arrays

An array is a collection of data items of the same data type under one name. The reason for using arrays as data storage is to enhance easier data referencing and economy of storage space. There are two types of arrays:

1. Single dimensional array
2. Multidimensional array

Single dimensional arrays

A single dimensional array stores data elements in a single row or a single column. Each element is referenced by a single index.

Single dimensional array declaration

An array must be declared for it to be used in a program. An array must have a valid name, size and the data type to be stored in it.

General syntax:

```
Var ArrayName : Array[startIndex..EndIndex] of DataType;
```

- ArrayName is any valid name accepted by Pascal
- Array is the reserved word indicating that the variable is an array
- startIndex..EndIndex indicates the size of the array
- DataType represents the type of data to be stored in the array.

Examples:

```
Var
Names : Array[1..100] of String;
Grades: Array[1..100] of char;
Marks: Array[1..50] of Integer;
Salaries: Array[1..10] of Real;
Population: Array[1..10] of LongInt;
```

Assigning values to an array

A value is assigned to an array as follows:

Syntax

```
ArrayName[Index] := value;
```

Examples

```
Names[1] := ‘Molly Cebet’;
Names[2] := ‘Dan Ouma’;
Grade[20] := ‘B’;
Grade[21] := ‘A’;
Salaries[3] := 450000;
Salaries[4] := 850000;

For index:=1 To 5o Do
Begin
      Writeln(‘Enter marks’);
      Readln(m);
       Marks[index] := m;
End;
```

Retrieving data from an array

Data stored in an array can be retrieved and placed in a variable or displayed on the screen. The variable to store the data from an array must have the same data type as the array. The content of an array can also be retrieved for manipulation to produce some results.

Example

```
Var name1, name2 : String;
Name1 := Names[1];
Name2 := Names[2];

Var sal1, sal2 : Real;
Sal1 := Salaries[3];
Sal1 := Salaries[6];

Var sum : real;
Sum := 0;
For index:=1 To 10 Do
Begin
       Sum:= sum + Salaries[index];
End;
Writeln(‘The sum = ’, sum);
```

Multidimensional arrays
A multidimensional array stores data elements in more than one row and one column. Each data item is referenced using more than one index. An element in a two dimensional array is referenced by two indexes, and that of a three dimensional array is referenced by three indexes.

A two dimensional array
A two dimensional array is made up of more than one row and one column. Each element is referenced by two indexes.

Declaration
Format
Var ArrayName : Array[row1..rowN, coulunm1..colunmN] of DataType;

Example
```
Var
Students : Array[1..10, 1..50] of String;
Marks : Array[1..30, 1..70] of Integer;
Wages : Array[1..5, 1..10] of Real;
```

Assigning values to a two dimensional array
General syntax
ArrayName[rowN, colunmN] := value;

Example
```
Students[1, 1] := ‘Ben Okelo’;
Students[1, 2] := ‘Rose Wanja’;
Marks[15, 40] := 69;
Marks[20, 55] := 90;
```

Retrieving data from a two dimensional array
Data can be retrieved from a two dimensional array and assigned to a variable or displayed on the screen.

Syntax
VariableName:=ArrayName[rowN, colunmN];

Example
```
Student1:= Students[3, 25];
Student2:= Students[5, 40];
Writeln(‘The marks = ‘ ,Marks[1, 2]);
```

Records
A record is a collection data items of different data types. For a record to be used in a program, it must be declared in the declaration section of the Pascal program structure. Once a record has been declared, it becomes a user defined data type that can be used to declare variables. A record must have a valid name and valid field names with their corresponding data types.

Record declaration

General syntax

```
Type
      RecordName = Record
              Field1 : DataType1;
               Field2 : DataType2;
                     :
                     :
              FieldN : DataTypeN;
End;
```

Example1

```
Type
   Person =  Record
            name   : String ;
            age    : Integer;
            height : Real;
            weight : Real;
    End; (* Declaration of Person  record*)
```

Example2

```
Type
   Student =  Record
            regNumber : String;
            name   : String ;
            unit : String;
            Marks : Integer;
            grade    : Char;
    End; (* Declaration of Student record*)
```

Declaring a variable of record type
Once a record has been defined, it becomes a user defined data type that can be used to declare other variables in the program. Taking the record Person declared above, the variables of its type can be declared as follows:

```
Var person1 : Person;
Var person2 : Person;
```

Assigning values to the variable of record type
Each variable of record type will contain all the values of the fields declared in the record structure. The values are assigned to the record variable using a dot notation symbol (.) as follows:

```
Var person1: Person; {Variable declaration}
            person1. name  : ‘Betty Brown’ ;
            person1.  age    : 45;
            person1. height : 6;
            person1. weight : 85;
```

OR

```
Writeln(‘Enter the name of the person: ’);
Readln(Person1.name);
Writeln(Enter age :);
Readln(Person1.age);
Writeln(Enter height :);
Readln(Person1.height);
Writeln(Enter weight :);
Readln(Person1.weight);
```

Retrieving data from a record
Data stored in a record can be retrieved and placed in a variable or displayed on the screen. The variables to store the data from a record must have the same data types as the data fields declared in the record.

Example: Retrieving person1 record

```
Var
 name : String;
age : Integer;
height : Real;
```

```
weight : Real;
name := Person1.name;
age := Person1.age;
height := Person1.height;
weight := Person1.weight;
```

OR

```
{Displaying a record on the screen}
Writeln('The name is : ', Person1.name);
Writeln('The age is : ', Person1.age);
Writeln('The height is : ', Person1.height);
Writeln('The weght is : ', Person1.weight);
```

Declaring an array of record
Once a record has been defined, it can be used as a data type when declaring an array.

Example using Person record declared above:

```
Var People : Array[1..60] of Person;
```

Assigning values to an array of record
The values of all the data fields of a record are assigned to a single array index in the case single dimensional arrays.

Example:

```
Var People : Array[1..60] of Person; {Array declaration}

{Assigning values to index 1 of people}
People[1].name := 'Pius Olima';
People[1].age := 60;
People[1].height := 5.5
People[1].weight := 85;
```

Retrieving data from an array of record
Data stored in an array of record can be retrieved and placed in a variable or displayed on the screen. The variables to store the data from a record must have the same data types as the data fields declared in the record.

Example: Retrieving data from index 5 of people array

```
Var
 name : String;
age : Integer;
height : Real;
weight : Real;
name := People[5].name;
age := People[5].age;
height := People[5].height;
weight := People[5].weight;
```

Text Files

A file is a storage unit that is used to store large volume of data permanently in the hard drive. Data is stored in a text file in the form of text strings.

Importance of text file in programming

- Used to store a collection of data at the same place
- Provides persistence storage of data

Handling of text files

File handling deals with the procedures and activities to be followed when storing and retrieving data from a text files. A file must be opened before data can be written to it or read from it. After data has been written or read from a file, the file must be closed. There are three modes that can be used to open a text file for manipulation:

- **Read mode:** Opens a text file to enable data to be retrieved from it. Reset procedure is used to direct the file pointer to the first record on the file.
- **Rewrite mode:** Opens a text file to enable data to be written in the file. Any already existing contents of the file will be deleted if this mode is used
- **Append mode:** Opens a text file to enable data to be written at the end of a text file. The already existing contents of the file will not be deleted as new data is added.

File variable

A file variable is a variable of type Text that is used as a link between the program and the text file for the purpose of communication. The program writes data to the file or reads data from the file through the file variable. A file variable of type text must be declared before any file handling can take place.

Opening a file in Rewrite mode:

Rewrite mode opens a text file to enable data to be written in the file. Any already existing contents of the file will be deleted if this mode is used. This mode should be used if you are creating a completely new file or if you intend to delete the entire contents of the existing file.

Example

```
{A program to accept data from the keyboard and write them to a text file}
Program PersonalRecord;
Var
    fileVariable : Text; {Declaration of file variable}
    filename, nationalId, name, gender : String;
    age : Integer;
Begin
        filename := ‘C:\PeopleRecord.TXT’;
        Assign(fileVariable, fileName); {Linking the file variable to the physical file}
         Rewrite(fileVariable); {Opening the text file for output}
        Writeln(‘Enter national ID:’);
        Readln(nationalId);
        Writeln(‘Enter name:’);
        Readln(name);
        Writeln(‘Enter age:’);
        Readln(age);
        Writeln(‘Enter gender:’);
        Readln(gender);
{Writing data into the text file}
        Writeln(fileVariable, nationalId, ‘  ‘, name, ‘  ‘, age, ‘  ‘, gender);
 Close(fileVariable); {Closing text file}
 End.
```

Opening a file in Append mode:

Append mode opens a text file to enable data to be written at the end of a text file. The already existing contents of the file will not be deleted as new data is added.

Example

```
{A program to accept data from the keyboard and add them to a text file}
Program PersonalRecord;
Var
   fileVariable : Text;
   filename, nationalId, name, gender : String;
   age : Integer;
Begin
      filename := ‘C:\PeopleRecord.TXT’;
      Assign(fileVariable, fileName);
       Append(fileVariable); {Opening text file for append}
      Writeln(‘Enter national ID:’);
      Readln(nationalId);
      Writeln(‘Enter name:’);
      Readln(name);
      Writeln(‘Enter age:’);
      Readln(age);
      Writeln(‘Enter gender:’);
      Readln(gender);
      Writeln(fileVariable, nationalId, ‘  ‘, name, ‘  ‘, age, ‘  ‘, gender);
    Close(fileVariable);
 End.
```

Opening a file in Read mode:

Read mode opens a text file to enable data to be retrieved from it. Reset procedure is used to direct the file pointer to the first record on the file.

Example

{A program to retrieve data from a text file, store them in variables then displays them on the screen}

```
Program DisplayInformation;
Var
    fileVariable : Text; {Declaration of file variable}
    filename, registrationNo, name, gender, course : String;
Begin
        filename := ‘C:\StudentRegistration.TXT’;
        Assign(fileVariable, fileName); {Linking a file variable to a physical file}
         Reset(fileVariable); {Moves the file pointer to the first record of the file}
 While NOT EOF fileVariable Do {A loop that enables all records of the file to be retrieved}
    Begin
      {Reading data and assigning them to variables}
        Readln(fileVariable, RegistrationNo, name, gender, course);
      {Displaying a record on the screen}
        Writeln(RegistrationNo ‘  ‘, name, ‘  ‘, gender, ‘  ‘, course);
      End;
   Close(fileVariable); {Closing the file}
   Writeln(‘Information retrieval completed’);
End.
```