COMPUTER PROGRAMMING
DCSA 1303

SCHOOL OF SCIENCE AND TECHNOLOGY
BANGLADESH OPEN UNIVERSITY

SCHOOL OF SCIENCE AND TECHNOLOGY

DCSA 1303

COMPUTER PROGRAMMING

Writer

Dr. M. Kaykobad
Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

Revised and Edited by
Mohammad Mamunur Rashid
School of Science and Technology
Bangladesh Open University

Co-ordinator
Dr. K. M. Rezanur Rahman
Professor
School of Science and Technology
Bangladesh Open University

Overall Supervision
Dean
School of Science and Technology
Bangladesh Open University
COMPUTER PROGRAMMING

Published by : Publishing, Printing & Distribution Division
Bangladesh Open University
Gazipur- 1705
©Bangladesh Open University

ISBN 984-34-4003-X
1st Edition: April 1997
Revised and Reprint: December 2010

Computer Composed & Desktop Processing by :
Md. Jakir Hossain, Sabina Yesmin

Cover Designer : Monirul Islam
Printed by : Ganim Printing & Packages
44/G, Azimpur Road, Dhaka-1205.
UNIT 1: PRINCIPLES OF PROGRAMMING
Lesson 1: Objectives of Computer Programming ........................................ 1
Lesson 2: Nature of Algorithms ............................................................... 5

UNIT 2: PROGRAMMING TOOLS
Lesson 3: Flowcharts-I ............................................................................ 9
Lesson 4: Flowcharts-II .......................................................................... 17
Lesson 5: Pseudocodes ........................................................................... 24
Lesson 6: Decision Table ....................................................................... 28
Lesson 7: Structured Diagram ............................................................... 32

UNIT 3: HIGH LEVEL LANGUAGE-I
Lesson 8: Characteristics ....................................................................... 37
Lesson 9: Data Elements ....................................................................... 41
Lesson 10: Data Structure-I ................................................................. 45
Lesson 11: Data Structure-II ................................................................. 48

UNIT 4: HIGH LEVEL LANGUAGE-II
Lesson 12: Operators ............................................................................ 51
Lesson 13: Precedence and Associativity ............................................... 55
Lesson 14: Statements .......................................................................... 57
Lesson 15: Procedures and Functions ................................................... 61

UNIT 5: PASCAL PROGRAMMING
Lesson 16: Pascal Fundamentals-I ......................................................... 65
Lesson 17: Pascal Fundamentals-II ......................................................... 72
Lesson 18: Pascal Fundamentals-III ....................................................... 77

UNIT 6: DATA TYPES
Lesson 19: Simple Data Type-I .............................................................. 83
Lesson 20: Simple Data Type-II .............................................................. 90
Lesson 21: Simple Data Type-III ........................................................... 96
Lesson 22: Input-Output ...................................................................... 104

UNIT 7: CONTROL STRUCTURES
Lesson 23: Control Structures-I ........................................................... 115
Lesson 24: Control Structures-II .......................................................... 120
Lesson 25: Control Structures-III ......................................................... 126
Lesson 26: Control Structures-IV ......................................................... 132

UNIT 8: PROCEDURES AND FUNCTIONS
Lesson 27: Procedures and Functions-I .................................................. 139
Lesson 28: Procedures and Functions-II ............................................... 143
Lesson 29: Procedures and Functions-III .............................................. 150
<table>
<thead>
<tr>
<th>Unit 9: Arrays and Records</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lesson 30: Arrays-I</td>
<td>155</td>
</tr>
<tr>
<td>Lesson 31: Arrays-II</td>
<td>159</td>
</tr>
<tr>
<td>Lesson 32: Records-I</td>
<td>163</td>
</tr>
<tr>
<td>Lesson 33: Records-II</td>
<td>166</td>
</tr>
<tr>
<td>Lesson 34: Sets</td>
<td>172</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit 10: Pointers and Recursion</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lesson 35: Pointers-I</td>
<td>175</td>
</tr>
<tr>
<td>Lesson 36: Pointers-II</td>
<td>179</td>
</tr>
<tr>
<td>Lesson 37: Recursion-I</td>
<td>184</td>
</tr>
<tr>
<td>Lesson 38: Recursion-II</td>
<td>189</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit 11: Executing Programs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lesson 39: Preparing and Running a Program</td>
<td>193</td>
</tr>
<tr>
<td>Lesson 40: Compiling and Running the Program</td>
<td>197</td>
</tr>
<tr>
<td>Lesson 41: Error Diagnostics</td>
<td>200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit 12: Programming Style</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lesson 42: Structured and Unstructured</td>
<td>207</td>
</tr>
<tr>
<td>Programming</td>
<td></td>
</tr>
<tr>
<td>Lesson 43: Top-down and Bottom-up Programming</td>
<td>211</td>
</tr>
<tr>
<td>Lesson 44: Modular Programming</td>
<td>215</td>
</tr>
</tbody>
</table>

| Answers to MCQs                               | 221|
Preface to the First Edition

Modern society relies heavily on automation and on automated handling, storing and processing of information. In the modern world efficiency of production of wealth depends heavily on various kinds of automation. Computers are used to automate methods for processing information. Apart from information processing, computers play ever increasing role in many other areas of modern life. Programming is a vital element in efficient utilisation of a computer. This book is written in a self learning style of distance education under Open University system. The book contains twelve units:

Unit 1 describes the principles of programming. Some general terms related to computer programming are defined and illustrated in this unit.

Unit 2 describes the programming tools in detail. The basic concepts of flowcharts are given here. Some examples on flowchart are also given here for clearing idea.

Unit 3 provides some idea on high level languages. First two lessons show some elementary data types of high level languages and the next two lessons give idea of data structures.

Unit 4 discusses the building blocks of any high level language. Different types of operators and their detailed characteristics are discussed here.

Unit 5 discusses the basic concepts of a programming language "Pascal". This unit gives few fundamental concepts of 'Pascal'.

Unit 6 describes different data types used in Pascal. Declaration methods of data types, their restriction and range are the main theme of this unit.

Unit 7 gives the knowledge of control structure of a Pascal program. Some of the structures are essential for writing Pascal programs.

Unit 8 describes procedures and functions. In some languages these are called subroutines.

Unit 9 describes arrays and records. The methods of accessing elements of arrays and declaring arrays are listed here.

Unit 10 discusses advanced notions the dynamic method of programming.

Unit 11 discusses the process of running programs. Facilities of debugging or correcting programs are also described here.

Unit 12 introduces some developed idea on programming style. The idea of structured and unstructured programming is given here.

At the end of each lesson of the unit there are exercises. One can easily check whether one understands the subject by answering the questions. An answers key contains at the end of this book.
Preface to the Revised Edition

The present edition of the book is revised. The content and exercises at the end of each lesson of the unit-1, unit-2, unit-3 and unit-4 have been brought up-to-date. Some examples are included using C programming language in this edition. The programming language “Pascal” (content of unit-5 to unit-11) remains as before for better understanding in language, but learners are requested to study programming with “C” and prepare themselves for examination using the study guide “Programming with C” of the book “Programme with C” by Byron Gottfried. The study guide has been supplied to learners with this book. Reference books contain at the end of this book for further reading.

We are grateful to our tutors and learners for their favourable appreciation of the book. Suggestions for further improvement will be highly appreciated.

Dean
School of Science and Technology
Bangladesh Open University
Unit 1: Principles of Programming

Introduction

In this unit you will learn some general terms related to computer programming such as languages and algorithms. You will be familiar with these terms after reading this unit. The importance of an algorithm in a programming language can be realised in lesson 2. Chronological development of different languages and their uses are described in this unit. A general rule of writing algorithm is also described here.

Lesson 1: Objectives of Computer Programming

1.1 Learning Objective

On completion of this lesson you will learn

- programming and programming language
- why programming is popular
- history of programming.

1.2 Definition of Programming

Programming is writing instructions for a machine specially a computer. The machine which works differently according to the instructions given to it is called a programmable machine. The jobs of these machines are not fixed. We can change the working plan of the machine by changing the instructions or programs. Computers are machines that can understand the instruction and work properly according to the instructions given.

In short, programming means step by step instructions for solving a problem by computer.

1.3 What is a Language?

The instruction given to the computer must have a particular format. Computers are unable to understand human language. There are several levels of format or language which a computer can understand. The interest of understanding computer languages increases day by day. History of computer languages is as old as computer science. This is described in the following sections.
1.4 Different Types of Language

1.4.1 Low level languages

In old days computers were very large and were not much reliable. Programming was done by switches and there was no facility to use the program. This approach of computer programming was completely a hardware technology. Next the approach of machine language comes up. In this approach instruction set of the computer consists of some numerical digits. This instruction set is used to build up a program. This code is difficult to understand for human beings. Next assembly language was introduced. In assembly language the numerical codes are replaced by suitable names which are understandable. This has made programming easier.

1.4.2 High level languages

This type of languages are like English language. But this is more structured. Actually this type of languages support mathematical notations used in arithmetic expressions. There are some special rules for writing programs in high level languages. Pascal, C, C++, Mathematica, Java, Math lab etc. Ada are all high level languages. High level languages are easy to write, compile and suitable for error correction.

1.4.3 Fourth Generation Languages

Scientists are trying to develop computer languages like human languages. They have not been successful but may be successful in near future. These languages have greater built-in facility such as database query, searching, sorting and interfacing. FoxPro, Cobol, Visual Basic, MS Access, Oracle Forms and Reports, SQL systems make use of such programs or languages. These are also known as the Fourth Generation Languages (4GL).

1.5 Purpose of Programming

There are several purposes of programming. These are discussed below.

1.5.1 Computer program for automated systems

Different offices need different system. So a ready made software is not enough for this purpose. Special programs should be written for office automation. These softwares may be used for accounting, payroll systems, machine control, industrial control, billing system of electricity, telephone and water, result processing of examination etc. It is important to develop these softwares properly because the output of the whole system depends on the output of the computer system.
Principles of Programming

1.5.2 Developing the system

Demand increases day by day. So new systems should be introduced. This is done by introducing new programs and new versions of programs. Sometimes new machines come up with extra facilities. Existing systems may not support these facilities. At that time developed programs are necessary for better utilisation of machines.

1.5.3 Scientific research

Scientific research requires computers for implementing the theory into practice. Huge numbers of calculations are required for this purpose. Only a computer program can help the scientists out. There are some programs which are useful for scientific calculations. For example, FORTRAN, C++, Mathematica, Math Lab. Simulation is also very necessary for research, which can be performed using simulation packages.

1.5.4 Development of programming languages

This is one of the main purposes of programming. High level languages are made from low level languages. Even top versions of high level languages are programmed by low version of high level languages. This idea of developing programming tool is called boot strapping. That is the objective of programming is to make the programs easier so that a layman can run his machine without the help of a programmer. This is actually the future trend of programming. Now the programmers are not concerned about the data structures and algorithms used. The compiler will use the proper data structure and efficient algorithm to solve the problem.

1.5.5 Object of this programming course

This course of programming is not all in all in computer programming. We have discussed some aspects of high level languages and then discussion on Pascal programming language will be found in subsequent lessons. This is actually an introductory course. Extra ordinary features of Pascal have been omitted from the book for reduction of complexity. Having learnt all the lessons, one will not only know the structure of Pascal and capable of writing small programs.

Learner! you will learn C Programming instead of Pascal from the study guide “Programming with C” and from its reference book.
1.6 Exercise

1.6.1 Multiple choice questions

1. Which of the following languages came first?
   a) Assembly
   b) C
   c) Pascal or C
   d) machine language.

2. High level language can be developed from
   a) low level language
   b) high level language
   c) assembly language
   d) C language.

3. Which of the following languages is nearest to structured English?
   a) High level
   b) Pascal or C
   c) machine
   d) Macro assembler.

1.6.2 Questions for short answers

1. What do you understand by programming?
2. Define the term ‘boot strapping’.
3. What is the language of programming?
4. How many types of languages are there?
5. Describe the role of computer programs in system development.

1.6.3 Analytical questions

1. Describe the chronological development of computer languages.
2. Describe the purpose of programming.
Lesson 2 : Nature of Algorithms

2.1 Learning Objectives

On completion of this lesson you will learn

- algorithm
- methods of writing algorithms
- properties of algorithms.

2.2 Definition and Properties of Algorithm

In Webster's dictionary, the word "Algorithm" is defined as "any special method of solving a certain kind of problem". But in computer science it has a special meaning. It means a step by step procedure for solving a problem by a computer. An algorithm has following properties.

1. An algorithm must be composed of a finite number of steps. Each step may be another algorithm composed of several steps.

2. Each step of the algorithm must be definite. The meaning of the operation must be clear. We cannot have an operation like "add 2 or 3 to x" in an algorithm.

3. The steps must be effective; each step can at least in principle be done by a person using pencil and paper in finite amount of time. Performing arithmetic on integers is an example of an effective operation, but arithmetic with real numbers is not necessarily effective, since some values may be expressible only by an infinitely long decimal expression.

4. The algorithm may have one or more inputs but it must have at least one output.

5. An algorithm must terminate after a finite number of operations. There is another word for an algorithm which obeys all of the above properties except termination, and that is computational procedure. An operating system of a digital computer is an example of a computational procedure since it does not terminate, but continues in a waiting state until a new job is entered.
2.3 Use of Algorithm in Programming

Whenever a problem is found the method of solving it should be written in an easier way. This is called an algorithm. The step of algorithm is converted to any programming language. A lot of intelligences is required to design an algorithm. Programming means translation of an algorithm into a programming language. The study of algorithms includes many important and active areas of research. Some of these are follows in the next subsection.

2.4 How to Design an Algorithm

Designing new algorithms means finding new methods of solving problems. This is done by mathematicians or computer scientists sometimes with the help of existing algorithms. Sometimes they are developed by changing the steps which improves the computational requirements. There are several fields where algorithms can be designed. These are sorting, searching, linear programming, pattern recognition, artificial intelligence etc. To design efficient algorithms clear theoretical background on these topics is necessary. Only programming knowledge is not sufficient for this purpose.

2.5 Writing or Expressing Algorithm

Pseudocode or structured English is suitable for expressing an algorithm. Here, unlike natural languages, each statement has an unambiguous meaning. Generally algorithm is expressed in any high level language.

2.6 Analysis of Algorithm

Whenever you design an algorithm you must analyse it. You have to determine the cost of algorithm. If the cost is less than the existing algorithm, this algorithm will be implementable. Generally the cost of multiplication, addition and data transfer is calculated. If the cost of a new algorithm is of lower order than previous one then we are assured that the algorithm will give better results. Let us begin to implement the algorithm in a program.

Example: There are three numbers. You have to find their median. Write the algorithm in structured English.

Consider $a$, $b$, $c$ are numbers.

\[
\begin{align*}
\text{if } b & \leq a \leq c \text{ then} \\
\text{median} & \leq a \\
\text{else if } a & \leq b \leq c \text{ then} \\
\text{median} & \leq b \\
\text{else } \text{median} & \leq c \\
\text{endif.}
\end{align*}
\]
2.7 Exercise

2.7.1 Multiple choice questions

1. Which one of the following statement is true?
   a) Algorithm is a part of programming
   b) Algorithm is a part of system design
   c) Algorithm is a part of problem solving
   d) Algorithm is a part of system analysis.

2. Which one of the following statement is true?
   a) An Algorithm is composed of finite number of steps
   b) An Algorithm is composed of any number of steps
   c) An Algorithm is composed of certain number of steps
   d) An Algorithm is composed of infinite number of steps.

3. Which of the following is false?
   a) Programming means translation of an algorithm into some machine code
   b) Programming means translation of an algorithm into structured English
   c) Programming means translation of an algorithm into a programming language
   d) Programming means translation of an algorithm into C programming language.

4. Which of the following is true?
   a) To design an efficient algorithm it is necessary to have good knowledge in mathematics
   b) To design an efficient algorithm it is necessary to have good knowledge in software
   c) To design an efficient algorithm it is necessary to have clear theoretical background on the topic
   d) To design an efficient algorithm it is necessary to have good knowledge in English.

5. Which of the following is suitable for expressing an algorithm?
   a) Structured English
   b) Machine language
   c) C
   d) Pascal.

2.7.2 Questions for short answers

1. What is the meaning of algorithm?
2. What is the use of algorithm?
3. How can algorithm can expressed?
4. What are the measures of algorithm?

2.7.3 Analytical question

1. Describe the different properties of algorithm.
Unit 2 : Programming Tools

Introduction

This unit will help learners about the programming tools like flowcharts, pseudocodes, use of which improves the efficiency of programming and programming skill. Flowcharts give a pictorial representation of the logic. This is described with several symbols and rules associated with them. Some examples on flowchart are also included for making the idea clear. Pseudocode is another method which is very similar to programming languages. The method of writing programmes with the help of pseudocode has been given in the lessons. Decision tables are another representation of programmes. Decision tables describes how conditional statements are represented in short cut format in a table. The last lesson of this unit describes some methods of representing data structures of input and output of the programme. Warnier-Orr diagrams are such diagrams. Structured chart gives a pictorial representation of programme by dividing main module into different modules and by parameter passing between the modules.

Lesson 3: Flow Charts-I

3.1 Learning Objectives

On completion of this lesson you will learn

- the basic concepts of flowcharts
- some examples and their representation by flowcharts
- symbols used to represent different actions.

3.2 FlowChart for Describing Algorithms

An algorithm is a set of instructions which when followed will produce the solution to a given problem. Algorithms occur in noncomputing contexts as well as in programming. You can think of the recipe for baking a cake as an algorithm- certainly a recipe is a set of instructions which, when followed, will result in a cake. Likewise, the instructions in a stereo kit are steps which, when followed, will produce a properly assembled electronic device. If the instructions are poorly written or if they are not followed precisely, the result is a soggy cake. Consider the following instructions that you need to make a cup of tea.
1. Pour water in a kettle.
2. Turn the oven on.
3. Place the kettle on the oven.
4. Turn the oven off.

Do you see what is wrong? The instructions never mentioned the time when the oven must be turned off. Of course, a human being would probably compensate for the inadequate instructions by making some assumptions or by his common-sense (which are not common to even sophisticated machines like computers). A machine, however, would not and this “program” would not work properly.

The step 4 can be executed only after being confirmed that the water in the kettle is already boiled. So, a condition must be tested satisfactorily before the step 4 can be executed. So the correct algorithm will be:

1. Pour water in a kettle.
2. Turn the oven on.
3. Put the kettle on the oven.
3a. Check whether water is boiled
4. Turn the oven off.

Note that in this diagram we have enclosed each instruction in a box and drawn an arrow to the instruction that follows. This type of diagram, called a flowchart, is very useful as a means of visualising the relationships among the statements of algorithms.

In order to design and analyse algorithms it is very useful to represent them in a convenient way. Flowcharts, pseudocodes and decision diagrams are some of the more frequently used representation tools. Of course, the tea making algorithms are quite straightforward, and the diagram is not needed; but many algorithms are far more complex.
Consider a series of steps you might follow if you were invited to a friend’s house for dinner. Part of the algorithm might be as follows.

1. Get dressed.
2. Catch a bus to friend’s house.
3. ...
4. ...
5. ...
6. ...
7. ...
8. ...
9. ...
10. Get dressed
11. If you are not late go to step 13.
12. Telephone friend
13. Catch a bus to friend’s house.
14. ...
15. ...
16. ...

If you feared you might be delayed, your plans might include the possibility of telephoning to say you would be late. To include this possibility you must introduce a decision point into the algorithm:

1. ...
9. ...
10. Get dressed
11. If you are not late go to step 13.
12. Telephone friend
13. Catch a bus to friend’s house.
14. ...
15. ...
16. ...

Statement 12 will be performed only if you were late and will not be executed otherwise.
The decision statement is statement 11. The flowchart for this algorithm is shown in Figure 3.1.

The exact way in which this algorithm will be executed depends on the data (that is, whether you are late or not):

If late

Get dressed.
Telephone friend's house

If not late

Get dressed.
Catch a bus to friend’s house.

But paths are implicit in the algorithm; the specific path executed will depend on the circumstances.

Next, let us look at a task a bit more complex- getting to the college in the morning.

0. Start.

1. Turn off the alarm.
2. Get up.
3. If there is no time to get washed, go to step 6.
4. Enter bathroom.
5. Get washed.
6. If there is no time to make breakfast, go to step 10.
7. Enter kitchen.
8. Get breakfast.
9. Go to step 11.
10. Grab some dry food.
11. If situation in university is tense go to 13.
12. Take the direct path to college. Go to step 14.
13. Take the path to college avoiding university.

### 3.3 Some Symbols of Flow Chart

Normally, we begin from step 0 and go to the next statement and finished at the space end unless directed to do otherwise by a GO TO instruction as in steps 3, 6, 9, 11 and 12 in the present example. Notice that this algorithm, like the previous one, contains an IF statement. IF statements permit jumping around and skipping some instructions. GO TO and IF statements cause a program to *branch*—that is, to make an out-of-sequence jump from one statement to another.

It is difficult to follow the branches; so in order to make the logic of the algorithm clearer; it is useful to represent the algorithm as a flowchart since visual representations are more effective for us. Before converting the algorithm for getting to college, let us introduce the symbols that are used to represent different actions in a flowchart. The symbols of the flowcharts are standard.

Each *block* in the flowchart has a characteristic shape. *Start* and *end* blocks look like this:

![Start and end block](image)

A *process* or *operation block* which gives instructions for work to be done looks like this:

![Process or operation block](image)

and a *decision block* (logical block) which asks a question and has two or more outputs looks like this:

![Decision block](image)

The flow chart of plan for making school can be shown in figure 3.2 using symbols described just above.
fig. 3.1.
Programming Tools

Fig 3.2

1. Time avail to breakfast?
   - Yes: Enter kitchen and get breakfast
   - No: Grab some dry food

2. Situation tense?
   - Yes: Avoid school path
   - No: Take direct path

End
3.3.1 Exercise

3.3.2 Multiple choice questions

1. Which one represents start and end blocks of a flowchart?
   a) Ellipse
   b) Circle
   c) Rectangle
   d) Cone.

2. Which one is the process or operation block, that gives instructions for work to be done looks like?
   a) Parallelogram
   b) Rhombus
   c) Line
   d) Rectangle.

3. Which one is the decision block, that asks a question and has two or move outputs looks like?
   a) Rhombus
   b) Circle
   c) Point
   d) Ellipse.

3.3.3 Questions for short answers

1. What do you mean by a flowchart?
2. Why are flowcharts essential?
3. Are the symbols of the flowcharts standard?
4. Is there any limit of the size of a flowchart?
5. What is an algorithm?
6. Write down the steps required to prepare tea.

3.3.4 Analytical questions

1. Draw a flowchart of GCD (Greatest Common Divisor) algorithm.
2. Write your daily routine with steps and draw a flowchart.
Lesson 4 : FlowCharts-II

4.1 Learning Objectives

On completion of this lesson you will learn

- advanced features of flowcharts like program loops and decisions
- how modularity of flowchart can be achieved.

4.2 Looping in Flowchart

Computers are efficient in executing repetitive operations. In programs, there are invariably repetitive operations, which are needed to be represented in flowcharts. Consider the following flowchart where the start block shows where the algorithm begins.

![Flowchart Diagram]

**Fig 4.1**

In figure 4.1 an unconditional infinite loop is shown. The exception in this flowchart is the flow direction from lower to upper. The last block of this flowchart will never be executed.
Figures 4.2 and figure 4.3 represent flowcharts containing conditional looping. Looping occurs according to the output of condition checking. In figure 4.2 condition checking is done at the end and in figure 4.3 condition checking is done in the beginning.

### 4.3 Examples of Looping

Suppose you wish to visit the botanical garden, and therefore, are looking for one of your friends to accompany you. So you take out your book of few phones and select a likely prospect. Dial the number, make few talk, and finally invite the friend to visit. If he (or she) accepts the offer, you have finished your work. If you are refused, then go back and select another number. These actions are shown in a flow chart in Figure 4.4. The essential concept in this algorithm is the branch back to an
A loop is a sequence of instructions which is repeated several times.

earlier statement to create a loop. A loop is a sequence of instructions which is repeated several times. Note that each time the programme goes through the loop a different telephone number is used; the instructions remain unchanged but the data operated on differs. If you eventually get an acceptance, you would reach the end block. However, if you keep on getting rejections, then you re-execute the loop instructions. Since you only have a finite number of telephone numbers in your book, you must stop when the data run out. Converting this flowchart to a series of instructions in English is straightforward:

0. Start.

2. Select prospect.
3. Dial number.
4. Make few talk.
5. If prospect does not accept, go to step 2.
6. end.
A similar algorithm could be constructed to find the right blood donor in a medical emergency. The program might be used if there were an urgent need for a donor with type B-positive blood. If the medical records of the students at a college were kept on a machine readable device, the computer could be used to search for the collection of data (called the file). Input and output are represented by specially shaped blocks:

![Flowchart](image)

The algorithm reads the name and blood type of a student. If the blood type is B-positive, then the name is printed; otherwise it loops back to read the next record. This process continues until all the records have been read.

A programmer might begin by mapping the steps visually, using flowchart like in Fig. 4.5.

![Flowchart](image)
The algorithm could be written out:

0. Start.

1. Read NAME, blood TYPE.

2. If TYPE not equal to B-positive, go to step 1.

3. Print NAME.

4. Go to step 1

5. End.

4.4 Modularity in Flowchart

Some times the same job may be repeated in different places in flowchart. In every place of flowchart same sequence of actions are included every time. This enlarges the flowchart and makes it unusually big. It is difficult to understand such flowchart. So another flowchart is used separately for this repetitive module. This is exactly like our traditional flowchart described so far. The only difference is the calling symbol, defined as Fig. 4.6 and said to be modularity in flow chart.

So another flow chart is used separately for this repetitive module. This is exactly like our traditional flowchart described so far.

Fig 4.6

For example we can consider the dialling process as a submodule. So the flowchart for visiting botanical garden can be modified as shown in Fig 4.7.
Fig 4.7
4.5 Exercise

4.5.1 Multiple choice questions

1. Which of the following is the symbol of I/O?
   a) Rectangle
   b) Triangle
   c) Rhombus
   d) Parallelogram.

2. Which one of the following statement is true for looping?
   a) not repeated
   b) repeated several times
   c) no use further
   d) repeated only one time.

3. Which of the following symbol is used for modular flowchart?
   a) Rectangle with two sides double lined
   b) Parallelogram
   c) Point
   d) Quadruple.

4.5.2 Questions for short answers

1. What is a loop?
2. Show different types of looping in flowchart.
3. How short modules are represented in a flowchart?
4. What is the reason of modularization in a flowchart?
5. How can you remove the problem when a flowchart is more than one page long?

4.5.3 Analytical questions

1. Draw a flowchart with all symbols so far you have learnt.
2. Draw a flowchart for result processing of 50 students.
3. Write an algorithm correspond to a given flowchart.
Lesson 5 : Pseudocodes

5.1 Learning Objectives

On completion of this lesson you will be able to learn

- pseudocode
- structure of pseudocode
- use of pseudocode.

5.2 Definition of Pseudocodes

Pseudocode, sometimes called structured English, is a method of expressing programme logic using plain English statements rather than a programming language. It uses narrative statements rather than graphic symbols such as trees of tables to describe a procedure. An advantage of pseudocode is that system builders can concentrate on developing processing logic independent of syntax guidelines of any programming language. Syntax guideline is the rules of formulating instructions. Pseudocode can be easily translated into a programming language.

5.3 Structure of Pseudocodes

Pseudocode uses some logic patterns as the basic control structures of structured programming. These are:

5.3.1 Sequence structure

The sequence structure is single steps or actions that follow one another without interruption. The series of actions does not depend on the existence of any condition. The pseudocode for the sentence structure is:

```
Do action1
Do action2.
For example:
Read customer bill
Print customer bill.
```

5.3.2 Decision structure

The decision structure, where two or more actions can be taken depending on which satisfies a stated condition. The pseudocode format for decision structure is:
Programming Tools

**In C language**

```c
if (expression)
statement 1
else
statement 2
```

*For example:*

```c
if (Income > Tk. 1,000)
income_tax=4*Income;
else
income_tax=0;
Printf("%f", income_tax);
```

**In Pascal**

```pascal
IF (condition 1 in true)
Do X
ELSE
DO Y
ENDIF
```

*For example:*

```pascal
IF (Income > Tk. 10,000)
income_tax=4*Income
ELSE
income_tax=0
ENDIF
```

### 5.3.3 Iteration structure

The *iteration structure*, when certain actions are repeated over and over while a specified condition occurs or until a condition occurs. The pseudocode format for the iteration structure is:

```
: do Statement
    While (expression);
: : :
```

*Here is an example of repetitive structure.*

**In C language**

```c
#include <stdio.h>
main() /*display the integers 0 through 9*/
{
    int digit = 0;
do
        printf("%d\n",digit++)
while(digit<=9);
}
```

**In Pascal**

*Here is an example of repetitive structure.*

```pascal
DO WHILE (End of file switch is off)
Read each student record
Add 1 to student Record counter.
DOWEND
```

**Example:** Write a pseudocode for money market fund monthly statement in Pascal

*IF investor's balance is greater than or equal to Tk.500*

Send monthly balance statement only

*ELSE*

*IF account fund has had activity*

Send monthly balance statement only

*ELSE*

Send monthly balance statement + warning notice

ENDIF

Indentation of student in pseudocodes increases its readability.
5.4 Use of Pseudocodes

1. Pseudocode is a structured language. It can be easily used for algorithm writing. Pseudocode is understandable to class of educated people. This can be used as communication media of computer people and its users.

2. This is used for communicating process specifications that shape program design. This is actually the primary design or draft design of programs.

3. Users can easily understand the program. So the operating principle or procedure can be easily expressed by pseudocodes. Pseudocodes are also used in documenting the program.

4. Different scientific problems solvable by computers are written in Pseudocodes. These are given in several books whenever needed.

5.5 Exercise

5.5.1 Multiple choice questions

1. Which one of the following statement is true?
   a) Pseudocodes a narrative sentence
   b) Pseudocodes a graphic symbol
   c) Pseudocodes structured English
   d) Pseudocodes is the machine language.

2. Which statement is used for termination of iteration structure in Pascale?
   a) ENDFOR
   b) ENDIF
   c) ENDWHILE
   d) DOWEND.

3. How many structures of pseudocodes you have found in this lesson?
   a) 2
   b) 3
   c) 4
   d) 5.
Programming Tools

5.5.2 Questions for short answer

1. What is a pseudocode?
2. What are the characteristics of Pseudocodes?
3. What is a decision structure? Write the pseudocode format for a decision structure.
4. What is the use of a pseudocode to programmers?
5. Describe how repetitive structures are represented in pseudocodes.

5.5.3 Analytical questions

1. Write a pseudocode of a sorting algorithm.
2. Write a pseudocode for money market fund monthly statement.
Lesson 6 : Decision Table

6.1 Learning Objectives

On completion of this lesson you will be able to learn:

- structure of a decision table
- use of a decision table.

6.2 What is a Decision Table?

*Decision table* is the tabular or graphical representation of problem solving method. So this is more precise than other method of problem solving like pseudocode. Different decisions are taken according to different conditions. Decision table must have the scope of representing the conditions. There are different rules which states one or more actions to be done for satisfying one or more conditions. Thus decision table can be said a tabular representation of the program.

6.3 Structure of Decision Table

There are 5 portions of a decision table. These are described below:

<table>
<thead>
<tr>
<th>Header</th>
<th>Condition entries</th>
<th>Condition stub</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.</td>
<td>2.</td>
</tr>
<tr>
<td></td>
<td>3.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action statements</th>
<th>Action entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td></td>
</tr>
</tbody>
</table>

1. **Header**: This is the name of the problem or programme in short. The header must have the quality that this is the mirror of the content of the table and says what we are doing in the programme. Header must be very few words specially 3 or 4 words.

2. The **body** of the table is divided into 4 parts. These are

*Condition Stub*: This is the upper left portion of table. Several conditions are listed here. The conditions are stated in structured English. Some notations like ‘<’ (less than), ‘>’ (greater than) are valid to use in stating conditions.
Programming Tools

**Action statements** : This is the lower left corner of the table. Several actions written in pseudocode are listed in this portion with their serial number.

**Rule** : The right portion of the table indicates the rules. Each column indicates one rule. Each rule satisfies one or more conditions and the entries for that condition is written by ‘Y’. The conditions which is not satisfied is indicated by 'N' entry. The ignored conditions are denoted by ‘-’. All these indicates condition entry that is upper right corner of the table.

The lower right corner is for action entry. The rule may support one or more action. The supported actions are denoted by 'X' entry and the others are denoted by blank entries in the corresponding column of the rule.

### 6.4 Example

Make a decision table for sending grade sheet to students. The decision table is shown below:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Y</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. marks ≥ 80%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. marks &lt; 40%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. marks ≥ 40%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Is the student defaulter?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. marks &lt; 50%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>X</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Send marksheet</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Send distinct certificate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Defer result</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Send warning sheet</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This table gives the following information:

1. If the student is a defaulter his result will not be published.
2. If the student gets marks below 40%, he fails and no result will be published.
3. If the student gets marks from 40% to 50% then marksheet will be issued with warning.
4. If the student gets marks from 60% to 80% then only marksheet will be issued.
5. If the student gets more than 80% then distinction award is given with marksheet.
6.5 Importance of the Decision Table

1. This is more modular. Statisticians can use this table for programming especially in database.
2. Tables are always attractive. Written information in decision table format is always helpful in decision making.
3. This is easier to programme. Programmers can code the problem in any programming language from decision table. So decision table is used by system analyst.
4. The operating principle of a machine can be represented by decision table, so users can take help from it.

6.6 Exercise

6.6.1 Multiple choice questions

1. How many words must be used for writing a header?
   a) 1 or 2 words
   b) 3 or 4 words
   c) 5 or 6 words
   d) 10 to 12 words.

2. Where are the action entries located?
   a) lower left
   b) bottom left
   c) top right
   d) bottom right portion of the decision table.

3. In which case 'X' sign is used?
   a) action entries
   b) condition stub
   c) header
   d) condition entry.

4. Which are satisfying the ignored condition?
   a) X
   b) ,
   c) -
   d) I.
6.6.2 Questions for short answers

1. What do you mean by a decision table?
2. What are the contents of condition stub and action stub?
3. How different rules are expressed in a decision table?
4. Why a decision table is necessary?
5. How a system analyst can use a decision table?

6.6.3 Analytical questions

1. Write a decision table for sending grade sheets to students.
2. Describe in your own words the process of making a decision table.
Lesson 7 : Structured Diagram

7.1 Learning Objectives

On completion of this lesson you will learn:

- structured diagram
- different types of structured diagram and their construction method
- use of the structured diagrams.

7.2 Concept of Structured Diagram

Structured diagram is the diagram which gives the modularization process of the programming problems into smaller subproblems. Each subproblems can be divided into several subproblems. So the diagram is like a tree of which head is the main problem and subtrees are subproblems. Actually all the structured diagrams which support modularization supports the tree structure. We will discuss two of them in the lesson.

7.2.1 Structured Chart

This is almost a tree with a tree root at the top. The whole problem is broken into smaller modules. This chart not only shows the way how they are broken but also the way how the programme is controlled by parameter passing between higher and lower level modules. There are several rules to express all these. Now we will show the structured diagram of result processing in fig 7.1 and then discuss certain rules of structured chart.

Result processing is done for every student so the every operation will iterate several times. The direction is denoted by directed arrows. The rectangular box denotes modules. The diamond with submodule denotes the conditional branch to another submodule.
Fig. 7.1 Structured diagram of result processing.

For example in the above structured diagram roll parameter is passed to mark finder. The mark finder module gives a Boolean parameter if this is true then mark found and grade is calculated otherwise absence of the student in that course is announced.

### 7.2.2 Warnier-Orr Diagram

The program works with the data. So construction of data helps to make a programme. Warnier-Orr diagram is that type of diagram which gives structure of input, output file database or final report in a structured way and which supports the tree structure. This is developed by Warnier and Orr. Actually the data is divided into submodules in this diagram. The root or main part of the system is written in the left portion and the submodules are written in the right of left curly braces as shown in the fig. 7.2.

**Warnier - Orr diagram gives structure of input, output file database or final report in a structured way and supports the tree structure.**
Each submodule follows another curly braces for further decomposition of the module. Now we will show the output of the students' marks record in a Warnier - Orr diagram in the following figure.

![Diagram](image-url)

**Fig 7.3**

The numbers in the parentheses denotes the repetition numbers. Here the computer science department has 50 students. So 50 set of data of the same size is required. There are 10 courses in each semester. Each course grade is divided into three examinations. They are class tests, attendance and theory examination. So courses are divided further.

### 7.3 Use of Structured Diagram
1. Structured chart is used to make a rough study of the problem. This is the previous state of algorithm design.
2. Warnier-Orr diagram is very useful in data structure design.
3. Design of input, output form can be easily done by Warnier - Orr diagram.
4. These diagrams favour a tool of system documentation. So all these are useful for system development, which is the main purpose of programming.

### 7.4 Exercise

#### 7.4.1 Multiple choice questions

1. **Which one is true?**
   a) Repetition is shown by the half circle
   b) Repetition is shown by the full circle
   c) Repetition is shown by the closed circle
   d) none of the above in Warnier-Orr diagram.

2. **Which one of the following statement is true?**
   a) Structured chart is the previous state of algorithm design
   b) Structured chart is the previous state of data structure
   c) Structured chart is the previous state of system development
   d) Structured chart is the previous state of program design.

3. **Which one supports the similarity of Warnier-Orr diagram and structured chart?**
   a) Repetition symbol
   b) Main structure
   c) Tree structure
   d) Looping.

4. **What is represented by Warnier-Orr diagram?**
   a) Program flow
   b) Output report
   c) Arithmetic computation
   d) Flow chart.

5. **Which of the following is used in Warnier-Orr diagram?**
Computer Programming

a) Circle  
b) Diamond  
c) Rectangle  
d) Curly braces.

7.4.2 Questions for short answers

1. What is the main principle of a structured diagram?  
2. Describe the use of a structured diagram.  
3. What is represented by Warnier - Orr diagram?  
4. What is represented by a structured chart?  
5. Describe the notation of forming Warnier - Orr diagram.

7.4.3 Analytical question

1. Make the Warnier - Orr diagram of the output of students marks record.
Unit 3 : High Level Languages -I

Introduction

This unit provides some idea on high level languages. This unit is not for any special language such as Pascal, C but for any kind of high level language. First two lessons show some elementary data types used in high level languages and next two lessons give idea of data structures. This unit gives idea for developing not only programmes but also algorithms. Some data structures are not readily available in high level languages but can be easily constructed by elementary data structures. They are queue, stack etc. They are very useful for algorithm construction and understanding. You will find a brief introduction of these topics in the last lesson of this unit. Records and arrays are common compound data structures in all languages. These will be discussed more elaborately.

Lesson 8 : Characteristics

8.1 Learning Objectives

On completion of this lesson you will learn :

• relative advantages and disadvantages of high and low level languages
• important features of a high level language.

8.2 Review of Levels of Languages

There are many different languages that can be used to programme a computer. These can be broadly divided into two classes, Low Level Languages and High Level Languages. Low level language is very detailed and consists of cryptic instructions that control the computer's internal circuitry. It is the natural dialect of the computer. Very few computer programmes are actually written in low level language, for two significant reasons. First, this is very cumbersome to work with. The programmer must still know the details of how a specific computer operates. He must also mentally translate complex operations and data structures into sequences of low level operations which use only the primitive data types that machine language provides. The programmer must also be intimately concerned with how and where data are represented within the computer. The second reason is that, they are not portable, i.e., a programme written for one type of computer cannot be run on another type of computer without significant alterations. Machine language is a low level language.
8.3 Why is High Level Language Developed?

To avoid these problems mentioned in the preceding section, High Level Languages (HLL) are developed. Basically in HLL the instructions are more compatible with human languages and human thought process. This is some sort of a compromise between human languages and machine languages. Hundreds of HLL exist. They differ in their degree of closeness to natural or mathematical language on one hand and to machine language on the other. They differ also in the type of problem for which they are best suited. Examples of high level language are Pascal, C, C++, Math Lab, FORTRAN, LISP etc. Generally, most of the computer programs are written in high level language.

8.4 Special Features of HLL

An HLL has some special features which facilitate reliable programming. Some of the features are described below.

8.4.1 Ease of understanding

An HLL program is generally easier to read, write and more correct than it is an assembly language programme, because an HLL usually provides a core natural notation for describing algorithms than does assembly language. They relieve the programmers from the burden of keeping the tedious details of a computer. These languages allow the programmers to utilize many of the symbols and much of the terminology that is already familiar. Even among HLLs, some are easier to use than others. An HLL provides the following features to design a clean and easy to understand program:

- operators, data structures, and ability to create such structures.
- control structure that clearly identifies the looping structures in a programme.
- scope rules that allow modifications to be made to a piece of program with unwittingly affecting other portions of the same programme.
- subroutines, allowing a program to be designed in small pieces.

8.4.2 Naturalness

Much of the understand ability of an HLL comes from the ease with which one can express an algorithm in that language. Some languages are clearly more suitable than others in this regard. FORTRAN is designed for numerical and mathematical computations, SNOBOL for string manipulation, DBASE, FOXPRO for database applications. A particular programme may be easy to develop in a particular language and not in others.
8.4.3 Portability

A high level language programme is relatively free from machine dependencies; they may be used with a little or no alteration on a variety of computers. Most HLLs have relatively well defined ‘standard version’ and programme conforming to the standard should run on any machine.

8.4.4 Efficiency of use

HLL have facilities for defining data structures, macros, subroutines, so that the programmers are able to implement programs in a way that makes efficient use of his time. The operating system and programming environment is also such that it reduces the programming time. An HLL programme needs to be translated into machine language so that computer can understand it to execute. This translation, often termed as compilation, is performed by a large programme called a compiler. The original program is called the source programme and the translated programme is called the object programme. A good HLL is able to translate the source programme into efficient object code.

8.5 Exercise

8.5.1 Multiple choice questions

1. Programming languages are usually divided into two classes
   a) high level and low level languages
   b) machine language and assembly languages
   c) machine language and high level languages
   d) assembly language and fourth generation languages.

2. Which of the following is not high level language?
   a) PASCAL
   b) C
   c) Machine language
   d) LISP.

3. Which one is developed to avoid major changes in written programme?
   a) high level languages
   b) low level languages
   c) assembly
   d) machine languages.
4. Which of the following is not a characteristic of a high level language?
   a) portability
   b) efficiency of use
   c) naturalness
   d) low productivity of programmers.

5. Which one is known as translator of source programme into machine language?
   a) compilation
   b) portability
   c) execution
   d) booting.

8.5.2 Questions for short answers

1. What is a low level language?
2. What are the problems of low level languages?
3. What are the special features of HLL?
4. Give four special features of HLL which makes program easy.
5. What is a compiler? What are a source program and an object program and how are they constructed?

8.5.3 Analytical question

1. Describe special features of high level languages with suitable examples.
Lesson 9 : Data Elements

9.1 Learning Objectives

On completion of this lesson you will learn

- basic data elements
- user defined data elements
- size taken by data elements
- limitation of capability of each data element
- operations available on data elements.

9.2 What are Data Elements?

Data elements are single items (numbers, characters etc.) which can be used with single names on one to one basis. An HLL supports many different data elements. There are four standard data elements - integer, real, character and Boolean. Some HLLs support user defined data elements, which include enumerated types and subrange types. Pascal, C supports all of them. These are discussed below:

9.3 Numerical Data

These include integers (whole numbers like 2, 3), real (numbers with floating point like 1.0, 2.56), complex numbers (numbers with imaginary part, like 2 + 3i) etc. The following operations on numerical data are defined:

<table>
<thead>
<tr>
<th>Operations</th>
<th>Data type</th>
<th>PASCAL Operator</th>
<th>C</th>
<th>Example Using Pascal</th>
<th>Example using C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>all</td>
<td>+</td>
<td>+</td>
<td>2.3 + 3.1 = 5.4</td>
<td>2.3 + 3.1 = 5.4</td>
</tr>
<tr>
<td>Subtraction</td>
<td>all</td>
<td>-</td>
<td>-</td>
<td>3.5 - 1.5 = 2.0</td>
<td>3.5 - 1.5 = 2.0</td>
</tr>
<tr>
<td>Multiplication</td>
<td>all</td>
<td>*</td>
<td>*</td>
<td>2.0 * 3.5 = 7.0</td>
<td>2.0 * 3.5 = 7.0</td>
</tr>
<tr>
<td>Division</td>
<td>all</td>
<td>/</td>
<td>/</td>
<td>6 / 3 = 2.0</td>
<td>6 / 3 = 2.0</td>
</tr>
<tr>
<td>Truncated Division</td>
<td>integer</td>
<td>DIV</td>
<td>/</td>
<td>9 DIV 4 = 2</td>
<td>9 / 4 = 2</td>
</tr>
<tr>
<td>Remainder after</td>
<td>integer</td>
<td>MOD</td>
<td>%</td>
<td>9 MOD 4 = 1</td>
<td>9 % 4 = 1</td>
</tr>
</tbody>
</table>

9.4 Character

Character type data are single characters enclosed in apostrophes like ‘a’, ‘B’, ‘$’ etc. Most computers, and virtually all microcomputers, make use of the ASCII (American Standard Code for Information Interchange) character set in which 128 different characters are ordered.
9.5 Logical or Boolean Data

Boolean-type data are truth values which are either true or false. Boolean-type expressions are formed by combining operands of the same type with relational operators. The following are the most common relational operators.

<table>
<thead>
<tr>
<th>Relational Operation</th>
<th>Pascal Operator</th>
<th>Expression</th>
<th>Value</th>
<th>C Operator</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal to</td>
<td>=</td>
<td>2 = 3</td>
<td>false</td>
<td>=</td>
<td>2 = 3</td>
</tr>
<tr>
<td>Not equal to</td>
<td>&lt;&gt;</td>
<td>2 &lt;&gt; 3</td>
<td>true</td>
<td>!</td>
<td>2! = 3</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
<td>4 &lt; 5</td>
<td>true</td>
<td>&lt;</td>
<td>4 &lt; 5</td>
</tr>
<tr>
<td>Less than or equal to</td>
<td>&lt;=</td>
<td>4 &lt;= 5</td>
<td>true</td>
<td>&lt;=</td>
<td>4 &lt;= 5</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;</td>
<td>6 &gt; 7</td>
<td>false</td>
<td>&gt;</td>
<td>6 &gt; 7</td>
</tr>
<tr>
<td>Greater than or equal to</td>
<td>&gt;=</td>
<td>3 &gt;= 2</td>
<td>true</td>
<td>&gt;=</td>
<td>3 &gt;= 2</td>
</tr>
</tbody>
</table>

Boolean type data can be subjected to logical operations (and, or, not etc.).

9.6 Pointers

These are data elements whose values address other data elements. The syntax may vary, the general idea is that we may write statement P := addr (X) to mean that P will now hold the address of X, or P will now point to X and P’s value will be address of variable X. Pointers are useful in allowing the programmer to create his own data structures, although their indiscriminate use can make the programs hard to read and debug.

9.7 Labels

These are data whose values are the position in the program. For example, when we write go to X, we mean that we are transferring the control of the program to the program location which is labeled or denoted by X. In the following example 100 is a label.

Example: In Pascal

```
100: X=4
```

```
readln (A)
```

If A=2 THEN go to 100
9.8 User Defined Data

9.8.1 Enumerated Data

An enumerated data type consists of an ordered sequence of identifiers (name of program element such as variable, function etc.), where each identifier is interpreted as an individual data item. These data items will collectively be associated with a particular name, which serves to identify the data type.

Example: For example a type 'days' can be declared as

```
TYPE days = {sun, mon, tue, wed, thu, fri, sat}
```

The variable `days` contains only these 7 values.

9.8.2 Subrange type Data

Subrange type data are the data items that fall within any defined subrange, thus forming a subset of contiguous, ordered data. For example, the range of marks obtained in an examination is 0 to 100. The type score_range is defined by

```
TYPE scorerange = 0 .. 100.
```

9.9 Exercise

9.9.1 Multiple choice questions

1. How many types of standard data elements are there in C?
   a) 2
   b) 3
   c) 4
   d) 5.

2. Which of the following statement is true?
   a) Subrange type data elements form a subset of integer data
   b) Subrange type data elements form a subset of any ordered data
   c) Subrange type data elements form a subset of any contiguous ordered data
   d) Subrange type data elements form a subset of data where each identifier is an individual data item.
3. Which of the following is not a standard data element?
   
   a) enumerated data  
   b) integer  
   c) real  
   d) Boolean.

4. Which one is valid?
   
   a) "2>3" integer  
   b) "2>3" character  
   c) "2>3" Boolean  
   d) "2>3" real data.

5. Which one of the following expression in Pascal is valid?
   
   a) \(24/5=4, 24 \text{ DIV } 5=4\)  
   b) \(24/5=4, 24 \text{ DIV } 5=4.8\)  
   c) \(24/5=4.8, 24 \text{ DIV } 5=4\)  
   d) \(24/5=5, 24 \text{ DIV } 5=4.8\).

9.9.2 Questions for short answers

1. What do you mean by data element?  
2. What are standard and enumerated data elements?  
3. Which arithmetic operators can be used with integer, real data types?  
4. Give the difference between DIV and '/' operators.  
5. What are the uses of pointer and label data types?

9.9.3 Analytical question

1. What do you mean by label, enumerated and subrange data types? Illustrate with examples.
Lesson 10 : Data Structure - I

10.1 Learning Objective

On completion of this lesson you will learn:

- use of data structures
- array as a data structure
- character as a data structure.

10.2 Concept of Data Structure

Data structure is a set of primitive data elements and other data structures, together with a set of structural relations among its components. A data structure is defined by (1) a specific way of arranging data, and (2) a set of operations performed on the data that make it useable in a certain way. Without a proper set of algorithms, the data structure is incomplete—just as an automobile, without a person who knows how to drive it, is only an inert pile of metal and glass. The most common data structures used by many languages are discussed in next sections.

10.3 Arrays

An array is a collection of elements of some fixed type, laid out in a $k$-dimensional rectangular structure. Each element of array can be accessed directly. An array is generally of fixed length, that is, the size of an array cannot be changed while the program is running. The main advantage of using an array is that, each of the array position can be used as a single variable and can be accessed directly. The entire array is referred to by an array name, and the individual item of data in the array can be obtained by suffixing a subscript, enclosed within parentheses, to the array name. The variable name, so obtained, representing individual item of data in an array is called the subscripted variable.

Consider the following one dimensional array, where each of its element is an integer. Suppose the array is defined as
In the definition, SCORE is the array name and so all the elements will be referred to by this name. The range within the bracket is called the subscript and determines the range of the location. The value of the subscripted variable SCORE(7) is 62. Each element can be accessed in this way. Now consider the following two dimensional array which is defined as SCORE(1..5, 1..9).

In the definition, two subscripts define the row and column number of the two dimensional array. In the subscripted variable, two subscripts indicate row and column number. For example, SCORE(3,8) means the element of 3rd row and 8th column.

### 10.4 Character Strings

These are really one dimensional array whose elements are characters. They can be represented as a sequence of characters bounded by “or” character. For example, “abcdef” is a string.
10.5 Exercise

10.5.1 Multiple choice questions

1. Which next the following is the definition of data structure?
   a) Data structure is a basic data type
   b) Data structure is a collection of basic data type
   c) Data structure is a way of arranging data
   d) Data structure is storing data.

2. Which of the following statement is valid?
   a) Generally the size of an array is fixed
   b) Generally the size of an array is dynamic
   c) Generally the size of an array is both fixed and dynamic
   d) Generally the size of an array is up to infinite.

3. What does indicate a subscripted variable?
   a) address of specific data
   b) memory element
   c) relative position of data from beginning
   d) Individual item of data.

4. Why does we use character storing data structure?
   a) for representing numeric
   b) for representing graphical
   c) for representing logical
   d) for representing a sequence of characters.

10.5.2 Questions for short answers

1. What is a data structure?
2. What is an array?
3. Give the definition of dimension and subscripted variable.
4. What is a character string?
5. Is there any similarity between array and character? Explain.

10.5.3 Analytical question

1. Describe the use of two dimensional arrays.
Lesson 11: Data Structure -II

11.1 Learning Objectives

On completion of this lesson you will learn:

- social data structures such as Record, Lists, Stacks, Queues
- accessing principles of these data structures
- use of these data structures.

11.2 Records

A record is a collection of elements of different types. The basic difference of array with record is that, all the array elements must be of same type, whereas the record elements may be of different types. The element of the record is called field, and each of the element may be accessed by mentioning the record name and the field name. Suppose we want to store a name and age of a collection of people. Now what we have to do is to define a record which has two fields. One field is of character string and will store the name, the other field is of integer type and it will store the age. Though the definition may vary from language to language, it may be like this:

```plaintext
record PERSON
  NAME CHARACTER(15)
  AGE INTEGER
```

We can access the NAME field of the record by mentioning PERSON.NAME (which contains the record name and the field name) and the AGE field by mentioning PERSON.AGE.

11.3 Lists

List is a sequence of data items. Each element is connected to its preceding and/or following elements by just knowing its address. If a list contains the variables A, B, C and D, then A will store the address of B, B the address of C, C the address of D. Such list is called one way list, because addressing continues from left to right. In two way list, A stores address of B, B store address of A, B stores address of C, C stores address of B and so on. The main difference of list from array is that, list elements must be sequentially accessed, that is, any of the list element cannot be directly accessed. For example, we cannot directly access C. We must start from A, as A knows the address of B, we can then move to B, and then knowing C’s address from B we can access C. The main advantage of list is that, if it is implemented by dynamic memory allocation, its size may be changed at runtime.
11.4 Stacks

A stack is a special type of list in which all insertions and deletions are made at one end of the list, commonly called “top” of the list. This is very much like a stack of trays in the cafeteria; the new tray is put on top of the stack and the tray at the top is removed. For this reason, stacks are also referred to as last-in-first-out (or LIFO) lists. Adding an item to a stack is called pushing it onto the stack, while deleting an item from stack is called popping it off the stack.

11.5 Queues

Queue is a special list where all insertions are made at one end (back) of the list and all deletions are made at another end (front) of the list. Unlike a stack, which is a last-in-first-out list, a queue is a first-in-first-out (or FIFO) list, because the element which is inserted first in the queue is deleted first. This works in the same way as a line at an airline ticket counter. When new passengers arrive, they join the line at the end. People who arrived earlier, however, leave the line at the front as they get their tickets. Perhaps queues are more familiar, so there are no special terms to describe adding items to queues and deleting items from queues.

Both stacks and queues are often used in real-life programming situation.

11.6 Exercise

11.6.1 Multiple choice questions

1. Which one is the valid?
   a) A record is collection of data types
   b) A record is collection of elements of different types
   c) A record is collection of array elements
   d) A record is collection of logical data elements.

2. Which one of the following statement is true?
   a) List a sequence of field
   b) List a sequence of data types
   c) List a sequence of data items
   d) List a sequence of characters.
3. Which one is valid?
   a) Stack is accessible only from top
   b) Stack is accessible only from bottom
   c) Stack is accessible insertion from top and deletion from bottom
   d) Stack is accessible insertion from bottom and deletion from top.

4. Which one of the following statement is true?
   a) Queue is a data structure facilitating first in first out
   b) Queue is a data structure facilitating last in last Out
   c) Queue is a data structure facilitating only deletion
   d) Queue is a data structure facilitating only insertion operations.

11.6.2 Questions for short answers

1. What type of data can be stored in Record?
2. What is the field of a Record?
3. What is a List? Give the advantages and disadvantages of this structure.
4. What is LIFO and FIFO?
5. What is the difference between list and array?

11.6.3 Analytical question

1. Describe the operating principle of stacks and queues.
Unit 4: High Level Language - II

Introduction

This unit gives the information about the format of any high level language. Different types of operators and their detailed characteristics have been discussed. Evaluation of different operators are shown in tabular form. In every compiler while evaluating arithmetic expression their sequence of operation is very important. This is called the precedence rule. This is described in lesson 13. Modular programming requires procedure. High level language must have this scope. Parameter passing is very important in this case. This is also described in this unit briefly.

Lesson 12 : Operators

12.1 Learning Objectives

On completion of this lesson you will learn

- some of the operators common to many High Level Languages
- different rules for applying operators in different operands
- string operators.

12.2 Arithmetic Operators

Arithmetic operators can be classified into two classes: unary and binary.

12.2.1 Unary Operators

They take only one argument. Examples are Negation (as in -X) in which it is called prefix operator (which precedes the operand) and Increment where the operator may be prefix or postfix (which follows the operand). For example, C language allows ++ as both a prefix and postfix unary operator, when X and Y are integer variables, X = ++Y means the same as

\[
\begin{align*}
Y &= Y + 1 \\
X &= Y
\end{align*}
\]

and X = Y++ means the same as

\[
\begin{align*}
X &= Y \\
X &= Y + 1
\end{align*}
\]
12.2.2 Binary Operators

They take two operands. Virtually all the binary operators are infix (which is placed between two operands). Examples are addition (X+Y), subtraction (X-Y) multiplication (X*Y), division (X / Y).

12.3 Relational Operators

A relational operator takes a pair of expressions as operands and returns a logical value, true or false. There are six common relational operators: ≥, ≤, =, ≠, < and >. They are used as shown in Table 9.2.

12.4 Logical Operators

A logical or Boolean operator has arguments with logical values, and the operator itself returns a logical value. The logical operators and, or, not, and “exclusive or” are common in many programming languages. The operands are interpreted as logical value, true or false. Not is a unary operator and the rule is that not true = false, not false = true. The other logical operators are binary and operates in following way:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A and B</th>
<th>A or B</th>
<th>A exclusive or B</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Relational operators and logical operators can be combined with arithmetic operators arbitrarily, as long as the types of results match the required types of operands. For example,

(A + B) and (X < Y)

is a legal expression in some language only if in that language the integer sum A + B may be interpreted as a logical value.

12.5 String Operators

The principal operations on strings that are common to many HLLs are

**Concatenation:** This operation adds a pair of strings into a single string. For example, concatenation of the string ‘abc’ and ‘de’ will produce the single string ‘abcde’.
Substring formation: This operation produces a certain consecutive portion of a string. For example, the substring of length four beginning at position three of the string 'abcdefgh' is 'cdef'.

Pattern searching: This operation searches for a pattern (substring) within a string and gives the starting location of the first occurrence of the pattern. For example, searching for the pattern 'cde' within the string 'abcdef' gives the answer three.

12.6 Exercise

12.6.1 Multiple choice questions

1. Let x=4, y=5, x=++y. What is the value of (x,y)?
   a) 4,5  
   b) 4,6  
   c) 3,5  
   d) 6,6.

2. Let x=4, y=5, x=y++. What is the value of (x,y)?
   a) 5,6  
   b) 5,5  
   c) 6,6  
   d) 8,9.

3. Which is the false statement?
   a) Logical operators may be unary
   b) Relational operators may be unary
   c) Arithmetic operators may be unary
   d) Logical operators may be binary.

12.6.2 Questions for short answers

1. How many types of operators you find in general?
2. Write differences between unary and binary operators.
3. What are the preconditions of using relational operator?
4. Mention logical operators and their operations.
5. Evaluate the following expressions for A=5, B=6, X=2, Y=8

i) \((A*B) \geq (B * Y)\)

ii) \((A<B) \ XOR (B >Y)\)

iii) \(A/B \geq X / Y\)

iv) \((B /Y\geq Y / Y) \ AND (A / X <X / B)\)

v) \((A + B) > (X - Y) \ NOR (\text{substr} ("mostofa", "tof"))\)

12.6.3 Analytical question

1. Describe string operators with suitable examples.
Lesson 13 : Precedence and Associativity

13.1 Learning Objectives

On completion of this lesson you will learn:

- how precedence is used for removing ambiguity
- precedence of different operators
- different types of associativity in evaluating expressions.

13.2 Precedence

Consider the arithmetic expression $A + B * C$. It may be interpreted in two ways:

- First add $A$ and $B$ and then multiply the result with $C$, $(A + B) * C$
- First multiply $B$ and $C$ and then add the result with $A$, $A + (B * C)$.

Obviously, the two results may not be the same. We, therefore, need the notion of precedence levels to indicate which operands are allowed to group their operands first. Normally, * and / are given higher precedence than + and -, meaning that $(A + B * C)$ is grouped first by finding operands to the left and right of *, then doing the same for +. That is, the grouping $A + (B * C)$ is correct. Different operators with same precedence level are treated as if they were the same operator and they are then evaluated by the associativity rule.

13.3 Associativity

When we combine operators to form expressions, the order in which the operators are to be applied may not be obvious. For example, $(A + B + C)$ can be interpreted as $((A + B) + C)$ or as $(A + (B + C))$. That is, we can go on addition from left to right or right to left of the expression. We say that the operator + is left associative if the operands are grouped left to right as in $(A + B + C)$. We say that it is right associative if the grouping is from right to left, as in $(A + (B + C))$.

Evaluation of expression with different operators on same precedence level is solely restricted by the associativity. For example, consider the expression $(20 / 5 * 2)$. Here * and / have same precedence. If the operators are left associative then the expression would be evaluated as $(20 / (5 * 2)) = 2$ while if the operators are right associative then the expression would be evaluated as $(20 / 5) * 2 = 8$.

Every HLL describes some precedence level and associativity to its operators.
13.4 Exercise

13.4.1 Multiple choice questions

1. Which one is equivalent to \( \frac{D}{B} \times C + B \) if the language supports left associative rule.
   
   a) \((B \times C) + \frac{D}{C}\)
   
   b) \(*C + \frac{D}{C}\)
   
   c) \((C + D)/C\)
   
   d) \(((D/B) \times C) + B\).

2. When two operators of same precedence is found in an expression then, which one is valid?
   
   a) evaluation depends on precedence rule
   
   b) evaluation depends on arbitrarily
   
   c) evaluation depends on associatively rule
   
   d) evaluation depends on addition.

3. If the language supports left associative rule, What is the value of \( 2 \times 2/4 + 6 - 8 - 7 \)?
   
   a) -8
   
   b) -9
   
   c) 5
   
   d) 6.

13.4.2 Questions for short answers

1. Why is precedence used?

2. "Precedence alone is not sufficient for removing ambiguity" - justify.

3. What special action is taken for some precedence operations?

4. How is associativity used for removing ambiguity?

5. Define left and right associativity with their examples.

13.4.3 Analytical question

1. Explain with example the precedence and associativity rules in computation.
Lesson 14 : Statements

14.1 Learning Objectives

On completion of this lesson you will learn

- different types of statements
- use of each type of statement in brief.

14.2 Moving Statements

The elementary actions of moving data from one place to another or moving the program in one direction to another are specified by the statements of a programming language. So at the highest level there are only two fundamental kinds of statements, first, data moving statements, and second, program moving statements.

14.3 Simple and Compound Statements

Statements can be either simple or compound. A simple statement performs a single action, while a compound statement does several actions in sequence. A simple statement does not contain any embedded statements. Examples of simple statements are as follows:

<table>
<thead>
<tr>
<th>In C</th>
<th>In Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) printf(.....);</td>
<td>1) WRITE (This is a simple statement);</td>
</tr>
<tr>
<td>2) int i := 12345;</td>
<td>2) Counter := Counter +1;</td>
</tr>
<tr>
<td>3) scanf(...........);</td>
<td>3) READ (X);</td>
</tr>
</tbody>
</table>

In certain languages, a program may be regarded as a sequence of simple statements (e.g., BASIC, SNOBOL, APL, C, C++).

A compound statement is composed of simple statements enclosed within a pair of braces {} in C programming. Compound statements enable computations that logically belong together to be grouped together. This ability helps make program readable. In PASCAL Compound statements use BEGIN, END, and the semicolon (;) to show where they begin and end. The following example of compound statements in C and Pascal.

<table>
<thead>
<tr>
<th>In Pascal</th>
<th>In C</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>{}</td>
</tr>
<tr>
<td>WRITE (Enter your name);</td>
<td>pi= 3.141593;</td>
</tr>
<tr>
<td>READIN (Name);</td>
<td>circumference= 2<em>pi</em>radius;</td>
</tr>
<tr>
<td>READIN</td>
<td>area= pi<em>radius</em>radius;</td>
</tr>
<tr>
<td>END</td>
<td>}</td>
</tr>
</tbody>
</table>
Of course, in other languages that have compound statements, the syntax may vary widely from these examples.

14.4 Types of Statements

Most of the statements in programming languages fall into one of the following major categories.

14.4.1 Computation statements

These are statements that apply operators to operands to compute new values and to move them to a new location. These are data-moving statements. Examples are:

In C
a) \( i = 0; \)
b) \( i = i + 1; \)

In Pascal
a) \( \text{Counter} := 1; \)
b) \( \text{Counter} := \text{Counter} + 1; \)

14.4.2 Structural statements

Certain statements such as END serve to group simple statements into structures.

14.4.3 Declaration statements

These statements generally produce no executable code. They only inform the compiler about the attribute of the name encountered in the program.

14.4.4 Input and output (I/O) statements

These statements serve the role indicated by their names. In many languages they are implemented by their library subroutines made available by the operating system to control input/output devices and convert data to or from human-readable form. They are also data moving statements.

The following are PASCAL I/O statements:
High Level Language - II

In Pascal

a) WRITE (This is displayed on the screen);
   {This statement moves a text from memory to the PC’s screen. }

b) READ (Counter);
   {This statement moves a number from screen to memory}

In C

a) Printf (..........................);
   {This statement moves a text from memory to the PC’s screen. }

b) Scanf (.................); {This statement moves a number from screen to memory}

14.4.5 Sequence control statements

These are program moving statements. In most languages, control automatically moves from one statement to the next. Certain statements such as goto (jumps to another location), call (execute another subroutine), return (returns to original location after executing the subroutine) etc. deliberately alter the flow of control. Examples of control statements are looping statements, which make the program repeat a sequence of operations until a certain condition is fulfilled; and branching statements, which, based on the value of some control variable, make the program take one path instead of another. Thus the following are sequence control statements in Pascal and C.

{ 
  a) FOR Counter := 1 to 10 DO 
  WRITELN (creates than five); 
  {looping statement, cause the program to execute an I/O statement 10 times } 

  b) IF Counter < 5 
  Then WRITELN (This is a loop statement); 
  Else WRITTEN (Less than or equal to five) 
  {Branching statement, cause the program to go in one direction or another based on the truth value of the expression Counter>5 }

In C

{While (count <=0) 
Printf (" X "); 
Scanf ("%f", X); 
sum+ = X; 
++ count; 
}
14.5 Exercise

14.5.1 Multiple choice questions

1. *How many types of statements are there in a programming language?*
   
   a) 5  
   b) 4  
   c) 2  
   d) 8.

2. *In PASCAL compound statements are bounded by*
   
   a) left and right curly braces  
   b) left and right braces  
   c) Begin, End and semicolon (;)  
   d) no special arrangement required.

3. *Which types of statements are not used for code generation?*
   
   a) I/O statements  
   b) Arithmetic statements  
   c) Declarative statements  
   d) Structural statements.

4. *FOR and DO-WHILE loops are*
   
   a) I/O  
   b) Structural  
   c) Sequence control statements  
   d) none of the above.

14.5.2 Questions for short answers

1. How many types of statements are there in a programming language?  
2. What are the uses of compound statements?  
3. Which statements are used to implement repetitive jobs?  
4. What is the difference between structural and sequence control statements?  
5. Give some example as sequence control statements.

14.5.3 Analytical question

1. Classify statements in different views.
Lesson 15: Procedures and Functions

15.1 Learning Objectives

On completion of this lesson you will learn:

- how information is transferred between called procedure and calling procedure
- use of procedure as shortening programmers effort
- different types of calling parameter passing methods of procedure.

15.2 Ability of Procedure

The ability to define and call procedure is a great asset in a programming language. Procedures:

1. Permit modular design of programs, by allowing large tasks to be broken into smaller units.
2. Permit economy in the size of programs and in the total programming effort, since similar computations need be specified only once, to
3. Add extensibility to language, since operators can be defined in terms of procedures, which can then be used as functions within expressions.

15.3 Parameter Transmission

One problem arising from the introduction of procedures is the method of transmitting information to and from procedures. The usual method of communication between two procedures, one of which calls the other, is through global variables and parameters of the called procedure. This section discusses various ways of passing information to a called procedure through its parameters. It is important to know what parameter passing convention a language uses, because the value of a program can depend on the convention used.

15.4 Parameters

At first, we need to make a distinction between the definition of a procedure and its use. Suppose the sequence

```
INTEGER PROCEDURE DIVIDE (X, Y) INTEGER X, Y;
    IF Y = 0 THEN RETURN 0
    ELSE RETURN X/Y
```

defines a procedure called DIVIDE. X and Y in this definition are called formal parameters, or just formulas. In use of this procedure, as in

```
A := DIVIDE (B,C)
```
B and C are called actual parameters or actuals. The terms B and C could be expressions, rather than simple variables. In general, the formal parameters of a procedure definition are place holders for values that will be supplied when the procedure is called. The actual parameters provide the values to be substituted for the formal parameters.

There are two common methods of passing parameters. They are: call-by-value and call-by-reference. Both these methods have their specific applications.

15.4.1 Call-by-value

This is the simplest possible method of passing parameters. The actual parameters are evaluated and the value is passed to the subroutine or procedures. The effect is that of initializing the formal parameters of the subroutines with the value of the actual parameters. The important fact is that, changing the value of the formal parameters within the subroutine will have no effect on the value of the actual parameters. In other words, any change of the parameters within the subroutines will not be reflected outside the subroutine. Consider the following procedure definition, where the parameter is passed by value:

```
PROCEDURE CHANGE(X, Y) INTEGER X, Y;
BEGIN
  X = 2;
  Y = 3
END
```

Now carefully see the following program segment:

```
A := 10;
B := 12;
CALL CHANGE(A, B)
CALL PRINT(A, B)
```

The program works in this way: A and B are initialized to 10 and 12 respectively. The procedure CHANGE is called with A and B as actual parameters. Value of A and B will substitute the value of X and Y. So at the starting of CHANGE value of X and Y will be 10 and 12 respectively. Within CHANGE the value of X and Y are reset to 2 and 3. As the parameters are passed by value, the change in X and Y will not affect the actual parameters A and B. So after returning from CHANGE, value of A and B will be 10 and 12 again and PRINT will print 10 and 12, not 2 and 3.

15.4.2 Call-by-Reference
When parameters are passed by reference, the calling program passes the memory locations (not the values, as in pass-by-value) of the actual parameters to the subroutine. Many languages (like PASCAL) restrict that, the actual parameter must be a variable and must not be a constant or expression. The important thing to note is that, when passed by reference, any change in the formal parameters within the subroutine will affect the value of the actual parameters, i.e., any change in the formal parameters will be reflected outside the subroutine. Consider the same procedure definition and program segment discussed in call-by-value part. Suppose that the parameters are now passed by reference. Then the program will work in this different way: A and B are initialized to 10 and 12 respectively. The procedure CHANGE is called with A and B as actual parameters. Now X and Y will have the same memory location of A and B respectively. So at the starting of CHANGE value of X and Y will be 10 and 12 respectively. Within CHANGE the value of X and Y are reset to 2 and 3 respectively. As the parameters are passed by reference, the change in X and Y will not affect the actual parameters A and B. So after returning from CHANGE, values of A and B will be 2 and 3 and PRINT will be print 2 and 3, not 10 and 12.

15.5.1 Exercise

15.5.1 Multiple choice questions

1. Procedure decreases
   a) programme size
   b) programme execution time
   c) compiling time
   d) none of the above.

2. Which is a false statement?
   a) Procedures decreases programming effort
   b) Procedures make programming modular
   c) Parameter passing is done by constants and variables
   d) Operators are substituted by procedures.

3. In call by value convention the parameter is passed
   a) arithmetic expression
   b) logical expression
   c) variables
   d) value.
4. For call by reference the value of the parameter must be
   a) expression
   b) operator
   c) variable
   d) array.

5. Consider the following procedure

   A = 2;
   B = 5;
   PROCEDURE swap (X,Y) INTEGER X,Y;
   BEGIN
   temp = X;
   X = Y;
   Y = temp;

   If procedure uses call by reference and swap (A,B) is called what will be the value of A and B
   a) 2,5
   b) 5,5
   c) 5,2
   d) 2,2.

6. What happened if call by value is used?
   a) 2,5
   b) 5,5
   c) 5,2
   d) 2,2.

15.5.2 Questions for short answers

1. What do you mean by actual and formal parameters?
2. Give an example of actual and formal parameters.
3. Mention different methods of parameter passing.

15.5.3 Analytical questions

1. Describe different calling conventions with examples.
2. Why procedures are used in writing programs?
Unit 5 : Pascal Programming

Introduction

So far we have discussed all the characteristics of programming languages in general. In this unit we are going to learn a special programming language called ‘Pascal’. This unit discusses a few fundamental concepts related to Pascal programming. Syntax and keywords of Pascal have been described in this unit. The ins and outs of declaring identifiers are shown below. Different data types have different rules and regulations. If we do not follow the rules we will find a syntax error at the time of computation. We can then correct the error to run the problem. So this unit works as a manual. Different types of statements are also introduced in the last lesson of the unit. Rules and regulations of declaring procedure and parameter passing are also introduced in this unit briefly.

Lesson 16 : Pascal Fundamentals I

16.1 Learning Objectives

On completion of this lesson you will learn:

- the Pascal character set and reserved words
- identifiers and standard identifiers
- numbers
- strings.

16.2 The Pascal Character Set

Pascal uses the letters A to Z (both upper and lower case), the digits 0 to 9, and certain special symbols as building blocks to form basic program elements (numbers, identifiers, expressions, etc.). The special symbols are listed below:

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

On some computers the symbols (* and *) are used instead of { and }. Also, the symbols (, and .) may be used in place of [ and ], and @ may replace ^. Note that some of the special symbols are composed of two separate, consecutive characters (e.g., <=, :=, ..).
16.3 Reserved Words

There are certain reserved words that have a standard, predefined meaning in Pascal. They are:

- **AND**
- **END**
- **NIL**
- **SET**
- **ARRAY**
- **FILE**
- **NOT**
- **THEN**
- **BEGIN**
- **FOR**
- **OF**
- **TO**
- **CASE**
- **FUNCTION**
- **OR**
- **TYPE**
- **CONST**
- **GOTO**
- **PACKED**
- **UNTIL**
- **DIV**
- **IF**
- **PROCEDURE**
- **VAR**
- **DO**
- **IN**
- **PROGRAM**
- **WHILE**
- **DOWNTO**
- **LABEL**
- **RECORD**
- **WITH**
- **ELSE**
- **MOD**
- **REPEAT**

These reserved words can be used only for their intended purpose, and therefore, cannot be arbitrarily redefined by the programmer. It is customary to display the reserved words within a Pascal program in either boldface or upper-case letters.

16.4 Identifiers

An identifier is a name that is given to some program element, such as a constant, a variable, a procedure or a program. Identifiers comprise letters or digits, in any order, except that the first character must be a letter. Both upper- and lower-case letters are permitted, and are considered to be indistinguishable. An identifier can be arbitrarily long, although some implementations of Pascal recognize only the first eight characters.

**Example:**

(a) The following names are valid identifiers:

- x, y12, sum, temperature, names, area, table 4.

(b) The following names are NOT valid identifiers:

- 4th - The first character must be a letter.
- Array - ARRAY is a reserved word.
- last name - Blank spaces are not allowed.

16.5 Standard Identifiers

Pascal contains a number of standard identifiers that have certain predefined meanings. These standard identifiers are:

- **abs**
- **false**
- **pack**
- **sin**
### 16.6 Numbers

Numbers can be written in several different ways in Pascal. In particular, numbers can include a sign, a decimal point and an exponent (or scale factor) if desired. The following rules apply to all numbers.

a) Commas and blank spaces cannot be included within the number.

b) The number can be preceded by a plus (+) sign or minus (-) sign if desired. If a sign does not appear the number will be assumed to be positive.

c) Numbers cannot exceed specified maximum and minimum values. These values depend upon the type of number, the particular computer and the particular compiler being used.

### 16.7 Integers

An integer contains neither a decimal point nor an exponent. Thus an integer is simply a sequence of digits, preceded (optionally) by a sign. The magnitude of an integer can range from zero to some maximum value that varies from one computer (and one compiler) to another. A typical maximum value on a microcomputer is 32767, though some computers allow integers that are much larger.

**Example:**

(a) The following integers are valid:
0, 1, +1, -1, 743, -5621, 60000000, -999999

(b) The following integers are invalid:

123,456 - Commas are not allowed
36. - A decimal point cannot appear in an integer number
10 20 7 - Blank spaces are not allowed

16.8 Real Numbers

A real number must contain either a decimal point or an exponent (or both). If a decimal point is included, it must appear between two digits. Thus a real number cannot begin or end with a decimal point. An exponent (scale factor) can be included to shift the location of the decimal point. (If a decimal point is not included within the number, it is assumed to be positioned to the right of the last digit.) This is essentially the same as scientific notation except that the base 10 is replaced by the letter E (or e). Thus, the number $1.2 \times 10^{-3}$ would be written as $1.2E^{-3}$ or $1.2e^{-3}$. The exponent itself must be either a positive or a negative integer.

Real numbers have a much greater range than integer numbers. Typically, the magnitude of a real number might range from a minimum value of approximately $1E^{-38}$ to a maximum of approximately $1E^{+38}$. (These values will vary from one computer (one compiler) to another.) In addition, the number 0.0 (zero, which is less than $1E^{-38}$) is also a valid real number.

**Example :**

(a) Valid real numbers--

0.0, 1.0, -0.2, 827.609, 5000.0067, -0.00871, 12.8

(b) Invalid real numbers--

1. - A digit must be present on each side of the decimal point
1,00.08 - Commas are not allowed
.33333 - A digit must be present on each side of the decimal point
50 - Either a decimal point or an exponent must be present

(c) The quantity $-5.026 \times 10^{-17}$ can be represented by--

$-5.026E^{-17}$, $-0.5026E^{-16}$, $-50.26E^{-18}$ or $-0.0005026E^{-13}$

The following are valid real numbers with exponents

$2E^{-8}$, $-0.006e^{-5}$, $1.6667E + 8$, $+ 0.121212e+12$

(d) The following real numbers are NOT valid
Pascal Programming

3.0E+10 - A digit must be represented on each side of the decimal point.
8e2.3  - The exponent must be an integer.
       (It cannot contain a decimal point)
.3333e-3 - A digit must be represented on each side of the decimal point.
3E 10   - Blank spaces are not allowed.

16.9 Numerical Precision

Remember that integer numbers are exact quantities whereas real numbers are approximations. Thus the real number 1.0 might actually be represented within the computer's memory as 0.99999999..., even though it might appear as 1.0 on the monitor or on a printout. Therefore real numbers cannot be used for certain purposes, such as counting, indexing etc., where exact values are required.

16.10 String

A string is a sequence of characters (i.e., letters, digits and special characters) enclosed by apostrophes. Both upper-case and lower-case letters can be used. The maximum number of characters that can be included in a string will vary from one version of Pascal to another. Most versions allow maximum string lengths of at least 255 characters, which is adequate for most purposes.

If a string includes an apostrophe, the apostrophe must be entered twice. Only one apostrophe will appear, however, when the string is printed out or displayed. Thus, a single apostrophe within is interpreted as a string delimiter, whereas a repeated apostrophe is interpreted as a single apostrophe within the string. Strings are normally used in write statements to label output.

Example :

(a) The following strings are valid---
   'GREEN'
   'Washington, D.C. 20005'
   '270-32-3456'
   '$19.95'
   'The correct answer is:'
   '2*(I+3)/J'

(b) A Pascal program contains the following string :
   'Please don't verb your nouns.'
   (Notice the repeated apostrophe in the word "don't").
   If the program causes the string to be printed or displayed,
however, it will appear as it should; namely, Please don't verb your nouns.

16.11 Exercise

16.11.1 Multiple choice questions

1. Most versions of PASCAL allow maximum string lengths of at least
   a) 255
   b) 128
   c) 8
   d) 64 characters.

2. If a string includes an apostrophe, the apostrophe must be entered
   a) twice
   b) once
   c) with a quote sign
   d) surrounded by quote signs.

3. The following is an invalid real number.
   a) 0.0
   b) 5000.0067
   c) 1,00.08
   d) -50.26e-18.

4. The quantity -5.021X 10^{-17} cannot be represented by
   a) -5.021E-17
   b) -.5021E-16
   c) -5 0.21e-18
   d) -5,00.021e-19.

5. The real number 5.E+11 is not valid because
   a) there must be a digit on each side of the decimal point
   b) Decimal point should not be there
   c) + sign should not be there
   d) 11 must be inside parentheses.
6. The following is not a valid real number

a) 8e2.3
b) 0.021
c) -6.012E-22
d) 20.67895.

16.11.2 Questions for short answers

1. What is an identifier? Write the conditions which must be satisfied to become an identifier.
2. Write the purpose of reserve words.
3. What is a standard identifier?
4. What does an integer data represent? How are real data types represented in programs?
5. Write the rules to represent a number in Pascal. Describe different parts of a real number.

16.11.3 Analytical questions

1. How is strings represented by Pascal? What is the constraint of representing string?
2. How are the problems of apostrophes resolved in Pascal string representation?
Lesson 17 : Pascal Fundamentals -II

17.1 Learning Objectives

On completion of this lesson you will learn:

- various data types
- constants
- variables
- expressions.

17.2 Data Types

One of the most important and interesting characteristics of Pascal is its ability to support many different types of data. These include:

17.2.1 Simple data types

These are single items (numbers, characters, etc.) that are associated with single identifiers on a one-to-one basis. Actually there are several different simple data types. These include the four standard data types:

(i) integer
(ii) real
(iii) char
(iv) Boolean.

The user defined simple types include:

(i) enumerated
(ii) subrange.

17.2.2 Structured data types

These data types consist of multiple data items that are related to one another in some specified manner. Each group of data items is associated with a particular identifier. The individual data items within each group can also be associated with corresponding individual identifiers. There are four types of structured data in Pascal:

(i) Arrays
(ii) Records
(iii) Files
(iv) Sets.
Pascal Programming

17.2.3 Pointer data types

These types of data are used to construct dynamic structured data types.

17.3 Constants

It is often convenient to associate a simple data item, such as a numerical value or string, with an identifier, thus providing a name for the data item. The identifier is called a constant if the data item is assigned permanently (i.e., if the value of the data item remains unchanged throughout the program). A constant must always be defined before it can appear in a Pascal statement.

This definition serves two purposes:

- it establishes that the identifier is a constant
- it associates a value with the constant.

The type of constant will implicitly be determined by the data item. The general form of a constant definition is expressed as:

\[
\text{CONST name} = \text{value}
\]

where, name is an identifier that represents the constant name
value is the actual data item that is assigned to name.

Example:

(a) A Pascal program requires frequent use of the numerical value 0.1666667. It may, therefore, be convenient to introduce a constant called fraction, which can be used in place of the actual number. This constant can be defined by writing:

\[
\text{CONST fraction} = 0.1666667;
\]

Subsequent references to the identifier fraction will be equivalent to referencing the actual number. Note that fraction is considered to be a real constant, since it is associated with a real number.

(b) A Pascal program makes frequent use of the string:

\[
\text{CONST title} = \text{'The Super-Duper Computer Company'};
\]

If we want to print out the actual string at a later point in the program, we merely refer to the identifier title in the appropriate output statement.
17.4 Variables

An identifier whose value is allowed to change during the execution of a program is called a variable. Every variable must be individually declared (i.e., defined) before it can appear in a statement. The variable declaration establishes the fact that the identifier is a variable (rather than a constant etc.), and specifies the type of the variable. Unlike a constant definition, however, a data item (e.g., a numerical value or string) is not associated with a variable within a variable declaration. The general form of a variable declaration is:

\[ \text{VAR name : type} \]

or if there are several variables of the same type,

\[ \text{VAR name1, name2, name3, ..., nameN : type} \]

Where, name1, name2, ... are identifiers that represent individual variable names type refers to the data type of the variables.

Example:

A Pascal program contains the integer variables \( \text{row} \) and \( \text{column} \), the real variable \( \text{value} \), and the char variable \( \text{flag} \). The program would, therefore, contain the following declarations:

\[
\begin{align*}
\text{VAR row, column : integer;}
\text{value : real;}
\text{flag : char;}
\end{align*}
\]

17.5 Expressions

An expression is a collection of operands (i.e., numbers, constants, variables, etc.) joined together by certain operators to form an algebra-like term that represents a value (i.e., a simple data item). There are two types of expressions in Pascal:

(i) Numerical Expressions - it represents a numerical value
(ii) Boolean Expressions - it represents a logical condition which is either true or false.

When constructing numerical expressions, care must be taken to distinguish between integer and real quantities. This is true of the operators and operands as well as the expression itself.

All expressions must satisfy the following general conditions:
Pascal Programming

- Two successive operators are not permitted. However, parentheses can be used to separate two successive operators. (Remember that parentheses must always be used in pairs).
- An expression can consist of a single identifier that is used as a constant or a variable.
- A function name (i.e., a function reference) can be used in place of a constant or variable identifier within an expression.

Example:

(a) A typical numerical expression is shown below:

\[(b*b - 4*a*c)/(2*a)\]

The identifiers a, b and c and the numbers 4 and 2 are called operands; and the symbols *, -, and / are the corresponding operators (which represent multiplication, subtraction and division, respectively). The parentheses are used to specify the order in which the operations are carried out. The entire expression represents a number. Thus, if a, b, c represent the values 1, 2, 3 respectively, the expression will represent the value -4.

(b) Here is an example of boolean (logical) expression:

\[\text{pay < 1000.0}\]

In this expression, pay is a variable of type real, 1000.0 is a real number and the symbol < is a boolean operator. (Note that pay and 1000.0 are operands in this boolean expression.) The expression will have the value true if pay represents a value that is less than 1000.0; otherwise the expression will have the value false.

(c) Boolean expressions are used in various control structures, such as the IF..THEN structure shown below:

\[
\text{IF pay < 1000.0 THEN writeln (employeenumber);}\]

This structure will cause the value of the variable employee number to be written out if the value of pay is less than 1000.0.
17.6 Exercise

17.6.1 Multiple choice questions

1. The following is not a structured data type
   a) Arrays
   b) records
   c) files
   d) integer.

2. How many types of variables are there in PASCAL?
   a) 2
   b) 3
   c) 4
   d) 5.

3. Pointer data types are used to
   a) construct dynamic structured data types
   b) records
   c) files
   d) static structured data types.

17.6.2 Questions for short answers

1. Why are constants used?
2. Write down the declaration method of constants.
3. What is the definition of variables?
4. What is the difference between a variable and a constant?
5. What are the components of an expression?

17.6.3 Analytical question

1. What is the taxonomy of different data types used in Pascal?
Lesson 18 : Pascal Fundamentals - III

18.1 Learning Objectives

On completion of this lesson you will learn:

- statements in a program
- introduction to procedures and functions
- Pascal Syntax diagram.

18.2 Statements

A Pascal statement is an instruction or a group of instructions, that causes the computer to carry out certain actions. There are two basic types of statements in Pascal:

18.2.1 Simple statements

These are essentially single, unconditional instructions that perform one of the following tasks:

- assign a data item to a variable, called an assignment statement.
- access a self-contained computational module, called a procedure.
- transfer program control unconditionally to another part of the program, (The GOTO statement).

18.3 Structured Statements

Pascal recognizes several different types of structured statements. These include:

- compound statements, which consist of a sequence of two or more consecutive statements
- repetitive statements, which involve the repeated execution of several simple statements
- conditional statements, in which one or more simple statements are executed only if some specified logical condition is satisfied.

Example:

(a) A typical assignment statement is shown below:

\[
\text{tax} := 0.14 \times \text{gross};
\]

In this example, it is assumed that tax and gross are both variables of type real, and that a real value has been assigned to gross. The assignment statement causes the value of gross to be multiplied by 0.14 and the product assigned to the variable tax. Notice that the symbol
which is used for assignment is ":=". Also note that the right hand side of the statement is a numerical expression.

(b) Here is a typical GOTO statement:

GOTO 100;

Thus the next statement to be executed will be the statement that is labeled 100. Use of GOTO statement is discouraged.

(c) A typical compound statement is shown below:

```
BEGIN
    read(radius);
    area := 3.14159 * sqr(radius);
    write(radius, area);
END
```

Note that the simple statements that comprise the compound statement are enclosed within the keywords BEGIN and END. Also notice that the simple statements are separated from one another by semicolons.

(d) Here is a typical repetitive statement:

```
FOR count := 1 TO 100 DO write(count);
```

This statement will be executed 100 times. Each time the statement is executed the current value of the variable count will be printed out or displayed. Thus the statement will cause the values 1 2 3 4 ... 100 to appear on the output device.

(e) A typical conditional statement is shown below:

```
IF pay < 1000.0 THEN write('Group 1') ELSE write('Group 2');
```

This statement causes the message "Group 1" to appear on the output medium if the variable pay represents a value that is less than 1000.0. If pay has a value that is greater than or equal to 1000.0, however, then the message "Group 2" will appear instead.

### 18.4 Procedures and Functions

Procedures and functions are self-contained program elements, sometimes referred to as modules, that carry out designated actions. These modules can be accessed from anywhere within a program. Moreover, if the same module is accessed from several different points, the module can be given different information (i.e., different values for the required data items) at each access point. When a module is accessed, the information provided is processed by the action statements within the module. Usually, this will cause new information to be generated. This information is then returned to the point at which the module was last accessed, and the program continues to execute from that point.
The information that is passed to a module is provided as a list of data items (i.e., constants, variables, expressions, etc.), called parameters. These parameters are separated by commas and enclosed in parentheses, immediately after the module name. Certain of the parameters may also be used to represent new information that is generated within the module. Thus, the parameters may represent information being returned by a module as well as information supplied to the module. Pascal supports both standard and user defined procedures and functions. The standard procedures and functions are included in the Pascal library, which is a part of the language.

All **procedures** have the following general characteristics:

- a procedure is accessed by a simple statement consisting of the procedure name followed by (optional) list of parameters.
- the parameters may represent information supplied to the procedure or, under certain conditions, information returned by the procedure.
- any number of data items can be transferred between a procedure and its reference point (i.e., the statement that accesses the procedure).

All **functions** have the following general characteristics:

- a function is accessed by specifying its name within an expression, as though it were an ordinary variable, followed by the an (optional) list of parameters.
- the function name can be followed by an (optional) list of parameters. These parameters are used only to transfer information to the function from its reference point.
- the function will return a single data item. This data item will be represented.
- a function must be of the correct data type for the expression in which it is accessed.

**Example:**

(a) The write statement is actually a reference (i.e., an access) to a standard Pascal procedure. Thus the statement

```
write(a,b,c);
```

will cause the values of the parameters `a`, `b`, and `c` to be printed out or displayed.

Note that the parameters here all represent information being supplied to the procedure; no new information is returned.

(b) Consider the statement:

```
area := 3.14159 * sqr(radius);
```
The right hand side is a numerical expression in which the variable \textit{radius} is passed to the standard function \textit{sqr}. This function returns the value of the \textit{radius} squared. The value of the \textit{radius} squared is then multiplied by 3.14159, and the result is assigned to the variable \textit{area}. Note that \textit{radius} is a parameter that represents a data item supplied to the standard function. The data item is returned by the function is represented by the function name \textit{sqr}. Also, note that the type of the function is numeric (specially, read); this is necessary if the function is to be used in numeric expression.

\subsection*{18.5 Pascal Syntax Diagram}

Syntax diagram is a method that is commonly used to represent the syntactical (i.e., the grammatical) constructs in Pascal.

\textbf{Example:}

The following figure shows a Pascal syntax diagram which explains the manner in which an identifier can be constructed. This diagram shows that an identifier must begin with a letter, as indicated by the left most box containing the word "letter". Following this box is a straight path with two optional return loops. Each loop contains a box representing a type of character that can be included as a part of the identifier name. Since the topmost box contains the word "letter" and the bottom box contains the word "digit", we conclude that an identifier must begin with a letter, followed by any number of letters and digits.

These syntax diagram can become quite complex, even for the relatively simple constructs. Therefore many beginners choose not to use them. Once the students have acquired some familiarity with Pascal, however, these diagrams appear much more understandable.

![Syntax Diagram for Identifier](image)

\textbf{Fig. 18.1 Syntax Diagram for Identifier}
18.6 Exercise

18.6.1 Multiple choice questions

1. How many types of basic statements are there in PASCAL?
   a) 2
   b) 3
   c) 4
   d) 5.

2. Which of the characteristics is not general to a function?
   a) A function is accessed by specifying its name.
   b) The function may return more than a single data item.
   c) A function must be of correct data type for the expression in which it is accessed.
   d) The function name can be followed by an optional list of parameters.

3. All procedures do not have the following general characteristics:
   a) A procedure is accessed by a simple statement consisting of the procedure name followed by an (optional) list of parameters.
   b) The parameters may represent information supplied to the procedure or, under certain conditions, information returned by the procedure.
   c) Any number of data items can be transferred between a procedure and its reference point (i.e., the statement that accesses the procedure).
   d) Procedure can be used in an expression like function.

4. The following is not a structured statement in PASCAL:
   a) Compound statements
   b) repetitive statements
   c) conditional statements
   d) assign a data item to a variable.

18.6.2 Questions for short answers

1. What do you mean by an assignment statement?
2. How many types of structured statements are there in Pascal?
3. Describe with examples of compound and statements.
4. What are the basic differences between function and procedure?
5. What is the use of syntax diagram?
18.6.3 Analytical questions

1. Give the representation of real numbers, identifiers with a syntax diagram.
2. What do you mean by procedure and function? Describe essential characteristics of a function and procedure.
3. Describe with example of different types structured statements.
Unit 6 : Data Types

Introduction

This unit describes different data types used in Pascal. Declaration methods of data types, their restriction and range are the main theme of this unit. Only simple data types are discussed in this unit. Compound data types are discussed in unit 9. Different operators which can be used for these data types are also shown in a tabular form. Constants are also useful in writing programs. The concept of constant has been introduced and method of using it is also listed. A brief description on Pascal arithmetic expression is also found in one of the lessons in this unit. It covers associative and precedence rules associated with it. The last lesson shows how the basic data types can be read or written. That is the input and output of the Pascal programming. There are several standard Pascal procedures which supplies some variable parameter routines. Formats of input and output are also described briefly in the lesson.

Lesson 19: Simple Data Type -I

19.1 Learning Objectives

On completion of this lesson you will learn :

- integer, real, char and Boolean types of data items
- how to use them
- syntax of their uses.

19.2 Different Data Types in Pascal

Pascal uses the following four standard types of data items :

(i) Integer-type
(ii) Real-type
(iii) Char-type
(iv) Boolean-type.

This section will cover first two types and the next section will cover the remaining two.

19.3 Integer Type Data

Integer type data are whole number (integer) quantities. Included within this category are integer-type constants, variables, functions and expressions.
Collectively, the operators that are used to carry out numerical-type operations are called arithmetic operators. There are six arithmetic operators that can be used with integer-type operands. Five of these will produce an integer-type resultant, the sixth will produce a quantity of type real, these operators are summarized below:

<table>
<thead>
<tr>
<th>Arithmetic operator</th>
<th>Purpose</th>
<th>Type of operands</th>
<th>Type of resultant</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Integer</td>
<td>Real</td>
</tr>
<tr>
<td>DIV</td>
<td>Truncated division</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>MOD</td>
<td>Remainder after division</td>
<td>Integer</td>
<td>Integer</td>
</tr>
</tbody>
</table>

Example:

Suppose that a and b are integer-type variables that have been assigned values 13 and 5, respectively. Several simple integer expressions and their resulting values are shown below.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+b</td>
<td>18</td>
</tr>
<tr>
<td>a - b</td>
<td>8</td>
</tr>
<tr>
<td>a*b</td>
<td>65</td>
</tr>
<tr>
<td>a/b</td>
<td>2.6</td>
</tr>
<tr>
<td>a DIV b</td>
<td>2</td>
</tr>
<tr>
<td>a MOD b</td>
<td>3</td>
</tr>
</tbody>
</table>

19.3.2 Common rules

Some of the more common rules, which apply to expressions involving only two integer operands, are given below:

1. The resultant of the two integer operands will be positive if both are positive otherwise it may be negative.
2. The two division operators (/ and DIV) and the MOD operator require that the second operand be non-zero.
3. Use of the DIV operator with a negative operand will result in truncation toward zero i.e., the resultant will be smaller in magnitude than the true quotient.
Data Types

4. According to the ISO standard, the second operand cannot be negative when using the MOD operator. However, many implementations of Pascal do allow this operand to be negative. In such cases the sign of the resultant will be determined so that the condition

\[(a \text{ DIV } b)*b + (a \text{ MOD } b)\]

is always satisfied, regardless of the signs of the individual operands.

**Example:**

Suppose that \(i\) and \(j\) are integer-type variables whose assigned values are 11 and -3, respectively. Several integer expressions and their results are given below:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i+j)</td>
<td>8</td>
</tr>
<tr>
<td>(i-j)</td>
<td>14</td>
</tr>
<tr>
<td>(i*j)</td>
<td>-33</td>
</tr>
<tr>
<td>(i \text{ DIV } j)</td>
<td>-3</td>
</tr>
<tr>
<td>(i \text{ MOD } j)</td>
<td>2 (nonstandard)</td>
</tr>
<tr>
<td>(i/j)</td>
<td>-3.6666666667</td>
</tr>
</tbody>
</table>

If \(i\) had been assigned a value of -11 and \(j\) had been assigned 3, then we would have

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i \text{ DIV } j)</td>
<td>-3</td>
</tr>
<tr>
<td>(i \text{ MOD } j)</td>
<td>-2</td>
</tr>
<tr>
<td>(i/j)</td>
<td>-3.6666667</td>
</tr>
</tbody>
</table>

Finally, if \(i\) and \(j\) were both assigned negative values (-11 and -3 respectively), then we would obtain

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i \text{ DIV } j)</td>
<td>3</td>
</tr>
<tr>
<td>(i \text{ MOD } j)</td>
<td>-2 (nonstandard)</td>
</tr>
<tr>
<td>(i/j)</td>
<td>3.6666666667</td>
</tr>
</tbody>
</table>

Note that the (nonstandard) condition

\[i = (i \text{ DIV } j)*j + (i \text{ MOD } j)\]

is satisfied in each of the above situations.

19.3.3 Range and a few words

Whole number or integer range varies from -32,788 to +32767. Now a new problem may arise regarding this range limitation of integers. Let us see this in an example.
**Example**: Suppose you are adding two numbers i and j and the values of them are as follows:

\[
i = 0100000000000000 \\
j = 0100000000000001
\]

The summation (i+j) will have a result

\[
i + j = 1000000000000001
\]

As we know an integer number is a 16-bit quantity so the left most bit will represent the *sign bit*. So the expected result of the summation would be 32,769 (=1000000000000001). But because of the *sign bit* it will be considered as -41! So be careful about your calculation, while the resultant is crossing the range of the integer quantity.

### 19.4 Real Type Data

Real-type data refers to data items that represent real numerical quantities. This includes real-type constants, variables, functions and expressions.

#### 19.4.1 Uses of operator on real type data

There are four arithmetic operators that can be used with real-type operands. They are:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
<th>Type of operands</th>
<th>Type of resultant</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Real</td>
<td>Real</td>
</tr>
</tbody>
</table>

**Example**:

Suppose that p and q are real-type variables whose assigned values are 12.5 and 0.5, respectively. Several simple real expressions, and their resulting values are shown below.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+q</td>
<td>13.0</td>
</tr>
<tr>
<td>p-q</td>
<td>12.0</td>
</tr>
<tr>
<td>p*q</td>
<td>6.25</td>
</tr>
<tr>
<td>p/q</td>
<td>25.0</td>
</tr>
</tbody>
</table>

#### 19.4.2 Common rules

Three of the most common rules, which apply to expressions containing only two operands, are summarized below:

For operators except '+' and '-'.
Data Types

1. The resultant will be positive if both operands are of the same sign. Otherwise the resultant will be negative.
2. The division operator requires that the second operand be non-zero.
3. If one operand is of type integer and the other is of type real, then the resultant will always be of type real.

**Example** : Let \( c \) and \( d \) be real variables whose assigned values are -0.66 and 4.50. Several real expressions and their corresponding values are given below :

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c+d )</td>
<td>3.84</td>
</tr>
<tr>
<td>( c-d )</td>
<td>5.16</td>
</tr>
<tr>
<td>( c*d )</td>
<td>-2.97</td>
</tr>
<tr>
<td>( c/d )</td>
<td>-0.14666667</td>
</tr>
</tbody>
</table>

19.4.3 Range of real numbers

The range of real numbers on most computers is much larger than the range of integers. For most purposes, you can pretend that real numbers have no limit, although in Turbo Pascal, the actual range is approximately plus or minus \( 10^{38} \) to \( 10^{38} \), equivalent to a 1 followed or preceded by 38 zeros.

19.4.4 Scientific notation

Real numbers are normally displayed in **scientific notation**, a convenient way for expressing large and small values. Dropping trailing zeros for clarity, the value 3.775E+02 is computer shorthand for the more common mathematical notation, \( 3.755 \times 10^2 \). To find the equivalent value just move the decimal point right for positive exponents, left for negative exponents. Thus, 5.123E+03 is equivalent to 5,123.00. 6.5E-03 is the same as 0.0065.

19.4.5 Mixing real and integer

Although the two data types real and integer are different, you may freely mix values of both types in expressions. While this breaks the strict type checking rule of Pascal, it would be awkward if expressions such as \( (2*3.1415) \) were not allowed. However, when you mix real numbers and integers in an expression, the result is always real and not integer.
19.5 Exercise

19.5.1 Multiple choice questions

1. Which one of the following expression is correct?
   a) \( x = x \div y + x \mod y \)
   b) \( x = (x \div y) \cdot y + (x \mod y) \)
   c) \( x \div (y \cdot y) + x \mod y \)
   d) \( x = x \div y \cdot (y + x) \mod y \).

2. In using DIV and '/' operator
   a) both operand must be positive
   b) at least one operand must be positive
   c) second operand is non zero
   d) first operand is non zero.

3. According to ISO standard second operand of MOD operation is:
   a) arbitrary number
   b) non zero
   c) negative
   d) non negative.

4. If you mix real with integers in arithmetic expression the result must be
   a) integer
   b) unknown
   c) logical
   d) real.

5. The range of real numbers in Turbo Pascal is
   a) \( 10^{-64} \) to \( 10^{+64} \)
   b) \( 10^{-32} \) to \( 10^{+32} \)
   c) \( 10^{-38} \) to \( 10^{+38} \)
   d) \( 10^{-16} \) to \( 10^{+16} \).

6. 0.0123 is displayed as
   a) 12.3E-03
   b) 1.23E-03
   c) 12.3E-04
   d) 0.123E-08.
Data Types

19.5.2 Questions for short answers

1. What does integer-type data represent?
2. What are arithmetic operators? Which arithmetic operators can be used with integer-type data?
3. How does the division operator (/) differ from the truncated division operator (DIV) when using integer-type operands?
4. What does real-type data represent?
5. What type of resultant is obtained if an arithmetic operator is used with one integer type and one real-type operand?

19.5.3 Analytical questions

1. Summarize the rules for using an arithmetic operator with two integer-type operands. In particular, summarize the restrictions that apply to the use of the two division operators (/ and DIV) and the MOD operator.
2. Summarize the rules for using an arithmetic operator with two real-type operands?
Lesson 20 : Simple Data Type-II

20.1 Learning Objectives

On completion of this lesson you will learn:

- what the Char-type data is
- what the ASCII character set is
- use of Boolean-type data.

20.2 Char-Type Data

Char-type data are single-character strings i.e., single characters enclosed in apostrophes. This data type includes single-character constants, identifiers that represent single-character constants, char-type variables and certain char-type functions. The complete set of characters that can be used with char-type data will vary from one compiler to another. In general, however, the letters A to Z (both upper and lowercase), the digits 0 to 9 and the commonly available special characters (+,-,=,$, etc.) are acceptable.

Example: The following are acceptable char-type data items.

<table>
<thead>
<tr>
<th>Example</th>
<th>Shows that the letter 'A' is encoded as(decimal) 65 in the ASCII code, and the letter 'B' is encoded as 66. Since 65 is less than 66,</th>
</tr>
</thead>
<tbody>
<tr>
<td>'P'</td>
<td>'5'</td>
</tr>
<tr>
<td>'t'</td>
<td>'$'</td>
</tr>
<tr>
<td>'z'</td>
<td>')'</td>
</tr>
</tbody>
</table>
Data Types

'A' is considered to precede 'B'. Similarly, 'A' precedes a because 65 is less than 97; and '0' precedes '1' because 48 is less than 49.

Table 20.1: ASCII character set

<table>
<thead>
<tr>
<th>ASCII VALUE</th>
<th>Character</th>
<th>ASCII VALUE</th>
<th>Character</th>
<th>ASCII VALUE</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>032</td>
<td>Blank</td>
<td>063</td>
<td>?</td>
<td>093</td>
<td></td>
</tr>
<tr>
<td>033</td>
<td>!</td>
<td>064</td>
<td>@</td>
<td>094</td>
<td>^</td>
</tr>
<tr>
<td>034</td>
<td>“</td>
<td>065</td>
<td>A</td>
<td>095</td>
<td>_</td>
</tr>
<tr>
<td>035</td>
<td>#</td>
<td>066</td>
<td>B</td>
<td>096</td>
<td>`</td>
</tr>
<tr>
<td>036</td>
<td>$</td>
<td>067</td>
<td>C</td>
<td>097</td>
<td>a</td>
</tr>
<tr>
<td>037</td>
<td>%</td>
<td>068</td>
<td>D</td>
<td>098</td>
<td>b</td>
</tr>
<tr>
<td>038</td>
<td>&amp;</td>
<td>069</td>
<td>E</td>
<td>099</td>
<td>c</td>
</tr>
<tr>
<td>039</td>
<td>'</td>
<td>070</td>
<td>F</td>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>040</td>
<td>(</td>
<td>071</td>
<td>G</td>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>041</td>
<td>)</td>
<td>072</td>
<td>H</td>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>042</td>
<td>*</td>
<td>073</td>
<td>I</td>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>043</td>
<td>+</td>
<td>074</td>
<td>J</td>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>044</td>
<td>,</td>
<td>075</td>
<td>K</td>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>045</td>
<td>–</td>
<td>076</td>
<td>L</td>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>046</td>
<td>.</td>
<td>077</td>
<td>M</td>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>047</td>
<td>/</td>
<td>078</td>
<td>N</td>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>048</td>
<td>0</td>
<td>079</td>
<td>O</td>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>049</td>
<td>1</td>
<td>080</td>
<td>P</td>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>050</td>
<td>2</td>
<td>081</td>
<td>Q</td>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>051</td>
<td>3</td>
<td>082</td>
<td>R</td>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>052</td>
<td>4</td>
<td>083</td>
<td>S</td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>053</td>
<td>5</td>
<td>084</td>
<td>T</td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>054</td>
<td>6</td>
<td>085</td>
<td>U</td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>055</td>
<td>7</td>
<td>086</td>
<td>V</td>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>056</td>
<td>8</td>
<td>087</td>
<td>W</td>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>057</td>
<td>9</td>
<td>088</td>
<td>X</td>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>058</td>
<td>:</td>
<td>089</td>
<td>Y</td>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>059</td>
<td>;</td>
<td>090</td>
<td>Z</td>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>060</td>
<td>&lt;</td>
<td>091</td>
<td></td>
<td></td>
<td>121</td>
</tr>
<tr>
<td>061</td>
<td>=</td>
<td>092</td>
<td>\</td>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>062</td>
<td>&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Point to remember: One thing to remember is that, none of the arithmetic operators can be used with char-type data, since char-type data items do not represent numerical quantities. Char-type data can be compared, however, using the relational operators described in next subsections.

20.4 Boolean Type Data
Booleans are truth values that are either true or false. This category includes Boolean-type constants, variables, functions, expressions. The two values that apply the Boolean-type data (true or false) represent an ordered set, with false preceding true. (Note that false is encoded as 0, and true is encoded as 1.)

20.5 Boolean Type Expressions
Boolean expressions are formed by combining operands of the same type with relational operators. These operators represent various conditions of equality and/or inequality. There are seven relational operators in Pascal, though we will consider only six of them in this lesson. These six are:

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

The seventh relational operator, IN, will be discussed later on.

These six operators can be used with operands of any type other than Boolean. When used with non-numerical operands, the inequalities refer to the order in which the operands are encoded. Some of these operators (namely =, <>, <=, >=) are also used in conjunction with set comparison.

Example
Here are some simple Boolean expressions involving numerical operands (within each expression, assume that both operands are of the same type.)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 = 3</td>
<td>false</td>
</tr>
<tr>
<td>2 &lt; 3</td>
<td>true</td>
</tr>
<tr>
<td>0.6 &gt;= 1.5</td>
<td>false</td>
</tr>
<tr>
<td>0.6 &gt;= 0.5</td>
<td>true</td>
</tr>
<tr>
<td>-4 &lt;&gt; 4</td>
<td>true</td>
</tr>
<tr>
<td>1.7 &lt;= -2.2</td>
<td>false</td>
</tr>
</tbody>
</table>
Data Types

Example: Suppose that $i$ and $j$ are integer-type variables that have been assigned the values of 3 and -5, respectively. Several Boolean expressions involving the use of these variables are shown below:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i \leq 10$</td>
<td>true</td>
</tr>
<tr>
<td>$i+j &gt; 0$</td>
<td>false</td>
</tr>
<tr>
<td>$(i-j) &lt; (i+j)$</td>
<td>false</td>
</tr>
<tr>
<td>$i-3 = j+5$</td>
<td>true</td>
</tr>
<tr>
<td>$2 \cdot i \geq i \div 2$</td>
<td>true</td>
</tr>
<tr>
<td>$(i \div 2) &gt; (j+6)$</td>
<td>false</td>
</tr>
</tbody>
</table>

Notice that the operands can be constants, variables or expressions.

Example: Suppose that $ch1$ and $ch2$ are char-type variables that have been assigned the characters 'P' and 'T', respectively. Several Boolean expressions involving the use of these variables are shown below.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ch1 = ch2$</td>
<td>false</td>
</tr>
<tr>
<td>$ch2 = 'T'$</td>
<td>true</td>
</tr>
<tr>
<td>$ch1 = 'p'$</td>
<td>false</td>
</tr>
<tr>
<td>$ch1 &lt; ch2$</td>
<td>true (because P precedes T)</td>
</tr>
<tr>
<td>$ch2 &gt; 'A'$</td>
<td>true (because T succeeds A)</td>
</tr>
<tr>
<td>'W' &lt;&gt; ch1</td>
<td>true</td>
</tr>
</tbody>
</table>

Effect of Boolean expressions on logical operators: Pascal also contains three logical operators. Two of these operators allow Boolean-type operands to be combined to form Boolean-type expressions; the third is used to negate (i.e., to reverse) the value of a Boolean-type operand. The logical operators are:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>Expression will be true if either operand is true (or if both operands are true).</td>
</tr>
<tr>
<td>AND</td>
<td>Expression will be true only if both operands are true.</td>
</tr>
<tr>
<td>NOT</td>
<td>Negate a Boolean operand.</td>
</tr>
</tbody>
</table>

Point to remember: To avoid ambiguities in the order in which the logical operations are carried out, the Boolean expressions should be enclosed with parentheses.

Example: Suppose that $n$ is an integer-type variable that has been assigned a value of 10, and $s$ is a char-type variable that represents the character 'A'. Several Boolean expressions involving the use of these variables are shown below:
<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n &gt; 0) AND (n &lt; 20)</td>
<td>true</td>
</tr>
<tr>
<td>(n &gt; 0) AND (n &lt; 5)</td>
<td>false</td>
</tr>
<tr>
<td>(n &gt; 0) OR (n &lt; 5)</td>
<td>true</td>
</tr>
<tr>
<td>(n &lt; 0) OR (n &gt; 20)</td>
<td>false</td>
</tr>
<tr>
<td>(n = 10) AND (s = 'A')</td>
<td>true</td>
</tr>
<tr>
<td>(n &lt;&gt; 5) OR (s &gt;= 'A')</td>
<td>true</td>
</tr>
</tbody>
</table>

**Example:** Suppose that the Boolean expression j > 6 is true. Then the expression

\[
\text{NOT } (j > 6)
\]

will be false. Also, the expression

\[
\text{NOT } (j \leq 6)
\]

will be true, since \((j \leq 6)\) will be false.

**Example:** Now suppose that the char-type variable ch represents the character 'G'. Then the expression \(ch > 'A'\) will be true, and the expression

\[
\text{NOT } (ch > 'A')
\]

will be false. Also, the expression \(ch = 'A'\) will be false, so that the expression

\[
\text{NOT } (ch = 'A')
\]

will be true.
Data Types

20.6 Exercise

20.6.1 Multiple choice questions

1. Which type of operator can not be used for character type data?
   a) arithmetic
   b) relational
   c) logical
   d) none of the above.

2. To represent Boolean data we need
   a) 1 bit
   b) 2 bits
   c) 8 bits
   d) 16 bits.

3. The total number of relational operator in Pascal is
   a) 5
   b) 2
   c) 8
   d) 7.

4. While programming with logical operators following actions should be taken
   a) operands should be enclosed in parentheses
   b) logical operations are written first
   c) operands should be enclosed in curly braces
   d) none of the above.

20.6.2 Questions for short answers

1. What does char-type data represent? Which characters can be used with char-type data?
2. What is meant by the ASCII character set?
3. How widely this character set is used?
4. In what general order the characters are arranged within the ASCII character set?
5. Describe in detail what the Boolean-type data represents?

20.6.3 Analytical question

1. What is Boolean type data and Boolean expression? Describe with example.
Lesson 21: Simple Data Type- III

21.1 Learning Objectives

On completion of this lesson you will learn:

- utility of constants in a program
- use of some standard constants
- use of some standard functions
- operator precedence and use of assignment statement.

21.2 The Constants

Figure 22.1 shows the block diagram for a constant declaration, which comes between the program declaration and body, but before any variable declaration part. Each constant has a descriptive name and an associated constant value.

```
PROGRAM Constants;
CONST
  Name      = 'Anne';
  Weight    = 118;
  Savings   = 375;
BEGIN
  Writeln('Name : ', Name);
  Writeln('Weight : ', Weight, ' pounds');
  Writeln('Name, has Tk. ', Savings, ' in savings');

The constant declaration part of the Program begins with key word CONST, followed by a list of constants separated by semicolons. Each constant has an identifier, an equality sign, a value and a semicolon. The value itself determines the type of constant. For example, line 3 declares a string constant, Name, while line 4 and 5 declare two integer, or whole number, constants, Weight and Savings.

21.3 Utility of Using Constants
Data Types

There are two main advantages of using *constants* in Pascal

1. First of all, well-chosen constant identifiers make a program more readable and, therefore, easier to understand. Calling 118 for Weight and 375 for Savings makes it harder to make out the meanings of the two value.

2. A second advantage is the concentration of constant values at the top of a program. Although a constant may appear many times in a program, you can easily change its declared value. When you recompile the program, all statements automatically recognize the new constant.

As an example of using constants advantageously, Program in section 22.2 employs the constant *Name* in two `Writeln` statements. If you declare your own name in line 3, then the program displays your name instead of Anne in lines 7 and 9. With constants, you avoid hunting through a program for values that need changing. In a typical commercial program with thousands of lines, that advantage is a necessary.

### 21.4 Standard Constants

Pascal includes three standard identifiers that represent constants. They are *maxint*, *false* and *true*. The first of these, *maxint*, specifies the largest value that may be an integer-type quantity. The two remaining constants, *false* and *true*, represent the two values that may be assigned to a Boolean-type data item. (You are again reminded that *false* and *true* represent an ordered set, with *false* preceding *true*.)

### 21.5 Standard Functions

Pascal also contains a number of standard functions that are used with various simple data types. (Standard functions are also referred to as *intrinsic* or *built-in* functions.) Some of these functions accept one type of parameter and return a value of the same type, while others accept parameters of one type and return a value of a different type. Information on several of these functions are summarized below.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Type of Parameter(x)</th>
<th>Type of Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>Compute the absolute</td>
<td>Integer or real</td>
<td>Same as x</td>
</tr>
</tbody>
</table>
The purpose of most of the functions presented above should be readily apparent. There are a few, however, whose purpose may be less obvious. Their use is illustrated in the following examples:

**Example:** The function abs(x) computes the absolute value of the number represented by the parameter x. So:

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(0.003)</td>
<td>0.003</td>
</tr>
<tr>
<td>abs(-0.003)</td>
<td>0.003</td>
</tr>
</tbody>
</table>

Thus if `diff` is an integer type variable whose value is -5 then
Data Types

<table>
<thead>
<tr>
<th>Function</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(diff)</td>
<td>5</td>
</tr>
</tbody>
</table>

**Example**: The functions `chr` and `ord` are used to determine the relationship between any Pascal character and its corresponding integer code. Thus if computer uses ASCII code then:

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>chr(65)</td>
<td>'A'</td>
</tr>
<tr>
<td>chr(112)</td>
<td>'p'</td>
</tr>
<tr>
<td>chr(53)</td>
<td>'5'</td>
</tr>
<tr>
<td>ord('A')</td>
<td>65</td>
</tr>
<tr>
<td>ord('p')</td>
<td>112</td>
</tr>
<tr>
<td>ord('5')</td>
<td>53</td>
</tr>
</tbody>
</table>

and so on. Note that

\[
\text{ord('A')} = \text{ord(chr(65))} = 65
\]

and

\[
\text{chr(65)} = \text{chr(ord('A'))} = 'A'
\]

and so on.

**Example**: Since integer, char and Boolean-type data all represent ordered sets, we can determine the predecessor or successor of any data item within one of these sets (or in any user-defined ordered set) with the functions `pred` and `succ`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>pred(3)</td>
<td>2</td>
<td>integer</td>
</tr>
<tr>
<td>pred('e')</td>
<td>'d'</td>
<td>char</td>
</tr>
<tr>
<td>pred(true)</td>
<td>false</td>
<td>boolean</td>
</tr>
<tr>
<td>succ(3)</td>
<td>4</td>
<td>integer</td>
</tr>
<tr>
<td>succ('e')</td>
<td>'f'</td>
<td>char</td>
</tr>
<tr>
<td>succ(true)</td>
<td>true</td>
<td>boolean</td>
</tr>
</tbody>
</table>

**Example**: The functions `round` and `trunc` can accept both positive and negative real-type numbers. Negative numbers are treated as though they were positive numbers, with the minus sign added after the rounding or truncation has been carried out. Thus

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>round(2.3)</td>
<td>2</td>
</tr>
</tbody>
</table>
Computer Programming

round(3.7)                             4
round(-1.8)                            -2
round(-6.1)                            -6

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>trunc(2.3)</td>
<td>2</td>
</tr>
<tr>
<td>trunc(3.7)</td>
<td>3</td>
</tr>
<tr>
<td>trunc(-1.8)</td>
<td>-1</td>
</tr>
<tr>
<td>trunc(-6.1)</td>
<td>-6</td>
</tr>
</tbody>
</table>

Point to remember: Finally remember that the parameters in a function reference can be constants, variables, expressions, or even references to other functions. The only restriction is that the parameter be of the proper type.

21.6 Operator Precedence

An expression can sometimes become quite complex, because of the presence of multiple operators within the expression. In such cases it becomes necessary to specify the order in which the various operations are carried out. This order may be determined by the natural operator precedence which is included within Pascal language. The precedence groups are tabulated below, from highest to lowest.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>1(highest)</td>
<td>NOT</td>
</tr>
<tr>
<td>2</td>
<td>* / DIV MOD AND</td>
</tr>
<tr>
<td>3</td>
<td>+ - OR</td>
</tr>
<tr>
<td>4(lowest)</td>
<td>= &lt;&gt; &lt; &lt;= &gt; &gt;= IN</td>
</tr>
</tbody>
</table>

Within a given precedence group the operations are carried out as they are encountered, reading from left to right.

Example: The numerical expression \( a-(b/c)\times\sqrt{d} \) is equivalent to the algebraic formula \( a-(b/c) \times \sqrt{d} \). Thus, if the real variables \( a, b, c, \) and \( d \) have been assigned the values 1, 2, 3, 4 respectively, the expression would represent the value \(-0.333333333\), since

\[
1-(2/3) \times \sqrt{4} = 1-(4/3) = -1/3 = -0.3333333
\]

Notice that the division is carried out first, since this operation falls within a higher precedence group than subtraction. The resulting quotient is then multiplied by \( \sqrt{4} \) (left-to-right rule within a precedence group). Finally, this product is subtracted from the value of the first variable.
21.7 Effect of Parentheses on Operator Precedence

The natural operator precedence can be altered through the use of parentheses, thus allowing the operations within an expression to be carried out in any desired order. In fact, the parentheses can be nested, one pair within another. In such cases the innermost operations are carried out first, then the next innermost operations, and so on.

Example: The numerical expression \((a-b)/(c*\sqrt{d})\) is equivalent to the algebraic formula \((a-b)/(c \times \sqrt{d})\). Thus, if the variables \(a\), \(b\), \(c\) and \(d\) have been assigned the values 1,2,3 and 4 respectively, the expression would represent the value \(-0.1666667\), since

\[
(1-2)/(3\times\sqrt{4}) = -1/6 = -0.1666667.
\]

Example: Consider the Boolean expression \((x > 0) \text{ OR } (y < 10)\) where \(x\) and \(y\) are integer-type variables. This expression will be true if \(x\) has been assigned a value greater than zero or if \(y\) has been assigned a value less than 10 (or both). If neither condition has been satisfied, then the expression will be false. Note that the parentheses are required within this expression. Without the parentheses, Pascal would first attempt to evaluate the expression \((0 \text{ OR } y)\) which is not meaningful (because the logical operator OR cannot be used with numeric operands).

Example: Now consider the numerical expression

\[2*((a \mod 5)*(4+(b-3)/\sqrt{c+2}))\]

If the variables \(a\), \(b\) and \(c\) have been assigned the values 8,15, and -4, respectively, then the given expression would be evaluated as

\[2*((3)*(4+12/\sqrt{-2})) = 2*(3*(4+3)) = 42\]

Point to remember: Sometimes it is a good idea to use parentheses to clarify an expression, even though the parentheses may not be required. On the other hand, the excessive use of parenthesis in complex expressions should be avoided if at all possible, as such expressions are frequent source of error (Example 22.8 contains an expression of this type).

21.8 More About Expression

Finally, the following grammatical rules must always be observed when constructing numerical and Boolean expressions.

1. Undefined identifiers cannot appear within an expression.
2. Preceding an identifier with a minus sign is equivalent to multiplication by -1. Thus -a*b is equivalent to -1*a*b.
3. Arithmetic operators cannot appear consecutively. Hence, the expression a*-b is not allowed, but a*(-b) is permitted.
4. Arithmetic operations cannot be implied. Thus, the expression 2*(x+y) is incorrect, but the expression 2*{x+y} is valid.
5. Arithmetic operations cannot be carried out on char or Boolean-type data. Therefore, expressions such as A + B and (n > 0) + (n < 20) are not allowed.
6. There cannot be an imbalance of parentheses. In other words, the number of left parentheses must be the same as the number of right parentheses.

21.9 Exercise

21.9.1 Multiple choice questions

1. What will be the value of the expression a-b/c*sqrt (d) if integer a, b, c, d have been assigned the values 1,2,3,4 respectively:

   a) -1/6
   b) -2/3
   c) -1/3
   d) none of the above.

2. What will be the value of the following function: round(-4.667)

   a) -5
   b) -4
   c) 5
   d) 4.

3. Indicate the value of the expression: -chr(ord(succ(chr(ord(pred (‘B’))+5)))-6)

   a) A
   b) B
   c) C
   d) D.

21.9.2 Questions for short answers

1. What three standard constants are included in Pascal? What type of data item is each constant? What is the purpose of each constant?
2. What is the purpose of the abs function? With what types of parameters it is used? What type of resultant is obtained?
Data Types

3. What is the purpose of the `chr` and `ord` functions? With what types of parameters are they used? What type of resultant is obtained in each case?
4. What is the purpose of the `pred` and `succ` functions? With what types of parameters are they used? What type of resultant is obtained in each case?
5. When using the ASCII character set, what relationships exist between the `chr` and `ord` functions and `pred` and `succ` functions? Why are these relationships not valid for all character sets?
6. What is the purpose of the `round` and `trunc` functions? With what types of parameters are they used? Give example of each function.
7. How are negative parameters treated by the `round` and `trunc` functions?
8. In what order are the operations within a precedence group carried out?
9. How can the natural operator precedence be altered?
10. In what order are the operations carried out within an expression that contains nested parentheses?

21.9.3 Analytical questions

1. Summarize the rules that must be observed when constructing numerical and Boolean expressions.
2. What are the main advantages of using constant in Pascal?
Lesson 22 : Input - Output

22.1 Learning Objectives

On completion of this lesson you will learn:

- methods for reading data into the computer and writing data out of the computer
- special input/output statements that are included within Pascal
- format of I/O instructions
- writing a variety of complete, though simple, Pascal programs.

22.2 Input and Output Files

Let us first consider the manner in which input data and output data are transferred into and out of the computer. For now we will consider a (non interactive) mainframe computing environment.

The input data items must be placed in a separate file, called the **input file**, before the corresponding program is actually run. This file is most often created via a text editor. However, it may also be entered into the computer via punch cards, using a keypunch machine and a card reader. In either case data items will be sorted sequentially, in the same order that they were entered. These individual data items will be grouped into logical lines, where each line corresponds to one line of typed data or one punch card.

Output data are transferred from the computer’s memory to an **output file**, which is similar but opposite to the input file. Again the data items are stored sequentially, in the same order that they were written. The data items will again be grouped into logical lines, which will correspond to physical lines of output when the data are printed or displayed.

It should be understood that the input data file and the output data file are stored within the computer as separate entities; they are **not** a part of a Pascal program. These files are associated with a Pascal program, however, by naming them as parameters within the program header.

**Example:** A Pascal program contains the following program header.

```
PROGRAM payroll(input , output);
```

The header specifies that the name of the program is payroll and that the program utilizes both input and output files. A program need not utilize both an input and an output file, though most programs do so, since most programs require input data and generate output data when they are executed.
Data Types

Example: A Pascal program called primes is used to generate the first twelve prime numbers. This program includes the following header.

\[ \text{PROGRAM primes(output);} \]

Notice that this program utilizes only an output file, since it does not require any input data. If the program were modified, however, so that it would generate the first \( n \) prime numbers, where \( n \) is an input quantity, then the program header would have to be changed to

\[ \text{PROGRAM primes(input,output);} \]

Such a modification would be a good idea, since the program would then be much more general.

Finally, remember that there are some interactive versions of Pascal that allow input/output operations to be carried out directly during program execution. Input and output files are not required in such situations. This is particularly common with microcomputers.

22.3 The Read Statement

The read statement is used to read data items from the input file and assign them to integer, real or char-type variables. The statement is written as

\[ \text{read (input variables);} \]

where the input variables are separated by commas. (Note that Boolean-type variables cannot be included in the list of input variables.)

Example: Here is a typical read statement.

\[ \text{read (a, b, c);} \]

This statement causes three data items to be read from the input file and assigned to the variables \( a \), \( b \), and \( c \), respectively. The data items are read from the input file and assigned to their respective variables in the same order that they are stored. Each variable must be of the same type as its corresponding data item.

Exception: An integer number can be assigned to a real-type variable. Each data item can be read only once.

Example: A portion of a Pascal program is shown below:
VAR a, b : real;
i, j : integer;
p, q : char;

read (a, b, i, j, p, q);

The read statement will cause two real numbers, two integer numbers and two single characters to be read from the input file and assigned to the variables a, b, i, j, p and q, respectively.

Some care should be given to the spacing of the input data items. Numerical data items must be separated from one another by blank spaces or by end-of-line designations. To be more precise, a numerical data item may be preceded by one or more blank spaces or end-of-line designations. Real numbers can be written with or without an exponent. Moreover, real numbers that represent integral quantities (e.g., 1, 0) can be written as integers (i.e., 1). Any number may be preceded by a plus or a minus sign provided there is no space between the sign and the number.

Char-type data must be treated somewhat differently, since all characters are significant. In particular, a char-type data item must not be enclosed in apostrophes. Also, a char-type data item cannot be separated from the preceding data item by a blank space or an end-of-line designation, since the separator will be interpreted (incorrectly) as the data item. (An end-of-line designation will be interpreted as a blank space if assigned to a char-type variable.

Example: Consider once again the portion of the Pascal program shown in the last example, i.e.,

VAR a, b : real;
i, j : integer;
p, q : char;

read(a, b, i, j, p, q);

Suppose that the variables are to be assigned the following values

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>12500</td>
</tr>
</tbody>
</table>
Data Types

\[
\begin{array}{ll}
  b & -14.8 \\
  i & 5 \\
  j & -9 \\
  p & X \\
  q & Y \\
\end{array}
\]

The data items might be entered into the input file as follows.

\[12500.0 -14.8 5 -9XY\]

Alternatively, the data items might be entered as

\[12500 -14.8 +5 -9XY\]

or as

\[1.25e4 -1.48e1 5 -9XY\]

If one read statement is followed by another, then the second read statement begins where the first read statement ended. More precisely, the second read statement will begin by reading the data item following the last data item that was read by the previous read statement. Thus, a new read statement does not necessarily begin by reading a new line of data.

**Example:** Let us again consider the Pascal program described in previous Examples. Now suppose that the read statement is replaced by the following two statements

```
read ( a, b, i);
read ( j, p, q);
```

The input file need not be changed, since the first read statement would read the first three values on the line, and the second read statement would read the last three values.

### 22.4 The Readln Statement

The readln statement, like the read statement, is used to read data items from the input file and assign them to integer, real or char-type variables. This statement is written in the form

```
readln(input variables);
```

The difference between the two statements is that the readln statement causes the next (not the current) read or readln statement to begin by reading a new line of data, whereas the read statement will allow the next read or readln statement to begin on the same line.
**Example:** A portion of a Pascal program is shown below.

```pascal
VAR  p1, p2, p3, p4 : integer;
    
read (p1, p2);
read (p3, p4);
```

Suppose that the input file contained the following eight numbers, arranged in two lines.

```
  1 2 3 4
  5 6 7 8
```

The read statements would cause the numbers 1, 2, 3 and 4 to be assigned to the variables `p1`, `p2`, `p3` and `p4`, respectively. If the read statements were placed with `readln` statements, i.e.,

```pascal
VAR  p1, p2, p3, p4 : integer;
    
readln(p1, p2);
readln(p3, p4);
```

then the variables `p1` and `p2` would still be assigned the values 1 and 2, but `p3` and `p4` would be assigned the values 5 and 6 (from the second line of input data). Note that it is the first `readln` statement that causes the values from `p3` and `p4` to be read from the second line of input.

If a `readln` statement is preceded by a `read` statement, then `readln` statement will begin where the previous `read` statement ended. Thus, the `readln` statement does not necessarily begin by reading a new line of data. (Note that the `readln` statement behaves the same as the `read` statement in this respect.)

**Example:** Now suppose that the read statements of the previous example are written as

```pascal
read(p1, p2);
readln(p3, p4);
```

The variables `p1`, `p2`, `p3` and `p4` would be assigned the values 1, 2, 3 and 4, respectively, since the `readln` statement would begin by reading the third value in the first line of data. However, the next `read` or `readln` statement would begin by reading the 5 from the second line of the input data. The `readln` statement is convenient for reading data on a line-by-line basis, as is common in batch processing.
Data Types

**Example:** A Pascal program contains the following three readln statements.

```
readln(a, b);
readln(c, d);
readln(e, f);
```

Suppose that the variables are all of type integer, and the input file contains the numbers

```
1 2 3
4 5 6
7 8 9
```

Then the values 4 and 5 will be assigned to the variables c and d; the values 7 and 8 will be assigned to e and f. It is not clear what values will be assigned to a and b, since it is not known where the first readln statement will begin. If the statement begins at the start of the line, then the values 1 and 2 will be assigned to a and b. On the other hand, if this statement is preceded by a read statement, e.g.,

```
read(x);
```

then x will be assigned the value 1, and a and b will be assigned the values 2 and 3 respectively.

22.5 The Write Statement

The write statement is used to write data items to the output file. This statement is written as

```
write (output data items);
```

The output data items can be strings, numerical constants or the values of variables or expressions. They may be of type integer, real, char-type or Boolean-type. (Each string must be enclosed within apostrophes). The data items must be separated by commas if there are more than one.

**Example**: Here is a typical write statement.

```
write ('x =', x);
```

This statement causes the value of the numeric-type variables x to be written to the output file, with a corresponding label. Thus, if x represents the value 123.456, then the following output will be written to the output file.

```
x = 1.2345600E+02
```

Similarly, the write statement

```
write('sum =', a + b);
```
causes the value of the numerical expression a+b to be written to the output file, with the corresponding label. Thus, if a and b represent the values 3 and -1, respectively, the write statement will produce the following output.

\[ \text{sum} = 2 \]

### 22.6 Field Width and a Few Words

Real numbers can be displayed in a number of different ways, though the default (standard) format is scientific notation, as illustrated in the above example. Boolean data items will be represented by the standard identifiers true or false, depending on their values.

Most versions of Pascal utilize a standard **field width** (i.e., number of spaces) to represent integer, real or Boolean data items. This field width will be different for each data type. Moreover, the field width that is used for a given type of data will vary from one version of Pascal to another. Typically, integer data might have a standard field width of 8, real data might have a standard field width of 14, with 7 digits to the right of the decimal point; and the Boolean data might have a standard field width of 6. If the field is wider than necessary (as is often the case), then the data item is placed within the right portion of the field. This results in one or more leading blank spaces, as illustrated in the previous example.

The standard field widths can easily be altered, thus allowing the programmer greater control over the appearance of the output items.

When string and char-type data items are written out the field width will be exactly equal to the number of characters in the data item. Hence there will be no leading or trailing blank spaces, unless they are included as a part of the actual data.

**Example:** A Pascal program contains the following write statement.

```
write ('RED', 'WHITE', 'BLUE');
```

This statement will generate the following line of output.

```
RED WHITE BLUE
```

Notice that the blank spaces separating the individual words are included within the last two strings. If the write statement had not included these blank i.e., if we had written

```
write('RED', 'WHITE', 'BLUE');
```

\[ \text{sum} = 2 \]
then the resulting output data would appear as an unbroken sequence of letters, i.e.,

**REDWHITEBLUE**

If one write statement is followed by another, then the second write statement will begin where the first one is ended. In other words, the first data item written by the second write statement will begin on the current line, immediately after the last data item that was written by the previous write statement. Therefore, a new write statement does not necessarily generate a new line of output data.

*Example:* A portion of a Pascal program is shown below.

```pascal
VAR a, b : real;
i, j    :    integer;
p, q  :    char;
.
.
write ('a =', a, 'b =', b, 'i =', i);
write ('j =', j, 'p =', p, 'q =', q);
```

Suppose that the variables have been assigned the following values.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>12500.0</td>
</tr>
<tr>
<td>b</td>
<td>-14.8</td>
</tr>
<tr>
<td>i</td>
<td>5</td>
</tr>
<tr>
<td>j</td>
<td>-9</td>
</tr>
<tr>
<td>p</td>
<td>X</td>
</tr>
<tr>
<td>q</td>
<td>Y</td>
</tr>
</tbody>
</table>

Then the resulting output data would appear on one line, as

```
a = 1.250000000E+04  b = -1.4800000E+01  i = 5  j = -9  p = X  q = Y
```

### 22.7 The Writeln Statement

The writeln statement is identical to the write statement, except that the writeln statement results in an end-of-line designation being written after the last data item. Therefore, any subsequent write or writeln statement will begin a new line of output. The statement is written as

```
writeln (output data items);
```

where the output data items can be strings, numerical constants or the values of variables or expressions of type integer, real, char or Boolean.
Example: Suppose that the write statements in the last example are replaced with writeln statements i.e.,

```pascal
VAR a, b : real;
i, j : integer;
p, q : char;
...
writeln (a =', a, 'b =', b, 'i =', i);
writeln (j =', j, 'p =', p, 'q =', q);
```

Then the following two lines of output would be generated.

```
a = 1.2500000E + 04  b = -1.480000E+01  i = 5
j = 5  p = X  q = Y
```

Notice that it is the first writeln statement that causes the original single line of output to be broken up into two lines. Then second writeln statement has no effect on this output, though any subsequent output would begin on a new line. Therefore, the statements

```pascal
writeln (a =', a, 'b =', b, 'i =', i);
write ('j =', j, 'p =', p, 'q =', q);
```

would produce the same two lines of output shown above, but the statements

```pascal
write (a =', a, 'b =', b, 'i =', i);
writeln (j =', j, 'p =', p, 'q =', q);
```

would generate the single line output shown in previous example. An empty writeln statement can be used to generate a blank line, as illustrated in the next example.

Example: Consider the following three writeln statements.

```pascal
writeln ('line one');
writeln;
writeln ('line two');
```

These statements will generate three lines of output (including one blank line), as shown below.

```
line one
line two
```

If the empty writeln statement were not present the blank line would not appear, and the two printed lines would be spaced closer together, i.e.,
**Data Types**

line one  
line two

### 22.8 Exercise

**22.8.1 Multiple choice questions**

1. Read statement is
   a) an arithmetic statement  
b) a procedure  
c) an I/O statement  
d) both b and c.

2. Which cannot be argument of read statement?
   a) Integer  
b) Boolean  
c) Real  
d) None of above.

3. The input variable of read statement are separated by
   a) tab  
b) blank space  
c) commas  
d) line feed.

4. Each string in write / written statement must be enclosed within
   a) apostrophes  
b) commas  
c) blank space  
d) none of the above.

5. The Boolean data might have a standard field width of
   a) 6  
b) 8  
c) 7  
d) 14.

### 22.8.2 Questions for short answers

1. What is an input and output file? How are data items entered into the input file?
2. How are the data items arranged within an input or an output file?
3. In what sense must the data items in the input file correspond to the variables in the read statement?
4. Summarize the rules for spacing numerical and char-type data items within the input file.
5. Can Boolean-type data be processed with a read statement?
6. How are data items read from the input file when one read statement follows another?
7. What happens when a readln statement is preceded by another read statement? What happens when a readln statement is preceded by another readln statement?
8. What is the purpose of the write statement? What types of output items can be included in this statement?
9. Can strings be included within a write statement? What types of output items can be included in this statement?
10. Can Boolean-type data be processed with the write statement?
11. What is meant by a field width? What are the default field widths at your particular installation?
12. How are data items written to the output file when one write statement follows another?
13. What is the purpose of the writeln statement?

22.8.3 Analytical questions

1. What are the purposes of read and readln statement? How does readln statement differ from the read statement? Describe with suitable examples.
2. a) How does writeln statement differ from write statement?
   b) What happens when writeln statement is preceded by a write statement or writeln statement? Illustrate with example.
   c) Why might an empty writeln statement be included in a Pascal program? Describe with example.
Unit 7 : Control Structures

Introduction

This unit gives you knowledge about the structure of Pascal program. These structural tools are basic structures of Pascal. Some of the structures are very essential and the others are useful for obtaining more efficiency. In first two lessons of this unit conditional structures IF and CASE have been explained. This structures are useful while controlling the flow of execution of programming depending on different situations. Next two lessons describe the repetitive structures. This type of structure is used when the same operation is allowed to do upon the same type of data. Three structure have been explained FOR, WHILE - DO and REPEAT - UNTIL. Specially in database applications this type of statements are often used. With all these statements a new but easy concept is introduced.

Lesson 23: Control Structures-I

23.1 Learning Objectives

On completion of this lesson you will learn:

- basic concept of simple and complex Boolean expression
- review of simple and compound statements and their use
- introductory idea of conditional statement.

23.2 The IF Statement

So far, the execution of statements in our programs has followed the statements’ physical order. First the statement at the top of the program is executed, then the next statement, then the statement after that, and so on, until the statement at the bottom of the page is executed.

Statement 1
Statement 2
Statement 3
Statement 4

But if we want the computer to execute either one statement or another statement depending on what has happened before. The IF statement is a statement that allows us to execute statements in a logical order that differs from their physical order. This statement allows...
us to ask a question and then do one thing if the answer is yes (true) and another if the answer is no (false).

23.2 Conditions and Boolean Expressions

To ask a question in Pascal, you make a statement. If the statement you make is true, the answer to the question is yes. If the statement is not true, the answer to the question is no. For example, if you wanted to ask “Are you having beef for dinner tonight?” You would instead say “we are having beef for dinner tonight.” If that statement is true, the answer to the question is yes. So, asking question in Pascal is a statement that may be either true or false.

23.2.1 Boolean Expressions

Boolean expressions are formed by combining operands of the same type (any type other than Boolean) with one of the six relational operators:

- `=` equal to
- `<>` not equal to
- `<` less than
- `<=` less than or equal to
- `>` greater than
- `>=` greater than or equal to.

**Example**: Several Boolean expressions are shown below.

```
count <= 100
sqrt(a+b+c) > 0.005
answer = 0
balance >= cutoff
ch1 < 'T'
```

The first four expressions involve numerical operands. Their meaning should be readily apparent. In the last expression, ch1 is assumed to be a char-type variable.
23.2.2 Logical operator

Relational operators compare things such as numbers and letters. Logical (Boolean) operators are the special symbols AND, OR and NOT, which are defined only for Boolean expressions. The first two (AND & OR) are used to combine Boolean operands to form logical expressions; the third (NOT) is a prefix that is used to negate a Boolean operand.

Example: Here are some Boolean expressions that illustrate the use of the logical operators.

(count <= 100) AND (ch1 <> 'x')
(balance < 1000.0) OR (status = 'R')
(answer < 0) OR ((answer > 0.5) AND (answer < 10.0))
(pay >= 1000.0) AND (NOT single)

Note that ch1 and status are assumed to be char type variables in these examples, and single is assumed to be Boolean. The remaining variables are assumed to be numeric (either integer or real). Also notice that the Boolean operands are enclosed in parentheses to avoid any ambiguities in the order in which the operations are carried out.

23.3 Types of Statement in Pascal

There are two basic types of statement in Pascal

1. Simple statement
2. Structured statement.

23.3.1 Simple Statement

Simple statements are essentially single, unconditional instructions that perform one of the following tasks.

1. Assign a data item to a variable (assignment statement).
2. Access a self-contained computational module, called a procedure.
3. Transfer program control unconditionally to another part of the program (the GOTO statement).

Example:

A typical assignment statement: tax := 0.14*gross;
A typical procedure call: move_ball(x,y,inc);
GOTO statement: GOTO 100;

23.3.2 Structured Statements

Pascal recognizes several different types of structured statement. These include:

Compound statements, which consist of a sequence of two or more consecutive statements
Computer Programming

- Compound
- Repetitive
- Conditional.

Compound statements, which consist of a sequence of two or more consecutive statements. Repetitive statements, which involve the repeated execution of several simple statements. Conditional statements, in which one or more simple statements are executed only if a specified logical condition is satisfied.

Example: A typical compound statement is given below

BEGIN
    read(radius);
    area := 3.1415*sqr(radius);
    write(radius,area);
END

Note that the simple statements that comprise the compound statement are enclosed within the keywords BEGIN and END. Also, notice that the simple statements are separated from one another by semicolons.

Example:

Here is a typical repetitive statement.

    FOR count:=1 TO 100 DO write (count);

This statement will be executed 100 times. Here is a more complex compound statement.

BEGIN
    sum :=0;
    FOR count:=1 TO n DO
        BEGIN
            read (x);
            sum := sum +x
        END;
    write('sum =', sum);
END

Example: A typical conditional statement is shown below:

    IF pay <1000.00 THEN
        write ('group1')
    ELSE write ('group2');
23.4 Exercise

23.4.1 Multiple choice questions

1. The value of the following expression (sqrt(5) > 2) AND (round(1.2)= 2)
   a) True
   b) False
   c) 5
   d) 2.

2. The simple statements are separated from one another by
   a) commas
   b) colons
   c) semicolons
   d) none of the above.

3. Compound statement is enclosed within the
   a) two second brackets
   b) two 1st brackets
   c) keyword BEGIN and END
   d) keyword IF and THEN.

23.4.2 Questions for short answers

1. What is Boolean expression?
2. What are the six relational operators used in Pascal?
3. How many types of statements are there in Pascal? Briefly describe them.
4. Give an example of a repetitive statement using FOR statement.
5. What is the purpose of GOTO statement?

23.4.3 Analytical question

1. Explain the use of compound statements with example.
Lesson 24: Control Structures-II

24.1 Learning Objectives

On completion of this lesson you will learn:

- conditional structure IF
- conditional structure IF - THEN - ELSE
- conditional jumping by CASE statement and its use
- use of nesting these structures.

24.2 The IF Structure

The IF structure is a conditional control structure that allows some action to be taken only if a given logical condition has a specified value (either true or false). This structure has two different forms. The simpler form is

\[
\text{IF Boolean expression THEN statement}
\]

The statement part of the structure will be executed if and only if the Boolean expression is true. If the Boolean expression is false, then the statement part of the structure will be ignored. The statement itself can be either simple or structured, though it is often a compound statement.

Example: One example of the IF-THEN structure is shown below.

```plaintext
IF count <= 100 THEN count:= count +1;
IF tag = '*' THEN
    BEGIN writeln(accountno); credit := 0 END;
IF test THEN BEGIN  x := 100; test := false END;
IF (balance < 100.0) OR (status = 'R') THEN
    writeln(balance);
```

In the first example, the integer variable count will be increased by 1 if its current value does not exceed 100. The second example causes the value of `accountno` to be written out, and a value of 0 to be assigned to `credit` if an `*` has been assigned to the char type variable `tag`. Notice that this example includes a compound statement.

The third example includes both a Boolean variable (test) and a compound statement. If test is originally true, then a value of 100 is assigned to `x` and `test` is set to false. In the last example, the value of `balance` is written out if its value is less than 1000.0 or if the char-type variable `status` represent the character `R` (or if both the conditions are true).

The second form of the IF structure is
Control Structures

**IF Boolean expression THEN statement 1 ELSE statement 2**

This is frequently referred to as the IF-THEN-ELSE structure. In this case, statement1 will be executed if the Boolean expression is true; otherwise, statement2 will be executed. Notice that exactly one of the statements is always executed (but never both). Again the individual statements may be either simple or structured and are often compound statements.

Semicolons should not appear in an IF-THEN-ELSE structure, except as separators within a compound statement. Beginners sometimes make the mistake of placing a semicolon before the keyword ELSE. This should be avoided, as it results in a compilation error.

**Example:** Here are several examples illustrating the use of the IF-THEN-ELSE structure.

```plaintext
IF status = 'S' THEN tax : = 0.20*pay ELSE tax : = 0.25*pay;
IF tag = '*' THEN BEGIN writeln(accountno); credit := 0; END
   ELSE credit := 100;
IF circle THEN BEGIN
   readln(radius);
   area := 3.1415* sqr(radius);
   writeln('Area of Circle : ', area)
END;
ELSE BEGIN
   readln(length, width);
   area := length*width;
   writeln('Area of rectangle : ', area)
END;
```

In the first example the value of tax is determined in one of the two possible ways, depending on the value that is assigned to the char-type variable status. The second example looks for certain ‘tagged’ accounts. If an account is tagged (i.e. if an '*' is assigned to the char-type variable tag), then the account number is written out and the credit limit is set to zero; otherwise the credit limit is set at 1000.

The third example shows an area can be calculated for either of the two different geometrical figures. If the Boolean type variable circle is true, then the radius of the circle is read into the computer, the area is calculated and then written out. If circle is false, then the length and width of a rectangle are read into the computer, the area is calculated and then written out.

Notice once again that there are no semicolons in the IF-THEN-ELSE structure, except as separators in the compound statements. IF structure can be nested within one another. Some of the forms that nested IF
structures can take on are shown below. The most general form of two-layer nesting is

\[
\text{IF } \text{be1 THEN IF } \text{be2 THEN s1 ELSE s2 ELSE IF } \text{be3 THEN s3 ELSE s4}
\]

where be1, be2, be3 represent Boolean expressions, and s1, s2, s3 and s4 represent statements. In this situation, one complete IF-THEN-ELSE structure will be executed if be1 is true and another will be executed if be1 is false. It is of course possible that s1, s2, s3 and s4 will contain other IF-THEN-ELSE structures. We would then have multilayer nesting.

Some other forms of two-layer nesting are

\[
\begin{align*}
\text{IF be1 THEN s1} \\
\text{ELSE IF be2 THEN s2 ELSE s3} \\
\text{IF be1 THEN IF be2 THEN s1 ELSE s2 ELSE s3} \\
\text{IF be1 THEN IF be2 THEN s1 ELSE s2}
\end{align*}
\]

In each of the first two cases, the subordinate nature of the inner IF-THEN-ELSE structure is indicated by the line on which it is written. In the last case, however, it is not clear which Boolean expression is associated with the ELSE clause. The answer is be2 because this is the associated with closest IF structure. Thus, this last example is equivalent to

\[
\text{IF be1 THEN BEGIN IF be2 THEN s1 ELSE s2 END}
\]

If we want to associate the ELSE clause with be1 rather than be2, we could write

\[
\text{IF be1 THEN BEGIN IF be2 THEN s1 END ELSE s2}
\]

This type of nesting must therefore be carried out carefully in order to avoid possible ambiguities.

\[\textbf{24.3 The CASE Structure}\]
Control Structures

The CASE structure is a conditional control structure that allows some particular group of statements to be chosen from several available groups. The selection will be based upon the current value of an expression, referred to as the selector. The general form of the CASE structure is

```
CASE expression OF
  case label list 1 : statement 1;
  case label list 2 : statement 2;
  .
  .
  case label list n : statement n;
END
```

The expression can be any simple-type expression other than real. It often takes the form of a single simple-type variable.

Each of the case labels represents one of the permissible values of the expression. Thus, if the expression is of type integer, the case labels would represent integer values that fall within the permissible range. The case labels need not appear in any particular order, though each of the case labels must be unique. Moreover, each label can appear in only one list. (Note: These labels are referred to as case labels to distinguish them from a different type of label, which will discuss in the next section.)

The statements can be either simple or structured. The use of compound statement is quite common. Null (empty) statements are also permitted, to indicate that no action is to be taken for certain values of the selector. The statements need not be unique (that is, the same statement may be used with two or more lists of case labels).

A statement will be executed if (and only if) one of its corresponding case labels matches the current value of the expression. Thus, the current value of the expression determines which of the statements will be executed. If the current value of the expression does not match any of the labels, then the action will be undefined. (Some versions of Pascal include an otherwise clause which specifies what action is to be taken if the value of the selector does not match any of the labels).

**Example**: A typical CASE structure is illustrated below. In this example, choice is assumed to be a char-type variable

```
CASE choice OF
  'R' : writeln('RED');
```
Computer Programming

'W' : writeln('WHITE');
'B' : writeln('BLUE');
END;

Thus, RED will be written out if choice is assigned the value R, WHITE will appear if W is assigned to choice and BLUE will be written out if B is assigned to choice. No output will be generated if choice is assigned some character other than R, W or B.

Example: Here is another typical CASE structure.

CASE trunc(x/10) OF
  1     : y:=y+5;
  3,5   : y:=y-2;
  6     : y:=2*(y+1);
  2,4   : ;
  9     : y:=0;
END;

In this example, x and y are real-type variables. The value of x/10 is truncated and hence converted to an integer, which will be used as the selector.

In a practical sense, the CASE structure may be thought of as an alternative to the use of nested IF-THEN-ELSE structures. However, it can only replace those IF-THEN-ELSE structures that test for equalities. In such situations, the use of the CASE structure is usually convenient.
24.4 Exercise

24.4.1 Multiple choice questions

1. In IF - THEN - ELSE structure
   a) semicolon is optional before ELSE
   b) semicolon is essential before ELSE
   c) semicolon must be used before ELSE for compound statements but not for simple statements
   d) none of the above

2. Nesting level of IF - THEN - ELSE structure is
   a) unlimited
   b) limited up to maximum allowed code size
   c) limited up to hard disk memory
   d) none of the above.

3. CASE structure is a
   a) conditional control structures
   b) unconditional control structures
   c) simple expression
   d) none of the above.

4. CASE statement is alternate of
   a) IF
   b) IF - THEN - ELSE
   c) FOR
   d) DO - WHILE.

24.4.2 Questions for short answers

1. What is the purpose of IF structure?
2. What are the forms of IF structure? Give examples.
3. What do you mean by multilayer nesting of IF - THEN - ELSE structure?
4. How is CASE structure used for making decisions?
5. Is it possible to use structured statements in the CASE structure?

24.4.3 Analytical questions

1. Write a program which takes a character from the keyboard and gives a message whether it is uppercase, lowercase, numeric digit or other numbers.
2. Is it possible to implement the problem by CASE structure? Show why CASE structure is beneficial.
Lesson 25: Control Structures-III

25.1 Learning Objectives:

On completion of this lesson you will learn:

- FOR loop structure
- different parts of FOR structure
- programming examples using this control structure.

25.2 The FOR Structure

The FOR structure is used to carry out unconditional looping in Pascal. That is, it allows some action to be repeated a specified number of times. The FOR structure has two different forms. The more common form is:

```
FOR control variable := value1 TO value2 DO statement
```

The statement part of the structure can be either simple or structured, though it is typically a compound statement that may include other control structures. This statement will be executed for each of the several consecutive values assigned to the control variable. The number of values assigned to the control variable therefore determines the number of times the statement will be executed.

The control variable must be a simple-type variable of any type other than real. Typically it will be either an integer variable or a user-defined variable. Initially, the control variable is assigned the value specified by `value1`. The control variable automatically takes on its next successive value each time the statement is repeated, until it finally takes on the value specified by `value2`. If the control is an integer-type variable, then it will automatically increase by 1 each time the statement is executed; hence, the statement will be executed `(value2 - value1 + 1)` times.

25.3 Examples on FOR Structure

To illustrate the use of the FOR structure, let us again consider the problem of writing out the first 20 positive integers, with one integer on each line. We require only a single FOR-TO statement to carry out this task. In particular, we can write:

```
FOR digit := 1 TO 20 DO writeln(digit);
```

where `digit` is assumed to be an integer type variable. In this example `digit` takes on the successive values 1,2,...,20 thus causing the loop to be...
Control Structures

executed 20 times. During each pass, the current value of digit will be written out on a separate line, as required. Similarly, the problem of determining the sum of the first $n$ integer quantities can be expressed as

\[
\text{sum} := 0;
\]

\[
\text{FOR digit} := 1 \text{ TO n DO sum} := \text{sum} + \text{digit};
\]

Here we begin by assigning a value of 0 to \textit{sum}. Each pass through the loop then causes the current value of \textit{digit} to be added to \textit{sum}. Therefore, \textit{sum} will represent the desired value $1+2+3+\ldots+n$ after the loop has been executed \textit{n} times. \textit{FOR} structure is usually the case when the required number of passes through the loop is known in advance.

There are a few rules that must be adhered to when writing a \textit{FOR} statement. Specifically, the values of \textit{value1} and \textit{value2} can be expressed as constants, variables or expressions. However, these values must be of the same data type as the control variable. Also, the value of \textit{value1} must be less than \textit{value2} if the statement is to be executed more than once. (If \textit{value1} and \textit{value2} are equal the statement will be executed once only; if \textit{value1} is greater than \textit{value2}, the statement will not be executed at all.)

\textbf{Example:} The use of the \textit{FOR} structure is really less complicated than it first appears, as illustrated by the following examples.

\begin{enumerate}
\item \hspace{1cm} \textit{sum} := 0;
\hspace{1cm} \text{FOR count} := 1 \text{ TO n DO }
\hspace{1cm} \hspace{1cm} \text{BEGIN readln} (x); \text{sum := sum} + x \hspace{1cm} \text{END;}
\hspace{1cm} \text{writeln ('sum =', sum);}
\item \hspace{1cm} \textit{sum} := 0;
\hspace{1cm} \text{FOR count} := n \text{ TO \hspace{0.5cm} (3\ast n+1) DO }
\hspace{1cm} \hspace{1cm} \text{BEGIN }
\hspace{1cm} \hspace{1cm} \hspace{1cm} \text{readln(x)};
\hspace{1cm} \hspace{1cm} \hspace{1cm} \text{sum := sum} + x
\hspace{1cm} \hspace{1cm} \text{END;
\hspace{1cm} \hspace{1cm} \text{writeln('sum =', sum);}
\end{enumerate}

The first example contains a compound statement that is executed \textit{n} times. (Note that \textit{n} is an integer-type variable whose value is assumed to be known, and that \textit{x} is a real-type variable). During each pass through the loop a new number (i.e., a new value for \textit{x}) is read into the computer and added to \textit{sum}. The sum of all \textit{n} numbers is then written out, after the loop has been completed. In the second example the initial and final values of the control variable are given by an integer-type variable and an integer-type expression, respectively. Notice that the compound statement is now spread out over several lines, with appropriate indentation. (This is the perfect format).
25.4 Another form of FOR Structure

The second form of the FOR structure is similar to the first form, except for the use of the key word DOWNTO in place of TO. Thus, the FOR structure can be written as

\[
\text{FOR control variable := value1 DOWNTO value2 DO statement}
\]

The action taken by this form of the FOR structure is similar to the first form, except that the control variable is evaluated backward rather than forward. Thus, if the control variable is an integer-type variable it will automatically decrease by 1, from value1 to value2, during successive passes through the loop. Therefore, value1 should be greater than value2. (if the values are equal, then the statement will be executed once only; and if value1 is less than value2, then the statement will not be executed at all).

Example : Here is an illustration of the second form of the FOR structure.

\[
\text{FOR i := 0 DOWNTO -12 DO}
\begin{align*}
\text{BEGIN} \\
\text{z := } 2 * i + 5; \\
\text{writeln('I =', i ', z =', z)}
\end{align*}
\text{END;}
\]

This example will cause 13 lines of text to be printed. Each line will contain the current value of the integer variable i, followed by the corresponding value of the formula \( z = 2i + 5 \). Notice that the successive values of i will decrease, from i =0 on the first line to i = -12.

It should be understood that value1 and value2 are evaluated only once, before the first pass through the loop. Therefore, the reader should not attempt to change either of these values within the loop. Also, the reader is cautioned not to use the control variable after the FOR structure has terminated, since it will normally be undefined.

25.5 Programming Example

Averaging a list of Numbers: Let us use the FOR structure to obtain the average of a list of n numbers. Our strategy will be based upon the use of a partial sum that is initially set equal to zero and is then updated as each new number is read into the computer. Thus, the very problem naturally lends itself to the use of repetition. The actual calculations can be carried out in the following manner.

1. Assign a value of 0 to the real variable sum.
2. Read in the value for n.
Control Structures

3. Do the following steps n times (i.e., for successive values of the control variable count ranging from 1 to n).
(a) Read in one of the numbers in the list. (Each number will be represented by the real variable x).
(b) Add the number to the value of sum.
4. Divide the value of sum by n to obtain the desired average.
5. Write out the calculated value for the average.

Here is the actual Pascal program.

PROGRAM average1(input, output);

(*This program calculates the average of n numbers using a FOR structure*)

VAR   n,count : integer ;
      x,sum,average : real ;
BEGIN (*action statements *)
      count := 1 ;
      sum := 0 ;
      readln(n);
      FOR count := 1 TO n DO BEGIN
          readln(x);
          sum := sum + x
      END;
      average := sum /n;
      writeln ('The average is', average);
END.

Notice that the control variable count is initially assigned a value of 1. This value increases by 1 each time the loop is executed, until count finally takes on a value of n. Thus, the loop will be executed exactly n times. Also observe that the sequence of statements included within the FOR structure is indented. This permits the physical extent of the structure to be easily identified.
25.6 Exercise

26.6.1 Multiple choice questions

1. Which of the following type is not control variable?
   
   a) Real
   b) Character
   c) Boolean
   d) User defined variable.

2. The initial and final value of FOR loop may be
   
   a) string
   b) logical operator
   c) relational operator
   d) expression.

3. Which of the following is equivalent?
   
   a) IF and IF - THEN - ELSE
   b) IF - THEN - ELSE and CASE
   c) FOR and IF
   d) FOR and CASE.

4. FOR loop is used for
   
   a) condition checking
   b) repetitive statement
   c) input statement
   d) parameter passing.

25.6.2 Question for short answers

1. When it is advantageous to use FOR control structure for looping?
2. What is the general format of FOR statement?
3. Is it possible to loop downward by using FOR structure? What is the format?
4. Explain how looping is done using FOR structure.
5. Identify all errors of the following program segment:

```pascal
VAR a,b,c,d : integer;
FOR       a:= b  DOWNTO  d DO
BEGIN
```

130
Control Structures

\[
\begin{array}{l}
\text{FOR } \ c := b \ TO \ d \ DO \\
\quad \begin{array}{l}
\quad \quad \\
\quad \quad \text{BEGIN} \\
\quad \quad \quad \\
\quad \quad \quad \text{END} \\
\quad \end{array} \\
\text{END;}
\end{array}
\]

25.6.3 Analytical questions

1. Write a programme segment using FOR structure to find the sum of the following series 1+5+9+16+......+n.
2. Write a programme segment using FOR structure to fund Sum, of the following series 1 + 2 + 3 + . . . + 10 and also calculate the average.
Lesson 26: Control Structures-IV

26.1 Learning Objectives

On completion of this lesson you will learn:

- two more repetitive control structures WHILE - DO and REPEAT - UNTIL
- equivalence of different control structures
- flow of execution of different loops.

26.2 The WHILE-DO Structure

The WHILE-DO structure is a repetitive control structure that is used to carry out conditional looping. The general form of this structure is

```
WHILE Boolean expression DO statement
```

The statement part of the structure will be executed repeatedly, as long as the Boolean expression remains true. This statement can be simple or structured, though it is usually a compound statement that includes some features that can alter the value of a Boolean expression. Suppose, for example, that we want to write out the integer quantities 1,2,...20, with one quantity on each line. This could be accomplished with the following WHILE-DO structure.

```
digit := 1;
WHILE digit <= 20 DO
   BEGIN
      writeln(digit);
      digit := digit +1
   END;
```

where digit is assumed to be an integer-type variable. We therefore begin with a value of digit =1. We proceed to write out the current value of digit, increase its value by 1 and then repeat the cycle. This process is continued as long as (i.e. WHILE) the value assigned to digit does not exceed 20. The net effect of this WHILE-DO structure is that the process of writing and incrementing will be repeated 20 times, resulting in 20 successive lines of output. Each line will contain a successive integer value, beginning with and ending with 20. Similarly, suppose that we want to determine the sum of the first n integer quantities, where n is a known integer-type variable. This could be accomplished by writing

```
sum := 0;
digit := 1;
```

```
Control Structures

WHILE digit <= n DO
BEGIN
    sum := sum + digit;
    digit := digit + 1
END;

or, equivalently,

sum := 0;
digit := 1;
WHILE digit < n + 1 DO
BEGIN
    sum := sum + digit;
    digit := digit + 1
END;

where sum and digit (as well as n) are assumed to be integer-typed variable. In either case, we begin with a value of sum = 0, then proceed to add successive values of digit to sum. The process of adding the current value of digit to sum and then increasing the value of digit by 1 will continue as long as (i.e. WHILE) the current value of digit is less than n + 1. At the conclusion of this process, sum will represent the sum of the first n integer quantities; i.e., 1+2+3+...+n. The WHILE -DO structure is used extensively in Pascal, as there are many programming applications that require this type of conditional looping capability. We will see many sample programs that utilize the WHILE-DO structure throughout this text.

26.3 Programming Example on WHILE-DO

Averaging a list of Numbers. Let us use the WHILE-DO structure to obtain the average of a list of n numbers. Our strategy will be based upon the use of a partial sum that is initially set equal to zero and is then updated as each new number is read into the computer like the previous example for FOR structure.

Here is the actual Pascal program using WHILE-DO structure.

PROGRAM average1 (input, output);
(*This program calculates the average of n numbers using a WHILE-DO loop*).
VAR n,count : integer;
x,sum,average : real;
BEGIN (*action statements*)
count := 1;
sum := 0;
readln(n);
WHILE count < n+1 DO
BEGIN
readln(x);
sum := sum+x;
count := count + 1
END;
average := sum /n;
writeln ('The average is', average)
END.

Notice that the WHILE-DO structure contains a compound statement which, among others causes the value of count to increase. Eventually, this will cause the Boolean expression

count < n+1

to become false, thus terminating the loop.

Also, note that the loop will not be executed at all if n is assigned a value that is less than 1. Physically, of course, it would not make any sense to do this. Finally, notice that the innermost compound statement is intended within the WHILE-DO structure. This causes the physical extent of the WHILE-DO structure to be readily identifiable.

26.4 The REPEAT-UNTIL Structure

The REPEAT-UNTIL structure is another repetitive control structure that is used to carry out conditional looping. It is similar to the WHILE-DO structure, and in some respects these two control structure complement one another. The general form of the REPEAT-UNTIL structure is

REPEAT sequence of statements UNTIL Boolean expression

The sequence of statements will be executed repeatedly until the Boolean expression becomes true. Note that the sequence of statements will always be executed at least once, since the Boolean expression is not tested until the end of the control structure. (This is in contrast to the WHILE-DO structure, where the Boolean expression is tested at the beginning of the control structure. The WHILE-DO structure will not be executed at all if the Boolean expression is initially false.)

Notice that this structure allows a sequence of statements to be included, whereas the WHILE-DO structure permits only one statement (though it can be compound). The sequence of statements within the REPEAT-UNTIL structure need not be included within BEGIN and END. Thus the keywords REPEAT and UNTIL act as brackets that indicate the beginning and the end of the statement sequence.
Control Structures

to include a compound statement or another control structure within this sequence. In addition, the sequence of statements will usually include some feature that will alter the value of the Boolean expression.

To illustrate the use of the REPEAT-UNTIL structure, consider once again the problem of writing out the integer quantities 1,2,...20 with one quantity on each line. We have already seen how this can be accomplished using the WHILE-DO structure. Now let us utilize the REPEAT-UNTIL structure for the same purpose.

```
digit := 1;
REPEAT
   writeln(digit);
   digit := digit + 1
UNTIL digit > 20;
```

Again, digit is assumed to be an integer-type variable.

We again begin with a value of digit =1, and then proceed to write out the current value of digit, increase its value by 1 and then repeat the cycle. The process is continued UNTIL the current value of digit (which was just increased in value) exceeds 20. The net effect of this REPEAT-UNTIL structure will be the same as the corresponding WHILE-DO structure presented in the last section. Thus, 20 successive lines of output will be generated, with each line showing a successive integer value.

Now let us consider the use of a REPEAT-UNTIL structure to determine the sum of the first n integer quantities, where n is a known integer-type variable. The required structure is

```
sum := 0;
digit := 1;
REPEAT
   sum := sum + digit;
   digit := digit + 1
UNTIL digit > n;
```

Again, sum and digit (as well as n) are assumed to be integer-type variables. In this case we begin with a value of sum = 0, and then proceed to add successive values of digit to some. The process of adding the current value of digit to sum and then increasing the value of digit by 1 will continue UNTIL the current value of digit exceeds n. At this time, sum will represent the sum of the first n integer quantities; i.e., 1+2+3+...+n.

The REPEAT-UNTIL structure, like the WHILE-DO structure, is used extensively in Pascal. Sometimes the choice of one structure over the other is simply a matter of personal preference. In other applications,
however, the choice is influenced by the desirability of testing the Boolean expression either at the beginning or at the end of the control structure.

### 26.5 Program using REPEAT-UNTIL STRUCTURE

Averaging a list of Numbers. Let us use the REPEAT-UNTIL structure to obtain the average of a list of n numbers. This is the same problem that we considered in the previous section using WHILE-DO loop.

Here is the actual Pascal program using REPEAT-UNTIL structure.

```pascal
PROGRAM average1(input, output);
(*This program calculates the average of n numbers using a WHILE-DO loop*).

VAR n, count : integer;
   x, sum, average : real;
BEGIN (*action statements*)
   count := 1;
   sum := 0;
   readln(n);
   REPEAT
      readln(x);
      sum := sum + x;
      count := count + 1
   UNTIL count > n;
   average := sum / n;
   writeln ('The average is', average)
END.
```

Notice that the sequence of statements contained within the REPEAT-UNTIL includes a statement that cause the value of count to increase. Eventually, this will cause the Boolean expression

\[
\text{count} > n
\]

to become true, thus terminating the loop. It should be understand that this loop will always be executed at least once, since the value of the Boolean expression is not tested until the end of the loop.

Finally, observe that the sequence of statements included within the REPEAT-UNTIL structure is intended so that the physical extent of the structure can easily be identified.
26.6 Exercise

26.6.1 Multiple choice questions

1. The expression in WHILE - DO structure must be
   a) conditional
   b) arithmetic
   c) logical
   d) a and c.

2. I=8
   WHILE I<15 DO
   BEGIN
       I=I+2;
   END
   How many times the loop will be executed
   a) 15
   b) 18
   c) 10
   d) 4.

3. Which structure need not be included within BEGIN and END
   a) WHILE - DO
   b) REPEAT - UNTIL
   c) IF - THEN - ELSE
   d) None of the above.

4. Which expression does not indicate the difference between WHILE
   DO and REPEAT - UNTIL procedure?
   a) Minimum iteration in WHILE - DO is 0 and in REPEAT - UNTIL is 1.
   b) In WHILE - DO structure including statements must be enclosed
      by BEGIN - END; but there is no restriction in REPEAT - UNTIL
   c) In WHILE - DO increment is only 1 and in REPEAT - UNTIL
      increment is not predefined
   d) None of the above.
26.6.2 Questions for short answers

1. What is the main feature of WHILE - DO structure?
2. What is the general format of the WHILE - DO statement?
3. What is the main difference between WHILE - DO and REPEAT - UNTIL structure?
4. Write down the general procedure how a WHILE - DO loop can be transferred into a FOR loop and REPEAT - UNTIL loop.
5. Show that WHILE - DO is sufficient to do any repetitive control statements.

26.6.3 Analytical questions

1. Write a program segment to calculate the sum of the following series using WHILE - DO structure 1+3+5+7+...+19.
2. Repeat the problem 1 by REPEAT - UNTIL structure.
3. Write a program segment to calculate the sum of the following series using by REPEAT - UNTIL 1+2+3+4+...+10 and calculate the average.
Unit 8 : Procedures and Functions

Introduction

Procedures and functions are introduced in this unit. An overall program may be decomposed into a sequence of individual subroutines. Procedures and functions in some other language are called subroutines. We discuss here various types of parameters and variables to be used in a program.

Lesson 27: Procedures and Functions-I

27.1 Learning Objectives

On completion of this lesson you will learn :

- what procedures and functions are
- how procedures work
- scope of identifiers.

27.2 Procedures and Functions

PASCAL programs can easily be designed in modular form. An overall program may be decomposed into a sequence of individual subproblems. There are two types of program modules in PASCAL. They are: procedures and functions. A procedure is a self-contained program structure that is included within a Pascal program. In some other programming languages, this type of structure is known as a subroutine.

The use of subroutines is important, because:

1. It makes a program easier to understand and modify.
2. Suppose you have designed a large PASCAL program in modular concept. It will help you to test and develop your program part by part. Coding and debugging may individually be done independently for each subroutine.
3. Modular programming will reduce your jobs. Suppose in a large program you are to perform some jobs in different parts of the program. You may write a subroutine for these jobs and use this subroutine as necessary at different parts of the program. This will speed up your working capability and make the program easy to read.
27.3 How procedures work

A PROCEDURE can be referenced simply by writing the procedure name followed by an optional list of parameters. The parameters must be enclosed in parentheses, if there are more than one, separated by commas. The procedure references are also known as procedure calls. When a procedure is referenced, control is automatically transferred to the beginning of the procedure. The action statements within the procedure are then executed. After executing all the action statements, control is automatically returned to the statement immediately after the procedure reference.

Example: Let us write a PASCAL program which reads in three integer quantities and then determines which quantity is the largest. Suppose we have a procedure named maximum which finds out the maximum of three numbers. By using this procedure we can write a program to solve our program.

```pascal
PROGRAM findmax;
VAR a, b, c : INTEGER;

PROCEDURE maximum;
VAR max : INTEGER;
BEGIN
  IF a > b THEN max := a
  ELSE max := b;
  IF c > max THEN max := c;
  writeln('The maximum number is:', max)
END;

BEGIN (*main action block*)
  READLN(a, b, c);
  WHILE a<>0 DO
  BEGIN
    maximum;
    readln(a, b, c);
  END
END.
```

When the reference to the procedure maximum is encountered, control is automatically transferred to the procedure. The procedure determines which of the three variables a, b and c has the largest value and then writes out the result. Control is then returned to the following READLN statement.

27.4 Scope of Identifiers
The scope of an identifier refers to the region within which the identifier is declared and can be utilized. The constants and variables that appear within the action statements of a procedure may have been declared externally, within a program block that contains the procedure declaration, or locally, within the procedure itself. In this respect there are two types of these:

(i) global   (ii) local.

The constants and variables that are declared within a block containing the procedure declaration are called global identifiers and can be utilized anywhere within that block, whether inside of or external to the procedure. On the other hand, the constants and variables defined inside a procedure are called local identifiers. They cannot be utilized externally.

Example: Observe the Pascal program shown in section 27.3. The program contains one procedure, named maximum. The variables a, b and c are declared outside of the procedure. They are utilized both within and outside of the procedure. Hence, they are global to the procedure maximum. On the other hand, the variable max is declared within the procedure maximum. Thus max is local to the procedure and cannot be utilized anywhere outside of this procedure.

27.5 Preference of Global and Local Identifiers

It is important to recognize when to use each type of identifier. In general, local identifiers are preferable to global identifiers, provided this is not inconsistent with the overall program logic. The use of local identifiers contributes to greater program legibility. It also minimizes the likelihood of programming errors caused by incorrect or inconsistent identifiers references. Same names can be used as local and global identifier. But only the name which is local to the current block is used or accessible. The name which is global is ignored in that position.

On the other hand, many programs require that certain data items be recognized both within and outside of a procedure. There are two ways to transfer such information across procedure boundaries:

(i) By using global identifiers.
(ii) By using parameters

By using global identifiers, the desired information may be made available to the whole program. How to pass information by using parameters will be discussed in the lesson.
27.6 Exercise

27.6.1 Multiple choice questions

1. Which is not used for a parameter passing?
   a) constants
   b) global variables
   c) arguments
   d) none of the above.

2. Which is not true for a subroutine?
   a) It makes the program modular
   b) It is easier to modify program
   c) It reduces execution time
   d) Debugging is easy.

3. A local variable can be used
   a) inside the procedure where it is called
   b) inside the procedure where it is declared
   c) inside the program where it is declared
   d) after complete execution of program.

27.6.2 Questions for short answers

1. What is the use of a subroutine?
2. How are procedures referenced?
3. Write the scope rule of Pascal.
4. What scope rule is used when same name is used as local and global variables?
5. Why local identifiers are preferable to global identifiers?

27.6.3 Analytical question

1. Write a subroutine which takes the radius of a circle as argument and returns with the area of the circle.
Lesson 28 : Procedures and Functions-II

28.1 Learning Objective

On completion of this lesson you will be able to learn:

- various types of parameters
- how value parameters work
- how variable parameters work
- when value and variable parameters will be used.

28.2 Passing Parameters to the Procedures

When we use procedures (or functions), we often need to exchange information between a procedure (or a function) and the point at which the procedure (or function) is referenced. One approach to accomplish this is to utilize global variables, as discussed in previous lesson. In previous lesson, we have also seen that there are some potentially undesirable aspects to the use of global variables. For example, altering the value of a global variable within a procedure may inadvertently alter certain information outside of the procedure, and vice versa. Furthermore, the transfer of multiple data sets cannot easily be accommodated with global variables.

The use of parameters offers a better approach to the exchange of information between a procedure and its reference point. Each data item is transferred between an actual parameter, which is included within the procedure reference, and a corresponding formal parameter, which is defined within the procedure. When the procedure is accessed, the actual parameters replace the formal parameters and thus information is exchanged between the procedure and its reference point.

Example: Go through the skeletal structure given below.

```
PROGRAM sample;
VAR a,b,c,d : real;
PROCEDURE  flash(x, y : real);
BEGIN    (*main action block*)
  (*process the values of x and y*)
END;
```

When the procedure is accessed, the actual parameters replace the formal parameters and thus information is exchanged between the procedure and its reference point.
Computer Programming

```
    .
    flash(a,b);
    .
    .
    flash(c,d);
    .
    .
END.
```

In this example the variables x and y are real-type formal parameters defined within the procedure flash. The actual parameters are the real variables a, b, c and d. We will assume that a, b, c and d have been assigned values elsewhere in the program, prior to the procedure references.

The first procedure reference causes the values of the actual parameters a and b to be transferred to the formal parameters x and y. Thus, the values of a and b are passed to procedure flash, where they are then processed.

This process is then repeated in the second procedure statement, this time transferring the values of c and d to x and y. The values of c and d are thus passed to flash, where they are processed accordingly. Notice that we have processed two different data sets simply by accessing the same procedure twice, with a different set of actual parameters each time.

There are certain rules that must be observed in order to establish a correspondence between a procedure reference and the procedure itself (i.e., when substituting actual parameters for formal parameters). They are:

1. The number of actual parameters in the procedure reference must be the same as the number of formal parameters in the procedure definition.
2. Each actual parameter must be of the same type as its corresponding formal parameter.
3. Each actual parameter must be expressed in a manner which is consistent with its corresponding formal parameter, as determined by the class of the formal parameter (the class of formal parameter is described below).

**Example**: Consider the following skeletal program structure.

```
PROGRAM sample;
```

Each actual parameter must be of the same type as its corresponding formal parameter.
VAR a,b : integer;
c,d : real;
PROCEDURE flash(x : integer; y : real);
BEGIN (*x and y formal parameter*)
  .
  (*process the values of x and y*)
  .
END;

BEGIN (*main action block *)
  .
  flash(a,c); (*a and c actual parameters*)
  .
  flash(b,d); (*b and d actual parameters*)
  .
END.

Note that each procedure reference includes two actual parameters, since two formal parameters (x and y) are defined within the procedure. Moreover, x is declared as an integer-type variable, and y is a real-type variable. Thus, each procedure reference must include one integer variable and one real variable, in that order.

28.3 Various Types of Formal Parameters

A procedure can contain four different classes of formal parameters. They are as follows:

(i) Value parameters
(ii) Variable parameters
(iii) Procedure parameters
(iv) Function parameters.

28.4 Value Parameters

Value parameters can best be thought of as input parameters for their respective procedures. By using value parameter, actual parameter is passed "by value" rather than "by reference". Values represented by
value parameters cannot, however, be transferred in the opposite direction, i.e., from the procedure to the calling portion of the program.

Value parameters are very simple to use. They are declared by simply including their names and corresponding data types within the procedure header, without any prefix (such as VAR). Absence of such a prefix automatically identifies this class of parameters.

Since in this case, the values of the actual parameters are transferred rather than the parameters themselves, an actual parameter may be expressed as a constant, a variable or an expression (provided the value of the parameter is of the proper data type).

28.4.1 Advantages and Disadvantages of Value Parameters

Any alteration to the value of a value parameter within the calling procedure will not affect the value of any of the actual parameters. This characteristics may limit the use of value parameters. However, such parameters are useful when one-way transfer of information is required (from the calling portion of the program to the procedure).

Example: Consider the following skeletal program structure,

```pascal
PROGRAM sample;
VAR a,b : integer;
    c,d : real;

PROCEDURE flash(x:integer; y : real);
    BEGIN
        ...
        (*process the values of x and y*)
        ...
        END;
    BEGIN (*main action block*)
        ...
        flash(3,a*(c+d)/b);
        ...
    END;
```

Any alteration to the value of a value parameter within the calling procedure will not affect the value of any of the actual parameters.
Notice that the formal parameters x and y, declared in procedure flash, are value parameters. The first of these (x) is of type integer, and the second (y) is real. Therefore, each reference to flash must contain two actual parameters, the first of which must be of type integer and the second real.

The main block includes two different procedure reference (i.e., two different references to flash). Each procedure reference contains two actual parameters, the first of which is of type integer and the second real, as required. Notice that two of these parameters are written as constants and two are written as expressions. Thus the first procedure reference transfers the value 3 to x, and the value of the real expression a*(c+d)/b to y. Similarly, the second procedure reference transfers the value of the integer expression 2*(a+b) to x and the value -0.5 to y.

### 28.5 Variable Parameters

In many applications, it is required that information must be transferred in both directions between the procedure and the procedure reference. In such situations, variable parameters are used. The format of using variable parameter is that the formal parameter must be preceded by keyword VAR in procedure declaration. Here is a program which swaps two arguments.

```pascal
PROGRAM change(input, output);
VAR x, y : integer;
PROCEDURE swap(VAR x : integer; VAR y : real);
VAR temp : integer;
BEGIN
  temp := x;
  x := y;
  y := temp
END;
BEGIN
  x := 10;
  y := 20;
  swap(x,y)
END.
```

At the end of program x=20 and y=10. When variable parameter is used the address of the parameter is referenced not the value of the parameter. So after complete execution of procedure the formal parameters passes
their values to the main program such that actual parameters possess the value of formal parameters. Thus information is passed from procedure to main program.

28.6 Exercise

28.6.1 Multiple choice questions

1. Which is not the rule for parameter passing?
   a) The type of actual and formal parameters must be same.
   b) The number of actual parameters and formal parameters are same.
   c) The order of parameters play no vital role.
   d) Each actual parameter must be consistent with its corresponding formal parameter.

2. Which may not be value parameters?
   a) Expression
   b) Procedure
   c) Variable
   d) Constants.

3. Variable parameter exchanges information in
   a) One way from procedure to main program
   b) Two way between procedure and main program
   c) One way from main program to procedure
   d) none of the above.

4. Variable parameter passes
   a) name of the actual parameter
   b) value of the actual parameter
   c) address of the actual parameter
   d) none of the above.

28.6.2 Questions for short answers

1. What are the problems of using global variables for transferring data elements between main program and procedure?
2. Explain rules associated with parameter passing.
3. Write down different types of parameters.
4. What are the main advantage and disadvantage of value parameter?
5. What do you mean by value parameter?
Procedures and Functions

28.6.3 Analytical question

1. Write a program which takes a number as argument and computes the square root by binomial expansion using variable parameters.
Lesson 29 : Procedures and Functions -III

29.1 Learning Objective

On completion of this lesson you will be able to learn:

- what functions are
- how functions work
- procedures and functions as parameters.

29.2 How Functions Work?

A function is a self contained program that is in many respects similar to a procedure. A function is very similar to a procedure. But there is one difference. It returns a value where its name appears. Moreover, a function is referenced by specifying its name within expressions as though it were an ordinary simple-type variable. The function name can be followed by one or more actual parameters enclosed in parentheses and separated by commas. In most cases, the actual parameters will transfer information to value parameters within the function and hence may be expressed as constants, variables or expressions.

The function itself consists of a function header and a block. The function header is written in the form

```
FUNCTION name : type
```

or, if parameters are included,

```
FUNCTION name(formal parameters) : type
```

The last item, type, specifies the data type of the result that is returned by the function. Generally, the formal parameters will be value parameters rather than variable parameters. This allows the corresponding actual parameters to be expressed as constants, variables or expressions.

The block is similar to that of a procedure and involves the same rules of scope as a procedure. Within the block, the identifier that represents the function name must be assigned a value of the appropriate type (as specified in the header). This is the value that the function returns to its reference point. Values can be assigned to the function name at two or more points within the block. Once an assignment is made, it cannot subsequently be altered.

**Example:** Observe the following function named *factorial* that calculates the factorial of a positive integer quantity.
FUNCTION factorial(n : integer) : integer;

(*calculates the factorial of n*)
VAR factor, product : integer;
BEGIN
  IF n<=1 THEN factorial := 1
  ELSE BEGIN
    product := 1;
    FOR factor := 2 TO n DO
      product := product*factor;
    factorial := product
  END
END

The first line, which contains the keyword FUNCTION, is the function header. Here, the header includes a declaration of the value parameter n. Also, the last item on the line(INTEGER), which states that the function will return an integer-type quantity.

This function accepts a value for n and then calculates the value of n! using two local integer variables, factor and product. The final result is assigned to the identifier factorial, which is also a function name.

Notice that there are two different assignments to factorial, but only one of these assignments is utilized when the function is executed. The choice depends upon the value that is assigned to n. Once factorial is assigned a value, it is not altered within the function.

To amplify on this last point, consider the following variation of the above function.

FUNCTION factorial(n : integer) : integer;

(*Calculates the factorial of n *)
VAR factor : integer;
BEGIN
  factorial := 1;
  IF n>1 THEN
    FOR factor := 2 TO n DO
      factorial := factorial*factor
  END;
END;

On the surface this version appears more appealing than the original, since it is simpler. But, This version is not valid because the value of factorial is altered after its initial assignment when n is greater than 1.
29.3 Functions and Procedures as Parameters

Sometimes it is desirable for a given procedure or function to access another procedure or function that has been defined outside the scope of the given procedure or function. For example, we may wish to have procedure A make use of function B, though function B has been defined outside of procedure A. This may be accomplished by passing the external procedure or function (e.g., function B) to the given procedure or function (e.g., procedure A) as a parameter. Thus, Pascal supports procedure parameters and function parameters as well as value parameters and variable parameters. The formal parameter declaration for a procedure or a function header, as illustrated below.

EXAMPLE: Suppose that the procedure process must access an externally defined real function that includes an integer-type parameter. Let us refer to this externally defined function as \( f \). The procedure, process might appear as follows.

```pascal
PROCEDURE process(FUNCTION f (u : integer) : real; c1, c2 : integer);
VAR c : integer
x : real;
BEGIN
    FOR c := c1 TO c2 DO
        BEGIN
            x := f(c);
            WRITELN(x =, x)
        END
END;
```

The main block might include the following reference to this procedure.
`process(calc, 1, 100);`

where the function \( calc \) is defined within the main block. Here is a skeletal structure of the entire main block.

```pascal
PROGRAM main;
.
.
FUNCTION calc(w : integer) : real;
.
.
```

Thus, Pascal supports procedure parameters and function parameters as well as value parameters and variable parameters.
Procedure and Functions

BEGIN (*function calc*)
   ...
   calc := ...;
END; (*calc*)

PROCEDURE process(FUNCTION f(u : integer) : real;
   c1, c2 : integer);

VAR c : integer;
   x : real;

BEGIN
   FOR c := c1 TO c2 DO
      BEGIN
         x := f(c);
         WRITELN('x=', x)
      END
   END; (*process*)

BEGIN (*main block*)
   ...
   process(calc, 1, 100);
   ...
END.

Procedure parameters and function parameters are particularly useful when the given procedure or function accesses different procedures or functions (i.e., makes use of different actual parameters) at different calling points. The formal procedure and function parameters and the actual procedure and function parameters must correspond with respect to their own parameters. This correspondence must include the number, class and type of parameters. Procedure parameters and function parameters are particularly useful when the given procedure or function accesses different procedures or functions (i.e., makes use of different actual parameters) at different calling points.
29.4 Exercise

29.4.1 Multiple choice questions

1. Which of the following is equivalent to a function?
   a) Procedure
   b) Procedure with value parameter
   c) Procedure with variable parameter
   d) Procedure with procedure parameter.

2. Function can be described as
   a) variable
   b) operator
   c) constants
   d) none of the above.

3. A function can return
   a) 1
   b) 2
   c) 3
   d) none of the above data elements.

4. A procedure can return
   a) 1
   b) 2
   c) 3
   d) none of the above data elements.

29.4.2 Questions for short answers

1. Write down the syntax of a function declaration.
2. What is the scope rule of a function?
3. What is the main difference between procedure and function?
4. What are the function parameter and the procedure parameter?
5. Show some uses of a function and a procedure parameters.

29.4.3 Analytical questions

1. Write a function which returns the following series $1 + 5 + 9 + 13 + \ldots$. The input is the number of elements in the series.
2. Write functions that generate 16-bit random number.
3. Write a function that calculates the factorial of a positive integer quantity.
Unit 9 : Arrays and Records

Introduction

Three important data structures have been discussed in this unit. Firstly array and then records and sets. Array is a data structure where large amount of identical data can be stored. The methods of accessing elements of arrays and declaring arrays are listed here. Record is an important data structure which has elements of different types. Accessing different elements of records will be shown in the corresponding lesson. Set is also an important data structure which can incorporate the idea of mathematics in Pascal programming language. Several examples incorporating the above ideas have been presented.

Lesson 30 : Arrays-I

30.1 Learning Objectives

On completion of this lesson you will learn :

- a new data structure array which allows to store and operate on many data of the same type.
- how array is defined in programming languages
- how this data structure can be accessed
- use of array with sample program segments.

30.2 Arrays

An ARRAY is a collection of elements of identical data type. Group of these elements are given a name to identify them collectively. Each element in the group can uniquely be identified and hence accessed by using its relative position in that group. One important thing to remember is that each element of the ARRAY is same in nature i.e., if an ARRAY is defined of type character then each cell of the ARRAY (each element) must be of character type.

During the time of defining an ARRAY there exists mechanism through which you can specify which type of data the ARRAY will contain, how many elements would there be, what would their respective indexes be i.e., position of each element within the ARRAY etc.

Following are some declaration syntaxes for some ARRAYS

```pascal
VAR LETTER : ARRAY [1..26] OF CHAR;
VAR POSITION : ARRAY [1..20] OF INTEGER;
TYPE COLOR = [RED, GREEN, BLUE];
VAR SHIRT : ARRAY [COLOR] OF CHAR;
TYPE LIMIT = 0..50;
TYPE MARKS = ARRAY [1..100] OF LIMIT;
```
Computer Programming

VAR STUDENT : MARKS;

In the first example above LETTER is the name of the array. "ARRAY" is the reserved word in PASCAL, [1..26] is the subscript also called index type, CHAR is the element type in the array LETTER.

Thus the general structure for the declaration of an array is

VAR <Arrayname> : ARRAY [<Range of index type>] of <element type>;

To access a particular cell (element in the array) we use the following syntax:

<Array name> [constant or variable of the index type]

Let’s look in detail at some of the preceding examples:

The declaration

VAR LETTER : ARRAY [1..26] of CHAR;

Sets up 26 cells in memory each of which contains an element of type character (CHAR). Each element (cell) can be accessed by its position in the group of 26 characters.

The declaration

VAR POSITION : ARRAY [1..20] of INTEGER;

Sets up 20 cells or room in the memory each of which contains an integer value. Each element can be accessed by using the notation POSITION [1], POSITION [2],...... etc.
30.3 Use of Array

The following is a simple program showing the use of arrays. This program finds the average of 20 numbers.

```
PROGRAM Average (INPUT, OUTPUT);
CONST  N : = 20;
VAR  SUM, I : INTEGER;
     MARK : ARRAY [1.. 20] OF INTEGER;
     AVE : real;
BEGIN (*MAIN*)
  SUM : = 0;
  FOR I = 1 TO 20 DO
    BEGIN
      READ (MARK [I]);
      SUM : = SUM + MARK [I]
    END;
  AVE : = SUM / N;
  WRITELN ("Average of 20 numbers is", AVE)
END. (* MAIN*)
```

30.4 Exercise

30.4.1 Multiple choice questions

1. Consider the following array declaration

   VAR  List : ARRAY [20..30] OF INTEGER

   i) The number of elements in the above list is

   a) 30
   b) 11
   c) 10
   d) none of above.

   ii) The number of bytes in the above list is

   a) 30
   b) 11
   c) 88
   d) 22.
iii) To get access to the 5th element, the array reference statement is

a) List[5]  
b) List[10]  
c) List[24]  
d) none of above.

iv) What happens if we want to get access to List[32]?

a) Error message  
b) Garbage value  
c) May get content of other data or instruction.  
d) none of above.

2. Subscript of array

a) must be positive integer  
b) no restriction about +ve or -ve  
c) must begin from 0  
d) none of above.

3. What does the last program in this lesson do?

a) takes 20 numbers  
b) takes 20 integers and calculates their summation and average  
c) takes 20 integers from keyboard and calculates their summation and average  
d) none of above.

30.4.2 Questions for short answers

1. What are the particular characteristics of an array as a structured data type?  
2. Give the pictorial representation of a one dimensional array.  
3. What is an array element? What data types can be used as array element? Give examples for each type.  
4. How can an individual array element be accessed?  
5. Explain the declaration syntax of a one dimensional array.

30.4.3 Analytical question

1. Write a program which takes 100 numbers and then sort them in descending or ascending order.
Lesson 31 : Arrays -II

31.1 Learning Objectives

On completion of this lesson you will learn :

- concept of multi-dimensional array
- definition of multidimensional array with examples
- accessing methods of multidimensional array and their use.

31.2 Multi-dimensional Array

Declaration syntax for a single or one-dimensional array is

```
VAR < Array name > : ARRAY [ Indextype ] OF < element type >
```

The concept of higher dimensional array comes into play when we think about linear collection of its lower dimensional arrays. For example, suppose we have an array of type integer having 100 elements. Then we can think about 10 such arrays each having the same number of integer elements (1000) arranged together as depicted below to form a two-dimensional array "Table".

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The cell highlighted is the 4th cell of 3rd array of the array group.

In Pascal, situation like this is handled by using two dimensional array.

Following would be declaration syntax for the 'Table' shown above:

```
VAR Table : ARRAY [1..10] OF (First array type);
```

or

```
VAR Table : ARRAY [1,...,10] OF ARRAY [1...100] OF integer;
```

or

```
VAR Table : ARRAY [1..10, 1..100] OF integer;
```

Using the above rules, following are some valid declarations of two-dimensional arrays:
TYPE ROW = ARRAY [1.....20] OF char;
    COL = ARRAY [1.....50] OF row;
VAR arr_row  : row;
VAR arr_col  : col;
VAR Square 1 : ARRAY [1....100, 1......1000] OF char;
    Square 2 : ARRAY [1....100, 200...299] OF integer;
    Square 3 : ARRAY [-100..99, 300....499] OF boolean;

In order to access any element in a two dimensional array the syntax is

    Table [I] [J]

Where I denotes an index in first dimension, J denotes an index in second dimension. These subscripts as shown may be a variable or constant. In the two dimensional array "Table" we have depicted previously that the highlighted element can be referenced by

    Table [3] [4]

31.3 Multi-dimensional Arrays

PASCAL has no limit with the dimension of an array. So an array may be of dimension 3, 4, 5 ... A typical 3 dimensional array "calendar" is depicted in the figure below:

![Multi-dimensional Arrays](image)

Fig. 31.1 Multi-dimensional arrays

Let us see what exactly above array means in step by step.

VAR Day   : ARRAY [1...........31] OF integer;
VAR Month : ARRAY [1...........12] OF integer;
VAR Year  : ARRAY [1900.....1999] OF integer;
Arrays and Records

These three one dimensional array declaration may be combined to obtain the declaration syntax of the 3 dimensional array calendar. The declaration syntax would be

\[ \text{VAR Calendar : ARRAY [1900..1999, 1..12, 1...31] OF integer;} \]

Thus it is obvious that a two-dimensional array is an array of another one dimensional array. Similarly a three-dimensional array would be an array of another two dimensional array.

In order to access a particular day say December 16, 1971 we will use

\[ \text{calendar [1971][12][16]} \]

which first enters into the year set (1900..1999) , then month set (1..12) within that year and finally picks up the desired day from within the day set (1..31) of that month.

Following is the accessing sequence for each day of the array calendar and is initialized to 0

\[ \text{FOR I = 1900 To 1999 DO FOR J = 1 TO 12 DO FOR K = 1 TO 31 DO Calendar [I][J][K] := 0 ;} \]

31.4 Exercise

31.4.1 Multiple choice questions

1. In the declaration of calendar how many places are useless and empty?
   a) 675
   b) 670
   c) 680
   d) none of the above.

2. Maximum number of dimensions of an array is
   a) 3
   b) 6
   c) limitless
   d) none of the above.
3. An array is declared as VAR A :ARRAY [1..10, 1..20] OF REAL

i) The number of bytes required in the above declaration is
   a) 200
   b) 800
   c) 20
   d) none of the above.

ii) First element of the above declaration is accessed by
   a) A[1]
   b) A[0][0]
   c) A[1][1]
   d) none of the above.

iii) The last element is of the above declaration is
   a) A[20]
   b) A[10][20]
   c) A[20][20]
   d) A[10][10].

31.4.2 Questions for short answers

1. How is multi-dimensional array declared or defined?
2. Give the pictorial representation of a multi-dimensional array.
3. How can the elements of a multi-dimensional array be accessed?
4. Are the indices of a multi-dimensional array of same type?
5. How can two dimensional array be expressed as a one-dimensional array? Can this concept be extended to multi-dimensional arrays?

31.4.3 Analytical question

1. Write a program segment which adds two of numbers.
Lesson 32 : Records-I

32.1 Learning Objectives

On completion of this lesson you will learn:

- how different types of data are grouped together into a record
- method of defining record data types
- access method of records
- nested definition of records.

Record is that structured data type whose constituent elements not be the same. One of the difficulties of using array is that each element of the array must be of the same data type. There are some situations where it is necessary to store heterogeneous data against a single object. As for example, it may be needed to store student ID, student name, courses undertaken and GPA of those courses for each student of a class. We can handle this situation using parallel arrays as follows:

<table>
<thead>
<tr>
<th>std-ID</th>
<th>std-NAM</th>
<th>std-CRS</th>
<th>std-GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>[1]</td>
<td>[1]</td>
<td>[1]</td>
</tr>
<tr>
<td>[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>[N]</td>
<td>[N]</td>
<td>[N]</td>
<td>[N]</td>
</tr>
</tbody>
</table>

Here using the same subscript in each of the 4 arrays we can have complete information for a student.

PASCAL provides an alternate and attractive mechanism to store such data using Records.

Declaration syntax for a Record type data structure is

```pascal
TYPE recname = RECORD
  field identifier : type;
  filed identifier : type
END;
```

Here <recname> is a user given name, <field identifiers> are the name of components in a record.
To store student information using record type data structure following should be the declaration syntax (assuming student id is INTEGER TYPE, student name is an array of 30 characters, std-CRS is an array of 10 characters, and std-GPA is of REAL type).

```pascal
TYPE std-Rec = RECORD
  std - id : integer;
  std - NAM : ARRAY [1...30] OF char;
  std - CRS : ARRAY [1...10] OF char;
  std - GPA : REAL
VAR  student : std - Rec;
```

### 32.2 Access Method for Records

If we wish to find the 5th character of the student name we should use the field selector operator (.) as follows:

```pascal
WRITE (student. std-NAM[5]);
```

### 32.3 Hierarchical Records

We have seen a simple record structure whose component can be of any type such as integer, character, array etc. A component of a record can itself be another record. Although the concept is a little bit complicated this hierarchical data structure has some definite advantages over some other scalar data types.

Following is a declaration for a hierarchical records:

```pascal
TYPE Date = RECORD
  Year : 1900 .. 2000;
  Month : 1 .. 12;
  Day : 1 .. 30
END;

History = RECORD
  Name : ARRAY [1..30] OF char;
  Salary : real;
  Join : Date
END;

Employee = RECORD
  Emp-Id : integer;
  Post : char;
  Detail : History;
  Retire : Date
END;
```
Arrays and Records

VAR Emp : Employee;

Thus joining year of an employee "Emp" is accessed using the field selector as shown below:


32.4 Exercise

32.4.1 Multiple choice questions

1. Record supports
   a) grouping of the same type of data
   b) grouping of different types of data
   c) both a and b
   d) none of the above.

2. If you want to get access to the 10th character of student name you have to give the following reference
   a) student[10]
   b) student.std-CRS[10]
   c) student.std-GPA
   d) student.std-NAM[10].

3. Joining date of an employee can be accessed using the following field selector from the record Employee mentioned in the example.
   a) Emp.Retire
   b) Emp.Retire.day
   c) Emp.Detail.Join.day
   d) none of the above.

32.4.2 Questions for short answers

1. What is a record? Why do such data types come up?
2. Give the syntax of a record data type.
3. What is the access method of a record?
4. How nested records are used?
5. What is the principal difference between arrays and records?

32.4.3 Analytical question

1. Give an example where fields of a record are also records.
Lesson: 33 Records-II

33.1 Learning Objectives

On completion of this lesson you will learn:

- definition of array of which each element is a record
- practical use of array of record in database of students
- easy method of referencing array of record using WITH statement
- nested use of WITH statement with example.

33.2 Array of Record

As we already have seen that array can be of any type e.g. CHAR, INTEGER, REAL, BOOLEAN or other user defined type, so can be of type "Record". Thus it is possible to define an array of type "Record". This is particularly important in some cases where different types of data for a single entity is required to store all information (name, Id, courses, GPA, department etc.) of all students of a class. If the class contains 40 students we would define an array of length 40 whose type would be of <Record type> where <Record type> contains details of each student.

Here is an example of a class roster represented by an array of student records:

```pascal
TYPE student = RECORD
  Name   : ARRAY [1..20] OF CHAR;
  Class  : 1 .. 6;
  GPA    : REAL
END;

LIST = ARRAY [1..500] OF student;

VAR ROLL : LIST;
```

This can be visualized as shown in the following figure:
Arrays and Records

[1] 1st record of type student

[2] 2nd record of type student

[3] 

[4] Roll[4].Name

Fig. 33.1 Array of Record

An element of ROLL is selected by a subscript 4. Thus ROLL [4] is the 4th element in the array of Record ROLL of type student. In order to access the GPA of the 4th student we use the following expression:

\[
ROLL[4].GPA
\]

Specifies the 4th element
Specifies the GPA field of that record

If we want to access the first character of the name of 4th student, we use the following syntax:

\[
ROLL[4].Name[1]
\]

4th element
 Specifies Name field
 Specifies 1st character

Fig. 33.2 Array of Records

33.3 WITH Statement

There are often arises situation where it is repeatedly necessary to access fields of a record. In this case normal mechanism to access the field of a record by using field selector (.) may become cumbersome. It is
necessary to write the name of the record plus its field name together with a dot (.).

PASCAL provides a shortcut mechanism to work with a record variable which is 'WITH'.

The format of the WITH statement is

```
WITH <record variable> Do statement
```

The use of WITH can be made clear by using a suitable example: we assume our previous record type "student"

```
TYPE student = RECORD
   Name : string [30];
   Roll : integer;
   GPA  : real
END;
VAR std-Lst : Array [1..30] OF student;
```

Following are two methods to read data into the record type variable std-Lst array.

**Method 1**

```
FOR I := 1 TO 30 DO
BEGIN
   READ (std-Lst[I]. Name);
   READ (std-Lst[I]. Roll);
   READ (std-Lst[I]. GPA)
END;
```

**Method 2**

```
FOR I := 1 TO 30 DO
   WITH std-Lst[I] DO
   BEGIN
      Read (Name);
      Read (Roll);
      Read (GPA)
   END;
```

Note that in method 2 we have used "Std-Lst[I]" 3 times. Thus within the scope of the WITH statement field identifiers are treated as variable identifier. Thus within the scope of WITH statement the record component can be selected by the field identifier alone - not by the full field selector.

WITH statements can be nested. For example the statement like
Arrays and Records

WITH <record variable 1> DO
   WITH <record variable 2> DO
   statement

is allowed. This statement can be abbreviated to:

WITH record variable 1, record variable 2, DO statement

This nesting is particularly important when processing array of records. Let us assume following declarations:

TYPE History = RECORD
   Name : ARRAY [1..30] OF char;
   DIST : ARRAY [1..20] OF char;
   SSN : integer;
END;

Lst  = RECORD
   UNIVERSITY : ARRAY [1..30] OF char;
   FACULTY : ARRAY [1..20] OF char;
   Member : student
END;

VAR
   Roaster : ARRAY [1..10] of Lst;
   std : student;
   detail : History;

Then the following are valid statements in the nested with:

I := 2;
WITH Roaster [I] DO
   WITH number DO
      :
      :
      :
33.4 Exercise

33.4.1 Multiple choice questions

1. To get access to the GPA of a roll number 7 from the list we have the following reference
   a) Roll[7].Class
   b) Roll[7].GPA
   c) Roll[7].Name[7]
   d) none of the above.

2. WITH statement is used for the following purpose
   a) Shortening the program size
   b) Shortening space requirement
   c) Shortening compiling time
   d) improving computational efficiency.

3. For the list Roaster mentioned in this lesson, to get access to the 5th character of faculty name of the 5th element of list Roaster we have to write
   a) Roaster[5].FACULTY[5]
   b) Roaster[5].Name[5]
   c) Roaster[5][5]
   d) none of the above.

4. WITH can be used
   a) any where
   b) in the loop only
   c) for a group of statements
   d) none of the above.

33.4.2 Questions for short answers

1. For what type of application would it be desirable for an array element to be a record?
2. What is the difference between array of records and record with element arrays?
3. What is WITH statement?
4. In what purpose is this statement used?
5. Summarize the rules of using WITH statement.

33.4.3 Analytical question
Arrays and Records

1. Identify errors of the following program segment.

PROGRAM sample (input, output);
    TYPE date = RECORD
        month : 1 .. 12;
        day : 1 .. 31;
        year : 1900 .. 2100
    END;
VAR birthday, today : date;
    count : 0 .. maxint;
BEGIN
    WITH today DO Write (month, day, year);
    count = 0;
    REPEAT
        WITH birthday DO
            Read (month, day, year);
            IF birthday = today THEN count := Succ (count);
        UNTIL year = 0;
    writeln (count);
END.
Lesson 34 : Sets

34.1 Learning Objectives

On completion of this lesson you will learn:

- concept of Set in Programming language
- different operations on Sets
- equality of Sets
- some examples indicating Set operations.

34.2 Concept of Set

In mathematics a set is a collection, group or class of objects. In Pascal we formally define a set as a collection of ordered, simple data items that are all of the same type. Thus a set may be a collection of integers or characters or enumerated data items. In order to utilize set concept we must first define a set type. We can then declare set type variables whose individual values are elements of that set type.

We begin by associating a group of ordered, simple type data items with a data type using a TYPE definition. This data type will be known as the base type. We can therefore establish the base type as

```pascal
TYPE base type = (data item1; … data itemn)
or, TYPE base type = first data item … last data item.
```

To declare a set type we use the following syntax:

```pascal
set type = SET OF base type;
```

To declare a set type variable use the following syntax:

```pascal
VAR set name : set type;
```

To declare a set of alphabet we use

```pascal
TYPE Letterset = SET OF 'A' .. 'Z';
VAR Vowels, consonants : Letterset;
```

To put elements into a set you must use an assignment statement:

```pascal
Vowels := ['A', 'E', 'I', 'O', 'V'];
```
Arrays and Records

We cannot access an individual element of a set but we can ask if a particular element is a member of a set variable. We can also do the standard set operations: union, intersection and difference.

Following are the operators used in PASCAL in relation to set operations.

`+` (Union) : The union of two set variables is a set made up of those elements which are in either or both.

`*` (Intersection) : The intersection of two set variables is a set made up of those elements occurring in both set variables.

`-` (Differences) : The difference between two set variables is a set made up of those elements in the first set variable but not in the second.

The following examples illustrate the three set operations. Let1 and Let2 are of type Letter set defined previously.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Results set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let 1 : = ['A', 'B', 'C'];</td>
<td>['A', 'B', 'C']</td>
</tr>
<tr>
<td>Let 2 : = Let 1 + ['X', 'Y', 'Z'];</td>
<td>['A', 'B', 'C', 'X', 'Y', 'Z']</td>
</tr>
<tr>
<td>Let 1 : = Let 1 - Let 2;</td>
<td>[ ] (empty set)</td>
</tr>
<tr>
<td>Let 1 : = Let 1 + ['X', 'Z'];</td>
<td>['X', 'Z']</td>
</tr>
<tr>
<td>Let 1 : = Let 1 + ['X', 'W'];</td>
<td>['X', 'W', 'Z']</td>
</tr>
<tr>
<td>Let 1 : = Let 1 * Let 2;</td>
<td>['X', 'Z'].</td>
</tr>
</tbody>
</table>

Relational operators together with the IN operator which is especially useful to test whether there exists a particular element in the set, is used in the processing of set and in each case they return TRUE or FALSE.

Following are the expressions with the result of operations:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return TRUE if</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET 1 = SET 2</td>
<td>SET 1 and SET 2 are identical</td>
</tr>
<tr>
<td>SET 1 &lt;&gt; SET 2</td>
<td>There is at least one element in SET1 not in SET 2 or there is at least one element in SET 2 not in SET 1.</td>
</tr>
<tr>
<td>SET 1 &lt;= SET 2</td>
<td>All the elements in SET 1 are in SET 2.</td>
</tr>
<tr>
<td>SET 1 &lt; SET 2</td>
<td>All the elements in SET 1 are in SET 2 and there is at least one element in SET 2 not in SET1.</td>
</tr>
<tr>
<td>SET 1 &gt;= SET2</td>
<td>All elements in SET 2 are in SET1.</td>
</tr>
<tr>
<td>SET 1 &gt; SET2</td>
<td>All the elements in SET2 are in SET1 and there is at least one element in SET1 not in SET2.</td>
</tr>
<tr>
<td>(element) IN SET 1</td>
<td>&lt; element &gt; is a member of SET 1.</td>
</tr>
</tbody>
</table>
34.3 Exercise

34.3.1 Multiple choice questions

1. The intersection of two set variable is a set made up of those elements occurring
   a) either set variable
   b) both set variable
   c) one set variable
   d) none of the above.

2. Which of the following is the assignment to set?
   a) \( V := \{ 'a', 'b', 'c' \} \)
   b) \( V := \{'a', 'b', 'c'\} \)
   c) \( V := ('a', 'b', 'c') \)
   d) none of the above.

3. '-' (Difference) operator means
   a) Mathematical difference in arithmetic
   b) Those elements who are available in 1st operand and not in 2nd operand
   c) Those elements who are available in 2nd operand and not in 1st operand
   d) none of the above.

34.3.2 Questions for short answers

1. What is the difference between concept of Set in mathematics and in Pascal?
2. What is the base type and how it is defined?
3. What is the set type and how it is defined?
4. Describe the operators used in Pascal in relation to set operation.
5. Describe the relational operators used for Pascal set.

34.3.3 Analytical question

1. What is IN operator? What is its use? Write down the results of the following Boolean expressions.
   \( X = [1,4,6] \quad Y = [6,7,8] \)
   a) \( [1,2,3]<>[3,2,1] \)
   b) \( [1,2,3]=[3,2,1] \)
   c) \( [1,2]<=[1,2,6] \)
   d) \( [1,5,7] >=[1,8,7] \)
   e) \( [5,7]<[7,8] \)
   f) 1 IN X
   g) \( X * Y >=Y \)
   h) \( X - Y = Y - X. \)
Unit 10 : Pointers and Recursion

Introduction

This unit discusses the dynamic method of programming. Firstly we will learn dynamic memory allocation. This is a very important concept. A separate type of data structure called pointer is provided by the Pascal compiler for this purpose. So first two lessons provides the declaration, referencing and practical use of pointer data type with examples. This is actually a very difficult concept for beginners. For this reason pictorial representation of pointers is also shown here. The last two lessons describe dynamic calling of subroutines by itself, that is recursion. Several rules to write recursive subroutines are described briefly. Execution flow of the recursive subroutines also listed with necessary examples.

Lesson 35 : Pointers-I

35.1 Learning Objectives

On completion of this lesson you will learn:

- introducing pointer data as a tool of dynamic data storage
- concept of pointer as address of static data
- defining pointers with examples.

35.2 Dynamic Variables

A static variable such as array is around as long as the part of the program (block) in which it is declared is executing. Which means that those variables declared in the VAR section of the main program are always there (during execution). Those variables declared in the VAR sections of procedures and functions are bound from the time they are called until control is passed back to the invoking routine.

PASCAL provides a mechanism for creating dynamic variables. That means you can define a type at compile time, but not actually create any variables of that type until run time.

This dynamic variables can be created or destroyed at any time during execution of the program. They can be defined as being of any simple or structured type.

We reference a dynamic variable not by name but through a pointer. A pointer is a variable which contains the address (location) in memory of
the dynamic variable it references. Every new dynamic variable created has an associated pointer to reference (select) it.

One of the difficulties of using array is that its size is predefined, and cannot be changed dynamically. This limitation may create a situation of wastage/shortage of memory.

We can use dynamic variables to overcome these problems of static variables like array. We can create new component for our list only when they are needed by using dynamic variables as components. By having each component contain the link or pointer to the next component in the list, we can create a dynamic data structure that can expand or contract as the program executes.

![List component (dynamic variable)](image)

Fig. 35.1 List of dynamic variables

**35.3 Pointers**
Pointers and Recursion

Dynamic variables are referenced through a pointer not by name. The pointer type is a predefined data type in PASCAL and (↑) is used to mention it.

Following is the declaration of some pointer type variable:

```
TYPE  Color  = (White, Black, Blue);
    Range = 1..25;

VAR   ptr1  : ↑Color;
      ptr2  : ↑Range;
      ptr3  : ↑char;
```

In the above declaration, ptr1, ptr2, and ptr3 are pointer variables as denoted by (↑) in their type definitions. The first declaration is read as "ptr1 is a pointer to a variable of type color". The declaration causes the compiler to name a memory location ptr1 the content of which will be a memory address.
35.4 Exercise

35.4.1 Multiple choice questions

1. Pointer is
   a) an address
   b) a memory location
   c) a data type which contains address of any location
   d) none of the above structure.

2. Problem of using array is
   a) array size must be mentioned previously
   b) array size can not be altered at any time
   c) deletion and insertion of elements in sorted list
   d) none of the above.

3. Pointer declaration syntax is given by
   a) VAR <ptr name> : *<type>
   b) VAR <ptr name> : <type>
   c) VAR <ptr name> : <type>
   d) none of the above.

35.4.2 Questions for short answers

1. What do you mean by a static variable? What are the problems of using static variables?
2. What is a dynamic variable? How do dynamic variables remove the problems for static variables?
3. What type of data structure is used to create linked structure in Pascal?
4. How is a pointer data type defined in Pascal?
5. What is meant by pointer data type?

35.4.3 Analytical question

1. Describe how the concept of dynamic data helps us in programming.
Lesson 36: Pointers-II

36.1 Learning Objectives

On completion of this lesson you will learn:

- allocation of memory address to previously defined pointer
- pictorial representation of pointers as a address of storage
- forming linked list dynamic data storage using pointer and records
- making allocated storage free
- few operations on pointer.

36.2 Referenced Variable

Referenced variables are accessed by a pointer variable not by the names. PASCAL provides a standard procedure NEW (ptr) to create a referenced variable. NEW (ptr) - creates a variable of the type referenced by pointer ptr, and stores the address of the new variable in ptr.

The statements

```
NEW (ptr1);
NEW (ptr2);
NEW (ptr3);
```

will put the address in ptr1, ptr2, and ptr3 of each of the newly created referenced variables, where ptr1, ptr2, ptr3 are defined in the previous lesson.

Any variable preceded by \( \uparrow \) operator gives the address. For example ptr.3 is a character. We can assign ptr3 = \( \uparrow \) ch, where ch is a character data declared in declaration part.

In order to access the newly created referenced variable we use the pointer variable followed by \( \uparrow \). For the referenced variables we just have created, following are valid assignments:

```
ptr1\( \uparrow \) := White;
ptr2\( \uparrow \) := 17;
ptr1\( \uparrow \) := 'A'.
```

Following figures show the results of these assignment.
PASCAL is a typed language. We cannot assign an integer value to a pointer, although value of pointer variable is numeric (integer in most machine). We can even not assign a pointer variable to another pointer variable if they do not refer to the same type of reference variable.

Pointers are important for creating dynamic data structures. As we need some way of linking each component to the next one in the structure, each component must contain a link or pointer that points to the next component in the structure. "Record" is ideal as a component of dynamic data structure. It can contain fields of different types with the pointer being one of them.

Following is such an example:

```
TYPE  ptrType  = ^student;

student = RECORD:
    name  :  ARRAY[1..20] OF char;
    next  :  ptrType
END;

VAR  ptr :  ptrType;

NEW(ptr);
```

An schematic diagram for above declaration and program segment is given below:

![Diagram of pointer data structure]

To create a link list of 20 elements we have to execute the following program segment:

```
New := ptr;
```
Pointers and Recursion

Head := ptr;
Tail := ptr;
FOR I = 1 TO 19 DO
BEGIN
  NEW (tempptr);
  Tail^.next := tempptr;
  Tail := Tail^.next
END;
Tail^.next := NIL;

The whole list is shown in the next figure.

36.3 Other Pointer Operations

When a dynamic variable such as ptr^ is no longer needed, DISPOSE(ptr) will return it so that memory spaces can be assigned later if needed. This process is called returning to available space.

Only assignment and a test for equality are legal operations on pointer variables. For example given these declarations.

```
TYPE Limit = 1..25;
VAR PTRA, PTRB : ^Limit;
```

We can do the following

```
NEW (PTRA);
NEW (PTRB);
READ(PTRA^, PTRB^);
WRITELN ('The sum of ', PTRA^: 1, ' AND', PTRB^: 1', ' is';
          PTRA^ + PTRB^: 1);
```

Consider the following operations :

```
PTRA^ := 18;
```
Then the comparison PTRA = PTRB evaluates to FALSE and PTRA<>PTRB evaluates to TRUE. But after the execution of the statement

PTRA = PTRB;
the PTRA = PTRB evaluates to TRUE and PTRA <> PTRB evaluates to FALSE.

The following diagram nicely explains the scenario:

Before execution of the statement PTRA = PTRB.

After execution of the statement PTRA = PTRB.

36.4 Exercise

36.4.1 Multiple choice questions

1. VAR ptr : ↑ char;
   ch : char;
   ch : = 5;
   ptr : =↑ch;

   What is the value of ptr ↑?

   a) Garbage
   b) 5
   c) NIL
   d) none of the above.

2. VAR PTR : ↑REAL;
   NEW (ptr);

   The number of bytes allocated for this declaration is

   a) 4 bytes
   b) 8 bytes
Pointers and Recursion

c) 2 bytes
d) none of the above.

3. VAR ptr : ↑ char;
   ch : char;
   ch := 6;
   ptr := ↑ ch;
   ptr↑ := 7;

   The value of ch is
   a) 6
   b) 5
c) garbage
d) none of the above.

4. What happens if dispose operation is done before NEW?
   a) Memory overflow error
   b) Memory underflow error
   c) System crash
   d) none of the above.

36.4.2 Questions for short answers

1. What is a referenced variable?
2. How are referenced variables declared?
3. Mention the rules and restrictions for pointer assignment.
4. Give the necessary declaration when linked list is formed.
5. What are the necessary tools used in Pascal for making the storage free in linked list?

36.4.3 Analytical questions

1. Give a pictorial representation of inserting deleting an element in the head, tail or middle of the list.
2. Write a program segment to create a link list of 20 elements.
Lesson 37 : Recursion-I

37.1 Learning Objectives

On completion of this lesson you will learn:

- a new concept of procedure calling, i.e., recursive calling
- use of recursion for determining recursive functions
- pictorial representation of recursive procedure call.

37.2 Concept of Recursion

Recursion is very powerful feature in PASCAL. Many programming languages do not support it. It is a little bit difficult to understand for the beginners but its power and compactness is attractive. By definition "The ability of a function or procedure to call or invoke itself is called recursion". Recursion is used in more advanced works. For the beginner, it is normally assumed that within a function we will not use the name of that function in the right side of any assignment statement. But recursion is such a technique where within a function we can use the name of that function in the right hand side of an assignment statement again (or rather call again). In other words a recursive function may call itself again and again.

Although operation of a recursive function is difficult to understand it will be comparatively easy to understand if we try to sketch the flow of control on the assumption that calling itself within a function is logically equivalent to calling another function only with the exception that the names of newly called function is same as the calling function.

37.3 Examples on Recursion

Let us start by giving an example of a function which calls another function, which in turn calls another function etc. The purpose of the following function FOURTH is to calculate the value of 4th power of its parameter. Thus the function FOURTH will return a value of $16 = 2^4$ if it is called with a parameter 2.

```pascal
FUNCTION fourth (x : integer) : integer;
VAR val16;
BEGIN
    val16 := x * third (x);
    fourth := val16;
END;

FUNCTION third (y : integer) : integer;
```

The ability of a function or procedure to call or invoke itself is called recursion. In other words a recursive function may call itself again and again.
Pointers and Recursion

VAR val8;
BEGIN
    val8 := y*second(y);
    third := val8
END;

FUNCTION second (z : integer) : integer;
VAR val4;
BEGIN
    val4 := z * first(z);
    second := val4
END;

FUNCTION first (p : integer) : integer;
VAR val2;
BEGIN
    val2 := p*1;
    first := val2
END;

Now let us define a recursive function power () which perform the same task more generally. Generally in the sense, we will make a call to power () with two parameter base and exponent. The function will return the value of the base raised to the power exponent. This means if the function is called with 2 and 5, it will return a value \(2^5=32\).

FUNCTION power (x, n : integer) : integer;
BEGIN
    if n = 1 then;
        power := x;
    else power := x * power (x, n - 1)
END;

37.4 Flow of Execution in Recursive Routine

We now see how this simple but powerful function calculates the value of \(X\) raised to the power \(N\).
We know

\[ X^N = X \times X \times X \times X \times X \times \ldots \times X \]

\rightarrow \ \text{N times} \rightarrow \n
= X \times (X \times X \times X \times \ldots \times X) \n
\rightarrow \ \text{N - 1 times} \rightarrow \n
= X \times X \times (X \times X \times X \times \ldots \times X) \n
\rightarrow \ \text{N - 2 times} \rightarrow \n
= X \times X \times X \times (X \times X \times X \times \ldots \times X) \n
\rightarrow \ \text{N - 3 times} \rightarrow \n
If we know the value of \( X^{N-1} \) we can easily calculate \( X^N \) because \( X^N = X \times X^{N-1} \). Similarly if we know the value of \( X^{N-2} \) we can easily calculate the value of \( X^{N-1} \) because \( X^{N-1} = X \times X^{N-2} \). Continuing this process we can go to the innermost product and \( X^1 = X \times X^0 = X \). This is the tricky process working here in the recursive function power( ).

Suppose power( ) is called with parameters 2, 3. The execution scenario would be as follows:

Initially the value of \( X = 2 \) and \( N = 3 \). So control will transfer to the 2nd statement (ELSE block). It will try to put a value in power. In order to calculate the right hand side of the statement it will see that right hand side invoke another function (although it is itself). At this situation the value of \( N \) is 3 and \( X \) is unchanged. So it will invoke power(2,2) because if evaluation of the function power(2,2) is not completed it can not multiply \( X \) with that evaluated value. So in order to execute its first task that is execution of the 2nd statement in the function on the first time it meets to evaluate and calculate the right hand side. So it will call power(2,2) blindly as if power( ) is another function (even if it's same!). One thing must be remembered that program is running and executing the first operation only and during this new call that is power(2,2), the original evaluation process is partly completed and control will have to return back to calculate \( X \times \text{power}(2,2) \) and then it will store this calculated value to power and then the execution of the function will be terminated.

Now to calculate power(2,2) it sees another task of power(2,1) should be performed to evaluate \( X \times \text{power}(2,1) \). So another task is pending in its hand (that is multiplying \( X \) with power(2,1)) and this task will have to be completed after getting return value from power(2,1) function call. The process continues like this and it maintains a chain of pending task. [When \( N \) becomes 2 then it is needed to calculate the value of \( X \times \text{power}(X,1) \) still it adds another pending task to its list and calls power(\( X,1 \)]. It immediately gets the success and needs no pending job to add into the pending list. Just after returning from the call power(2,1) with a returning value 2 in hand, it will perform the immediate pending task that is the task which was added in the pending list during the time it invoked
the function call power (2,1). This task is multiplying X with power (2,1). It will perform the task. This means it has only completed the function evaluation phase of its previous call that calculation of power (2,2) is just completed, so it will then perform that task which was added in the pending list during the time of invoking the function call power (2,2). This way it will perform all the pending job and the initial call (from outside the function power ( ) would be completed.

37.5 Pictorial Representation of Recursive Routine

Following figures best describe the above scenario:

Fig. 37.1 Recursive routine (subroutine)
37.6 Exercise

37.6.1 Multiple choice questions

1. Which programming language does not support recursion?
   a) C
   b) PASCAL
   c) PROLOG
   d) FORTRAN.

2. To calculate $2^{10}$ how many times the procedure power () is called internally?
   a) 10
   b) 11
   c) 9
   d) none of the above.

37.6.2 Questions for short answers

1. What do you mean by recursion?
2. What are the advantages and disadvantages of recursion?
3. Give a pictorial description of execution flow of a recursively called subroutine.

37.6.3 Analytical questions

1. 1, 1, 2, 3, 5, 8, 13 are called the numbers of fibonacci series. Write a program to find out $n$th Fibonacci number.
2. Write a recursive subroutine which can calculate $n!$. 


Lesson 38: Recursion-II

38.1 Learning Objectives

On completion of this lesson you will learn:

- different portions of recursive subroutine
- flow of working principle of recursive subroutine.

38.2 Exit path of Recursive Subroutine

Almost every recursive function has a similar structure. In each recursive function there must be an "exit" path and call to itself or induction. A recursive function calls itself with a subset or partial parameters or other instances of parameters. So each time it enters into the function and try to execute it makes another call to itself. Thus the process does not end if a suitable 'exit' path is not provided.

This 'exit' path is such statements which would be executed due to some conditional cases. That is, there must be alternative steps for the program control to be chosen instead of calling itself at least once during running of the program.

In our recursive example of the previous lesson we have used the statement

\[
\text{IF } N = 1 \\
\text{THEN } \text{power} := X
\]

This is actually the 'exit' path in the recursive function. A closer look at the function reveals that each recursive call made to the function power() is actually with two parameters one of which (i.e., N) is reduced in each call. This ensures that if we start with INTEGER N and \( N \geq 1 \) there must be a call with \( N = 1 \). At this point there is no need to call power( ) itself again and hence return from its previous call.

Once this return happens all pending jobs can easily be computed because all the pending jobs are only simple multiplications involving no call to itself.

38.3 Tabular Representation of a Recursive Subroutine

Table 39.1 presents the hypothetical diagram to understand the operation of \( \text{power}(2,3) \):

[For convenience we are rewriting the function power( ) with statement number for reference, this is only for clarification purpose and during]
expressing control flow we are using $\alpha$ as the future value of power (2,2) and $\beta$ as of power (2,1)].

FUNCTION power(X, N : INTEGER) : INTEGER;
BEGIN

1 : IF N = 1 THEN power := X

2 : ELSE power := power(X, N-1)

END;

Table 38.1

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Statement #</th>
<th>Trying to perform operation</th>
<th>Pending task</th>
<th>New call initiated</th>
<th>Storing value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,3</td>
<td>2</td>
<td>power = $\alpha$</td>
<td>X*$\alpha$</td>
<td>power(2.2)</td>
<td>NIL</td>
</tr>
<tr>
<td>2,2</td>
<td>2</td>
<td>power = $\beta$</td>
<td>X*$\beta$</td>
<td>power(2.1)</td>
<td>NIL</td>
</tr>
<tr>
<td>2,1</td>
<td>1</td>
<td>power = 2</td>
<td>NIL</td>
<td>NIL</td>
<td>$\beta=2$</td>
</tr>
<tr>
<td>--</td>
<td>2</td>
<td>Pending task $X*\beta$</td>
<td>NIL</td>
<td>NIL</td>
<td>$\alpha=2*2$</td>
</tr>
<tr>
<td>--</td>
<td>2</td>
<td>Pending task $X*\alpha$</td>
<td>NIL</td>
<td>NIL</td>
<td>POWER= $(2*2)^2$</td>
</tr>
</tbody>
</table>
38.4 Exercise

38.4.1 Multiple choice questions

1. Maximum number of recursive call of a function is
   a) One
   b) Two
   c) unlimited
   d) limited upto the program code size.

2. If there is no exit path in a recursive function then
   a) program will fall in infinite loop
   b) program will run correctly
   c) more time required to get result
   d) memory storage will terminate program.

3. The exit of recursive routine must have
   a) arithmetic expression
   b) assignment statement
   c) I/O statement
   d) condition checking.

38.4.2 Questions for short answers

1. What are the different parts of recursive versions?
2. What is exit path of recursive routine?
3. What is the common structure of "exit" portion of recursive function?

38.4.3 Analytical questions

1. Write a program which computes the factorial of any number using recursive routine.
2. Give a tabular or pictorial representation when factorial (4) is called in a program.
Unit 11 : Executing Programs

Introduction

While you are able to write a program you must run it by a compiler of that language. This unit discusses the process of running programs. First lesson of this unit discusses some common words about programming. This lesson describes good programming habit which increases programming speed of a quality programming. In the next lesson we will learn some technical aspects of programming. Facilities of debugging or correcting programs are also described in this lesson. This is actually a manual of compiler program. The last section describes different types of errors you can face in programming. The methods of recovering common errors are also shown here. Some suggestions for program writing are also given.

Lesson 39 : Preparing and Running a Program

39.1 Learning Objective

On completion of this lesson you will learn :

- concept of structural programming
- how better programs can be written.

39.2 Designing Your Program

Many books have been written about program design, but people are still arguing about the best ways to do it. Nevertheless, there are a few simple rules that you can follow to create a good design without having to become a master theoretician.

39.2.1 Never try to code first

First try to understand the problem deeply. Sketch out all the details of the problem step by step in normal English.

Example : Let us have a problem of finding out the area of circles of any radius.

Our steps would be as follows :
start program
clear the screen
state what your program will do
take input of radius from the user
calculate the area using formula
show it
wait for any key

Gradually refine this into real code.

39.2.2 Follow structured programming techniques

Subdivide the program’s tasks in outline form and plan to use subroutines for each divisions. Hide everything inside the subroutine from the other part of the program. This structured approach will give you tremendous help while you are trying to find a rather complex error inside the program.

Be careful about the variable declarations. Some variables may be required by several subroutines, these variables must be global; and you must be careful enough about the changed value of these variables in different parts of the program.

At this stage you need to decide what data structure you must use. You recall that data structures are parts of your program that hold data, and you design many of these structures yourself.

You will be able to follow all these features in the later lessons and now we are presenting a simple demonstration program.

```pascal
PROGRAM Area - Circle;

USES CRT; {for CLRSCR}

CONST
  pi = 3.14159;

VAR
  area, radious : real;

PROCEDURE Wait_for_a_key;
VAR
  ch : char;

BEGIN
  write ('Press any key to continue...');
  ch := READKEY;
  writeln
```
Executing Programs

END;

BEGIN

CLRSCR;
write ('enter the radius of a circle :');
read (radius);
area := pi * sqr (radius);
writeln;
write ('The area is :', area);
Wait_for_a_key

END.

You should recall the following points:

1. The semicolon is used as a separator in Pascal. Hence it is used between successive statements and declarations.

2. BEGIN and END are actually brackets that indicate the beginning and end of a compound statement. So after BEGIN a semicolon is not needed and before an END, semicolon is not needed.

3. Every complete program must be terminated by a period ( . ). Thus after the final END, you must put a period.

39.3 Some Notes on Better Programming Practice

Better programming is not something you learn from a class or a book—you learn by doing. And now we are going to discuss about some better programming styles.

- make CONST declarations for all numeric constants to make your programs easier to read.

- try to use comments in your program. This enhances readability of your program and later modifications become easier.

- try to make subroutines independent and hide everything inside it.
39.4 Exercise

39.4.1 Multiple choice questions

1. What is the purpose of subdividing programs?
   a) Hiding one portion from others for group work
   b) Reducing the size of program
   c) Decreasing execution time
   d) none of the above.

2. Semicolons are used as
   a) instructions
   b) I/O statements
   c) separator of statements
   d) none of the above.

3. Period (.) is used as
   a) termination of program
   b) termination of subroutine
   c) termination of statement
   d) separator of fields.

39.4.2 Questions for short answers

1. What are the problems of trying to code at first?
2. What do you mean by structural program design?
3. Why are comments necessary in program?
4. What is the use and advantage of constant declaration?
5. What are the reasons of using comments in a program?

39.4.3 Analytical question

1. Write a program using the process of program design in this chapter.
40.1 Learning Objectives

On completion of this lesson you will learn:

- idea of compiling
- process of compiling
- some techniques of tracing for finding errors.

40.2 Compiling

Your computer cannot understand ordinary words, either in English or Pascal. Before you can run your program, you have to compile it—that is, translate it into a form that your PC can understand and execute.

A variety of compiler is widely used and of course, most popular is the Turbo Pascal compiler. In Turbo Pascal Environment, press F2 to save the program you typed and to compile it, press ALT-F9. This will convert your program from words (source code) into object code, which your PC will understand.

No matter how carefully you have checked your program, you will experience the error free compilation at first time with very rare success. Almost always the compiler will find some problem in the program: a misspelled command word, a syntax error, a mistake in punctuation, a missing argument, or anything else that would prevent the program from running correctly. These are called compile-time errors.

Once you have corrected all the errors (if there were any), you can press ALT-F9 again to compile the program. You surely would be cheered up with the message: ‘Compile successful: press any key’.

40.3 Running

To run your program now, press CTRL-F9. This command actually makes the execution file. From compile menu there are two options of making execution file. One is to make EXE file in memory which is used to run file from Turbo Pascal environment. To run the program from DOS environment we have to mention the destination of EXE file in the "Compile" menu and destination submenu.

40.4 Tracing
If the program behaves unusually then we have to trace the program carefully. There are several rules of tracing. These are given below:

**F7 key** : By this key the program halts after execution of each statement and waits for F7 key.

**F8 key** : This key is similar to F7 key but ignores all subroutine calls. For F7 key the tracing inside particular subroutine is going on when a subroutine call statement is found.

**F4 key** : Place the cursor at the line which you want to observe and press the key. The tracing program will run from this line. Other lines will not be traced.

**CTRL F7 key** : This key is used to watch the condition of any variable. It invokes an editor to write the variable name. This variable is placed in the watched window with their values they possess just now.

**CTRL F8 key** : This key is used for fixing some break points. While running program, it halts at the breakpoints. For marking breakpoint place the cursor at the desired line and press the CTRL F8 key.

### 40.5 Exercise

#### 40.5.1 Multiple choice questions

1. What is the purpose of ALT - F9 key?
   a) Running the program
   b) Debugging program
   c) making the object code of the source code
   d) none of the above.

2. You cannot run program before
   a) correcting logical errors
   b) correcting your data structure
   c) correcting your compiling errors
   d) none of the above.

3. Break points are given by
   a) CTRL F8
   b) F8
   c) SHIFT F8
   d) all of the above.
Executing Programs

4. What is the purpose of CTRL - F7 key?
   a) Debugging program
   b) Watching the value of constants for modifying
   c) Watching the value of variables for correcting programs
   d) all of the above.

5. Break points are placed
   a) where errors can be determined
   b) where errors can be occurred
   c) all of the above
   d) non of the above.

40.5.2 Questions for short answers

1. What are the different phases of running a program?
2. Why is tracing done?
3. Describe different facilities for tracing.
4. What are the differences in tracing by F7 and F8?
5. What actions should be taken for running programs in DOS environment written by Pascal syntax?

40.5.3 Analytical question

1. Write down some common errors you have faced in running the perfect number assignment in previous chapter.
Lesson 41: Error Diagnostics

41.1 Learning Objective

On completion of this lesson you will learn:

- different types of errors
- possible sources of errors
- how these errors can be detected.

41.2 The Study of Bugs

‘Bug’ is just a name for an error in your program. Anything that prevents a program from compiling and running correctly is a bug. Some program bugs are catastrophic in their effects, while others are not so dangerous and still others are so obscure that no one will ever discover them.

There are three main types of errors:

1. SYNTAX errors
2. RUN-TIME errors
3. LOGIC errors.

41.3 Syntax Errors

The compiler can easily detect the syntax errors. Usually you get an error message and the line of your program causing this error is highlighted; and the cursor stays at that line. Again there may be some special occasions when the cursor position is not the true location of the error at all.

Now let's have a look at some of the most common syntax errors in Pascal and also discuss what to do with them.

41.3.1 ";" Expected

It is one of the most common bugs. You know, semicolons are used to separate statements from other statements in a Pascal program. To correct this bug, just add a semicolon at the end of the line from which it is missing. Normally the compilation error occurs on the line immediately after the line without the semicolon.

41.3.2 Unexpected end of file
Executing Programs

The compiler keeps track of nested BEGIN..END pairs, so it knows when the main body of the program begins and when it ends. In a unit or a program, you will get this error if the final END statement is missing.

If you don’t know why you got this message, check your comments, starting at the end of the program and working back toward the start. An open comment with no other comments after it will cause the full last part of the program to disappear from the compiler's view. So check it!

41.4 Error in Statement

When you get this error message, first check the highlighted statement to make sure that it is not the source of the trouble. If it is not, somewhere above that statement, another statement or a comment is not terminated properly: you may have too few ENDS, mixed-up comment brackets, or some variation on those errors. All you can do against this type of error is to have a deep knowledge of the rules for constructing statements and expressions in Pascal (the syntax of Pascal).

41.4.1 Type identifier expected

You might have attempted to define a type in the header of a procedure or function, for example

```pascal
PROCEDURE testing (S : String[35]);
```

The only types you can use in parameter lists are types that have already been defined in your program. In this example, define a type

```pascal
String35 = STRING[35]
```

and use that type in your procedure definition.

41.4.2 ',' Expected

In general, this means you haven't passed all of the needed parameters to a procedure or function you're calling. If the compiler knows the routine takes three parameters, then it expects a comma after the second.

41.4.3 ')' expected

You'll get this message if you pass too many parameters to a procedure or function call. It will also come up if you accidentally separate two parameters in the header of a procedure or function with a colon instead of a semicolon.

41.4.4 '=' expected
After typing a lot of VAR declarations, it’s easy to forget and use a colon instead of an equal sign when writing a TYPE declaration. That is the common cause of this error.

### 41.4.5 Ordinal type expected

Ordinal types are those that have a direct correspondence with whole numbers. Byte, ShortInt, Word, Integer, and LongInt types are ordinal, and so are any subranges of these types. The remaining ordinal types are Char, Boolean, and user defined enumerated types. Floating-point types, arrays, sets, files, records, strings, and pointers are not ordinal. If you attempt to use a nonordinal type in a Set declaration, you’ll get this error.

### 41.5 Run-time errors

Your program may be syntactically correct, but when it runs an error turns up. For example, there is nothing that the compiler would find wrong with dividing one variable by another; but if the divisor becomes zero in the course of the program, that is a fatal error. Some run-time errors may cause a crash! Unexpected input values, invalid calculations, flawed program logic, disk access problems, inappropriate screen output commands all of these can cause run-time errors.

#### 41.5.1 Attempt to Assign Out-of-Range Value

This error can occur when you are using numeric types such as bytes, which have very limited range of values. It also can occur with integers, enumerated types, and subrange types.

For example, let us have a variable of type byte named “Val”, an attempt of assigning a value ‘550’ to it causes a run-time error. As ‘Val’ can take values of only 0 to 255.

#### 41.5.2 Failure to Initialize Variable Before Use

Some variables need to be initialized at the beginning of a program or subroutine; other variables do not.

The only type of variable that absolutely must be initialized is the pointer. An uninitialized pointer variable might contain the memory address of part of the operating system or part of Pascal! And the result would be very dangerous. All pointer variables should be set to NIL at the beginning of a program or subroutine, and should be set to NIL again as soon as they are disposed of and are no longer needed.

Variables used to control WHILE and REPEAT loops should be initialized.
Executing Programs

41.5.3 Nonmatch in CASE statement: failure to handle it

Consider the following example:

CASE choice of

1 : Do Something1;
2 : Do Something2;
3 : Do Something3;

END;

As long as the user enters 1 to 3, everything is fine. However, if the user enters 4, then the program will stop with a run-time error. So you have to write some code to handle such situation. For example, you can write an ELSE clause with a CASE statement.

41.5.4 Hooray for error messages

Error messages are personal messages to you, the programmer. By gaining a thorough understanding of them, you become a more effective programmer. You also gain a more complete knowledge of the abilities and quirks of the compiler itself. You don’t have to memorize the list of error messages, but be sure you take advantage of what the compiler tells you.

41.6 Logic Errors

Now, what is a logic error? If you are like me, you should say to your computer, ‘Do what I mean, not what I say’. Remember, the computer does exactly what it’s told. And the logic error can be understood by observing the difference between what the computer did and what you said.

Logic errors occur when the programmer analyzes a problem incorrectly and comes up with the wrong solution. The best way to avoid or minimize logic errors is to keep your code as simple as possible and also by following structured programming style (as described before).

Example:

See carefully the following two programs:

```{----------------------------------------------------}
PROGRAM  no_ 1
VAR
```
Computer Programming

a, b, c: INTEGER;

PROCEDURE Add_two_num (a, b, c: integer);
BEGIN
  c := a + b
END;

BEGIN  {main body of program}
  a := 1;
  b := 1;
  c := 0;
  Add_two_num (a, b, c)
{GUESS : what would be the value of c}
END.

{----------------------------------------------------}
PROGRAM no_2

VAR a, b, c: INTEGER;

PROCEDURE Add_two_num (a, b: INTEGER; VAR c: INTEGER);
BEGIN
  c := a + b
END;

BEGIN  {main body of program}
  a := 1;
  b := 1;
  c := 0;
  Add_two_num (a, b, c)
  {GUESS : what would be the value of c}
END.

Surely you wish to add a and b and to store the value to c. But in the first case c equals 0 and in the second case c equals 2.

So, any variables that are to be changed by a subroutine must be passed to it as VAR parameters.

41.6.1 Error in Loop Iteration Number

You must be careful enough in using WHILE and REPEAT loops. Novice Pascal programmers often make mistake in guessing the number of times their loops would execute.

Remember, a WHILE statement evaluates its loop control variable before going through the loop each time. A REPEAT statement, on the other
Executing Programs

hand, evaluates the control variable at the end of each loop, so no matter
what the initial value of the control variable, a REPEAT loop will always
execute once.

41.6.2 Detecting logical errors

Logical errors are the most difficult kind of errors to locate. This can
sometimes be accomplished by testing a new program with data that will
yield a known answer. If the correct results are not obtained, then the
program obviously contains errors. Even if the correct results are
obtained, however, one cannot be absolutely certain that the program is
error-free, since some errors cause incorrect results only under certain
circumstances (as, for example, with certain values of the input data or
with certain program options). So a new program should be thoroughly
tested. This is especially true of complicated programs or programs that
will be used extensively by others.

As a rule, a calculation will have to be carried out by hand, with the aid
of a calculator, in order to obtain a known answer. for some problems,
however, the amount work involved in carrying out a hand calculation is
problematic.

To detect logical errors almost every current Pascal compilers are well
equipped with a debugger. With the debugger, you can trace the
complete flow of the program, and you can see how each variable
changes every step of the way. In previous chapter several commands
have shown already.

41.7 Final Words

Errors both large and small are a fact of programming life. Strangely
enough, the big errors are usually the easiest to correct. Sometimes, it is
difficult even to be sure that a small error is present until the program
crashes, of course. When you ask programmers how they learnt the
process of debugging, you are likely to get a single word in response :
experience.

41.8 Exercise

41.8.1 Multiple choice questions

1. Which of the following is not a syntax error?
   a) ';' expected
   b) ',' expected
   c) function 'sqr' not found
   d) ')' expected.
2. Logical errors can be detected by only
   a) operating system
   b) programmer
   c) compiler
   d) file server.

3. Which of the following data type must be initialized before using?
   a) Pointer
   b) Character
   c) Integer
   d) Array.

4. Generally errors occurred in WHILE and REPEAT loops. This is due to misunderstanding on
   a) number of iterations
   b) concept of semicolons
   c) concept of BEGIN - END
   d) none of the above.

41.8.2 Questions for short answers

1. What do you mean by ordinal type? Give examples.
2. Which type of bug is the most dangerous and why?
3. Explain why structured programming methods make easier debugging process.
4. How is the semicolon used in Pascal? Where must it appear?
5. What is the purpose of BEGIN and END?

41.8.3 Analytical question

1. How many type of logical and syntax errors you can find while programming? Explain briefly.
Unit 12: Programming Style

Introduction

In this unit we introduce some developed idea on programming style. We are going to present about structured and unstructured programming. In structured programming we shall present about design methodologies, such as: top-down, object oriented and data-structured design. Besides, we shall introduce also modular programming in this unit.

Lesson 42: Structured and Unstructured Programming

42.1 Learning Objectives

On completion of this lesson you will learn:

- what structured programming is
- what the benefits of structured programming are.

42.2 Concept of Modularity

To accomplish any complex task successfully we follow some methods. For example, to perform a complex task it is a common practice to decompose it into simpler logically connected subtasks, each of which is much easier to perform. We do it because it is easier to complete the job this way, because if something goes wrong, or if some modification is necessary in changed circumstances we can effectively do it bringing in necessary changes only in the concerned subtasks. Modularity is measured in terms of two criteria: coupling and cohesion.

42.3 Adaptability

We know that computers are very efficient in performing repetitive tasks. But in real life quite often we have to perform a job not exactly identical. Consider, for example, the process of importing goods from abroad. The whole process may not be identical if there is some change in import policy. However, most of the process remains identical with some changes here and there. If we have to automate the process of importing goods we must keep provisions of bringing in changes in subtasks that may have to be modified due to changes in policy. This means we must make the automated process easily adaptive to probable changes that can take place. In fact, the process should be designed in such a way that expected changes can be accommodated by modifying the corresponding modules keeping other modules unaffected. For any automated system...
adaptability is a very desirable property. In order to enjoy these facilities during and after the system building, we must formulate criteria for decomposing a system into a set of well-organized modules. Several design methodologies provide such decomposition criteria. These include:

1. Structured (top-down) design
2. Object-oriented design
3. Data-structured design.

42.4 Structured Design Technique

The design technique that has probably received the most attention and popularity is referred to as structured design. This technique, as described by Larry Constantine, and further advanced by Edward Yourdon and Genford Myers, uses functional decomposition of a system into a set of well-structured modules. We have discussed other moderating is the paragraph 42.2. The main goal of structured design is to generate a system structure in which the modules are loosely coupled and highly cohesive.

It is desirable that the higher-level modules be relatively machine-independent, that is, can be performed in any machine without modification. Usually each higher-level module specifies what actions to be taken. A lower-level module, on the other hand, specifies how the action is to be performed. Important decisions are made at the top of the hierarchy, while less important decisions are made at lower levels. The methodology is based on data flow in, out and through a system. This flow of data (and its associated processing) is represented by data flow diagrams. Designs are obtained from these data-flow diagrams.

There are two main strategies for generating a system (or module) structure that matches the problem structure as described by data-flow diagrams. In the first method, each independent data flow is regarded as a transaction that requires a special response by the system. The structure of a design contains a transaction centre, which consists of modules that handle each type of transaction. This approach is called transaction analysis or transaction-centred design. A second approach, transform analysis, involves identification of important transformations (or functions) in the system. Such transformations become a transform centre in the design structure. Structured design also provides other criteria (besides coupling and cohesion) for evaluating the quality of a design. Structured design focuses primarily on the operations or functions performed by a system, with less attention being given to the design of data structures.

42.5 The Necessity of Structured Programming
The necessity of structured programming can be well understood from the statistics which asserts that even good programmers use 15% of their time in program development, whereas the other 85% time is spent on program debugging. It may be mentioned here that maintenance of a software system requires approximately five times the effort spent on its development. These figures indicates the importance of following a design technique, in which both the tasks of debugging and maintenance can be performed easily. Modularization of the system helps to perform these jobs efficiently. Coupling and cohesion are very important criteria for modularization. To explain the terms let us consider the following example of an electronic equipment. You will see several boxes in it housing components used for performing some function. While components housed in each box have a lot of connections amongst themselves, the number of communications between boxes is not very big. This helps to understand easily how the system works. Therefore, it becomes easier to carry out troubleshooting in such systems. In programming too, it is easier to understand the flow of a program if the main program consists of a small number of statements, each of which is responsible for performing the logically related operations, and there are not too many communications interference's among modules. These programs more or less relocate our thought process and, therefore, we can easily understand such programs, and carry out necessary modifications. In unstructured programs of these properties are absent. While structured programs have a flow forward, unstructured programs usually contain statements with backward flows, which does not conform with our logical thinking. Flowcharts corresponding to unstructured programs contain too many arcs going backwards. These usually correspond to GOTO-like statements. However, structured programming does not necessarily allow the use of GOTO-like statements. But too many uses of such statements indicate violation of this property. Early programming languages like FORTRAN-2 or FORTRAN-4 used to have these features. However, recent versions of FORTRAN like FORTRAN-77 or programming languages like PASCAL, C advocates for structured programming styles, and supports it with a variety of constructs and facilities.
42.6 Exercise

42.6.1 Multiple choice questions

1. Modularity is measured in terms of two criteria
   a) data flow diagram and systems analysis
   b) coupling and cohesion
   c) data structure and flowchart
   d) efficiency and simplicity.

2. The main goal of structured design is to generate a system structure in which the modules are
   a) loosely coupled and highly cohesive
   b) loosely coupled and lowly cohesive
   c) highly coupled and highly cohesive
   d) highly coupled and lowly cohesive.

42.6.2 Questions for short answers

1. What do you mean by structured design?
2. What are the two criteria of modularity?
3. What is data-flow-diagram (DFD)?
4. What is modularity?
5. What is transaction analysis?

42.6.3 Analytical question

1. What is the difference between structured and unstructured programming?
Lesson 43: Top-Down and Bottom-Up Programming

43.1 Learning Objectives

On completion of this lesson you will learn:

- definition of top-down and bottom-up programming
- merits and demerits of top-down and bottom-up programming.

Let us illustrate the two techniques top-down and bottom-up with an example - the game of matching coins. In this game, two boys, say Rahim and Karim, spin a coin. If the coins match (both heads or both tails) Rahim wins; otherwise Karim wins. We will use the computer to determine who has won the game.

Assuming Rahim Toss and Karim Toss to have the values 'H' or 'T' depending upon whether 'Head' or 'Tail' comes up in their spin we may organize our program as follows:

```pascal
PROGRAM CoinMatch(INPUT,OUTPUT);
VAR Rahim_Toss, Karim_Toss : char;
BEGIN
  WRITELN('Input Value of Rahims_Toss');
  READLN(Rahim_Toss);
  WRITELN('Input Value of Karims_Toss');
  READLN(Karims_Toss);
  IF Rahim_Toss := Karim_Toss THEN WRITELN(Rahim wins)
                                   ELSE WRITELN(Karim wins)
END {CoinMatch}
```

Let us extend this example to allow several consecutive games to be played. Suppose Rahim is playing for a Taka a game and will continue to play until he wins Taka 100 or loses everything starting with Taka 50. Clearly we would like to execute the same basic section of code repeatedly, keeping track of his fortune until Rahim reaches his goal. Since a program executed by a computer is stored in the computer’s memory prior to execution, it is possible for any section of code to be re-executed as often as desired. This is called a program loop. Programming languages provide ways of describing loops. In Pascal, one way is the repeat until statement. For this problem it takes the form

```pascal
REPEAT
  statements to handle one play
UNTIL condition
```

211
Computer Programming

until Rahim’s Money : = 0 or Rahim’sMoney = 100

where Rahim’s money is the amount of Taka currently possessed by Rahim during the game. The code to handle one play must change the value of Rahim’s Money up or down by one Taka. The variable Rahim’s Money has been declared as an INTEGER and is increased by the statement Rahim’s Money := Rahim’s Money + 1. So final program looks like

PROGRAM CoinMatching(INPUT,OUTPUT);

VAR Rahim’s_Toss, Karim’s_Toss : char;
    Rahim’s Money : integer;

BEGIN
    Rahim’s Money := 50;
    WRITELN(‘Input Value of Rahim’s_Toss’);
    READLN(Rahim’s_Toss);
    WRITELN(‘Input Value of Karim’s_Toss’);
    READLN( Karim’s_Toss);
    IF Rahim’s Toss = Karim’s_Toss
        THEN Rahim’s Money := Rahim’s Money + 1
        ELSE Rahim’s Money := Rahim’s Money - 1;
    WRITELN(‘Rahim’s winning are’, Rahim’s Money);
    UNTIL (Rahim’s Money = 100) or (Rahim’s Money = 0)
END {CoinMatch}

In this example, we developed the final version of the program by bottom-up design: we first designed a code to handle a single case and then built the rest of the program around it. (It might seem more appropriate to call this ‘inside-out’ development, but the details are viewed as being part of the ‘lowest level’. For this and other reasons, the name ‘bottom-up’ is used.) If we had initially been given the second version of the problem, that is the game should be continued until Rahim is either pennyless or possesses taka 100, we would probably have used top-down design, first planning the structure of the program using a loop, as in

INITIALIZE Rahim’s Money to 50
REPEAT
    play one game, increasing or decreasing Rahim’s Money
    output the current value of winnings
UNTIL Rahim’s Money : = 100

Having seen the form of code needed in the loop, we could program it directly.
Programming Style

Usually, top-down design leads to a clearer programs. The top or outer levels are designed first, uncluttered with the details of the inner or lower levels, which are left to be designed later. However, there are some important provisions. Code must eventually be written for each lower level that is left to be completed later. If it subsequently becomes impossible to write such a code, the higher levels will have to be redesigned. If it is not clear that a lower level is possible, the programmer must first examine that lower level and, if necessary, design it completely. Consequently, top-down design becomes useful only after a programmer has had enough experience to recognize what is possible and practical. Beginning programmers should start with bottom-up design and change slowly as they accumulate experience. Even experienced programmers will encounter tasks in which top-down must give way temporarily to bottom-up.

In top-down design it is often desirable to write an incomplete program sketch, such as the example above. The descriptions of sections of code yet to be written can be given as comments by surrounding them with braces. If left in the final version, they form valuable documentation. However, the beginning programmer should endeavor to write all programs immediately in the language being used since this is a valuable practice in using the language to express thoughts. Programs should be sketched in a free-form way only after a measure of fluency in the language has been achieved.
43.2 Exercise

43.2.1 Multiple choice questions

1. In top-down design it is often desirable to
   a) write an incomplete program sketch
   b) write individual modules first
   c) do the documentation first
   d) code first.

2. Programs should be sketched in a free-form way
   a) only after a measure of fluency in the language has been achieved
   b) only when there is a lack of knowledge in programming language
   c) only by beginning programmers
   d) none of the above.

43.2.2 Questions for short answers

1. What is top-down programming?
2. When do we use bottom-up approach?
3. When do we use top-down programming?
4. What is program loop?
Lesson 44: Modular Programming

44.1 Learning Objectives

On completion of this lesson you will learn:

- the concept of modular programming and its advantages
- the related concepts of module coupling and module cohesion.

Although it is generally agreed that any large system, and hence any large program, should be divided or partitioned into modules if it is manageable, this division is no guarantee of a well-organized system. It is imperative, however, that this partitioning be carried out so as to make the modules independent of one another as much as possible. Let us first examine this criterion of module independence, that is, module coupling. A second criterion for partitioning a system is that each module should carry out a single, problem-related function. We shall examine this criterion of module cohesion next. In addition to the criteria of coupling and cohesion, several other criteria or guidelines can be used to evaluate and improve a design. A discussion of several of these guidelines will be discussed at the end of this lesson.

44.2 Module Coupling

*Coupling* refers to the degree of interdependence among modules. The degree of coupling can be used to evaluate the quality of a system design. We want to minimize coupling among modules, that is minimize their interdependence. One extreme in coupling would be to have a system design (hierarchy of modules) totally uncoupled. Since a system must perform some sets of functions or tasks in some organized fashion, it cannot consist of a set of totally uncoupled modules. The other extreme would be to have a hierarchy of modules tightly coupled; that is, there is a high degree of dependence between each pair of modules in the design. Between these two extremes, there are many degrees of coupling.

Having a loosely coupled system facilities:

1. The replacement of one module by another so that only a few modules will be affected by the change

2. The tracking down of an error and the isolation of the defective module causing that error.
Having established the desirability of loosely coupled systems, we examine now the various kinds of coupling that can exist between two modules. Such a classification is given in Table 3.1.

Table 3.1 Classification of Module Coupling

<table>
<thead>
<tr>
<th>Type of Coupling</th>
<th>Degree of Coupling</th>
<th>Degree of Maintainability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>High (Tight)</td>
<td>Low</td>
</tr>
<tr>
<td>Common</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stamp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>Low (Loose)</td>
<td>High</td>
</tr>
</tbody>
</table>

We examine the five types of coupling, from the least desirable (that is, tightest coupling) to the most desirable (that is, loosest coupling). The strength of coupling between two modules is influenced by the type of connection, the type of communication between them, and the overall complexity of their interface.

44.2.1 Content Coupling

As indicated in the table above, certain types of coupling are more desirable than others. The least desirable type is content coupling. Two modules exhibit content coupling if one module alters the other. In such a situation, the modified module is highly dependent on the modifying module. Examples of content coupling are when one module directly alters internal data or statement within the other. Content coupling also occurs when one module transfers control into the middle of a routine in another module.

44.2.2 Common Coupling

The degree of coupling can be reduced by placing data used by two or more modules in a global or common declarative area, such as variables declared in an outermost Pascal block, a FORTRAN COMMON block, or the accessible portion of an Ada programming language package dependence among modules is still strong, since a change in the common data are potentially affects all modules that share the data. This type of coupling is known as common coupling.

44.2.3 Control Coupling

Two modules are said to be control-coupled if one passes data items, called control flags, to the other in order to control the latter’s inner logic. For example, assume that we have a module that can perform either an insertion or a retrieval in a given data structure such as an array. One of the parameters required by this module is a Boolean control flag. A value of true might denote an insertion and false a retrieval. If a
subordinate module passes a control flag to superordinate, then the former controls the operation of the latter giving rise to an undesirable situation.

44.2.4 Stamp Coupling

Two modules are said to be *stamp-coupled* if a data structure is used as a communication medium. Both modules refer to the same data structure and require a knowledge of its form and contents. Stamp coupling is similar to common coupling except that coupled modules share the data selectively. As a result a module often gets exposed to more data than it requires to perform its assigned task.

44.2.5 Data Coupling

Two modules are termed *data-coupled* if they communicate through a parameter list, each parameter in the list being either a single data item or a homogenous table (such as a vector of numbers). This is the simplest and most desirable kind of coupling between two modules.

44.3 Module Cohesion

Coupling is one of the criteria for evaluating how well a system has been modularized. This criterion suggests that a well-modularized system is one in which the module interfaces are clear and simple. Another criterion for judging a design is to examine each module in a system, and determine the strength of binding within that module. The internal strength of a module, that is, how strongly related the parts of a module are, is referred to as *cohesion*. A module whose parts are strongly related to each other and to the purpose or function of that module is said to be strongly cohesive. On the other hand, a module whose parts are not related to each other is weakly cohesive.

In addition to module cohesion there are other forms of cohesion, like coincidental cohesion when parts of a module are completely unrelated to one another, logical cohesion when elements are related to a number of tasks of the same general kind or category (for example module handling all input of a program), temporal cohesion when elements are involved in various activities that are related only in time, procedural cohesion which occurs when the parts of a module may have to be performed in a certain order. Moreover, there are communicational, sequential, functional and informational cohesion's. In order to classify cohesion the following procedure can be very useful.

If a module performs one problem-related activity then if this activity is one of a set of activities associated with a complex data structure
then we have informational cohesion
else we have functional cohesion
else if the activities are related by data
then if the sequencing of activities is important
then we have sequential cohesion
else we have communicational cohesion
else if the activities are related by flow of control
then if sequencing is important
then we have procedural cohesion
else we have temporal cohesion
else if the activities are of the same general kind
then we have logical cohesion
else we have coincidental cohesion

The following are some additional guidelines for modularization:

Implementation of a module should not exceed one or two pages.

The fan-out (number of subordinate modules) of a module should be within $7\pm2$.

The fan-in (number of modules which calls this module.) of a module should be as high as possible.
44.4 Exercise

44.5.1 Multiple choice questions

1. Implementation of a module should not exceed
   a) one or two pages
   b) five pages
   c) three subroutines or functions
   d) none of the above.

2. The fan-out of a module should be within
   a) $7 \pm 2$
   b) $8 \pm 2$
   c) $9 \pm 2$
   d) $11 \pm 2$.

3. The fan-in of a module should be
   a) as low as possible
   b) as high as possible
   c) as short as possible
   d) none of the above.

44.4.2 Questions for short answers

1. What do you mean by coupling?
2. What is cohesion?
3. What is data coupling?
4. What do you mean by common coupling?
5. How many types of coupling are there in this lesson?

44.4.3 Analytical questions

1. Write short notes on different types of cohesion.
2. Write short notes on different types of coupling.
Answers to MCQS:

Unit 1:
Lesson 1:
1. d, 2. a, 3. b
Lesson 2:
1. c, 2. a, 3. c, 4. c, 5. a

Unit 5:
Lesson 16:
1. a, 2. a, 3. c, 4. d, 5. a, 6. a
Lesson 17:
1. d, 2. b, 3. a
Lesson 18:
1. a, 2. b, 3. d, 4. d

Unit 9:
Lesson 30:
1. i) b, ii) b, iii) a, iv) a, 2. a, 3. c
Lesson 31:
1. a, 2. c, 3. i) b, ii) c, iii) b
Lesson 32:
1. b, 2. d, 3. c
Lesson 33:
1. b, 2. a, 3. a, 4. c
Lesson 34:
1. b, 2. b, 3. b

Unit 2:
Lesson 3:
1. a, 2. d, 3. a
Lesson 4:
1. d, 2. b, 3. a
Lesson 5:
1. c, 2. d, 3. b
Lesson 6:
1. b, 2. d, 3. a, 4. c
Lesson 7:
1. a, 2. a, 3. c, 4. b, 5. d

Unit 6:
Lesson 19:
1. b, 2. c, 3. d, 4. d, 5. c, 6. a
Lesson 20:
1. a, 2. a, 3. d, 4. a
Lesson 21:
1. d, 2. a, 3. a
Lesson 22:
1. c, 2. b, 3. c, 4. a, 5. a

Unit 10:
Lesson 35:
1. c, 2. a, 3. c
Lesson 36:
1. b, 2. c, 3. a, 4. c
Lesson 37:
1. d, 2. a
Lesson 38:
1. c, 2. a, 3. b

Unit 3:
Lesson 8:
1. a, 2. c, 3. a, 4. d, 5. a
Lesson 9:
1. c, 2. c, 3. a, 4. c, 5. c
Lesson 10:
1. c, 2. a, 3. d, 4. d
Lesson 11:
1. b, 2. c, 3. a, 4. a

Unit 7:
Lesson 23:
1. b, 2. c, 3. c
Lesson 24:
1. c, 2. b, 3. a, 4. b
Lesson 25:
1. a, 2. d, 3. b, 4. b
Lesson 26:
1. a, 2. d, 3. b, 4. c

Unit 11:
Lesson 39:
1. a, 2. c, 3. a
Lesson 40:
1. c, 2. c, 3. a, 4. c
Lesson 41:
1. c, 2. b, 3. a, 4. a

Unit 4:
Lesson 12:
1. d, 2. a, 3. b
Lesson 13:
1. d, 2. c, 3. a
Lesson 14:
1. a, 2. c, 3. c, 4. c
Lesson 15:
1. a, 2. c, 3. d, 4. d, 5. c, 6. a

Unit 8:
Lesson 27:
1. a, 2. c, 3. b
Lesson 28:
1. c, 2. c, 3. b, 4. c
Lesson 29:
1. a, 2. a, 3. d, 4. d

Unit 12:
Lesson 42:
1. b, 2. a
Lesson 43:
1. a, 2. a
Lesson 44:
1. a, 2. a, 3. b
FURTHER READING


