M.Sc. Final Year
Mathematics
Option (I)

PROGRAMMING IN C

मध्यप्रदेश भोज (मुक्त) विश्वविद्यालय – भोपाल

MADHYA PRADESH BHOJ (OPEN) UNIVERSITY - BHOPAL

Reviewer Committee

1. Dr Anjana Yadav
   Assistant Professor
   IEHE College, Bhopal

2. Dr Anil Rajput
   Professor
   Govt C.S.Azad (PG) College, Sehore

3. Dr Rajkumar Bhimte
   Assistant Professor
   Govt College, Vidisha, MP

Advisory Committee

1. Dr Jayant Sonwalkar
   Hon'ble Vice Chancellor
   Madhya Pradesh Bhoj (Open) University, Bhopal

2. Dr L.S.Solanki
   Registrar
   Madhya Pradesh Bhoj (Open) University, Bhopal

3. Dr Neelam Wasnik
   Dy Director Printing
   Madhya Pradesh Bhoj (Open) University, Bhopal

4. Dr Anjana Yadav
   Assistant Professor
   IEHE College, Bhopal

5. Dr Anil Rajput
   Professor
   Govt C.S.Azad (PG) College, Sehore

6. Dr Rajkumar Bhimte
   Asst. Professor
   Govt College, Vidisha, MP

COURSE WRITERS

Rohit Khurana, Faculty and Head, ITL Education Solutions Ltd., New Delhi
Units: (1.0-1.2, 1.2.2, 1.3)

Dr. Subburaj Ramasami, Former Professor and Consultant, SRM University, Chennai
Units: (1.2.1, 1.2.3-1.2.4, 1.4-1.13, 2, 3, 4, 5)

Copyright © Reserved, Madhya Pradesh Bhoj (Open) University, Bhopal

All rights reserved. No part of this publication which is material protected by this copyright notice may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the Registrar, Madhya Pradesh Bhoj (Open) University, Bhopal

Information contained in this book has been published by VIKAS® Publishing House Pvt. Ltd. and has been obtained by its Authors from sources believed to be reliable and are correct to the best of their knowledge. However, the Madhya Pradesh Bhoj (Open) University, Bhopal, Publisher and its Authors shall in no event be liable for any errors, omissions or damages arising out of use of this information and specifically disclaim any implied warranties or merchantability or fitness for any particular use.

Published by Registrar, MP Bhoj (open) University, Bhopal in 2020

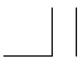
Vikas® is the registered trademark of Vikas® Publishing House Pvt. Ltd.

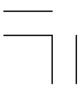
VIKAS® PUBLISHING HOUSE PVT. LTD.
E-28, Sector-8, Noida - 201301 (UP)
Phone: 0120-4078900 • Fax: 0120-4078999
Regd. Office: A-27, 2nd Floor, Mohan Co-operative Industrial Estate, New Delhi 1100 44
• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

SYLLABI-BOOK MAPPING TABLE

Programming in C

Syllabi	Mapping in Book
Unit-I C Programming: Basic Concepts An Overview of Programming, Programming Language, Classification of Programming Language, Procedural Oriented, Object Oriented Programming Language, Characteristics of a Good Programming, Introduction to C, Basic Structure of C, Program Programming Style, Execution of C Program, C Tokens, Keywords and Identifiers, Constants, Variables, Data Types, Declaration of Variable, Assigning Value to Variable.	**Unit-1:** An Overview of Programming **(Pages 3-53)**
Unit-II Operators and Expressions Operators and Expression, Arithmetic Operators, Relational Operator, Logical Operator, Assignment Operator, Increment Operator, Decrement Operator, Conditional Operators, Bitwise Operator, Special Operator, Arithmetic Expression, Evolution of Expression, Operator Precedence and Associativity, Input and Output Statement, Formatted Input and Formatted Output.	**Unit-2:** Operators and Expressions **(Pages 55-84)**
Unit-III Decision Making, Branching, Arrays and Pointers Decision Making and Branching, `if`, `else`, `nested if else`, `if Ladder`, `switch Statements`, Conditional Operator `GOTO` Operator, Looping Statement, `While`, `do`, `for`, Jumps in Loops Arrays, One Dimensional Arrays, Two Dimensional Array, Multidimensional Array, Pointers, Declaration of Pointer Accessing the Address of a Variable, Initiating Pointers, Accessing a Variable through its Pointer, Pointer and Arrays, Pointer and Function, Pointer and Structure.	**Unit-3:** Decision Making, Branching, Arrays and Pointers **(Pages 85-178)**
Unit-IV Handling of Character String, Function and Structure Handling of Character String, Declaring and Initializing String Variables, String Handling Functions, User Define Function from of C, Function, Return Values and their Types, Calling a Function by Value and Reference, Nesting of Function Recursion, Function with Array and Structures and Union, Structure Initialization, Array of Structures, Structure with in Structure, Structure and Function.	**Unit-4:** Handling of Character Strings, Function and Structure **(Pages 179-229)**
Unit-V File Management in C File Management in C, Defining and Opening a File, Closing a File, Input/Output Operations on File, Error Handling, Input/Output Operator, Random Access to the Files, Command Line Argument, Preprocessors, Macro Substitution, ANSI Editions, Computer Control Directives.	**Unit-5:** File Management in C **(Pages 231-272)**





CONTENTS

INTRODUCTION 1

UNIT 1 AN OVERVIEW OF PROGRAMMING 3-53

1.0 Introduction
1.1 Objectives
1.2 Introduction to C Programming Language
  1.2.1 Classification of Programming Language
  1.2.2 Procedural Oriented Programming Language
  1.2.3 Object Oriented Programming Language
  1.2.4 Characteristics of a Good Programming Language
1.3 Basic Structure of C Programming Style
1.4 Execution of C Program
  1.4.1 Linking and Loading
  1.4.2 Sample C Program
  1.4.3 Variation in the `main()` Function
  1.4.4 Syntax and Semantic Errors
  1.4.5 Reserved Words
1.5 Tokens in C
1.6 Data Types
1.7 Constants
1.8 Variables
  1.8.1 Types of Variables
  1.8.2 Declaration of Variable
  1.8.3 Assigning Value to Variable
1.9 Answers to 'Check Your Progress'
1.10 Summary
1.11 Key Terms
1.12 Self Assessment Questions and Exercises
1.13 Further Reading

UNIT 2 OPERATORS AND EXPRESSIONS 55-84

2.0 Introduction
2.1 Objectives
2.2 Operators in C
  2.2.1 Arithmetic Operators
  2.2.2 Relational Operators
  2.2.3 Logical Operators
  2.2.4 Assignment Operators
  2.2.5 Conditional Operator
  2.2.6 Increment and Decrement Operators
  2.2.7 Bitwise Operator
  2.2.8 Special Operator
2.3 Expression in C
2.4 Operator Precedence and Associativity
2.5 Input and Output in C
  2.5.1 Use of `printf()`
  2.5.2 Single Character Input/Output
  2.5.3 Strings—`gets()` and `puts()`
2.6 Answers to 'Check Your Progress'
2.7 Summary

2.8 Key Terms
2.9 Self Assessment Questions and Exercises
2.10 Further Reading

UNIT 3 DECISION MAKING, BRANCHING, ARRAYS AND POINTERS 85-178

3.0 Introduction
3.1 Objectives
3.2 Conditional Statements
3.2.1 `if` Statement
3.2.2 `if...else` Statement
3.2.3 Nesting of the `if...else` Statements
3.2.4 Logical Operators and Branching
3.2.5 Conditional Operator and `if...else`
3.3 Switch Statement
3.4 Break, Continue, Goto Statements
3.5 Looping Statements
3.6 Arrays in C
3.6.1 Declaring Arrays
3.6.2 One Dimensional Array
3.6.3 Two Dimensional Array
3.6.4 Multidimensional Array
3.7 Pointers in C
3.7.1 Pointer to `void`
3.7.2 `NULL` Pointer
3.7.3 Declaration of Pointer Variables
3.7.4 Accessing the Address of a Variable
3.8 Pointer to Array
3.8.1 Multi Dimensional Arrays
3.9 Pointer and Function
3.9.1 Chain of Pointers/Array of Pointers
3.9.2 Pointers to Functions
3.10 Pointer and Structure
3.10.1 Passing Structure by Reference
3.10.2 Array of Pointers
3.10.3 Pointer to Pointer
3.11 Answers to 'Check Your Progress'
3.12 Summary
3.13 Key Terms
3.14 Self Assessment Questions and Exercises
3.15 Further Reading

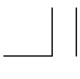
UNIT 4 HANDLING OF CHARACTER STRINGS, FUNCTION AND STRUCTURE 179-229

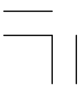
4.0 Introduction
4.1 Objectives
4.2 Strings in C
4.2.1 Use of `scanf()` and `printf()` with Strings
4.2.2 Reading and Writing Strings: `gets` and `puts`
4.2.3 Library Functions for String Handling
4.2.4 Two-Dimensional Character Arrays
4.3 User Defined Function Form of C
4.3.1 Function Declaration
4.3.2 Differences between User Defined Functions and Stored Procedures

4.4 Functions in C
4.4.1 Function Call – Passing Arguments to a Function
4.4.2 Scope: Rules for Functions
4.4.3 Function Parameters
4.4.4 Return Values and their Types
4.5 Calling a Function by Value and Reference
4.6 Function Recursion
4.7 Arrays and Functions
4.8 Structures and Functions
4.8.1 Declaration
4.8.2 Processing a Structure
4.8.3 User-Defined Data Type
4.8.4 Structure Elements Passing to Functions
4.8.5 Structure Passing to Functions
4.8.6 Structure Within Structure
4.9 Function with Union
4.10 Answers to 'Check Your Progress'
4.11 Summary
4.12 Key Terms
4.13 Self Assessment Questions and Exercises
4.14 Further Reading

UNIT 5 FILE MANAGEMENT IN C **231-272**

5.0 Introduction
5.1 Objectives
5.2 Introduction to Files in C
5.3 Opening and Closing a File in C
5.4 Input / Output (I/O) Operations on File
5.5 Error Handling and I/O Operator
5.6 Character I/O
5.7 File I/O and FEOF
5.7.1 FEOF
5.8 STDIN, STDOUT and STDERR
5.9 Random Access to the Files
5.10 Command Line Arguments
5.11 C Preprocessors
5.12 ANSI Editions
5.13 Answers to 'Check Your Progress'
5.14 Summary
5.15 Key Terms
5.16 Self Assessment Questions and Exercises
5.17 Further Reading



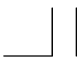


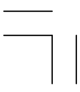
INTRODUCTION

NOTES

C is a programming language and is substantially different from C++ and C#. Many operating systems are written using C, UNIX being the first. Later, Microsoft Windows, Mac OS X and GNU/Linux were written in C. Not only is C the language of operating systems, it is the precursor and inspiration for almost all the popular high-level languages available today. Perl, PHP, Python and Ruby are also written in C. In fact, one of the strengths of C is its universality and portability across various computer architectures. Therefore, C can be used for the development of different types of applications that include real-time systems and expert systems. C also provides flexibility to users for introducing new types of features in their programs, depending upon the requirement and definition of user defined functions. The various features of C—algorithms, flow charts, decision making statements, functions, arrays, structures and pointers—are useful for program developers.

This book is divided into five units that attempt to give the students a fair idea of Introduction to C Programming Language, operators and expressions decision making, branching, arrays and pointers handling of character strings, function and structure file management in C. The book follows the Self-Instructional Mode or SIM format wherein each unit begins with an 'Introduction' to the topic followed by an outline of the 'Objectives'. The detailed content is then presented in a simple and structured manner interspersed with Answers to 'Check Your Progress' questions. A list of 'Key Terms', a 'Summary' and a set of 'Self-Assessment Questions and Exercises' is also provided at the end of each unit for effective recapitulation.





UNIT 1 AN OVERVIEW OF PROGRAMMING

NOTES

Structure

1.0 Introduction
1.1 Objectives
1.2 Introduction to C Programming Language
    1.2.1 Classification of Programming Language
    1.2.2 Procedural Oriented Programming Language
    1.2.3 Object Oriented Programming Language
    1.2.4 Characteristics of a Good Programming Language
1.3 Basic Structure of C Programming Style
1.4 Execution of C Program
    1.4.1 Linking and Loading
    1.4.2 Sample C Program
    1.4.3 Variation in the `main()` Function
    1.4.4 Syntax and Semantic Errors
    1.4.5 Reserved Words
1.5 Tokens in C
1.6 Data Types
1.7 Constants
1.8 Variables
    1.8.1 Types of Variables
    1.8.2 Declaration of Variable
    1.8.3 Assigning Value to Variable
1.9 Answers to 'Check Your Progress'
1.10 Summary
1.11 Key Terms
1.12 Self Assessment Questions and Exercises
1.13 Further Reading

1.0 INTRODUCTION

C language was designed and developed by Brian Kernighan and Dennis Ritchie, at The Bell Research Labs in 1972, the 'C' programming language is one of the most popular computer languages in today's computer world. It was created so as to allow the programmer access to almost all of the machine's internals, registers, I/O (Input/Output) slots and absolute addresses. Structured programming, also known as procedural programming, was a powerful and an easy approach of writing complex programs. In procedural programming, programs are divided into different procedures, also known as functions, routines or subroutines, and each procedure contains a set of instructions that performs a specific task.

Martin Richards developed a language called BCPL (Basic Combined Programming Language) in the mid 1960s. It had many good features. In the year 1970, Ken Thompson, also working at AT&T Bell Labs USA, developed a language called B, which was influenced by BCPL. He developed B as a compact

NOTES

language for system programming. Subsequently, in 1972, Dennis Ritchie, also working at Bell Labs, designed and implemented C language on the UNIX operating system on a Digital Equipment Corporation (DEC) PDP-11 computer. The C language was, therefore, developed under the influence of BCPL and B. The C language is, however, rich in data types unlike the other two languages. The B language can be considered as the C language without types. The C language added data types, such as char and float. By 1973, the C language was ready. The C language further added new features between 1973 and 1980. The C language was used to write UNIX kernel for PDP-11 computer.

A variable is referenced to a named area of storage mechanism that holds a single value, such as numeric or character. The name of each variable in C language is declared to use and its type before using it. The C programming language has two main variable types known as local variables and global variables. Local variables scope is confined within the block or function where it is defined. Local variables must always be defined at the top of a block. Global variable is defined at the top of the program file and it can be visible and modified by any function that is referenced by it.

In this unit, you will study about the introduction of C programming language, classification of programming language, procedural oriented programming language, object oriented programming language, characteristics of a good programming language, basic structure of C programming style, execution of C program, C tokens, keywords and identifiers, variables and data types, declaration of variable and assigning value to variable.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basics of C programming language
- Discuss the classification of programming language
- Interpret on the procedural oriented programming language
- Define the significance of object oriented programming language
- Analyse the characteristics of a good programming language
- Explain the basic structure of C programming style
- Describe about the execution of C program
- Discuss the C tokens
- Understand the importance of keywords and identifiers in C
- Define the variables and data types in C
- Explain how the variable is declared
- Assigning value to variable

1.2 INTRODUCTION TO C PROGRAMMING LANGUAGE

NOTES

Designed and developed by Brian Kernighan and Dennis Ritchie, at The Bell Research Labs in 1972, the 'C' programming language is one of the most popular computer languages in today's computer world. It was created so as to allow the programmer access to almost all of the machine's internals—registers, I/O (Input/ Output) slots and absolute addresses. In addition to this, 'C' allows for as much data handling and programmed text modularization as is needed to allow very complex multi-programmer projects to be constructed in an organized and timely fashion.

Although this language was originally intended to run under UNIX, there has been a great interest in running it under the MS-DOS operating system on the IBM PC and compatibles. It is an excellent language for this environment because of the simplicity of expression, the compactness of the code, and the wide range of applicability.

Also, due to the simplicity and ease of writing a C compiler, it is usually the first high level language available on any new computer, including microcomputers, minicomputers, and mainframes.

Programming in C can be a great help in the areas where you need to use Assembly Language but would prefer to keep it simple to write and easy to maintain the program. The time saved in coding of C can be quite rewarding in such cases.

Even though the C language has a good record when programs are transported from one implementation to another, there are differences in compilers that you will find anytime you try to use another compiler. You come to know of many differences when you use nonstandard extensions such as calls to the DOS BIOS when using MS-DOS. Nevertheless, these differences can be minimized by careful choice of programming constructs.

When the C programming language gained popularity among users, using a wide range of computers, representatives of the software sector met to propose a standard set of rules for the use of the C programming language. The group represented all sectors of the software industry and after many meetings, and many preliminary drafts, they finally wrote an acceptable standard for the C language. It has been accepted by the American National Standards Institute (ANSI), and by the International Standards Organization (ISO). Although implementation of the program is not compulsory, it would be the loss of the individual/organization not opting for it.

C is a general-purpose high-level programming language that was developed by Dennis Ritchie at the Bell Laboratories, USA in the early 70s. The language was named 'C' because it was the successor to a language named 'B', which was developed by Ken Thompson in 1969–70. Dennis Ritchie extended the features of B and turned-into C by adding some more features in it. C was developed as a high-level language that could be used to rewrite the UNIX operating system which was earlier written in assembly language. C was initially used to develop

NOTES

system software, such as operating systems, compilers, interpreters, assemblers, databases, text editors, utilities, etc. One of the most interesting features of C language is that its compiler is written in C itself.

In 1978, Brian Kernighan and Dennis Ritchie wrote a book 'The C Programming Language'. Because of this book, the de facto standard for C programming language was known as K. & R. standard for many years. However, there were many changes made unofficially to the C language that were not present in the K. & R. standard. Due to this reason, a group of compiler vendors and software developers approached the American Standards Institute (ANSI) in 1983 to build a standard for the C language, and by the end of 1989 the committee approved the ANSI standard for C programming language.

Basics of C

This section discusses some basic concepts of C which lay the foundation for harnessing the features and capabilities of this language. First the basic elements required to construct simple C statements will be discussed. This includes the C character set, followed by the various tokens in C (such as keywords, identifiers, constants, operators and punctuators). In addition, we will discuss the various data types in C and working with variables, operators and expressions.

C Character Set

A character set can be defined as a set of characters that either individually or in combination, represents a meaning in a language. In C, a character set includes the upper case (A–Z) and lower case (a–z) letters, decimal digits (0–9), blank spaces (tabs, new line, space, etc.) and special characters (characters that perform a specific action based on the context in which they are used). The special characters are listed in Table 1.1.

Table 1.1 *Special Characters*

Special Characters				
+	>	/	[	\
!	;	"	]	{
<	*	.	%	}
:	^	,	~	#
-	(	=	_	|
?	)	'	&	

Apart from the upper and lower case letters, decimal digits and special characters, C also uses a combination of characters. For example, the escape sequences such as '\b', '\c' and '\t' are a combination of the special character '\' and a letter ('b', 'c' and 't').

Advantages of C

Some of the advantages of C language are as follows.

- **Readability**: It is easier to learn and understand because it is very close to human languages.
- **Machine Independent**: Programs written in C language are portable as codes. Once written they can be used on different platforms (machines) also.

NOTES

- **Easy Debugging**: Programs written in this language are easy for debugging. Debugging means to rectify the errors in a program. High-level languages require an interpreter or a compiler to detect error(s) and helps programmers to correct errors.
- **Easy Maintenance**: Programs written in C are modified easily as they are similar to human languages.
- **Reusability**: C codes are reusable, that is, once the programs are loaded into a library they can be used by other applications also.
- **Structured Programming**: Code written in C can be broken down into several functional programs. This can be developed independently and then combined into a single program.
- **Low Development Cost**: Programs written in C language increase programmers' productivity (number of lines of code generated per hour).
- **Easy Documentation**: Programs written in C provide easy documentation for future maintenance.

1.2.1 Classification of Programming Language

Modular programming is a good programming concept. Instead of writing one large program, it is better to divide it into a number of sub tasks and code each one of them separately. Here, each function is coded as a separate module. This modularization has many advantages. One of them being the modules can be reused in other programs. The modules are tested individually and later integrated with other modules.

The use of iteration constructs, such as `for`, `while`, etc., is encouraged in modular programming. The module is designed in such a manner that it has one entry point and one exit point. Modular programming is a prerequisite not only for structured programming but also for object oriented programming.

Structured Programming Concept

Two mathematicians Corrado Bohm and Giuseppe Jacopini proved that any computer program could be written only with three program structures as follows:

- Sequences
- Decisions
- Loops

This discovery is considered to be a precursor methodology for modern programming known as structured programming. Professor Edsger W. Dijkstra even advocated that the `goto` statement should be abolished from all high level languages because of its ill effects on programs. The programs that include `goto` statements can cause innumerable problems, particularly during maintenance of the software. The program, which uses `goto` statement, is called spaghetti code – code that has no simple direct logical structure.

Structured Programming is the name given to good programming practices. It is a preferred methodology for programming in procedure oriented languages. Structured programming consists of guidelines for designing programs. This programming concepts were evolved to improve the quality of programs.

NOTES

Features of Structured Programming

It is a philosophy of a concept that facilitates design of programs. They are easily understandable, modifiable and do not cause surprises. Some of the features of structured programming are given below.

- Flow of control in the program should be as simple as possible.
- The program should be constructed using independent modules or functions or sub routines.

It uses only three types of logical structures as given below:

- **Sequences:** Statements that are executed one after another.
- **Decision:** One of the two blocks of a program code is executed based on the outcome of testing a particular condition. Example is `'if… else'` structure.
- **Loops or Iteration:** One or more statements are executed repeatedly as long as a particular condition or a combination of conditions remains true.

Structured programming also means that we develop an understandable and maintainable program. Many high level languages, such as Applied Decision Analysis (ADA), Pascal, C, FORTRON, etc., support structured programming.

The primary purpose of structured programming is to create the right procedures that enable the design of a program correct, understandable, testable and modifiable. Structured programming results in additional advantages as follows:

Since top down structured design is advocated during the high level design of the program, a number of modules would have been identified with clear specifications. Therefore, a number of programmers work in parallel, designing one module. Furthermore, the programmer understands the role of each of the modules in the overall program structure since he starts from the top level design document very early in the project.

It has also been found that structured programming reduces the time to develop programs, because of work parallelism as explained above as well as clarity in the complete structure of the program. Furthermore, structured programming facilitates divide and conquer of the problem. This, thereby helps in focussing on one problem at a time, that too smaller problems. All these lead to quick turn around time for development of programs.

1.2.2 Procedural Oriented Programming Language

Which provided a structured way of writing programs. Structured programming (also known as procedural programming) was a powerful and an easy approach of writing complex programs. In **procedural programming**, programs are divided into different procedures (also known as functions, routines or subroutines) and each procedure contains a set of instructions that performs a specific task. This approach follows the top-down approach for designing the program. That is, first the entire program is divided into a number of subroutines. These subroutines are again divided into small subroutines and so on, until each subroutine becomes an indivisible unit (Refer Figure 1.1).

NOTES

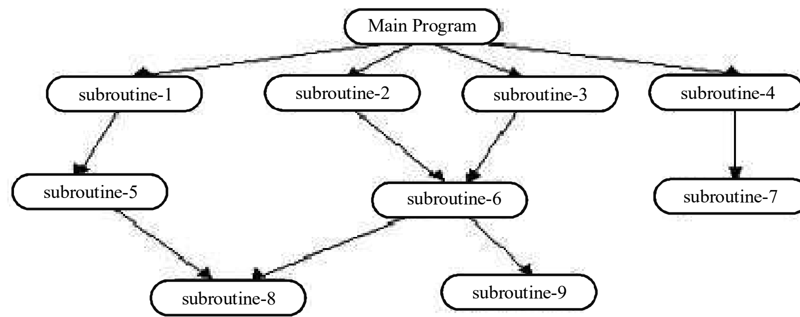


Fig 1.1 *Top-Down Approach*

Programs in procedural programming consist of a controlling procedure known as the **main**, which controls the execution of other procedures. When a call to a procedure is made, the program control is passed to that procedure and all the instructions in that procedure are executed one after another. After executing all the instructions, the program control returns to the procedure, from where the call is made.

For example, to write a program that can add, modify, find and delete students' records from a database using the procedural programming, the entire program can be divided into four different procedures, namely, `add ()`, `find ()`, `delete ()` and `modify ()` (Refer Figure 1.2). In addition to these procedures, the program consists of a main procedure. If a user wants to add a record of a student in a database, the control is passed from the main procedure to the *add* procedure. Once all the instructions are executed in *add* procedure, the control is transferred back to the main procedure. The same steps are followed while calling other procedures.

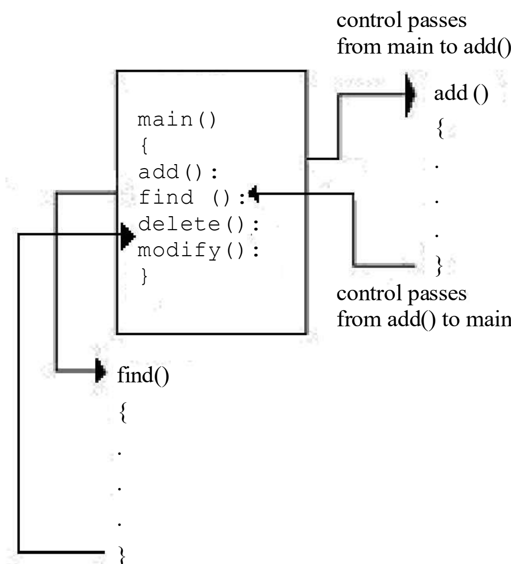


Fig 1.2 *Procedural Programming*

1.2.3 Object Oriented Programming Language

High level programming languages, such as FORTRAN, Pascal, BASIC, C and COBOL are known as function oriented or procedure oriented languages. They

have been used for developing important and mission critical applications. In these languages, computer programs are organized as a sequence of instructions. In function oriented programming, the emphasis is on procedures or instructions and data integrity does not get due importance. In large software projects, a number of programmers will be involved in programming. Each one will develop programs for some portion of the product. An inexperienced programmer might spoil the integrity of some data items inadvertently due to poor understanding. This is due to the inherent design of the function oriented programming languages where data and function are not tied to each other closely. They are independent of each other causing major problems while the software is put to use.

The principle of Object Oriented Programming (OOP) is to combine both data and the associated functions into a single unit called a class. An object in programming means data. Data is, therefore, predominant in object oriented programming. However, in the object oriented paradigm, accessibility of data is restricted and therefore, accidental corruption of data is minimized or eliminated. Thus, OOP ensures integrity of data. C++, Java, C #, etc., are some languages which facilitate OOP.

In both programming paradigms namely; structured programming using procedure oriented languages and object oriented programming, the following two approaches are possible:

- Top Down Approach
- Bottom Up Approach

Top Down Approach

The program is constructed and frequently designed using top down design methodology.

The top down design model involves designing the overall program structure first followed by designing individual functions.

Bottom Up Approach

The opposite of top down methodology is bottom up methodology wherein individual functions are designed first and finally the structure of the program is made.

Top Down Structured Design

One of the earliest methodologies for software analysis and design is top down structured design. This is the method promoted by the early high level languages, such as FORTRAN and COBOL. C programs are also developed using top down structured design. In a C program, the `main()` function is at the top of the program structure. It calls other functions. Hence, a C program is represented in the form of a tree, with a single node at the top, which calls a number of other nodes. The nodes may in turn call some other nodes and so on. This methodology is called top down structured design. In this methodology, we divide and conquer by partitioning the functions to be implemented.

NOTES

1.2.4 Characteristics of a Good Programming Language

We have to develop an understandable and maintainable program. Some of the popular codes of good programming practices are as follows:

- Write only one statement per line.
- Coin meaningful names for constants, variables and functions.
- Use capitals for names of constants.
- Divide programs into functions or classes as the case may be.
- Each function and class defines one task.
- Each function and class must have at least one comment statement.
- Skip a line between functions and classes.
- Skip a line after declaration statements in the `main()` function as well as each function.
- Bring clarity by skipping lines wherever required.
- Put only one brace on each line.
- Align all opening braces and closing braces.
- Indent as much as possible to bring out logical structure of the program.
- Some of the program statements that could be indented are body of function, class, body of loop, body of `if..else` statements.
- Indent each `case` in `switch` statement.
- Indent an item within a `struct/class` declaration.

Features of a Good Computer Program

The features of a good program are as follows:

- **Efficiency:** There are two aspects which are considered to estimate the efficiency of computer software as given below:
  - o Response time or speed of execution.
  - o Main memory consumed to execute the program.

A program is efficient if its speed of execution is faster and at the same time it consumes lesser memory.

- **Correctness or Integrity:** This is a measure of accuracy of the results produced when executing the program.
- **Degree of Structured Programming:** This attribute measures to produce programs with clear flow and design along with hierarchical structure.
- **Degree of Modularity:** This attribute refers to a process in which programs are divided into separately named and addressable components called modules which are integrated to satisfy problem requirements.
- **Simplicity:** This attribute measures the ease with which the program can be understood.

NOTES

- **Maintainability:** This is a measure of ease of modifications of the program.

  The time taken to locate the module that contains error will be much shorter in modular structured programs. Since the functions are cohesive and there is lesser coupling between functions, the side effects of maintenance will also be minimal.
- **Reusability:** Reusability is facilitated by the modular design of the program. Each module is a good candidate for reusability in different programming projects.
- **Ease of Debugging:** Small codes are easy to debug rather than large programs. The wrong understanding of the specifications of each module will be avoided while developing and testing the program. Therefore, modular programming enhances ease of debugging.
- **Enhanced Understandability:** Structured programming advocates adequate comment statements and documentation. It also advocates defining meaningful names for the functions as well as variables and constants. All these facilitate ease of understanding of the complete program.

1.3 BASIC STRUCTURE OF C PROGRAMMING STYLE

Programs are a sequence of instructions or statements. These statements form the structure of a C program. To understand the structure of a program, consider Figure 1.3.

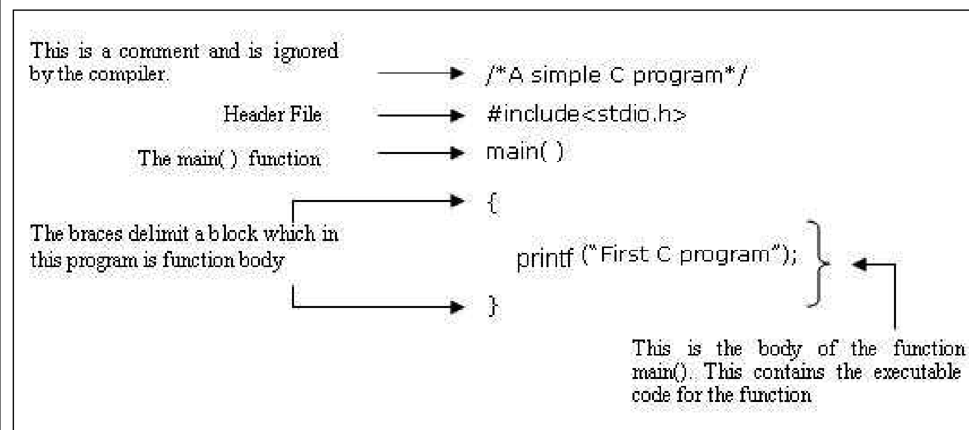


Fig 1.3 *Structure of a C Program*

The structure of a C program can be broadly classified into *header files* and the *main() function*. They can be described as follows:

- **Header files**: While writing programs, programmers use various programming elements defined in the Standard C Library. These pre-defined programming elements include library functions, variables, constants, etc. In order to use such pre-defined elements in a program, appropriate standard header files must be included in the program. The standard header files are files that contain the information (like the return type of library functions,

data type of constants, etc.) required by C compilers in order to use the pre-defined elements. In other words, they contain declarations of library functions, type definitions and so on. As a result, programmers do not need to explicitly declare (or define) the programming elements. Standard header files are specified in a program through the preprocessor directive, `# include`

In Figure 1.3, the `stdio.h` header file is used. When the compiler processes the instruction `#include <stdio.h>`, it includes the contents of `stdio.h` in the program. Once the contents of the `stdio.h` header file are included, programmers can work with the *standard input*, *output* and *file system* facilities. The name `stdio.h` file stands for standard input-output header file. The `stdio.h` file contains numerous prototypes and macros to perform Input or Output (I/O) operations for C program. The standard I/O functions defined in `stdio.h` are discussed here.

- **printf()**: It is the standard library function used to display any message or values on screen. It takes as arguments a format string and an optional list of variables or literals to output. The `printf()` takes the following form:

  ```
  printf("format-control-string",variable-list);
  ```

  The `format-control-string` controls the format of the output and it can consist of the following:

  1. Sequence of characters
  2. Conversion specifications that always begin with a `%` sign and end with a conversion specifier
  3. Escape sequences that always begin with a \ sign.

  The `variable-list` (which is optional) consists of variable names separated by commas whose values are to be printed. This list should include a variable corresponding to each conversion specifier.

- **scanf()**: It is also a standard library function used to receive formatted input from keyboard. The `scanf()` takes the following form:

  ```
  scanf("format-control-string", variable-address-
  list);
  ```

  The `format-control-string` describes the format of the data to be input. It can consist of conversion specifications and character literals. The `variable-address-list` is a list of addresses of the variables in which the input values are to be stored. This list should include an address for each conversion specification. The address of variable is obtained by using ampersand (`&`) followed by the variable name.

- **main()**: The first executable instruction in all C programs is the `main()` function. That is, programs begin their execution from this instruction. Once all the instructions in the `main()` function are executed, the control passes out of `main()`, terminating the entire program. Generally, large and complex programs are divided into smaller subprograms known as functions. Out of all the functions defined in a program, the `main()` function is one of the most important functions.

NOTES

In Figure 1.3, `main()` is the beginning of its function definition. The word `main` refers to the name of function and the following '`()`' indicates that it is a function. The opening and closing curly braces '`{}`' enclose the `main()` *function's body*. All C statements that need to be executed are written within the `main()` function's body. The statement `printf("First C Program";` written within the `main()` function's body, displays the message `First C Program` with the help of the `printf()` function.

Note: Every C program must have one and only one `main()` function.

1.4 EXECUTION OF C PROGRAM

Martin Richards developed a language called *BCPL* (Basic Combined Programming Language) in the mid 1960s. It had many good features. In the year 1970, Ken Thompson, also working at AT&T Bell Labs USA, developed a language called *B,* which was influenced by *BCPL*. He developed *B* as a compact language for system programming. Subsequently, in 1972, Dennis Ritchie, also working at Bell Labs, designed and implemented C language on the UNIX operating system on a Digital Equipment Corporation (DEC) PDP-11 computer. The C language was, therefore, developed under the influence of *BCPL* and *B*. The C language is, however, rich in data types unlike the other two languages. The *B* language can be considered as the C language without types. The C language added data types, such as `char` and `float`. By 1973, the C language was ready. The C language further added new features between 1973 and 1980. The C language was used to write UNIX kernel for PDP-11 computer.

The languages Basic Combined Programming Language (BCPL), *B* and C are procedure oriented languages similar to FORTRAN. However, they are more compact having few keywords. Simple compilers translate them. They use programs in the standard library for input/output and other interactions with the operating system. During the 1980s, the C language compilers became available for most computer architectures and operating systems. It became a programming tool in PC, which increased its popularity.

Ritchie along with Kernighan published a book '*The C Programming Language*' in 1978. The Kernighan and Ritchie (K&R) description of the language became a kind of industry standard, popularly called K&R C. Since then, as many C compilers came into existence and there were minor compatibility problems due to variations in implementation, the American National Standards Institute (ANSI) formed a committee (X3JII) for bringing out a standard for the C language. This committee brought out a standardized definition of the C language in 1989. The international standard on the C language, namely ISO/IEC 9899 was released in 1990 by adopting the ANSI standard on the C language. This standard is called C90. Subsequently it was revised in 1999. The revised standard is called C99. Thus, there are three forms of C, namely K&R C, C90 and C99. Even thereafter, minor amendments to the standard are taking place. You will now learn the C language that conforms to the International Organization for Standardization/ International Electrotechnical Commission (ISO/IEC) 9899: 1990 standard, hereinafter called the Standard.

NOTES

Note:

ISO—International Organization for Standardization

IEC—International Electro-technical Commission

Developing a C Program

A number of Integrated Development Environment (IDE) tools for developing and executing C programs are available in the market. Freeware compilers for C and a host of other languages are available on the Internet from an organization called The Free Software Foundation. This is called gcc C of GNU. It can be downloaded from the following Uniform Resource Locators (URLs):

`http://www.gnu.ai.mit.edu/software/gec` to work with UNIX operating system `http://www.delorie.com/djgpp/` to work under DOS/Windows environment

A typical methodology for the development of a C program using GNU C in the Windows environment in the DOS prompt, is given in Figure 1.4.

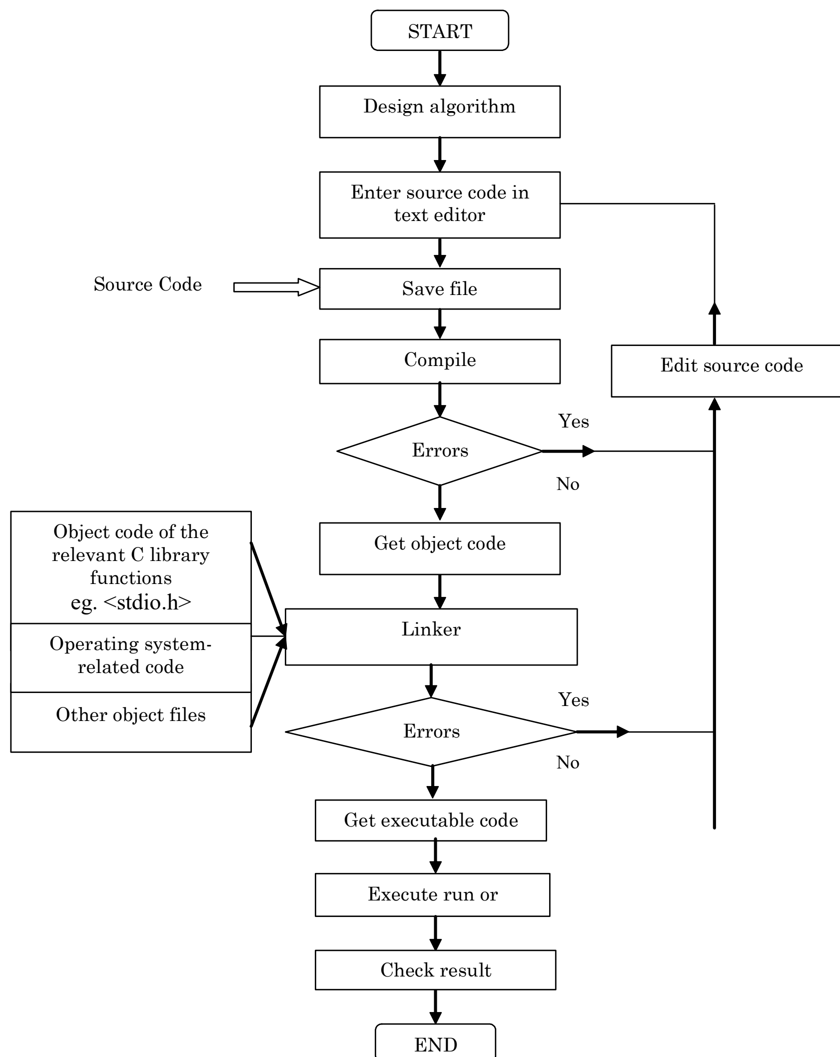


Fig. 1.4 *Steps Involved in Developing a C Program*

NOTES

The various steps involved in program development given in Figure 1.4 are briefly discussed below.

Object Code

Now we are ready to compile the source code. We have to call the compiler and submit our source code for compilation. If we are using a GNU compiler, we type it in the Disk Operating System (DOS) command for compilation as follows:

```
gcc–2.1.c
```

Assume that you have another source file called `px.c`. You will get `px.o` when the compilation is successful.

1.4.1 Linking and Loading

During this process, you combine all the object files. In this case, Ex 1.1.c and `px.o`. Additionally, you will add the object code corresponding to the header files included in the source code or program under execution. For example, you will combine the object code corresponding to the `printf()` function in the `<stdio.h>` file.

Another important code is that corresponding to the particular operating system of the computer system. This is essential for executing the program in the given hardware and the operating system. The linker to get an executable file for the source file will combine all these. The executable file will be in the machine code. It will be generated automatically on successful linking and loaded for execution. You get Ex 1.1.c.exe in this case after successful linking.

1.4.2 Sample C Program

A program in the C language is given below, which displays a text. Have a close look at the program.

```
/* Example 1.1*/
/* program for displaying a text */
#include <stdio.h>
int main()
{
printf("Om Vinayaga");
}
```

Create a new file in the text editor. Then type in the program exactly as given above. Save this program as Example 1.1. Next compile the program. The compiler after compilation will give a message. Look at the message box. If the C program was compiled in a C++ compiler, there may be warnings, but you can safely ignore them. However, the errors, if any, should not be ignored. The compiler will give the errors and line numbers. Sometimes, even an experienced programmer will find it difficult to understand the error messages. If you encounter the same difficulty in understanding the error messages, you need not worry. Open the program file again and check whether you have typed it exactly as in the book including the semicolons, quotation marks and brackets and the program is a working program. Keep checking and compiling till the compiler says ‘success’, meaning that there is no error in the program after compilation. Now you have to

link the object code to get the executable code. When there are no errors even after linking, you have to execute or run the program. Use the right command for 'run'. On execution, you will get the result as given below:

Result of the program

```
Om Vinayaga
```

You have succeeded in establishing a communication link with the computer. You can now talk to it regularly by learning new commands and using better communication methods. You have now become familiar with the methodology for learning a programming language. The steps involved are summarized as follows:

- Type the program in a text editor.
- Save and give a name to the program.
- Compile the program.
- Look for errors and correct them.
- Link the object code to produce executable code.
- Execute/Run the program and analyse the result.

The methodology for the preparation of the source code, compilation, linking and execution may vary from system to system. Learn the correct operations of the IDE/system being used for the various steps mentioned above.

1.4.3 Variation in the `main()` Function

Most compilers have implemented K&R C. Some have implemented C90. Not many have implemented the new features introduced in C99. Hence, it is important that the reader checks what standard his compiler supports. In the Example 2.1, we did not explicitly return anything. On successful execution, the program returns zero. Return of any other integer means that the program execution was not successful. Hence, you can add a statement `return 0` at the end of the program.

The main function did not receive any value which was indicated by the empty parentheses. We can indicate nothing by the keyword `void`. The modified program including the above two features is given below:

```
/* Example 1.2*/
/* program for displaying a text */
#include <stdio.h>
int main (void)
{
printf ("Om Vinayaga");
return 0;
}
```

Result of the program

```
Om Vinayaga
```

Thus, both these programs have carried out the task successfully. However, in the rest of the book, the style given in the previous program will be used. You may adopt the style that works in your compiler.

1.4.4 Syntax and Semantic Errors

NOTES

A C program must contain a function known as `main()` function. It can be consisted of one or more than one function. In C, `main()` function is called when the program starts up. A simple function consists of a header followed by an opening brace followed by opening brace and the body of the function and then followed by closing brace. Each C statement is an instruction to the computer which is ended with a semicolon (;). The syntax of the C programming language is a set of rules that specifies whether the * of chaqracters in a file is conforming C The semantics of a statement is its meaning. The C compiler is used to detect the syntax errors. But, semantics error shows up the program's behavior only after it is compiled. To detect the semantic errors tracing process is involved in the program's state, i.e., the values of variables after each program is stepped up. C keywords are considered as vocabulary of this programming language. The semantic and syntax errors are explained as follows:

Syntax Errors

Syntax errors are those errors which result from statements that are not in the correct form, for example if semicolon (;) is missed at the end of a statement an error will occur at the run time of the programs. The compiler recognizes all the syntactic errors which occur in C program. Syntax errors occur in C programming during the parsing of input code and are caused by grammatically incorrect statements. Typical errors might be an illegal character in the input, a missing operator, two operators in a row, two statements on the same line with no intervening semicolon, unbalanced parentheses, a misplaced reserved word, etc. Semantic errors occur during the execution of the code after it has been parsed as grammatically correct. These have to do not with how statements are constructed but with what they mean. Such things as incorrect variable types or sizes, nonexistent variables, subscripts out of range are considered as semantic errors. A syntax error is an error in the typing of the code or statement. A semantic error basically means invalid logic. In C, if system defined function "`printf();`" has a syntax error as the command was mistyped. Some syntax errors are very common in C language especially for beginners. Proper syntax in C language helps you to identify and correct many syntax errors in whatever program you are currently working on. C program errors are recognized as syntax and semantics errors. Following types of errors are considered as syntax errors:

Undeclared Variables: A variable must be declared before it can be used. This type of error is easily recognized.

Variable and Symbol Names are Case Sensitive: Sometimes, programmers declare a variable in the program but errors could occur as an undeclared variable error. This type of error occurs in the C program because one or more variable names appear in the wrong case. For example, consider the following code declaration:

```
int val1, val2, pp;
val1=val1*PP;
```

The compiler will output the error message "`Undefined variable PP.`"

NOTES

Missing Parenthesis or Brace or Brackets: A missing parenthesis can cause other type of errors too. This type of error can be avoided by entering the matching parenthesis while the time of entering the matching parenthesis whenever the left parenthesis is entered before entering the statements.

Missing Semicolon: In the C code that follows, three declarations are given. Line numbers (chosen in all examples arbitrarily) are shown to the left of each line. Consider the following example:

```
5 int num;
6 float value
7 double bigNum;
```

A C compiler would generate an error something like the following:

```
Syntax Error: semi-colon expected
Line 6 of program myprog.c
```

To fix this error you need to add a semicolon after the identifier value as in `6 float value;`

Undeclared Variable Name: If the preceding code were compiled and included an assignment statement, as in

```
5 int num;
6 float value
7 double bigNum;
8 bigNum = num + value;
```

Consider the following program:

```
1 #include <conio.h>
2 int main ()
3 {
4 printf("Hello World!!!");
5 return 0;
6 }
```

A compiler will generate an error message as follows:

```
Syntax Error: undeclared identifer "printf"
Line 4 of program myprog.c
```

To correct this error you need to add a header file as `#include <stdio.h>` which contains `printf()` system defined function.

Mismatch of Array name: A string literal that initializes a character array contains more characters than the array can hold. Consider the following expression:

```
char ca[3] ="abcd";
```

An error occurs in the following way:

```
line 1: warning: 1 extra byte(s) in string literal initializer
ignored
```

Illegal Pointer Declaration: A pointer to `void` may not be used to access an object. An expression is written to indirect through a (possibly qualified) pointer to `void`. The indirection is ignored although the rest of the expression (if any) is possible. Consider the following program:

```
f(){
  volatile void vp1, vp2;
   (vp1 = vp2);          / assignment does get done /
}
```

NOTES

An error occurs in the following way:

```
line 3: warning: access through "void" pointer ignored
```

Null Directive: The file name in #include directive is null. For example, consider the following statement:

```
#include <>
```

Since, no header file is declared in the `#include` directive therefore an error occurs in the following way:

```
line 1: empty file name
```

Following steps are required to eliminate the syntax errors:

- To display the current list of syntax errors.
- To start at the first error listed, try to correct it and then re-compile your program. Sometimes many errors will drop out after one error is fixed.
- If you are having trouble with a particular error listed for a specific line then search for a syntax error in the lines that line, starting with the line immediately preceeding the line under consideration and working backwards.

Semantic Errors

Semantic errors are logical errors due to illogical statements in program. Semantic and logical errors are not easy to debug live the syntax errors. They are those types of errors where the code is syntactically correct but it does not produce desired result due to logical error occurs in the programs. The compiler does not know what program intends to achieve with the program so semantic errors are not detected. This type of errors can be solved by performing the following functions:

Set Breakpoints: Setting breakpoints allow user to examine the program execution results at the loophole point.

Set Up Watch List: It allows user to find out program execution results quickly at only breakpoints.

Program might terminates immediately or enter into infinite loop. An infinite loop which repeats indefinitely is often created when a programmer writes a loop in which the expression tested never becomes false. For example, consider the following code in which for loop is used:

```
for(int 1=100;i<=0;i++)
{
  printf("%d", i);
}
```

The `for` loop will run infinitely after executing the above program.

1.4.5 Reserved Words

NOTES

The **reserved words** have special meaning within the language and are predefined in the language's formal specifications. Typically, reserved words include labels for primitive data types in languages that support a type system and identify programming constructs, such as data types, loops, blocks, conditionals and branches. The list of reserved words in a language are defined when a language is developed. Reserved words may not be redefined by the programmer, unlike predefined functions, methods or subroutines, which can often be overridden in some capacity.

The syntax rules or grammar of C defines certain symbols and keywords to have a unique meaning within a C program. These symbols and keywords are known as reserved words and must not be used for declaring data and variables. This is because most of the facilities which C offers are in libraries that are included in programs. Once a library has been included in a program, its functions are defined and you cannot use these reserved words as user-defined keywords, for example variables. As per C syntax specifications, all of these reserved words must be in lower case. Each is considered to be a single word like the special keywords. The word symbols cannot be redefined within the C program, i.e., they cannot be used for anything other than their predefined and intended use. The C language uses a number of keywords such as `int`, `char` and `while`. A keyword has a special meaning in the context of a C program and can be used for that specific purpose only, for example `int` can be used only to specify that the data type is integer. All keywords are written in lowercase letters only. Thus, `int` is a keyword but `Int` and `INT` are not. Keywords are the reserved words. It means you cannot use them as identifiers. Table 1.2 shows the list of reserved words, which are not used in declaring variables:

Table 1.2 *Reserved Words in C*

auto	break	case	char	const
continue	default	do	double	float
else	enum	extern	if	goto
int	long	register	for	include
return	short	signed	sizeof	static
void	volatile	while	exit	main
struct	switch	typedef	union	unsigned
short	asm	system	getc	putchar

Check Your Progress

1. Who developed C language?
2. What is procedural programming?
3. Write the three characteristics of good programming language.
4. Define the term reserved word.

1.5 TOKENS IN C

There are six classes of tokens in C programming language:

NOTES

(i) Keyword
(ii) Identifier
(iii) Constant
(iv) String Literal
(v) Operator
(vi) Punctuators

Tokens are similar to atomic elements or building blocks of a program. A C program is constructed using tokens. There are certain other building blocks of a program that do not form part of any of the above. They are as follows:

- Blanks
- Horizontal Tabs
- Vertical Tabs
- New Line Characters
- Form Feed
- Comments

The C Character Set

The C language supports and implements the American Standard Code for Information Interchange (ASCII) for representing characters. The ASCII uses 7 bits for representing each character or digit. The characters are coded from 0000000 (decimal 0) to 1111111 (decimal 127). Therefore, the ASCII consists of code for 128 characters in all. The ASCII values (decimal equivalent of the 7 bits) of some alphabets and digits are given in Table 1.3.

Table 1.3 *ASCII Values of Selected Alphabets*

ASCII Value	Character or Digit
48	0
49	1
57	9
65	A
66	B
67	C
89	Y
90	Z
97	a
98	b
121	y
122	z

The digits and alphabets are organized sequentially and hence, it is easy to get the ASCII value; for instance, the ASCII value of D is 68, E is 69, 8 is 56, *x* is 120 and so on.

NOTES

Keywords in 'C' Language

Keywords are the words whose meaning has already been explained to the 'C' complier, which are also called as reserved words. The keywords cannot be used as variable names because if we do so we are trying to assign a new meaning to the keyword which is not allowed by the complier.

There are 32 words defined as keywords in C. These have predefined uses and cannot be used for any other purpose in a C program. They are used by the compiler as an aid to compiling the program. They are always written in lowercase. A complete list is as follows:

```
auto      double    int       struct
break     else      long      switch
case      enum      register  typedef
char      extern    return    union
const     float     short     unsigned
continue  for       signed    void
default   goto      sizeof    volatile
do        if        static    while
```

Identifiers in C

An identifier is used for any variable, function, data definition, etc. In the C programming language, an identifier is a combination of alphanumeric characters, the first being a letter of the alphabet or an underline, and the remaining being any letter of the alphabet, any numeric digit, or the underline.

Two rules must be kept in mind when naming identifiers.

- The case of alphabetic characters is significant. Using INDEX for a variable name is not the same as using index and neither of them is the same as using Index for a variable name. All three refer to different variables.
- According to the ANSI-C standard, at least 31 significant characters can be used and will be considered significant by a conforming ANSI-C compiler. If more than 31 are used, all characters beyond the 31st may be ignored by any given compiler.

1.6 DATA TYPES

Data is used in a program to get information. In a program used to find out the greater of two numbers, the numbers are data, and the output which says which number is greater, is information. C is a versatile language and handles many different types of data in an elegant manner.

Bit stands for binary digit, i.e., 0 or 1. Each byte is a collection of 8 bits, i.e., 8 consecutive bits of '0' or '1'. Data is handled in a computer generally in terms of bytes and therefore, will be in the form of multiples of 8 bits. Each ASCII character is represented by one byte.

NOTES

Fundamental Data Types

An item that holds data is also called an object. An object has a name or identifier associated with it. Each object can hold a specific type of data. There are five basic data types in C, as follows:

- Character
- Integer
- Real Numbers
- Void (Comprising an Empty Set of Values)
- Enum (Which will be Introduced Later)

You have to first understand how a computer works. Assume that two numbers `a` and `b` are to be multiplied. First of all, the two numbers have to be stored in the memory. Then the required calculation has to be performed. The result has also to be stored in the memory. Each number is of a specific data type; for instance, all three of them can be declared to be integers. Each data type occupies a specific size in the memory. What does one mean by size? It is the amount of storage space required; each bit needs one storage space. One byte needs eight storage spaces. If a number is of type integer declared as `int`, it is stored in 2 bytes. The number depending on its type gets stored in different forms. If a number is of `float` type, it takes 4 bytes to store it. All characters can be represented according to the ASCII table and hence, 1 byte, i.e., 8 bits are good enough to store a character, which is represented as `char`.

These sizes may vary from computer to computer. The header files `<limits.h>` and `<float.h>` contain information about the sizes of the data types.

Real numbers can be expressed with single precision or double precision. Double precision means that real numbers can be expressed more precisely. Double precision also means more digits in mantissa. The type `'float'` means single precision and `'double'` means a double precision real number. Table 1.4 indicates the size of various data types.

Table 1.4 *Size of Data Types*

Data Type	Size
char	1 byte
int	2 bytes
float	4 bytes
double	8 bytes

Maximum and Minimum Magnitudes of Data Types

The maximum and minimum values of data types are not limitless. For example, `<limits.h>` specifies the minimum and maximum magnitudes for integers and characters. Since `char` is stored in a byte, it is as good as a short integer. `char` can be stored as an unsigned character, which means all the 8 bits can be used to store it. The maximum value of an 8-bit number is 255 when all bits are 1. If it is a `signed char`, the first bit will be reserved for storing the sign. The sign bit will be 0 for a positive number and 1 for a negative number. The integer values of `signed` and `unsigned chars` are given below:

NOTES

CHAR-BIT		8	bits in a `char`
SCHAR	MAX +	127	maximum value of `signed char`
SCHAR	MIN –	127	minimum value of `signed char`
UCHAR	MAX	255	maximum value of `unsigned char`

Now, let us look at the maximum and minimum magnitudes for the integer data type. They are:

INT	MAX +	32767	maximum value of `int`
INT	MIN –	32767	minimum value of `int`

Integer means signed integer. The data type integer occupies 2 bytes or 16 bits. The most significant bit is reserved for sign. It will be '0' for a positive number and '1' for a negative number. Therefore, you can easily calculate how the limits for the integer data types have been arrived at.

The standard has also provided for another type of integer called `short int` with the same maximum and minimum values.

1.7 CONSTANTS

The following are the types of constants:

- Integer Constant
- Character Constant
- Float Constant
- Enumeration constant
- String Constant
- Symbolic constant

All these types are explained below.

Integer Constants

The following are the types of integers:

```
int
unsigned int
long
unsigned long
```

Variations in Integer Types

We can use the sign bit also for holding the value. In such cases, the variable will be called `unsigned int`. The maximum value of an `unsigned int` will be equal to 65535 because we are using the Most Significant Bit (MSB) also for storing the value. The minimum value will obviously be 0.

A long integer is represented as `long int` or simply `long`. The maximum and minimum values of long are given below:

NOTES

LONG	MAX	+	2147483647
LONG	MIN	–	2147483647

Unless otherwise specified, integers or long integers will be signed, i.e., the first bit will be reserved for the sign. The `long int` obviously uses 4 bytes or 32 bits.

The magnitudes of `long` can also be doubled by using an unsigned long integer denoted as `unsigned long`.

However, integers are not suitable for very low values and very large values. This can be overcome by floating point or real numbers.

An integer constant may be suffixed by the letter `u` or `U` to specify that it is an `unsigned integer.` Similarly, if the integer is suffixed with `l` or `L`, it signifies a `long integer`. If we specify `unsigned long integer` we suffix the constant with `ul` or `UL`.

The following are the examples of valid and invalid integers:

Valid integers

```
+345      /* integer */
 345      /* integer */
-345      /* integer */
729u      /* unsigned integer */
729U      /* unsigned integer */
-112345L  /* Long integer */
112345UL  /* Unsigned Long integer */
+112345l  /* Long integer */
112345l   /* Long integer - if no sign precedes, it is a positive
number */
```

Invalid integers

```
345.0       /* decimal point not allowed */
112, 345L   /* no comma allowed */
112 345UL   /* = blank not allowed */
112890345L  /* exceeds the maximum */
+112 345UL /* unsigned cannot have + */
(345l       /* ( not allowed */
-345s       /* illegal characters */
```

We have so far considered only decimal numbers. The C language, however, entertains other type of numbers as well. The octal numbers will be preceded by 0 (zero).

The following are the examples of valid and invalid octal numbers:

Valid octal number

```
0346
0547
0120
```

NOTES

Invalid octal number

```
0394        /* 8 or 9 are not allowed in an octal number */
0 x 345     /* prefix has to be zero only */
```

The C language also supports hexadecimal numbers. Here, since the base is 16, we use alphabets also in the numbers as given in Table 1.5.

Table 1.5

a	or	A	for	10
b	or	B	for	11
c	or	C	for	12
d	or	D	for	13
e	or	E	for	14
f	or	F	for	15

Additionally, hexadecimal numbers will be preceded by `0X` or `ox`, i.e., zero followed by `x`.

The following are the examples of valid and invalid hexadecimal numbers:

Valid hexadecimal numbers

```
0x345
0xA132
0x100
0x20B
```

Invalid hexadecimal numbers

```
0x, 123/* no comma */
0x      /* cannot be empty */
0A00    /* x is missing */
```

Character Constants

A character constant is a single character enclosed in single quotes as in `'x'`. Characters can be alphabets, digits or special symbols.

The following are the examples of valid and invalid character constants:

Valid character constants

```
'A'
'Z'
'C'
'c'
```

Invalid character constants

```
'\n'
'\t'
'\u'
'\b'
AA
```

```
'AA'
"AA"
'1a'
```

NOTES

A character constant represents its integer value as defined in the character set of the machine. Therefore, you can add two characters. For example, the ASCII values of digit 1 = 49 and C = 67. When we add these values we get code 116 whose equivalent character is `t`.

Let us verify this with the following example:

```
/* Example 1.3
demonstrates that chars can be treated like integers*/
#include <stdio.h>
int main()
{
const char ALPHA1='1';
char alpha2='C';
char alpha3;
alpha3=ALPHA1+alpha2;
putchar(alpha3);
return 0;
}
```

Result of the program

```
t
```

Therefore, characters can be treated like integers as well, although they are declared as character variables and constants. Since characters are of type `int`, we could add them. Characters can also be defined as integers as given in the following example:

```
/* Example 1.4
Demonstrates that a char can also be declared as int*/
#include <stdio.h>
int main()
{
int x;
x='1'+'C';
printf("x as integer=%d\n", x);/*x printed as integer*/
printf("x as character=%c\n", x);/*x printed as character*/
return 0;
}
```

Result of the program

```
x as integer=116
x as character=t
```

NOTES

Floating Point or Real Numbers

Let us enumerate the difference between floating point and integer numbers.

- Integers are whole numbers without decimal points but a float has always a decimal point. Even if the number is a whole number, it is written with a decimal point. For instance, 42 is an integer, while 42.0 is floating point number.
- Floating point numbers occupy more space for storage as we have already seen.

A real number in the simple form consists of a whole number followed by the decimal point and also one or more decimal numbers following the decimal point, which makes the fractional part. This form of representation is known as fractional form. It must have a decimal point. It could be either positive or negative. As usual the default sign is positive. No commas or blanks or special characters are allowed in between.

The following are the examples of valid and invalid float types:

Valid floats

```
144.00
226.012
```

Invalid floats

```
+144   /* no decimal point   */
1,44.0 /* comma not allowed  */
```

Scientific Notation: Floating point numbers can also be expressed in scientific notation. For example, `3.0E`$_2$ is a floating point number. The value of the number will be equal to $3.0 \times 10^2 = 300.0$

Instead of the upper case `E`, the lower case `e` can be used as in
`0.453e+05`, which will be equal to `0.453` $\times 10^5 =$ `45300`

There are two parts in the scientific notation of a real number which are as follows:

- Mantissa (before E)
- Exponent (after E)

In the scientific form the following rules are to be observed:

- The mantissa part can be positive or negative.
- The exponent must have at least one digit, which can be a positive or negative integer. Remember the exponent cannot be a floating point number.

`type float` is a single precision number occupying a storage space of 4 bytes.

`type double` represents floating-point numbers of double precision and hence occupies 8 bytes.

If you look at the file `<float.h>` you will find the maximum and minimum floating point numbers as given below.

FLT – MAX	1E	+ 37	maximum floating point number
FLT – MIN	1E	– 37	minimum floating point number

Floating Point Constants: The constants are suffixed as given below:

NOTES

`F` or `f`	–	`float`
no suffix	–	`double`
`L` or `l`	–	`long double`

If an integer is suffixed with `L` or `l`, then it is a `long integer`.

If a float is suffixed with `L` or `l`, then it is a `long double` floating point number.

Examples

Valid floating point constants

```
1.0 e 5
123.0 f       /* float        */
11123.05       /* double          */
23467.34 e 5 l/* long double  */
```

Invalid real constants

```
245.0         /* invalid float, but valid double  */
456           /* It is an integer                         */
1.0 e 5.0     /* exponent cannot be a real number */
```

When they are declared as variables, they can be declared as follows:

```
float  a = 3.12;
float  a, b, c;
float val1;
float val2;
long double val3;
```

The values of constants cannot be altered in programs. They can be defined as follows:

```
const int PRINC = 1000;
const float INT_RATE = 0.12 f;
```

The values of `PRINC` and `INT_RATE` cannot be changed in the program even by introducing another assignment statement when they are declared as constants using const. The following example verifies this statement:

```
/*Example 1.5
Demonstrates that constants cannot be changed
even with assignment statements. To verify, include
statements 7, 8 & 9 in the program by removing
the comment specifiers at the beginning of the program
statement 7 and the end of statement 9*/
#include<stdio.h>
main()
```

NOTES

```
{
const int PRINC =1000;
const float INTST=0.12f;
printf("PRINCIPAL=%d INTEREST=%f\n", PRINC, INTST);
/*PRINC =2000;
INTST=0.24f;
printf("PRINCIPAL=%d INTEREST=%f\n", PRINC, INTST);*/
}
```

Key in the example and execute the program. After successful compilation, you will get the result as follows:

Result of the program

```
PRINCIPAL = 1000; INTEREST = 0.1200
```

Now include the second part of the program by removing `/*` and `*/` at statements 7 and 9, respectively. Earlier this was treated as a comment. Now this part will get included in the program. Now compile it. You will get a compilation error. This is due to your attempt to redefine the constants `PRINC` and `INTST`, which is not allowed. Incidentally, the technique of including or excluding a program segment at will using `/*` and `*/` is a convenient method for program development.

Enumeration Constant

The keyword for this data type is `enum`. We can define *guardian* as follows:

```
enum guardian
{
   father,
   husband,
   guardian
};
```

Note that there are no semicolons, but only a comma after the members except the last member. There is not even a comma after the last member. There is no conflict between both guardians. The top one is `enum` and the bottom one is a member of `enum` guardian. See the similarity between `structure`, `union` and `enum`. The `enum` variables can be declared as follows:

```
enum guardian          emp1, emp2, emp3;
```

This is similar to structures and unions. The first part is the declaration of the data type. The second part is the declaration of the variable.

The initial values can be assigned in a simpler manner as given below:

```
emp1 = husband;
emp2 = guardian;
emp3 = father;
```

We have to assign only those declared as part of the `enum` declaration. Assigning constants not declared will cause error. The compiler treats the enumerators given within the braces as constants. The first enumerator father will

be treated as 0, the husband as 1 and the guardian as 2. Therefore it is strictly as per the natural order starting from 0.

The enumerated data type is never used alone. It is used in conjunction with other data types. We can write a program using `enum` and `struct`. It is given below:

NOTES

```
/*Example 1.6
enum within structure*/
#include <stdio.h>
main()
{
    enum guardian
    {
       father,
       husband,
       relative
    };
    struct employee
    {
       char *name;
       float basic;
       char *birthdate;
       enum guardian guard;
    }emp[2];
    int i;
    emp[0].name="RAM";
    emp[0].basic= 20000.00;
    emp[0].birthdate= "19/11/1948";
    emp[0].guard= father;
    emp[1].name="SITA";
    emp[1].basic= 12000.00;
    emp[1].birthdate= "19/11/1958";
    emp[1].guard=husband;
    for(i=0;i<2;i++)
       {
          if( emp[i].basic ==12000)
          {
             printf("Name:%s\nbirthdate:%s\nguardian:
%d\n",
             emp[i].name, emp[i]. birthdate, emp[i].guard);
          }
       }
}
```

NOTES

Result of the program

```
Name:SITA
birthdate:19/11/1958
guardian: 1
```

The program clearly assigns the relationships between the `employee` and the `guardian`. `enum guardian` is the data type and `guard` is a variable of this type.

However, when you are printing `emp [i].guard`, you are printing an integer. Hence 0, 1 or 2 will only be printed for the status, and this is a limitation. This can be overcome by modifying the program. The program modified with a switch statement is given below:

```
/*Example 1.7 expanding enum*/
#include <stdio.h>
main()
{   enum guardian
    {  father,
       husband,
       relative };
    struct employee
    {
       char *name;
       float basic;
       char *birthdate;
       enum guardian guard;
    }emp[2];
    int i;
    emp[0].name="RAM";
    emp[0].basic= 20000.00;
    emp[0].birthdate= "19/11/1948";
    emp[0].guard= father;
    emp[1].name="SITA";
    emp[1].basic= 12000.00;
    emp[1].birthdate= "19/11/1958";
    emp[1].guard=husband;
    for(i=0;i<2;i++)
    {
       if( emp[i].basic ==12000)
          {printf("Name:%s\nbirthdate:%s\nguardian:",
          emp[i].name, emp[i]. birthdate);
          switch(emp[i].guard)
          {  case 0:printf("father\n");
                break;
             case 1:printf("husband\n");
```

NOTES

```
                break;
            case 2:printf("relative\n");
                break;
        }
    }
  }
}
```

Result of the program

```
Name:SITA
birthdate:19/11/1958
guardian:husband
```

Even though the conversion of an integer to actual name is additional work, `enum` is an useful construct since it improves readability in addition to number of other advantages.

For example, we can define boolean as follows:

```
enum boolean {
        false,
        true
    };
```

Here '`false`' will contain an integer value `0` and true `1`. This can be used to assign values for '`found`' in our sorting programs. We can define found as follows after declaring boolean.

```
enum boolean found;
```

We have allowed the system to assign integer values to the members of `enum`, but we can also assign specific values to the various members of `enum`. When values are assigned, then it takes precedence over what the system assigns.

We can define boolean as:

```
enum boolean
{   yes = 1,
    no = 0
};
```

This is similar to defining using `#define`. The `#define` equivalent for this will be:

```
#define yes 1
#define no 0
```

Although `#define` and `enum` provide a way to associate symbolic names, such as boolean with constants, there are difference between them. The differences are as follows:

- `enum` can generate values itself unlike `#define` where you have to specify the replacement constant.
- The compilers need not check the validity of what is stored in the `enum` variable, but the `#define` replacement constant will be checked for valildity.

- It is possible to print out the values of `enum` variables in symbolic form, but this is not possible with `# define`. Anyway, either of them can be used depending on the context.

NOTES

String Constants

A character constant is a single character enclosed within single quotes. A string constant is a number of characters, arranged consecutively and enclosed within double quotes.

Examples of valid strings:

```
"God"
"is within me"
" "
```

You may be surprised about the third string constant, which has no characters. This is called a `NULL` or empty string and is allowed in C.

The string constant can contain blanks, special characters, quotation marks, etc. within the string. In order to distinguish the end of a string constant, the compiler places a null character `\0` (back slash followed by zero) at the end of each string before the quotation mark. The null character is not visible when the string appears on the screen. The programmer does not include the null character either. It is the compiler which automatically inserts the null character at the end of every string.

Invalid string:

```
'Yoga' /* should be enclosed in double quotes */
```

Symbolic Constants

The format for symbolic constant is as follows:

```
# define name constant
```

For example, we can define:

```
# define INITIAL 1
```

Which defines `INITIAL` as 1.

The `INITIAL` type of definition is called symbolic constants. They are not variables and hence, they are not defined as part of the declarations of variables. They are specified on top of the program before the main function. The symbolic constants are to be written in capital or upper case letters. Wherever the symbolic constant names appear in the program, the compiler will replace them with the corresponding replacement constants defined in the `#define` statement. In this case, 1 will be substituted wherever `INITIAL` appears in the program. Note that there is no semicolon at the end of the `#define` statement.

Any name is an identifier. Just as the name of a person, street or city helps in the identification of a person or a street or a city, the identifier in the C language assigns names to files, functions, constants, variables, etc. An identifier in the C language is defined as a sequence of alphanumeric characters, i.e., alphabets or digits. The first character of an identifier has to be an alphabet. In the C language, lowercase alphabets and uppercase alphabets are considered to be different. For instance, `VAR` and `var` represent different names in the C language.

NOTES

VALID IDENTIFIERS

```
C1
PROC1
P34
VAR_1
EX1
a
bc
Ual1
Aa
```

INVALID IDENTIFIERS

```
1PROGA
4.3
A-B
```

Any function name is also an identifier. For instance, '`printf`' is the name of function available with the C language system. The function helps in printing. Therefore, identifiers can be constructed with alphabets (A...Z), (a...z), (0...9). In addition, underscore can also be used in identifiers. Unless otherwise specified, small letters are usually used for identifiers.

1.8 VARIABLES

A variable is referenced to a named area of storage mechanism that holds a single value, such as numeric or character. The name of each variable in C language is declared to use and its type before using it. The C programming language has two main variable types known as local variables and global variables as follows:

Local Variables

Local variables scope is confined within the block or function where it is defined. Local variables must always be defined at the top of a block. When a local variable is defined it is not initialized by the system. User can initialize it in the program. When execution of the block starts the variable is available and when the block ends the variable dies, i.e., it does not work. The following C program of Example 1.8 shows how local variables are used in the C program:

```
/* Example 1.8.
   Using Local Variables in the program*/
   #include <stdio.h>
   int main()
    {
      int i=4;
      int j=10;
      i++;
      if (j > 0)
      {
```

NOTES

```
    /* i defined in 'main' can be seen */
    printf("i is %d\n",i);
   }
   if (j > 0)
   {
    /* 'I' is defined and so local to this block */
     int i=100;
    printf("i is %d\n",i);
   }/* 'I' (value 100) dies here */
 printf("i is %d\n",i); /* 'I' (value 5) is now visable.*/
 return 0;
  }
```

Result of the program

```
 i is 5
i is 100
i is 5
```

In the above program, ++ increment operator is called for incremental operator and it increases the value of any integer variable by 1. Thus, `i++` is equivalent to statement `i = i + 1;`. You will see – operator in the program which is called decrement operator and it decreases the value of any integer variable by 1. Thus, `i–` is equivalent to statement `i = i - 1;`

Global Variables: Global variable is defined at the top of the program file and it can be visible and modified by any function that is referenced by it. Global variables are initialized automatically by the system when you define them, such as data type and initialize as shown in Table (1.6).

Table 1.6 *Data Type and Initializer*

Data Type	Initializer
int	0
char	'\0'
float	0
pointer	NULL

If same variable name is being used for global and local variable then local variable takes preference in its scope. But, it is not a good practice in C programming to use global variables and local variables with the same name. The C program of Example 1.9 shows how global variables are used in the C program:

```
/* Example 1.9.
  Using Global Variables in the program*/
  #include <stdio.h>
  int i=4;     /* Global definition  */
  int main()
   {
    i++;     /* Global variable   */
    func();
```

NOTES

```
    printf("Value of i = %d — main function\n", i );
    return 0;
  }
func()
  {
    int i=10;   /* Local definition */
    i++;     /* Local variable  */
    printf("Value of i = %d — func() function\n", i );
  }
```

Result of the program

```
Value of i = 11 — func() function
Value of i = 5 — main function
```

In the above program, `i` in the `main()` function is global and will be incremented to `5`. Variable `i` in `func` is incremented to `11`. When control returns to `main()` function the internal variable will die and any reference to `i` will be to the global.

1.8.1 Types of Variables

A variable has two specifiers, namely `data type` and `storage class`.

- data type (e.g. `int, float, char`, etc.)
- storage class (e.g. `auto, static`, etc.)

Data type specifies the types of data stored in a variable. Storage class specifies the segments of the program where the variable is recognized and how long the storage of the value of the variable will last.

There are four types of storage class specifications as follows:

- `automatic`
- `register`
- `static`
- `extern`

You have so far been defining only the data type of the variables but not the storage class. You may wonder then, how your programs worked. Then you as a programmer have to specify the type when it is required to operate in a particular manner. If the storage class is not specified, the compiler will assume the type on its own. The storage class is applicable to all types of variables and is prefixed to the data type declaration as given below:

```
auto char  z ;
extern int a, b, c ;
static float x ;
register char  y ;
```

The basic characteristics of each storage class are discussed in the following sections:

NOTES

Automatic Variables

Any variable declared in a function is assumed to be an automatic variable by default. Automatic Variables:

- **Storage Location:** Except for `register` variables, the other three types will be stored in memory.
- **Scope:** `auto` variables are declared within a function and are local to the function. This means the value will not be available in other functions.

Any variable declared within a function is interpreted as an `auto` variable, unless a different type of storage class is specified.

`Auto` variables defined in different functions will be independent of each other, even if they have the same name.

`Auto` variables are local to the block in a function. If an `auto` variable is defined on the top of the function after the opening brace, then it is available for the entire function. If it is defined later in a block after another opening brace, it will be valid only till the end of the block, i.e., up to the corresponding closing brace.

The following program illustrates the concept.

/*Example 1.10*

```
/* to demonstrate the use of auto variable*/
#include <stdio.h>
int main()
{
    auto int x=10;
    void f1(int x);
    int f2(int x);
  {
       auto int x =20;
       printf("x = %d in the first block\n", x);
       x=f2(x);/*20 is passed to f2 and returned value assigned
to x*/
       printf("x = %d after the return from f2\n", x);
  }
 printf("x = %d after the first block\n", x);
  {
       auto int x=30;
       printf("x = %d in the second block\n", x);
  }
 printf("x = %d after the second block\n", x);
 f1(x);/*x=10 is passed to the function f1*/
  printf("x = %d after return from function will be 10\n",
x);
  }
 void f1(int a)
  {
```

NOTES

```
        auto float x=5.555;/*integer x will be lost*/
        printf("x = %f in the function\n", x);
    }
     int f2(int x)
    {
        auto int y=100;
        y+=x; /*y will be 120*/
        printf("y = %d in the function\n", y);
        return y;
    }
}
```

Execute the program and you will get the following results:

Result of the program

```
x = 20 in the first block
y = 120 in the function
x = 120 after the return from f2
x = 10 after the first block
x = 30 in the second block
x = 10 after the second block
x = 5.555000 in the function
x = 10 after return from function will be 10
```

This gives a clear idea about the scope of the `auto` variables.

- **Initial Values:** The `auto` variable will contain some garbage values unless initialized. Therefore, they must be initialized before use.
- **Life:** How long will the value stored in the `auto` variable last?

It will last as long as the function or block in which it is defined is active. If the entire function has been executed and the value has been returned, the value of the `auto` variables of the functions will be lost. You cannot call it later. This point should be noted.

External Variables

External variables are also known as global variables. What is global? The scope of external variables extends to all the functions of a program. You have so far created all the functions in a single file. You can create functions in more than one file. However, for the sake of simplicity, assume that all the functions are in one file. Global variables will be declared like other variables with a storage class specifier `extern`. For example,

```
extern  int a, b
extern  float c, d
```

The scope of the variables starts from the point of declaration to the end of the program. The value of the external variable at any point of time is that of the last assignment. Assume that the main function may assign `a = 10`. The function `z` may then use it and perform a calculation and at the end assign a value 20. If

printed at that point of time, the value will be 20. It may be called by another function p where its value may become zero. If, at this point of time, the main function calls or z calls it, the value will be 0. Thus, external variable is accessible and transparent to all the functions below it. We will write a program to demonstrate this concept.

NOTES

/*Example 1.11*

```
/* to demonstrate use of external variable*/
#include <stdio.h>
extern int ext_a=10;
int main()
{
    int f1(int a);
    void f2(int a);
    void f3();
    printf("ext_a = %d in the main function\n", ext_a);
    f1(ext_a);
    printf("ext_a = %d after the return from f1\n", ext_a);
    f2(ext_a);
    printf("ext_a = %d after the return from f2\n", ext_a);
    ext_a*=ext_a;
    printf("ext_a = %d \n", ext_a);
    f3();
    printf("ext_a = %d after return from f3\n", ext_a);
}
int f1(int x)
{
    ext_a-=10;
    return ext_a;
}
void f2(int x)
{
    ext_a+=20;
}
void f3()
{
    ext_a/=100;
}
```

Result of the program

```
ext_a = 10 in the main function
ext_a = 0 after the return from f1
ext_a = 20 after the return from f2
ext_a = 400
ext_a = 4 after the return from f3
```

How does the program work?

NOTES

ext_a is declared as an external variable with value 10 before main(). Therefore, ext_a will be recognized all through the program .

f1, f2 and f3 are functions.

f1 returns an integer and f2 returns void, i.e., it does not return a value.

f3 neither receives nor returns any value.

In the first printf(), we get ext_a = 10.

Now f1 is called. In f1 ext_a = 10 is passed as an argument.

The value of ext_a is 0 now in function f1. The second printf in main prints ext_a = 0

Now f2 is called .

ext_a becomes 20 now. It does not return any value. However, the third printf prints the value as 20. How does this happen? It is because the current value of ext_a is known to main even without f2 passing it.

We discussed that a function can return only one value. However, by using a global variable, you can overcome this limitation as follows:

You square ext_a, i.e., ext_a = 400 now.

The 4th printf() statement confirms this. Now, you call f3. In spite of the fact that you neither passed an argument nor returned any value from f3, ext_a is known to f3 as 400. Then, 100 divides ext_a. Therefore, ext_a will be 4 as confirmed by the fifth printf() statement.

This program illustrates the concept of external variables in simpler situations where the name of the global variable is not assigned to the function's local variables.

It is perfectly legal to use the same name for different local variables in a function. We can even use the name and declare it as another data type.

For example, you can define ext_a as a float in another function f1. Then how is the conflict to be resolved? You will reserve the answer to the question for a few minutes.

You should be careful while handling external variables because the variables may be disturbed in a remote corner inadvertently. Global variables when declared on top of main() can be identified easily and therefore, the storage class specifier extern need not be specified in such situations. If it cannot be easily recognized by declaration elsewhere in the program, it should be specified clearly.

The initial value of an external variable is zero, if not assigned. The life of the variable is till the termination of program execution. The scope extends from the point of declaration till the end. It will be stored in memory.

Static Variables

The initial value of static variables is zero unless otherwise specified. This is also stored in memory. Static variables are declared as follows:

```
static int x, y, z ;
static char a ; etc.
```

NOTES

`static` variables are local to the functions and exist till the termination of the program. Therefore, when the program leaves the function, the value of the `static` variable is not lost. If the program calls the function again, the `static` variable will execute the function with the value it already possesses. Assume that `f1` is a function containing a `static` variable as given below:

```
main ()
{
   int f1 (—);
   f1 (—);
}
int f1 (—)
{
   static int var = 0 ;
}
```

When `f1` is called the first time, `var` will be initialized to zero. If `var` is finally assigned the value 10 at the end of `f1`, then `var = 10` will remain till the program stops execution and if main calls `f1` again, the value of `var` will not be initialized to 0 again but will remain as 10. The initialization `var = 0` will not have any effect. However, `var` can further be modified depending on the statements in `f1`. Had it been an `auto` variable, `var` would have been initialized each time to zero. This is essentially due to the value of `auto` variable being lost immediately after the program leaves the block. This is one of the differences between `static` and `auto` variables. `Static` variables; however, will not be known outside the function, i.e., in other functions, such as `main()` or any other functions called by `main()`.

Now, consider the conflict arising out of external variables and local variables (`auto/register, static`) having the same names. In such cases, the local variables take precedence over the external variables. This means that in a function the local variable of the same name would only be recognized. The function will be blind to the external variable of the same name. However, the global variables of other names will be recognized as explained already. All other conditions remain the same. The value of a local variable does not affect the global variable and vice versa. The initial value of the local variable will be dependent upon whether it is `static` or `auto`. The program in Example 1.12 explains the concept of the working of the different storage classes.

/*Example 1.12*

```
/* to demonstrate scope of variables*/
#include <stdio.h>
char chara,charb,charc; /*global variables*/
int main()
{
    int disp();
    char prn(char m);
    chara='x';
    charb='y';
```

NOTES

```
        charc='z';
        prn(charc);
        putchar(chara);
        disp();
        putchar(charb);
        disp();
    }
    int disp()
    {
        static int charb;
        charb=charb+1;
        printf("%d\n",charb);
        return charb;
    }
    char prn(char charn)
    {
        auto char chara;
        putchar(charc);
        chara=charn;
        putchar(chara);
        return chara;
    }
```

Result of the program

```
zzx1
y2
```

Let us understand how the program functions.

You declare `chara, charb` and `charc` as global variables and in `main()`, you assign *chara* = x, *charb* = y, *charc* = z.

You have declared `disp` as a function passing no variable but returning an integer.

You have declared `prn` as a function passing and returning a character.

You call function `prn`. You have a `chara` of type `auto` in `prn`. The statement `putchar (charc)` will display the value of `charc` in the `main` function, which is `z`.

The next `putchar (chara)` in `prn` will display `z` because of `chara = charc. z` is returned to `main`.

Now `putchar (chara)` will print `x` and not `z` since in the `main` function, `chara` refers to the global variable.

Now you call `disp ()`. Although, you have not initialized it, the initial value of `b` will be zero and therefore, it will print 1. Now the program returns to `main()`.

NOTES

The next putchar (charb) will print y because the global variable is active in the main function. Now, you call disp again. Since the old value of b is not lost, the next time, the program prints 2. Thus, the value of a static variable is maintained between function calls. Local variables get precedence over global variables.

Register Variables

Register variables have characteristics similar to auto variables. The only difference between them is that while auto variables are stored in memory, register variables are stored in the register of the CPU. The initial value will be an unpredictable or garbage value; the variables are local to the block and they will be available as long as the blocks are active.

Why then do you need to declare one more storage class? The CPU registers respond much faster than the memory. After all, you want to access, store and retrieve the stored variables faster so that the computing time is reduced. Registers are faster than memory. Therefore, those variables which are used frequently can be declared as register variables.

They are declared as,

```
register int i ;
```

A memory's basic unit may be 1 byte but depending on the size of the variable even 10 contiguous bytes of memory can be used to store a long double variable. Such an extension of size is not; however, possible in the case of registers. The registers are of fixed length like 2 bytes or 4 bytes and therefore, only integer or char type variables can be stored as register variables. Since registers have many other tasks to do, register variables may be defined sparingly. If a register variable is declared and if it is not possible to accept it as a register variable for whatever reasons, the computer will treat it as an auto variable. Therefore, the programmer may specify a frequently used variable in a program as a register variable in order to speed up the execution of the program.

1.8.2 Declaration of Variable

The names of variables and constants are identifiers. The names are made up of alphabets, digits and underscore, but the first character of an identifier must be an alphabet. C allows up to 31 characters for the identifier (names) and therefore, the naming of the variables should be carried out in an easily understandable manner. For example, in the program for the calculation of,

```
Simple interest I = pnr/100,
```

you can declare them with actual names,

```
p = principal, r = rate_of_interest, n = number_of_ years
```

Naturally, programmers may not like typing long names for fear of making mistakes elsewhere in the program apart from being reluctant to increase their typing workload. Meaningful names can, however, be assigned to data types even with few letters. For example,

```
p = princ; r = intrate; n = years
```

NOTES

Some compilers may allow names with up to 31 (thirty one) characters, but may consider the first eight characters for all purposes. Hence, a programmer could coin shorter yet meaningful names, instead of using single alphabets for variable names. One should, however, be careful not to use the reserved words, such as 32 keywords, for variable names as they have a specific meaning to the compiler. If they are used as variable names, then the compiler will get confused. Be careful not to use the reserved words as identifiers.

A program to find out the square of integer 5 is given as follows:

```
/*Example 1.13*/
/*program to find square of 5*/
#include <stdio.h>
int main()
{
printf("square of %d= %d", 5, 5*5);
return 0;
}
```

Result of the program

```
square of 5= 25
```

You have now achieved the objective of finding the square of 5. Later on, you may want to find out the square of another number, say 7, for example. We would have to write the same program again replacing 5 by 7 and then compile and run it. This would waste a lot of time. To save time, we can, therefore, write a general purpose program as given below:

```
/*Example 1.14*/
/*program to find square of any given number*/
#include <stdio.h>
int main()
{
int num;
printf("Enter the integer whose square is to be found\n");
scanf("%d", &num);
printf("square of %d= %d", num, num*num);
return 0;
}
```

Here, we define `num` as an integer variable. When '`&`' precedes `num`, it indicates the memory address of `num`.

At the first `printf`, the message appears as it is and the cursor goes to the next line because of the new line character `\n` at the end of the message, but before the closing quotation mark. The next statement is new to you. It is used to

receive an integer typed on the console. You can type in any integer number, and the number typed will be stored in the memory at the memory location named '`num`'. The purpose of the statement is, therefore, to get the integer (because of the appearance of `%d` within quotes) and it is stored at the memory address '`num`'. The next statement prints the number typed and its square.

NOTES

When you execute the program, the following message appears on the monitor:

Enter the integer whose square is to be found.

Since we want to find out the square of 25 type:

```
25
```

Promptly, the reply will be as shown as follows:

```
square of 25 = 625
```

The next time you may want to find out the square of another number, say 121. Then simply run the program and when prompted, type 121 to get the answer.

Here the number whose square has to be found out has been declared as a variable. The variable has a name and is stored at the location pointed to by the variable name. Therefore, the execution of the program for finding out the square of any number is easy with the above modification as in Example 1.13.

Variables and constants are fundamental data types. A variable can be assigned only one value at a time, but can change value during program execution. A constant, as the name indicates, cannot be assigned a different value during program execution. For example, `PI`, if declared as a constant, cannot have its value varied in a given program. If `PI` has been declared as a constant = 3.14, it cannot be reassigned any other value in the program. Programs may declare a number of constants. Variables are similarly useful for any programming language. If `PI` has been declared as a variable, then it can be changed in the program to any value. This is one difference between a variable and a constant. Whether an identifier is constant or variable depends on how it is declared. Both variables and constants belong to one of the data types like `int, float`, etc. The convention in 'C' is to indicate the names of constants by the uppercase letters.

```
PI
SIGMA
```

Variable names are, on the other hand, indicated by the lower case letters.

```
int a
float xy
```

1.8.3 Assigning Value to Variable

The C programmer should understand how much memory storage each variable type occupies in the IDE used by him. The following example will help us to find the size of each variable type. The result will be in terms of bytes occupied by the variable type. A new keyword `sizeof` is used to find out the size. The syntax for using the keyword is as follows:

```
sizeof (<data type>)
```

or

```
sizeof (<expression>)
```

Consider the following example:

NOTES

```
/*Example 1.15*/
/*program to find out the sizes of various types of integers*/
#include<stdio.h>
int main()
{
printf("size of char =%d\n", sizeof(char));
printf("size of short=%d\n", sizeof(short));
printf("size of int =%d\n", sizeof(int));
printf("size of unsigned int=%d\n", sizeof(unsigned));
printf("size of long int=%d\n", sizeof(long));
printf("size of unsigned long int=%d\n", sizeof(unsigned
long));
printf("size of float =%d\n", sizeof(float));
printf("size of double=%d\n", sizeof(double));
printf("size of long double%d\n", sizeof(long double));
}
```

Result of the program

```
size of char = 1
size of short = 2
size of int = 2
size of unsigned int = 2
size of long int = 4
size of unsigned long int = 4
size of float = 4
size of double = 8
size of long double = 10
```

Therefore, it is obvious that a long double occupies 10 bytes and stores long floating-point numbers with double precision.

Note that the size of `short int` will be either equal to or less than the size of an integer variable.

Variables, which require more than 1 byte for storage, will be stored consecutively in the memory.

Check Your Progress

5. State about the fundamental data types.
6. Define the term string constants.
7. What is local variable?
8. Define the term variables.

NOTES

1.9 ANSWERS TO 'CHECK YOUR PROGRESS'

1. C language was designed and developed by Brian Kernighan and Dennis Ritchie, at The Bell Research Labs in 1972. The 'C' programming language is one of the most popular computer languages in today's computer world. It was created so as to allow the programmer access to almost all of the machine's internals—registers, I/O slots and absolute addresses.
2. Structured programming (also known as procedural programming) was a powerful and an easy approach of writing complex programs. In procedural programming, programs are divided into different procedures (also known as functions, routines or subroutines) and each procedure contains a set of instructions that performs a specific task.
3. Three characteristics of good programming language are:
   - Write only one statement per line.
   - Coin meaningful names for constants, variables and functions.
   - Use capitals for names of constants.
4. The reserved words have special meaning within the language and are predefined in the language's formal specifications. Typically, reserved words include labels for primitive data types in languages that support a type system and identify programming constructs, such as data types, loops, blocks, conditionals and branches. The list of reserved words in a language are defined when a language is developed. Reserved words may not be redefined by the programmer, unlike predefined functions, methods or subroutines, which can often be overridden in some capacity.
5. An item that holds data is also called an object. An object has a name or identifier associated with it. Each object can hold a specific type of data. There are five basic data types in C, as follows:
   - Character
   - Integer
   - Real Numbers
   - Void (Comprising an Empty Set of Values)
   - Enum (Which will be Introduced Later)
6. The string constant can contain blanks, special characters, quotation marks, etc., within the string. In order to distinguish the end of a string constant, the

NOTES

compiler places a null character \0 (back slash followed by zero) at the end of each string before the quotation mark. The null character is not visible when the string appears on the screen. The programmer does not include the null character either. It is the compiler which automatically inserts the null character at the end of every string.

7. Local variables scope is confined within the block or function where it is defined. Local variables must always be defined at the top of a block. When a local variable is defined it is not initialized by the system. User can initialize it in the program.
8. A variable is referenced to a named area of storage mechanism that holds a single value, such as numeric or character. The name of each variable in C language is declared to use and its type before using it. The C programming language has two main variable types known as local variables and global variables.

1.10 SUMMARY

- C language designed and developed by Brian Kernighan and Dennis Ritchie, at The Bell Research Labs in 1972, the 'C' programming language is one of the most popular computer languages in today's computer world. It was created so as to allow the programmer access to almost all of the machine's internals—registers, I/O slots and absolute addresses.
- C is a general-purpose high-level programming language that was developed by Dennis Ritchie at the Bell Laboratories, USA in the early 70s.
- Structured programming (also known as procedural programming) was a powerful and an easy approach of writing complex programs. In procedural programming, programs are divided into different procedures (also known as functions, routines or subroutines) and each procedure contains a set of instructions that performs a specific task.
- High level programming languages, such as FORTRAN, Pascal, BASIC, C and COBOL are known as function oriented or procedure oriented languages. They have been used for developing important and mission critical applications.
- The principle of Object Oriented Programming (OOP) is to combine both data and the associated functions into a single unit called a class.
- The first executable instruction in all C programs is the `main()` function. That is, programs begin their execution from this instruction. Once all the instructions in the `main()` function are executed, the control passes out of `main()`, terminating the entire program.
- The reserved words have special meaning within the language and are predefined in the language's formal specifications. Typically, reserved words include labels for primitive data types in languages that support a type system and identify programming constructs, such as data types, loops, blocks,

conditionals and branches. The list of reserved words in a language are defined when a language is developed. Reserved words may not be redefined by the programmer, unlike predefined functions, methods or subroutines, which can often be overridden in some capacity.

- Keywords are the words whose meaning has already been explained to the 'C' complier, which are also called as reserved words.
- Data is used in a program to get information. In a program used to find out the greater of two numbers, the numbers are data, and the output which says which number is greater, is information. C is a versatile language and handles many different types of data in an elegant manner.
- A real number in the simple form consists of a whole number followed by the decimal point and also one or more decimal numbers following the decimal point, which makes the fractional part. This form of representation is known as fractional form.
- The string constant can contain blanks, special characters, quotation marks, etc. within the string. In order to distinguish the end of a string constant, the compiler places a null character \0 (back slash followed by zero) at the end of each string before the quotation mark. The null character is not visible when the string appears on the screen. The programmer does not include the null character either. It is the compiler which automatically inserts the null character at the end of every string.
- Local variables scope is confined within the block or function where it is defined. Local variables must always be defined at the top of a block. When a local variable is defined it is not initialized by the system. User can initialize it in the program.
- A variable is referenced to a named area of storage mechanism that holds a single value, such as numeric or character. The name of each variable in C language is declared to use and its type before using it. The C programming language has two main variable types known as local variables and global variables.
- Global variable is defined at the top of the program file and it can be visible and modified by any function that is referenced by it.

1.11 KEY TERMS

- **C language:** C is a general-purpose high-level programming language that was developed by Dennis Ritchie.
- **Procedural programming:** Structured programming (also known as procedural programming) was a powerful and an easy approach of writing complex programs.
- **Object Oriented Programming (OOP):** High level programming languages, such as FORTRAN, Pascal, BASIC, C and COBOL are known as function or procedure oriented language.

- **Reserved words:** Keywords are the words whose meaning has already been explained to the 'C' complier, which are also called as reserved words.
- **Object:** An item that holds data is also called an object.
- **Symbolic constants:** The `INITIAL` type of definition is called symbolic constants.
- **Global variables:** Global variable is defined at the top of the program file and it can be visible and modified by any function that is referenced by it.

1.12 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Define the classification of programming language.
2. State the concept of structured programming.
3. Why object oriented programming language is used?
4. Write the features of good computer program.
5. Define the term syntax errors.
6. Name the tokens used in C.
7. Define the term data types.
8. What are integer constants?
9. Differentiate between local and global variables.
10. How a value is assigned to a variable in C?

Long-Answer Questions

1. Briefly discuss the significant features of C language and its advantage with the help of examples.
2. Discuss about the procedural programming with the help of relevant examples.
3. Describe the structure of C programming with the help of relevant examples.
4. Briefly explain the execution method of C program giving appropriate examples.
5. Discuss in detail the token in C with the help of example program.
6. Briefly discuss the data types with the help of relevant examples.
7. Discuss about the constants in C and its types giving appropriate examples.
8. Describe the types of variables with the help of example.
9. Briefly explain the register variables giving appropriate examples.

1.13 FURTHER READING

Gottfried, Byron S. 1996. *Programming with C*, Schaum's Outline Series. New York: McGraw-Hill.

Jeyapoovan T. 2006. *Computer Programming - Theory & Practice*. New Delhi: Vikas Publishing House Pvt. Ltd.

Khurana, Rohit. 2005. *Object Oriented Programming*. New Delhi: Vikas Publishing House Pvt. Ltd.

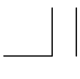
Kanetkar, Yashwant. 2003. *Let Us C*. New Delhi: BPB Publication.

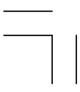
Saxena, Sanjay. 2003. *A First Course in Computers*. New Delhi: Vikas Publishing House Pvt. Ltd.

Subburaj, R. 2000. *Programming in C*. New Delhi: Vikas Publishing House Pvt. Ltd.

Ghosh, Smarajit. 2009. *All of C*. New Delhi: PHI Learning Pvt Ltd.

Bronson, Gary J. 2000. *A First Book of ANSI C*, 3rd edition. California: Thomson, Brooks Cole.





NOTES

UNIT 2 OPERATORS AND EXPRESSIONS

Structure

2.0 Introduction
2.1 Objectives
2.2 Operators in C
 2.2.1 Arithmetic Operators
 2.2.2 Relational Operators
 2.2.3 Logical Operators
 2.2.4 Assignment Operators
 2.2.5 Conditional Operator
 2.2.6 Increment and Decrement Operators
 2.2.7 Bitwise Operator
 2.2.8 Special Operator
2.3 Expression in C
2.4 Operator Precedence and Associativity
2.5 Input and Output in C
 2.5.1 Use of `printf()`
 2.5.2 Single Character Input/Output
 2.5.3 Strings—`gets()` and `puts()`
2.6 Answers to 'Check Your Progress'
2.7 Summary
2.8 Key Terms
2.9 Self Assessment Questions and Exercises
2.10 Further Reading

2.0 INTRODUCTION

The symbols which represent various computations (such as, addition, subtraction, etc.) performed on various data items are known as operators. The data items on which the operators act are known as operands. Depending on the function they perform, C operators are classified into various categories. This includes arithmetic operators, relational operators, logical operators, the conditional operator assignment operators, bitwise operators and other operator.

The input and output functions of a computer facilitate interactions between the computer and the user. The user has to input data in order to process it in the computer and get the result as the output. The `printf ()` statement can be programmed to give the output in the desired manner. The functions `gets ()` and `puts ()` are appropriate when strings are to be received from the screen or sent to the screen without errors.

In this unit, you will study about the operators in C, arithmetic operators, relational operator, logical operator, assignment operator, increment operators, decrement operator, conditional operators, bitwise operator, special operator, expression in C, operator precedence and associativity, input and output statement, formatted input and formatted output.

2.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the operators used in C
- Discuss the arithmetic and relational operators
- Understand the significance of logical and assignment operator
- Define the increment and decrement operator
- Analyse the conditional and bitwise operator
- Explain the special operator
- Describe about the expression in C programming
- Discuss the operator precedence and associativity
- Understand the input and output statements
- Define the formatted input and formatted output

2.2 OPERATORS IN C

Various types of operators are explained below.

2.2.1 Arithmetic Operators

The basic arithmetic operators are:

+ addition, e.g., `c = a + b`

− subtraction, e.g., `c = a − b`

`*` multiplication, e.g., `c = a * b`

/ division, e.g., `c = a/b`

% modulus, e.g., `c = a % b`

When we divide two numbers, we get a quotient and a remainder. To get the quotient we use `c = a/b;`

```
/* c   contains quotient */
```

To get the remainder we use `c = a % b;`

```
/* c   contains the remainder */.
```

% is also popularly called modulus operator. Modulus cannot be used with floating-point numbers.

Therefore, `c = 100/6;` will produce `c = 16`.

`c = 100 % 6`, will produce `c = 4`.

In expressions, the operators can be combined.

For example, `a = 100 + 2/4;`

What is the right answer?

Is it 100 + 0.5 = 100.5

or 102/4 = 25.5

NOTES

To avoid ambiguity, there are defined precedence rules for operators in 'C'. Precedence refers to the evaluation order of operators. However, in an expression there may be the same type of operators at a number of places, which may lead to ambiguity as to which one to evaluate first. In addition, more than one operator may have the same precedence. For example, `*` and `/` have the same precedence. To avoid ambiguity in such cases, there is a rule called associativity. The precedence and associativity of operators in 'C' are given in Table 2.4.

Have a look at Annexure 1. Associativity says either left to right or vice versa. This means that when operators of the same precedence are encountered, the operators of the same precedence have to be evaluated from left to right, if the associativity is left to right.

Now refer to the previous example. Since `/` has precedence over `+`, the expression will be evaluated as `100 + 0.5 = 100.5.`

In the precedence table, operators in the same row have the same precedence. The lower the row, the lower the precedence.

For example, `()`, which represents a function, has a higher precedence than `!`, which is in the lower row. Similarly `*` and `/` have higher precedence over `+` and `-`.

Whenever you are in doubt about the outcome of an expression, it would be better to use parentheses to avoid the ambiguity.

Consider the following examples:

```
1) 12 - 3 * 3 = 12 - 9 = 3 and not 27.
2) 24 / 6 * 4 = 4 * 4 = 16 and not 1.
3) 4 + 3 * 2 = 4 + 6 = 10 and not 14.
4) 8 + 8 / 2 - 4 * 6 / 2
   = 8 + 4 - 4 * 6 / 2
   = 8 + 4 - 24 / 2
   = 8 + 4 - 12 = 0
```

Note the steps involved in the previous example.

2.2.2 Relational Operators

Two variables of the same type may have a relationship between them. They can be equal or one can be greater than the other or less than the other. You can check this by using relational operators. While checking, the outcome may be true or false.

For example, if `a = 5` and `b = 6`;

`a` equals `b` is false.

`a` greater than `b` is false.

`a` greater than or equal to `b` is false.

a less than b is true.

a less than or equal to b is true.

Any two variables or constants or expressions can be compared using **relational operators**. Table 2.1 below gives the relational operators available in 'C'.

Table 2.1 *Relational Operations in C*

Operator	Example	Read as
< less than	a < b	Is a < b
> greater than	a > b	Is a > b
<= less than or equal to	a <= b	Is a < or = b
>= greater than or equal to	a >= b	Is a > or = b
== equal to	a == b	Is a equal to b
!= not equal to	a != b	Is a not equal to b

Note that for checking equality the double equal sign is used, which is different from other programming languages. The statement a = b assigns the value of b to a. For example, if b = 5, then a is also assigned the value of 5. The statement a == b checks whether a is equal to b or not. If they are equal, the output will be true; otherwise, it will be false.

Now look at their precedence from Table 2.4.

>   >=   <   <=   have precedence over   ==   !=

Note that arithmetic operators + - * / have precedence over the relational and logical operators.

Therefore, in the following statement:

```
(x - 3 > 5)
```

x - 3 will be evaluated first and only then the relation will be checked. Therefore, there is no need to enclose x - 3 within parenthesis as in ((x - 3) > 5.

2.2.3 Logical Operators

You can use logical operators in programs. These logical operators are:

&& denoting logical And

|| denoting logical Or

! denoting logical Negation

The relational and logical operators are evaluated to check whether they are true or false. 'True' is represented as 1 and 'False' is represented as 0.

It is also by convention that any non-zero value is considered as 1 (true) and zero value is considered as 0 (false).

For example,

```
if     (a - 3)
       {s1}
else
       {s2}
```

If a is 5, then s1 will be executed.

If a = 3, then s2 will be executed.

If a = –5, s1 will still be executed.

To summarize, the relational and logical operators are used to take decisions based on the value of an expression.

2.2.4 Assignment Operators

Assignment operators are written as follows:

identifier = expression;

For example,

```
i = 3;
```

Note: 3 is an expression

```
const A = 3;
```

'C' allows multiple assignments in the following form:

identifier 1 = identifier 2 = ...= expression.

For example,

```
a = b = z = 25;
```

However, you should know the difference between an assignment operator and an equality operator. In other languages, both are represented by =. In 'C' the equality operator is expressed as = = and assignment as =.

Shorthand Assignment Operators

You have been looking at simple and easily understandable assignment statements. This can be written in a different manner when the RHS includes LHS; or in other words, when the result of computation is stored in one of the variables in the RHS.

The general form is `exp1 = exp1 + exp2`.

This can be also written as exp1 + = exp2.

Examples:

simple form	special form
`a = a + b;`	`a += b;`
`a = a + 1;`	`a += 1;`
`a= a - b;`	`a - = b;`
`a = a - 2;`	`a - = 2;`
`a = a*b;`	`a*= b;`
`a = a*(b + c);`	`a*= b + c;`

```
a = a/b;            a / = b;
a = a/2;            a / = 2;
d = d - (a + b);    d - = a + b
```

NOTES

The **assignment operators** `=, + =, - =, * =, / =, % =`, have the same precedence or priority; however, they all have a much lower priority or precedence than the arithmetic operators. Therefore, the arithmetic operations will be carried out first before they are used to assign the values.

2.2.5 Conditional Operator

The condition operator is also termed as ternary operator and is denoted by?:

The syntax for the conditional operator is as follows:

(Condition)? statement1: statement2;

What does it mean? If the condition is true, execute statement1; else, execute statement2. The conditional operator is quite handy in simple situations as follows:

```
(a > b)?  print a greater
       :  print b greater
```

Thus, the operator has to be used in simple situations. If nothing is written in the position corresponding to `else`, then it means nothing is to be done when the condition is false.

An example is as follows:

```
/*Example 2.1
Demonstrates use of the ? operator*/
#include <stdio.h>
int main()
{
    unsigned int a,b;
    printf ("enter two integers\n");
    scanf("%u%u", &a, &b);
    (a==b)?printf("you typed equal numbers\n"):
    printf("numbers not equal\n");
}
```

Result of the program

```
enter two integers
123 456
numbers not equal
```

2.2.6 Increment and Decrement Operators

C contains two increment and decrement operators which are present in postfix and prefix forms. Both forms are used to increment or decrement the appropriate variable. The statement `++i` (prefix form) increments `i` before using its value, while `i++` (postfix form) increments it after its value has been used. Both the forms will produce different outputs when evaluated.

Increment Operator ++

The **++ (increment) operator** adds 1 to the value of a scalar operand or if the operand is a pointer then increments the operand by the size of the object to which it points. The operand receives the result of the increment operation. The operand must be a modifiable `lvalue` of arithmetic or pointer type. You can put ++ before or after the operand. If it appears before the operand, the operand is incremented first and then used in the expression. If you put ++ after the operand, the value of the operand is used in the expression *before* the operand is incremented. The following statement shows the increment operator concept:

```
play = ++play1 + play2++;
```

This statement is similar to the following expressions:

```
int temp, temp1, temp2;
temp1 = play1 + 1;
temp2 = play2;
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 + 1;
play = temp;
```

The result has the same type as the operand after integral promotion. The usual arithmetic conversions on the operand are performed. In prefix operation, the value is incremented or decremented first and then applied, while in postfix the value is applied first and then incremented or decremented.

```
/*Example 2.2
#include<stdio.h>
main( )
{
    int i = 3,j = 4,k;
    k = i++ + ++j;
    printf("i = %d, j = %d, k = %d",i,j,k);
}
```

Result of the Program

```
i = 4, j = 5, k = 8
```

Decrement Operator

The **– (decrement) operator** subtracts `1` from the value of a scalar operand or if the operand is a pointer decreases the operand by the size of the object to which it points. The operand receives the result of the decrement operation. The operand must be a modifiable `lvalue`. You can put – before or after the operand. If it appears before the operand the operand is decremented and the decremented value is used in the expression. But if – appears after the operand then the current value of the operand is first used in the expression and then the operand is decremented. The following statement shows the decrement operator concept:

```
play = —play1 + play2—;
```

NOTES

This statement is similar to the following expressions:

```
int temp, temp1, temp2;
temp1 = play1 - 1;
temp2 = play2;
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 - 1;
play = temp;
```

2.2.7 Bitwise Operator

Binary operators are usually assumed to be operators which operate on binary digits, i.e., bits. However, this is not the convention in C. Binary operators refer to those operators, which operate on 2 operands, i.e., `a + b, a % b`, etc. In the same way, unary operators are those, which operate on single operand, such as `a++, a--` , etc.

Bitwise operators access the internal representation of the numbers, which are bits 0 or 1. These operators apply only to the integer family operands including `char`. There are six operators for bitwise operation or manipulation. The operators and their symbols are as follows:

```
& bitwise AND
| bitwise OR
^ bitwise exclusive OR
<< left shift
>> right shift
~ one's complement
```

Conversion of Decimal to Binary

In C, you can represent numbers as decimal, octal and hexadecimal numbers. They are all stored as integers. Since the decimal number is an integer, it is stored in 1 word or 2 bytes. If you had a mechanism for storing a bit, it would have been more economical. As there are no exclusive representations for binary numbers, they have to be stored using the existing storage mechanism. Therefore, each bit of a binary number will also be considered as an integer and stored one after another. If a short integer is available, that is enough, since you have to store only a 0 or 1. Each bit has to be stored as a word of 1 byte or 2 bytes, and the 8 bits of a number have to be stored as an array of 8 integers.

Any problem involving bitwise operation involves conversion of the numbers into binary numbers. Now, consider an algorithm to convert decimal into binary. You repetitively divide the number by 2 and the remainder is assembled to form the binary number.

The following is the example to convert `19` into binary.

```
2  19
2  9 - 1
2  4 - 1
2  2 - 0
```

```
2  1 - 0
   0 - 1
```

The equivalent of 19 in binary is 10011.

NOTES

Therefore, when you divide by 2, the first remainder is the Least Significant Bit (LSB) and the last remainder is the Most Significant Bit (MSB). For simplicity, you assume that you convert them into 8-bit numbers.

Algorithm 1 gives the method of converting a decimal number into an array of bits.

Algorithm 1

```
Step 1:  Store the decimal number in integer variable
num.
Step 2:  Declare an array of size 8 to store 8 bits.
Step 3:  for (i = 0 ; i <= 7; i + + )
    {
    b [ i ] = num % 2 ;
    num = num / 2 ;
    }
```

Let us convert 19 using the algorithm to confirm.

```
Step 1 b [ 0 ]  =     19 % 2      =     1
Step 2 b [ 1 ]  =     9 % 2       =     1
Step 3 b [ 2 ]  =     4 % 2       =     0
Step 4 b [ 3 ]  =     2 % 2       =     0
Step 5 b [ 4 ]  =     1 % 2       =     1
Step 6 b [ 5 ]  =     0 % 2       =     0
Step 7 b [ 6 ]  =     0 % 2       =     0
Therefore, 19₁₀ =     0010011
b [ 0 ]         =     LSB
b [ 7 ]         =     MSB
```

Now you can understand all the bitwise operations easily.

In software, you organize the digits byte-wise, but the hardware handles it bitwise. Therefore, bitwise operators are very useful for directly interacting with the hardware. However, when you do programming you will be using decimal numbers. The bitwise operator recognizes the number, carries out the bitwise operation and gives the result in decimal numbers. To check whether the operation is correct, you have to convert the operand and the result into bits using Algorithm 1 and then see whether it is correct. This is only required initially till you gain confidence, and may not be necessary later on. Although bitwise operators manipulate the bits, they understand the decimal, octal and hexadecimal numbers and carry out the operation at one go.

Hexadecimal and Octal Representation

If you use decimal numbers, you have to convert them into binary format using Algorithm 1. Let us use hexadecimal or octal numbers for understanding the use of

NOTES

the bitwise operator, since it will be easy for us to convert them into binary form and vice versa. You have to specify the operand in the hex or octal number system, and the result will also be in the same system. Since integers are represented as 16-bit numbers, you can consider integer as 4 hexadecimal numbers each of width 4 bits.

Take a decimal number 6666 and write a program to convert it into binary form and print out its equivalent octal and hexadecimal numbers. You have to use Algorithm 1 for conversion from decimal to binary. Hex and octal numbers can be directly printed. The program is as follows:

```
/*Example 2.3*/
/* To convert decimal to binary, octal and hexadecimal*/
#include <stdio.h>
int b[16], i; /*global variables*/
int main()
{
    int num=6666;
    void convert (int num);
    convert(num);
    printf("\n16 bit binary number equal to %d\n", num);
    for (i=15; i>=0; i--)
    {
       printf("%d", b[i]);
    }
    printf("\nequivalent hexadecimal number\n%x", num);
    printf("\nequivalent octal number\n%o", num);
    return 0;
}
void convert(int a)
{
    for (i=0; i<=15; i++)
    {
       b[i]=a%2;
       a= a/2;
    }
}
```

Result of the program

```
16 bit binary number equal to 6666
0001101000001010
equivalent hexadecimal number
1a0a
equivalent octal number
15012
```

NOTES

What do you notice? The hexadecimal number groups 4 bits each from the left and gives an equivalent hexadecimal of the 4 bits before the number is printed. For example, the first four bits (from LSB) are 1010 whose equivalent hex number is A or a. Similarly, the other three groups of 4 bits are converted to hex. Thus, you have 4 digits of hexadecimal numbers for the 16-bit binary number.

Similarly, the 16-bit binary number is grouped into 3 bits each from the LSB. Therefore, 6 octal numbers will be printed. The last group contains only the MSB. It can only be 0 or 1. The other numbers could be from 0 to 7. Therefore, it is easy to convert octal numbers into binary form. In this case, since MSB is zero, it is omitted and there are only 5 digits in the octal representation.

Now, analyse the program. This program uses new concepts. An integer array of size 16 is declared as a global `array b[16]`. The function `main()` passes num to `convert`. In `convert`, the binary conversion is completed and stored in `array b[0]` to `b[15]`. If you print it straightaway, you will get `b[0]` on the left. To print `b[15]` on the leftmost side, you print `b[15]` first in the `for` loop in main. Although the function `convert` does not pass any value, the array is known because it is a global array. You have indirectly passed the array through global variable.

Now using examples, see the operation of bitwise operators in detail:

One's complement operator

It changes `1` to `0` and `0` to `1`, if `x = 1100, ~x = 0011.`

The usage is as follows:

Let us declare `i, j` as integers;

```
i = 6666;
j = ~ i;
```

Table 2.2 for one's complement of octal numbers is as follows:

Table 2.2 *One's Complement of Octal Numbers*

Octal Number	Binary	Complement in Binary	Complement in Octal
0	000	111	7
1	001	110	6
2	010	101	5
3	011	100	4
4	100	011	3
5	101	010	2
6	110	001	1
7	111	000	0

Therefore, you can write the one's complements from this table.

```
j = ~ i
i = 015012
j = 762765
```

This can be used for encryption of information for security purposes.

NOTES

Right Shift Operator

```
unsigned int i , j ;
    j = i >> 2 ;
```

Here, the bits in *i* will be shifted 2 places to the right and stored in `j`.

If you write `i >> 6`, the bits will be shifted right by 6 places. What happens to the leftmost bits? They will all be filled with zeros.

Now, `i = 6666` in `hex` will be `1A0A`

Let, `j = i >> 4 ;`

The bits will be shifted by 4 places.

Therefore, `j = 01A0`

`i = 015012 in octal`

`if` `j = i >> 3 then`

`j = 001501;`

Left Shift Operator

```
j = i << p;
```

This expression will shift `i` by `p` bits and will store it in `j`.

What happens to the rightmost bits shifted? Zeros will be inserted whenever a bit is shifted.

```
i = 015012  / * octal number * /
j = i << 3;
```

Their result will be `150120.`

```
i = 1A0A;  / * hexadecimal number * /
j = i << 4 ;
```

The shifted number will be,

```
A0A0
```

Shifting again by 4 bits will give `0A00`.

Check by shifting the octal number 150120 to the left by 3 bits. The result will be 501200. Find out the reason yourself.

AND Operator

It compares two bits, and if both are 1, the output is 1, otherwise zero. This can be used to compare the sets of bits. Assume that you want to check only the 16th bit in the 16-bit word of a number, you can carry out the task using AND operator on the number and word with the 16th bit 1 and all other 15 bits 0. When you compare using AND, the 15 bits will be 0 and 16th bit will be a 0 or 1, depending on the number.

Remember to use a single `&`. You know `&&` stands for logical AND.

This will operate on 2 operands.

Let us have, `a = 015012 octal`

`b = 177777`

`c = a & b` will provide an output of `c = 015012` octal. Verify by converting into bits.

`A = 1A0A hexadecimal`

`b = 0000`

`c = a & b` will produce `c = 0000` hexadecimal because `b` is all zeros.

OR Operator

This is also a binary operator. The output of `OR` operator will be `0`, if both the inputs are `0`, and `1` otherwise.

Let, `a = 015012 octal`

`b = 000000`

`c = a | b` will produce `c = 015012 octal`

`Let, a = 015012 octal`

`b = 177777 octal`

`c = a | b` will produce `c = 177777 octal.`

This is because `b` is all ones and hence `a | b` will automatically produce all `1s`, even without looking at `a`.

When `b` is all zeros, the output will be `1` wherever `a` is `1`. Therefore, the output will be same as `a`.

Exclusive OR Operator

When one of the 2 operands is 1, we get the output of exclusive OR as 1; otherwise, the output will be 0.

```
      a    = 1A0A hex
           = 0001101000001010
Let,  b    = 1111111111111111  =    FFFF hex
  a ^ b    = 1110010111110101  =    E5F5 hex
Let,  c    = 0000000000000000  =    0000 hex
  a ^ c    = 0001101000001010  =    1A0A hex
  b ^ c    = FFFF hex
```

Encrypting Selected Bits

Write a program that will convert selected bits of a number into 1 and leave the rest unchanged.

Algorithm 2

```
Step 1:   Get the number num
Step 2:   Get the bitfrom & bit_to, which will be changed
```

NOTES

```
to 1 .
Note:    LSB = 1 and MSB = 16.
```

Step 3: `Convert the number with MSB = b [15], LSB = b[0]`

Step 4: `Change the bits to 1 at the chosen places and leave it as it is, at other places.`

The program is given below:

```
/*Example 2.4 */
/* To convert selected bits of a given number
to 1 and leave the rest as they are*/
#include <stdio.h>
int b[16], c[16], i; /*global variables*/
int main()
{
    int num, bitfrom, bit_to, temp;
    void convert (int num);
    void build(int bitfrom, int bit_to);
    printf("Enter the number to be manipulated\n");
    scanf("%d", &num);
    printf("enter bit from and then bit_to for change to
1\n");
    printf("MSB=16th bit, LSB 1st bit\n");
    scanf("%d%d", &bitfrom, &bit_to);
    /*Still some will give wrongly as from =4, to=8; This
error can be corrected as follows:*/
    if (bit_to > bitfrom) /*exchange*/
       {temp=bit_to;
       bit_to=bitfrom;
       bitfrom=temp;}
    convert(num);
    printf("\n binary number equal to number given by
you\n");
    for (i=15; i>=0; i--)
    {
       printf("%d", b[i]);
    }
    build (bitfrom, bit_to);
    printf("\n binary number with changed bits\n");
    for (i=15; i>=0; i--)
    {
       printf("%d", b[i]);
    }
}
void convert(int a)
{
    for (i=0; i<=15; i++)
    {
       b[i]=a%2;
       a= a/2;
    }
```

NOTES

```
}
void build(int bitfrom, int bit_to)
{
    for (i=0; i<=15; i++)
    {
       if (i >= (bit_to-1) && i <= (bitfrom-1))
          b[i]=1;
    }
    return 0;
}
```

Result of the program

```
Enter the number to be manipulated
123
enter bit from and then bit_to for change to 1
MSB=16th bit, LSB 1st bit
3 12
binary number equal to number given by you
0000000001111011
binary number with changed bits
0000111111111111
```

You can see how it works using a simple example.

Let, `b = 1001`

Assume that you want to convert bit 2 into bit 1 as 1.

Therefore, from bit `b[bit_to-1]` to `b[bitfrom -1]` as 1.

`b[0]=1, b[1]=1`, rest unchanged.

The changed number = `1011`

Note here that you called `LSB` as the 1st bit and `MSB` as the `16th bit` and given the `bitfrom` and `bit_to`, they will be 1 more than the actual element number in the `array b`. That is the reason, 1 has been subtracted in the program. The program takes care of operator errors. If you study the result, you will notice that `bitfrom` and `bit_to` have been interchanged, but the program was executed without interruption, since it was designed to take care of such mistakes.

Bitwise Assignment Operators

These operators combine bitwise operators and assignment operators. The assignment operator is always to the right of the bitwise operator.

You can see their usage with examples, assuming 6-bit numbers for simplicity.

Let `x` be `010110`, i.e., `x = 026 octal`

Usage of OR AND = Operator

`x | = 014` is equivalent to,

`x = x | 014`

014 = 001100

The OR of x AND 014 = 011110

Therefore, x|=014 is equal to 036 octal.

NOTES

Let us see the usage of other bitwise assignment operators as well.

Usage of & and = Operator

Let, x = 066 = 110 110

x & = 044 means x = x & 044, 044 = 100100

Therefore, the AND of x and 044 = 100100 = 044

Let, x = 055 = 101 101

x & = 020 means,

x = x & 020

020 = 010000

x & 020 = 000000 = 0

^= Operator

x ^ = a means,

x = x^ a

Let x be 11001

a be 01110

x^ a = 10111

Therefore, x = 10111

>>= Operator

x >>= a means,

x = x>> a

or x >> = 2 means,

x = x >> 2

if x = 10110, it will be shifted right twice

Therefore, x = 00101.

2.2.8 Special Operator

The sizeof operator is basically worked as C function and it also yields the size in bytes of the operand, which can be an expression or the parenthesized name of a type. A sizeof expression has the form. The type of the result is the unsigned integral type size_t defined in the header file stddef.h. The sizeof operator applied to a type name yields the amount of memory that would be used by an object of that type, including any internal or trailing padding. The sizeof operator may not be applied to a bit field, a function type , an undefined structure or class and

an incomplete type, such as `void.` The `sizeof` operator applied to an expression yields the same result as if it had been applied to only the name of the type of the expression. At compiling time, the compiler analyzes the expression to determine its type, but does not evaluate it. None of the usual type conversions that occur in the type analysis of the expression are directly attributable to the `sizeof` operator. However, if the operand contains operators that perform conversions, the compiler does take these conversions into consideration in determining the type. The following code causes the usual arithmetic conversions to be performed and it is also assumed that a `short` uses 2 bytes of storage and an `int` uses 4 bytes:

```
short x; ... sizeof (x)         /* the value of sizeof
operator is 2 */
short x; ... sizeof (x + 1)     /* value is 4, result of
addition is type int */
```

The result of the expression `x + 1` has type `int` and is equivalent to `sizeof(int)`. The value is also `4` if `x` has type `char, short` or `int` or any enumeration type. Types cannot be defined in a `sizeof` expression. In the following example, the compiler is able to evaluate the size at compile time. The operand of `sizeof,` an expression, is not evaluated. The value of `b` is the integer constant 5, from initialization to the end of program run time:

```
#include <stdio.h>

int main(void){
  int b = 5;
  sizeof(b++);
  return 0;
}
```

Except in preprocessor directives, you can use a sizeof expression wherever an integral constant is required. One of the most common uses for the `sizeof` operator is to determine the size of objects that are referred to during storage allocation, input, and output functions. Another use of `sizeof` is in porting code across platforms. You should use the `sizeof` operator to determine the size that a data type represents in the following way:

```
sizeof(int);
```

The operand of the `sizeof` operator can be a vector type or the result of dereferencing a pointer to vector type and in this case the return value of `sizeof` is always `16`.

```
vector bool int v1;
vector bool int *pv1 = &v1;
sizeof(v1); // vector type: 16.
sizeof(&v1); // address of vector: 4.
sizeof(*pv1); // dereferenced pointer to vector: 16.
```

```
sizeof(pv1); // pointer to vector: 4.
sizeof(vector bool int); // vector type: 16.
```

NOTES

The result of a `sizeof` expression depends on the type it is applied to, for example with an array. With an array, the `sizeof` operator's result is the total number of bytes in the array. For example, in an array with 10 elements, the size is equal to 10 times the size of a single element. The compiler does not convert the array to a pointer before evaluating the expression. The `sizeof` operator to get the size of the various datatypes, for example, if you declare a variable `int x;` you can get the size of `x` with `sizeof(x);`. It is also worked with structures or arrays. For example, if you have a variable of a `struct` type with the name `a_struct`, you can use `sizeof(a_struct)` format to find out how much memory it is taking up, for example, `sizeof(int);` and you can get a better idea by the following example:

```
FILE *fp;
fp=fopen("c:\\test.bin", "wb");
char x[10]="ABCDEFGHIJ";
fwrite(x, sizeof(x[0]), sizeof(x)/sizeof(x[0]), fp);
```

The `sizeof` is not a function but a compile time operator. The compiler does not the value of declared variable `i` but only its type. It evaluates this at compile time and works as a constant it.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
main()
{
   size_t sz;
   sz=sizeof(sz);
   printf("Size of sizeof is %lu\n", (unsigned long)sz);
   exit(EXIT_SUCCESS);
}
```

The result of the above code returns the size of the `sizeof operator` in terms of `unsigned long`.

2.3 EXPRESSION IN C

An **expression** is a combination of variables, constants and operators written according to the syntax of C language. In C, every expression evaluates to a value, i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of C expressions are shown in Table 2.3.

NOTES

Table 2.3 Representation of Arithmetic Expressions in C

Algebraic Expression	C Expression
$a \times b - c$	`a * b - c`
$(m + n)(x + y)$	`(m + n) * (x + y)`
(ab / c)	`a * b / c`
$3x^2 + 2x + 1$	`3*x*x+2*x+1`
$(x / y) + c$	`x / y + c`

Evaluation of Expressions

Expressions in C are evaluated using an assignment statement of the following form: `variable = expression;`

`variable` is any valid C variable name. When the statement is encountered then the `expression` is evaluated to replace the previous value of the variable on the left hand side. All variables used in the expression must be assigned values so that no error occurs at the time of evaluation. The following are some examples of evaluation statements:

```
x = a * b - c;
y = b / c * a;
z = a - b / c + d;
```

The following program illustrates the effect of presence of parenthesis in expressions.

```
/*Example 2.5
void main ()
{
float a, b, c, x, y, z;
a = 9;
b = 12;
c = 3;
x = a - b / 3 + c * 2 - 1;
y = a - b / (3 + c) * (2 - 1);
z = a - ( b / (3 + c) * 2) - 1;
printf ("x = %fn",x);
printf ("y = %fn",y);
printf ("z = %fn",z);
}
```

Result of the program

```
x = 10.00
y = 7.00
z = 4.00
```

NOTES

An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators. The two distinct priority levels of arithmetic operators in C are as follows:

```
High priority * / %
Low priority + -
```

Rules for Evaluation of Expression

The following are the rules for evaluation of expression in C language:

- First parenthesized sub expressions are evaluated from left to right.
- If parenthesis is nested, the evaluation begins with the innermost sub expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
- The associability rule is applied when two or more operators of the same precedence level appear in the sub expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parenthesis is used, the expressions within the parenthesis assume the highest priority.

Types of Expressions

The three types of expressions in C language are as follows:

1. **Arithmetic:** It evaluates a number, for example `a=12;`.
2. **String:** It evaluates character or text string, for example `'text'` or `'12345'`.
3. **Logical:** It retains 'True' or 'False' value.

Conditional Expression in C

The conditional expression holds two values based on the generated condition. The following syntax is used to write a conditional expression:

```
(condition) ? val1 : val2;
```

In C, the standard expression can be declared as follows:

```
Status_of_person = (age >= 18) ? "adult" : "minor";
```

In this conditional expression either of the two values can be returned which is based on the value of age. If age is greater than 18 the assigned value to the Status_of_person will be 'adult', and if age is less than 18 then the assigned value will be 'minor'.

Check Your Progress

1. List the basic arithmetic operators.
2. Differentiate between increment and decrement operator.
3. State about the binary operator.
4. Define the term expression.

NOTES

2.4 OPERATOR PRECEDENCE AND ASSOCIATIVITY

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate value to the proper type so that the expression can be evaluated without losing any significance. Each operator in C has a precedence associated with it. The **precedence** is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to any one of these levels. The operators of higher precedence are evaluated first. The operators of same precedence are evaluated from right to left or from left to right depending on the level. This is known as associativity property of an operator. Operator precedence describes the order in which C reads expressions. For example, the expression `a=4+b*2` contains two operations, an addition and a multiplication. Questions are raised like does the C compiler evaluate `4+b` first, then multiply the result by `2` or does it evaluate `b*2` first, then add `4` to the result? The given operator precedence chart will help you to get the correct answers. Operators higher in the chart have a higher precedence, meaning that the C compiler evaluates them first. Operators on the same line in the chart have the same precedence and the associativity column on the right gives their evaluation order. Two operator characteristics determine how operands group with operators: precedence and associativity. Precedence is the priority for grouping different types of operators with their operands. **Associativity** is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first. The grouping of operands can be forced by using parentheses. For example, in the following statements, the value of `5` is assigned to both `a` and `b` because of the right-to-left associativity of the = operator. The value of `c` is assigned to `b` first and then the value of `b` is assigned to `a`.

```
b = 9;
c = 5;
a = b = c;
```

The above statements only assign value to the expression. To evaluate these expressions we have to use arithmetic operators. The * and / operations are performed before + because of precedence. The variable b is multiplied by c before it is divided by d because of associativity rule. The order of precedence and associativity is given in the given table that C uses for evaluating declared expressions.

```
a + b * c / d
```

You can explicitly force the grouping of operands with operators by using parentheses to specify the order of precedence.

NOTES

Table 2.4 *Operators Precedence and Associativity*

Operator Name	Associativity	Operators
Primary	left to right	`()  [ ]  . ->`
Unary	right to left	`++  --  +  -` `!  ~  &  *  (type_name)  sizeof`
Multiplicative	left to right	`*  /  %`
Additive	left to right	`+  -`
Bitwise Shift	left to right	`<<  >>`
Relational	left to right	`<  >  <=  >=`
Equality	left to right	`==  !=`
Bitwise AND	left to right	`&`
Bitwise Exclusive OR	left to right	`^`
Bitwise Inclusive OR	left to right	`|`
Logical AND	left to right	`&&`
Logical OR	left to right	`||`
Conditional	right to left	`? :`
Assignment	right to left	`=  +=  -=  *=  /=  <<=  >>=` `%=  &=  ^=  |=`
Comma	left to right	`,`

2.5 INPUT AND OUTPUT IN C

The input and output functions of a computer facilitate communication with the external world. They also facilitate interaction between the computer and the user. The user has to input data in order to process it in the computer and get the result as output. The peripheral device for input is the **keyboard**. Keying in of the input data into the computer at run-time is achieved easily by library functions, which are supplied along with the 'C' compiler and have been standardized. The functions, which enable keying in of the input data in the keyboard, are given as follows:

- `scanf()`
- `getchar()`
- `getch()`
- `getche()`
- `gets()`

The computer communicates its output, i.e., the result of computation, either through the console video monitor, printer, disk drive or input/output ports. Since we are learning 'C' through the PC, we will get the output through the video

NOTES

monitor. Just as there are library functions for input, there are standard library functions for output as well. They are given as follows:

- printf()
- putchar()
- putch()
- puts()

Giving input and getting output are achieved by using the standard library functions. Note that all the function names are followed by parentheses. The parentheses are meant for passing arguments or getting arguments. The arguments may be absent in certain functions as in the case of main(). However, function names must be followed by parentheses to indicate to the compiler that they are functions. Arguments can be either variables or constants.

2.5.1 Use of printf()

Initially, formatted input/output statements will be discussed. Here, one must categorically specify the data type of variables to be read or written and their formats. The printf() and scanf() are formatted input and output statements. The printf() statement can be programmed to give the output in the desired manner. Example 2.6 is given below to illustrate the use of the printf() function in printing the required values.

```
/*Example 2.6*/
/* To demonstrate the print function*/
#include <stdio.h>
main()
{
    printf("welcome to more serious programming\n");
}
```

Here the first statement after the comment line directs the compiler to include the standard input/output header file. Note that in the printf() statement whatever is given within quotes except those immediately following '\' and '%' symbols will be printed as it is. Those with the '\' symbol like '\n', '\t' are escape sequences. Those of the '%' symbol such as '%d', '%f' are known as conversion characters. They have a specific meaning to the printf() function.

Result of Program

```
welcome to more serious programming
```

Then the cursor will go to the next line. The cursor is made to go to the next line because of the new line character \n. The escape sequences carry out the functions assigned when their turn for execution is reached in the printf() statement.

Now correct the statement as given below:

printf("\tWelcome"); and execute the program.

Welcome will appear from the first tab.

Execute the program again. You will find that the message is printed from the next available tab position on the same line.

2.5.2 Single Character Input/Output

NOTES

Characters can be scanned through the `scanf()` function and printed through the `printf()` function as given in Example 2.7.

```
/*Example 2.7*/
/*input and output of character through scanf() and printf()*/
#include<stdio.h>
int main()
{
    char alpha;
    printf("Enter a character\n");
    scanf("%c", &alpha);
    printf("\n The character typed by you is:- %c\n", alpha);
}
```

The program is simple to understand.

A variable `alpha` is declared of type `char`. The message directs the user to enter a character. The `scanf()` function uses the conversion character `%c` since we have to scan a character type variable. The (ampersand) `&alpha` indicates the address where the scanned character is to be stored.

There are two points to be noted here. We have to specify the format as `%c` and after entering the character, we have to hit the `Return` key. The interaction with the computer while executing the program was captured and given below:

```
Enter a character
S
The character typed by you is:- S
```

2.5.3 Strings—gets() and puts()

A **string** is an array of characters. The functions `gets()` and `puts()` are appropriate when strings are to be received from the screen or sent to the screen without errors.

Standard Library for Strings

There are a number of library functions for string manipulation as given below:

`strlen(CS)`—returns the length of string `CS`.

`char * strcpy (s, ct)`—copy string `ct` to string `s`, including NULL and return `s`.

`char * strcat (s, ct)`—concatenate string `ct` to end of string `s`; return `s`.

`int strcmp (cs, ct)`—compare string `cs` to string `ct`; return < 0 if `cs < ct`,

`0 if cs == ct or > 0 if cs > ct`

`char * strchr (cs, c)`—returns the pointer to the first occurrence of `c` in `cs` or NULL if not present.

NOTES

There are some more string functions.

If these are to be used, `<string.h>` should be included before the main function.

Use of gets() and puts()

One can use `scanf()` to receive strings from the screen. The program using `scanf()` for reading a name is as follows:

```
char name [25];
scanf (" %s ", name);
```

Strings can be declared as an array of characters as shown above. In the `scanf()` function, when we get the array of characters as a string, it is enough to indicate the name of the array without a subscript. When we get a string, there is no need for writing `'&'` in the `scanf()` function. We can indicate the name of the string variable itself.

Strings may contain blanks in between. If you use a `scanf()` function to get the string with a space in between such as 'Rama Krishnan', Krishnan will not be taken note of since space is an indicator of the end of entry of the input. But `gets()` will take all that is entered till the enter key is pressed. Therefore, after entering the full name, the `enter()` key can be pressed. Thus, using `gets()` is a better way for strings. We can get a string in a much simpler way using `gets()`. The syntax for `gets` is, `gets(name);`

Similarly `puts()` can be used for printing a variable or a constant as given below:

```
puts (name);
puts ("Enter the word");
```

However, there is a limitation. `printf()` can be used to print more than one variable and `scanf()` to get more than one variable at a time in a single statement. However, `puts()` can output only one variable and `gets()` can input only one variable in one statement. In order to input or output more than one variable, separate statements have to be written for each variable. As you know that `gets()` and `puts()` are unformatted I/O functions, there are no format specifications associated with them.

We will take another interesting example. If a word is a palindrome, we will get the same word when we read the characters from the right to the left as well.

Examples are : nun

malayalam

These words when read from either side give the same name. We will write a program to check whether a word is a palindrome or not.

This program uses a library function called `strlen()`. The function `strlen(str)` returns the size or length of the given string. Now let us look at the program.

```
/*Example 2.8*/
/* To check whether a string is palindrome*/
#include <stdio.h>
```

NOTES

```
#include <string.h>
#define FALSE 0
int main()
{
    int flag = 1;
    int right, left, n;
    char w[50]; /* maximum width of string 50*/
    puts("Enter string to be checked for palindrome");
    gets(w);
    n=strlen(w)-1;
    for ((left = 0, right = n); left <= n/2; ++left,
--right)
    {
        if (w[left]!=w[right])
        {
            Flag = FALSE;
            break;
        }
    }
    if (flag)
    {
        puts(w);
        puts("is a palindrome");
    }
    else
        printf("%s is NOT a palindrome");
}
```

Output of the program

```
Enter string to be checked for palindrome
palap
palap
is a palindrome
```

If `strlen()` or `gets()` or `puts()` are used in a program, we have to include `<string.h>` before the `main()`.

Step 1

Now let us analyse the functioning of the above example.

We are defining a symbolic constant FALSE as 0.

We initialize `flag` as 1. We define a string `w` as an array of 50 characters.

`gets(w);` returns the word typed and stores it from location `&w [0]`

Let us assume that we typed 'nun' and analyse what happens in the program.

`strlen (w)` will return the length of the word typed. In this case `strlen (w) = 3`.

We subtract this by 1 to get the subscript of the rightmost character. The subscript of the leftmost character is obviously 0.

We are initializing the `for` loop with the following:

```
left = 0      right = n = 2
flag = 1
```

We check whether `left < = n/2` and

Since it is so, we check whether `w [0] ! = w [2]`.

The condition is false since `w[0] = w[2] = 'n'`.

Therefore, the groups of statements following `if` are skipped : `flag` remains 1.

Step 2

Now left is incremented to 1 and right is decremented to 1.

Again `w[1] ! = w[1]` is false, `flag` remains 1.

Therefore, control returns to the `for` statement.

Now, `left = 2    right = 0`

Since left is greater than `n/2`, control comes out of the `for` loop. Now the statement `if (flag)` will be executed.

It will check whether `flag` is true. In this case, `flag` is still true.

Therefore, the computer prints that the word is palindrome. If the word is not a palindrome, then what happens?

Let us now assume that we typed 'book' and see what happens in the program.

To start with, `left = 0    right = 3`

`w [left] ! = w[right],`

Therefore, the statements within the `{ }` will be executed, `flag` will be set to false and then the `break` statement will be executed.

The statement `break` causes immediate exit from the loop. Now `flag` is false. Therefore, the `else` statement is executed to say that the word is NOT a palindrome.

Check Your Progress

5. What is operator precedence?
6. Define the term associativity.
7. What is the significance of `printf ()` statement?
8. When are `gets()` and `puts()` function used?

2.6 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The basic arithmetic operators include addition, subtraction, multiplication and modulus operators.

NOTES

2. The ++ (increment) operator adds 1 to the value of a scalar operand or if the operand is a pointer then increments the operand by the size of the object to which it points. The operand receives the result of the increment operation. The — (decrement) operator subtracts 1 from the value of a scalar operand or if the operand is a pointer decreases the operand by the size of the object to which it points. The operand receives the result of the decrement operation.
3. Binary operators are usually assumed to be operators which operate on binary digits, i.e., bits. However, this is not the convention in C. Binary operators refer to those operators, which operate on 2 operands, i.e., a + b, a % b, etc. In the same way, unary operators are those, which operate on single operand, such as a++, a— , etc.
4. An expression is a combination of variables, constants and operators written according to the syntax of C language.
5. Precedence refers to the evaluation order of operators. In an expression, there may be the same type of operators at a number of places, which may lead to ambiguity as to which one to evaluate first. To avoid ambiguity, there are defined precedence rules for operators in C.
6. Associativity is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence.
7. The `printf()` statement can be programmed to give the output in the desired manner.
8. The functions `gets()` and `puts()` are appropriate when strings are to be received from the screen or sent to the screen without errors.

2.7 SUMMARY

- The basic arithmetic operators include addition, subtraction, multiplication and modulus operators.
- Two variables of the same type may have a relationship between them. They can be equal or one can be greater than the other or less than the other. This can checked this by using relational operators. While checking, the outcome may be true or false.
- Binary operators are usually assumed to be operators which operate on binary digits, i.e., bits. However, this is not the convention in C. Binary operators refer to those operators, which operate on 2 operands, i.e., a + b, a % b, etc. In the same way, unary operators are those, which operate on single operand, such as a++, a— , etc.
- An expression is a combination of variables, constants and operators written according to the syntax of C language.

NOTES

- An expression is a combination of variables, constants and operators written according to the syntax of C language.
- Precedence refers to the evaluation order of operators. In an expression, there may be the same type of operators at a number of places, which may lead to ambiguity as to which one to evaluate first. To avoid ambiguity, there are defined precedence rules for operators in C.
- Associativity is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence.
- The input and output functions of a computer facilitate communication with the external world. They also facilitate interaction between the computer and the user.
- The user has to input data in order to process it in the computer and get the result as output. The peripheral device for input is the keyboard.
- The computer communicates its output, i.e., the result of computation, either through the console video monitor, printer, disk drive or input/output ports. Since we are learning 'C' through the PC, we will get the output through the video monitor.
- The `printf()` and `scanf()` are formatted input and output statements. The `printf()` statement can be programmed to give the output in the desired manner.
- A string is an array of characters. The functions `gets()` and `puts()` are appropriate when strings are to be received from the screen or sent to the screen without errors.
- In the `scanf()` function, when we get the array of characters as a string, it is enough to indicate the name of the array without a subscript. When we get a string, there is no need for writing '&' in the `scanf()` function. We can indicate the name of the string variable itself.
- Strings may contain blanks in between. If you use a `scanf()` function to get the string with a space in between such as 'Rama Krishnan', Krishnan will not be taken note of since space is an indicator of the end of entry of the input. But `gets()` will take all that is entered till the enter key is pressed.

2.8 KEY TERMS

- **Assignment operators:** These are operators that assign values to variables.
- **Expression:** It is any legal combination of symbols that represents a value.
- **Associativity:** Associativity is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence.
- **`printf()`:** It is a function used to print the character, string, float, integer, octal and hexadecimal values onto the output screen.
- **`Scanf()`:** It is a function used to take input from the user or console.

2.9 SELF ASSESSMENT QUESTIONS AND EXERCISES

NOTES

Short-Answer Questions

1. Differentiate between relational and logical operators.
2. Define the term bitwise assignment operators.
3. State about the special operators.
4. What are the different types of expressions?
5. State the different input and output functions.
6. What is the significance of `printf()` and `scanf()` function?

Long-Answer Questions

1. Briefly discuss the various types of C operators giving appropriate examples.
2. Explain bitwise and special operators giving example programs.
3. Discuss about the expression in C with the help of example programs.
4. Describe the operator precedence and associativity giving appropriate examples and programs.
5. Briefly explain the input and output in C with the help of examples.
6. Write a program to demonstrate the use of `printf ()` and `scanf ()` function.
7. Discuss `gets()` and `puts()` in detail with the help of example programs.

2.10 FURTHER READING

Gottfried, Byron S. 1996. *Programming with C*, Schaum's Outline Series. New York: McGraw-Hill.

Jeyapoovan T. 2006. *Computer Programming - Theory & Practice*. New Delhi: Vikas Publishing House Pvt. Ltd.

Khurana, Rohit. 2005. *Object Oriented Programming*. New Delhi: Vikas Publishing House Pvt. Ltd.

Kanetkar, Yashwant. 2003. *Let Us C*. New Delhi: BPB Publication.

Saxena, Sanjay. 2003. *A First Course in Computers*. New Delhi: Vikas Publishing House Pvt. Ltd.

Subburaj, R. 2000. *Programming in C*. New Delhi: Vikas Publishing House Pvt. Ltd.

Ghosh, Smarajit. 2009. *All of C*. New Delhi: PHI Learning Pvt Ltd.

Bronson, Gary J. 2000. *A First Book of ANSI C*, 3rd edition. California: Thomson, Brooks Cole.

UNIT 3 DECISION MAKING, BRANCHING, ARRAYS AND POINTERS

Structure

3.0 Introduction
3.1 Objectives
3.2 Conditional Statements
 3.2.1 `if` Statement
 3.2.2 `if...else` Statement
 3.2.3 Nesting of the `if...else` Statements
 3.2.4 Logical Operators and Branching
 3.2.5 Conditional Operator and `if...else`
3.3 Switch Statement
3.4 Break, Continue, Goto Statements
3.5 Looping Statements
3.6 Arrays in C
 3.6.1 Declaring Arrays
 3.6.2 One Dimensional Array
 3.6.3 Two Dimensional Array
 3.6.4 Multidimensional Array
3.7 Pointers in C
 3.7.1 Pointer to `void`
 3.7.2 `NULL` Pointer
 3.7.3 Declaration of Pointer Variables
 3.7.4 Accessing the Address of a Variable
3.8 Pointer to Array
 3.8.1 Multi Dimensional Arrays
3.9 Pointer and Function
 3.9.1 Chain of Pointers/Array of Pointers
 3.9.2 Pointers to Functions
3.10 Pointer and Structure
 3.10.1 Passing Structure by Reference
 3.10.2 Array of Pointers
 3.10.3 Pointer to Pointer
3.11 Answers to 'Check Your Progress'
3.12 Summary
3.13 Key Terms
3.14 Self Assessment Questions and Exercises
3.15 Further Reading

3.0 INTRODUCTION

Conditional statement are used for solving complex problems. Depending on the occurrence of a particular situation, `if` and `else` keywords are used for branching to different segments of the program. C is ideal for handling branching because the syntax is clear and unambiguous. If the condition is true, then a single statement or group of statements following if will be executed. If more than one statement is to

be executed, then the statements are grouped within braces. To perform the same operation a number of times, loop or iteration is used.

An array refers to a data structure which holds multiple variables of the same data type using an indexing system to find each variable stored within it. For example, a string is a one dimensional array of characters. An array is also defined as a vector defined as simple data structure which holds a fixed number of equal size data elements of the same type. Multi-dimensional arrays operate on the same principles as single dimensional arrays. Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as call by reference. When a function is called by reference any change made to the reference variable will effect the original variable. Array of pointers is the array that holds memory locations.

In C, a pointer is a variable that points to or references a memory location in which data is stored. Each memory cell in the computer has an address that can be used to access that location using a pointer variable. We can access and change the contents of this memory location by declaring the pointer. The asterisk informs the compiler that you are creating a pointer variable.

In this unit, you will study about the conditional statements, `if` and `if...else` branching statements in your C programs, `switch, break, continue` and `goto` statement, looping statements, `if`, `for`, `while` and `do...while` in a program, arrays in C, one dimensional array, two dimensional array, multi-dimensional array, pointers in C, pointer accessing the address of a variable, pointer and arrays, pointer and function, pointer and structure.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the significance of conditional statements
- Use `if` and `if...else` branching statements in your C programs
- Define `switch, break, continue` and `goto` statement
- Understand the basic of looping statements
- Use `if`, `for`, `while` and `do...while` in a program
- Explain the concept of arrays in C
- Describe the one dimensional array and two dimensional array
- Discuss about the multi-dimensional array
- Understand the basic of pointers in C
- Define the pointer accessing the address of a variable
- Elaborate on the initiating pointers
- Explain the concept of pointer and arrays
- Discuss about the pointer and function
- Define the significance of pointer and structure

3.2 CONDITIONAL STATEMENTS

Real-life application programs do not merely consist of simple multiplication or addition. They call for solving complex problems. Depending on the occurrence of a particular situation, we may follow different paths; the `if` and `else` keywords are quite handy in branching to different segments of the program. 'C' is ideal for handling branching because the syntax is clear and unambiguous. We will now discuss the branching constructs. Relational operators are used in conjunction with branching constructs. Hence, we look at them first.

3.2.1 if Statement

The syntax of the `if` statement is given below.

```
if (condition)
    {statements}
```

If the condition is true, then a single statement or group of statements following the `if` will be executed. If more than one statement is to be executed, then the statements are grouped within braces. If it is a single statement, then curly braces are not required.

If the condition turns out to be false, then the next statement after those belonging to the `if` will be executed. Example 3.1 will make the concept clear.

Input two integers from the keyboard. If they are equal, then the program will print, 'You typed equal numbers'; otherwise, it will print nothing.

```
/*Example 3.1
To demonstrate the use of if*/
#include <stdio.h>
main()
{
unsigned int a,b;
printf ("enter two integers\n");
scanf("%u%u", &a, &b);
if (a==b)
    {
    printf("you typed equal numbers\n");
    }
}
```

Each open curly brace has to have a matching closing curly brace. The first closing curly brace corresponds to the `if` statement and the second one to the main function. Execute the program by first keying in two equal valued unsigned integers.

Result of the program

```
enter two integers
56 56
you typed equal numbers
```

NOTES

After you are satisfied, you can try the program with unequal numbers. You will not get any message.

3.2.2 `if...else` Statement

We did not get any message when the numbers were unequal and this can be avoided by using the else statement.

The usage of `if .. else` is shown below.

```
if (condition true)
{
   statements s1
}
else
{
   statements s2
}
statements s3;
```

The statement `else` is always associated with an `if`.

If the condition is true, then statements `s1` will be executed. After executing them, the program will skip the `else` block and control goes to statement `s3` that follows the `else` block.

If the condition is false, then the statements in the `else` block, i.e., `s2` will be executed followed by statement `s3`.

Statements `s1` will not be executed at all when the condition becomes false. The usage of braces clearly brings out which statements belong to the `if` block and which to the `else` block.

Example 3.2 brings out the usage of `if... else`.

```
/*Example 3.2
To demonstrate the use of if.. else*/
#include <stdio.h>
main()
{
    unsigned int a,b;
    printf ("enter two integers\n");
    scanf("%u%u", &a, &b);
    if (a==b)
    {
       printf("you typed equal numbers\n");
    }
    else
    {
       printf("numbers not equal\n");
    }
}
```

NOTES

The output of the program when unequal numbers were keyed in is as follows.

Result of the program

```
enter two integers
17 13
numbers not equal
```

3.2.3 Nesting of the `if...else` Statements

We witnessed the usage of a single `if` statement in Example 3.1. We saw `if` followed by `else` in Example 3.2. There is no restriction to the number of `if`**,** which can be used in a program. This applies to `else` as well, but `else` can only follow an `if` statement.

We can have the following in a program:

```
{
if (condition1)
    {
       if (condition2)
          {statements-s1}
       else
          if (condition3)
             {statements-s2,}
    }
else
    {statements-s4}
statements-s5
}
```

This is called a nested `if` and `else` statement. As the level of nesting increases, it will be difficult to analyse and logical mistakes will be made more easily.

In the above example, when `condition1` is false, `statements-s4` will be executed. If `condition1` is true and `condition2` is also true, then `statements-s1` will be executed.

If `condition1` is true and `condition2` and `condition3` are false, `statements-s5` will be executed directly.

To execute statements–`s2`

`Condition1` has to be true;

`Condition2` has to be false,

And `condition3` has to be true.

Try to analyse this yourself. There are better methods to solve the above problem, which will be discussed later.

For example, three unequal integers are keyed in and are called `x`, `y` and `z`. Write a program to find the greatest of the three numbers.

NOTES

Before writing a program, we must write the algorithm. We should not straight away get down to programming.

Algorithm 1

Algorithm for finding the largest of 3 integers.

Step 1. Print a message to enter 3 integers.

Step 2. Get three numbers and store them at `&x`, `&y` and `&z`.

Step 3. Check if `x > y`

Step 4. If false, go to step 9

Step 5. If true

Step 6. Check if `x > z`

Step 7. If true, write `x` is the largest; End

Step 8. If false, write `z` is the largest; End

Step 9. Check if `y>z`

Step 10. If true, write `y` is the largest; End

Step 11. If not, write `z` is the largest.

End

Let us now code these steps into a 'C' program, which is shown in Example 3.3.

```
/*Example 3.3
To demonstrate the use of the nested if.. else*/
#include <stdio.h>
main()
{
    int x,y,z;
    printf ("enter three unequal integers\n");
    scanf("%d%d%d", &x, &y, &z);
    if(x>y)
    {
      if(x>z)
       {
          printf("x is largest\n");
       }
      else
       {
          printf("z is largest\n");
       }
    }
    else
    {
       if(y>z)
```

NOTES

```
        {
           printf("y is largest");
        }
        else
        {
           printf("z is largest");
        }
     }
  }
```

Test the correctness of the program by giving a different set of values for `x`, `y` and `z`.

Result of the program

```
enter three unequal integers
908 231 907
x is largest
```

Look at the example. It uses multiple nesting of `if ... else`.

Take care to see that every opening brace has a corresponding closing brace. It is better to indent the braces as shown in the example so that no mistake is committed. Take care to see that `else` matches with the corresponding `if` and each opening brace `{` matches with a corresponding closing brace `}`; if either an opening {or closing} is extra, then an error will result.

3.2.4 Logical Operators and Branching

In the above examples we have been checking one condition at a time. It would be nice if we could check more than one condition at a time. 'C' provides three logical operators for combining more than one condition. These are as follows:

Logical `and` represented as `&&`

Logical `or` represented as ||

`Negation` or `not` represented as `!` (exclamation).

Let us see some examples of usage of the logical operators. In Example 3.3 we concluded that,

`if x > y` and `if x > z`, then `x` is the largest.

We will represent the same as,

```
if ((x > y) && (x > y))
printf ("x is the largest");
```

You will see that the program has become much more elegant.

The syntax for `&&` is,

```
if ((condition1) && (condition2))
   {
   statements-s1
   }
```

NOTES

Statements–s1 will be executed only if both the conditions are true.

The syntax for 'or' is as follows:

```
if ((condition 1 ) ||(condition 2))
    {
    statements-s2
    }
```

In this case, even if one of the conditions is true, the `statements-s2` will be executed. At least one condition has to be true for the execution of `s2`. However, if both are false, `s2` will not be executed.

The ***NOT*** **operator** with symbol **!** can be used along with any other relational or logical operator or as a stand-alone operator. It simply negates the operator following it. The syntax of `'!'` is as follows:

```
if ! (condition) statement s3;
```

`s3` will be executed only when the condition is not true or the condition is false.

Let us rewrite Algorithm 1 by using the logical operators. The revised Algorithm 2 is given below:

Algorithm 2

Step 1: `If` (`x` > `y`) and (`x` > `z`), `x` is the largest.

Step 2: `Else if` (`x`<`y`) and (`y`>`z`), `y` is the largest.

Step 3: `Else` print z is the greatest.

The complete program is given in Example 3.4.

```
/*Example 3.4
To demonstrate the use of logical operators*/
#include <stdio.h>
main()
{
    int x,y,z;
    printf ("enter three unequal integers\n");
    scanf("%d%d%d", &x, &y, &z);
    if ((x>y) && (x>z))
       printf("x is largest\n");
    else
    {
       if((x<y) && (y>z))
          printf("y is largest\n");
       else
          printf("z is largest\n");
    }
}
```

Result of the program

```
enter three unequal integers
12 23 78
z is greatest
```

Let us now write a program to convert a lower case letter typed into an upper case letter. For this purpose you may have to refer to the ASCII table in Annexure 2.

It is obvious that if we subtract 32 from the ASCII value of a lower case alphabet we will get the ASCII value of the corresponding upper case letter. Let us write an algorithm for the conversion of lower case to an upper case letter. It is given in Algorithm 3.

Algorithm 3

Step 1: Send a message for getting a character

Step 2: Get a character

Step 3: Check whether the character typed is `>=a` and `<=z`

(This is essential since we can only convert a lower case alphabet into upper case.)

Step 4: If so, subtract `32` from the value of the character; if not, go to step 6

Step 5: Output the character with the revised ASCII value; END

Step 6: Print "an invalid character" END

The algorithm is implemented in Example 3.5.

```
/*Example 3.5
Conversion of lower case letter to upper case*/
#include <stdio.h>
main()
{
    char alpha;
    printf ("enter lower case alphabet\n");
    alpha=getchar();
    if ((alpha >='a') && (alpha<='z'))
    {
       alpha= (alpha-32);
       putchar (alpha);
    }
    else
       printf("invalid entry; retry");
}
```

Now you can test the program by giving both the valid and invalid inputs; valid inputs are the lower case letters and invalid inputs are all other characters.

Result of the program

The result for the invalid input is given below:

```
enter lower case alphabet
8
invalid entry; retry
```

The result when tried with a valid input is given below:

```
enter lower case alphabet
n
N
```

The programs should be executed, i.e., tested with both the valid and invalid inputs.

3.2.5 Conditional Operator and `if...else`

The syntax for the conditional operator is given below:

```
(Condition)? statement1: statement2;
```

What does it mean? If the condition is true, execute `statement1`; else, execute `statement2`. Here nesting is not possible. The `if...else` statement is more readable than the `conditional(?)` operator. However, the conditional operator is quite handy in simple situations as given below:

```
(a > b) ?  print a greater
           :  print b greater;
```

Thus, the operator has to be used in simple situations. If nothing is written in the position corresponding `to else,` then it means nothing is to be done when the condition is false.

Example 3.2 is rewritten using the `?` operator in Example 3.6.

```
/*Example 3.6
To demonstrate the use of the ? operator*/
#include <stdio.h>
main()
{
    unsigned int a,b;
    printf ("enter two integers\n");
    scanf("%u%u", &a, &b);
    (a==b)?printf("you typed equal numbers\n"):
    printf("numbers not equal\n");
}
```

Result of the program

```
enter two integers
123 456
numbers not equal
```

3.3 SWITCH STATEMENT

Switch statements allow clear and easy implementation of multiway decision-making. Assuming that a number is received from the keyboard and that depending on the value, we want to carry out some operations, the `switch` statement can be used effectively in this situation. In simpler situations `if...else` could be used, and in complex situations, `switch` can be used. For example, if we get numbers starting from 1 to 4 and print their values in words, we can use the `if...else` statement as given in Example 3.7:

```
/*Example 3.7
Converts the digits 1-4 in words using if*/
#include <stdio.h>
#include <conio.h>
main()
{
    int a;
    char ch ='c';
    while (ch=='c')
       {
       printf("\nEnter a digit 1 to 4\n");
       scanf("%d",&a);
       if(a== 1)
          printf("One\n");
       else
          if (a== 2)
             printf("Two\n");
          else
             if(a== 3)
                printf("Three\n");
             else
                if(a==4)
                      printf("Four\n");
                else
       printf("Illegal character\n");
       printf("enter 'c' if you want to continue\n");
       printf("or any other character to end\n");
                           ch=getche();
                           if (ch!='c')
                                 printf("End of
Session");
          }
}
```

Result of the program

```
Enter a digit 1 to 4
3
Three
enter 'c' if you want to continue
or any other character to end
c
Enter a digit 1 to 4
1
One
enter 'c' if you want to continue
or any other character to end
c
Enter a digit 1 to 4
2
Two
enter 'c' if you want to continue
or any other character to end
nEnd of Session
```

We have struggled hard to do this exercise. Assuming that we want to get a one digit number up to 9 and print its value in words, it would become a more complex task. The `switch` statement comes in handy in such situations. The syntax of the `switch` statement is as follows:

```
switch (expression)
{
   case  constant or expression : statements
   case  constant or expression : statements
   ..
default : statements }
```

When the **`switch`** keyword is encountered, the associated expression is evaluated. The program now looks for the **`case`**, which matches with the value of the expression. Execution then starts from the statement corresponding to the **`case`** which matches. Each **`case`** has to be accompanied by integer expressions, which must be unique, as otherwise the program will not know where to start. For example, if the first two cases are as given below:

```
case 10 : s1;
case 10 : s2;
```

In this case the program would not know whether to execute `s1` or `s2` when the expression of `switch` evaluates to 10. Therefore, the constant expressions following the `case` keyword should all be unique. There may be occasions when none of the constant expressions matches the `switch` expression in which case the `default` statements will be executed. Thus `switch` allows branching of the program execution to appropriate place. A program to print the values of the digits in words is given below:

```
/*Example 3.8
converts the digits 0-9 in words*/
#include <stdio.h>
```

```
#include <conio.h>
main()
{
    int a;
    char ch ='c';
    while (ch=='c')
       {
          printf("\nEnter a digit 0 t0 9\n");
          scanf("%d",&a);
          switch(a)
          {
             case 0:printf("Zero\n");
                break;
             case 1:printf("One\n");
                break;
             case 2:printf("Two\n");
                break;
             case 3:printf("Three\n");
                break;
             case 4:printf("Four\n");
                break;
             case 5:printf("Five\n");
                break;
             case 6:printf("Six\n");
                break;
             case 7:printf("Seven\n");
                break;
             case 8:printf("Eight\n");
                break;
             case 9:printf("Nine\n");
                break;
             default:printf("Illegal character\n");
          }
          printf("enter 'c' if you want to continue\n");
          printf("or any other character to end\n");
          ch=getche();
          if (ch!='c')
             printf("End of Session");
       }
}
```

Result of the program

```
Enter a digit 0 t0 9
6
Six
```

```
enter 'c' if you want to continue
or any other character to end
c
Enter a digit 0 t0 9
7
Seven
enter 'c' if you want to continue
or any other character to end
nEnd of Session
```

The `do...while` concept has been used in this example also. The program first enters the `do` loop. After the execution of the loop, the program asks you to enter `c` if you want to continue or any other character to end the program. If you enter `c`, the program will continue. Thus, if you want to exit, you can press any other letter, say `'n'`. If you press `'n'` the program stops instantly. This is the right way of using the `do...while` loop.

Now, the `switch` is analysed. The program asks for entry of a digit `0` to `9`. The entered digit is stored in variable `a`. If `a = 5`, then the program goes to case `5`. It is followed by printing the value Five and then break. If you press 8, then case 8 matches and Eight will be printed.

What is a `break` statement ?

Assume that all the `break` statements are removed from the program. Then if you press 1, 'One' will be printed and all the statements following that will be executed. This means that Two, Three, ... till Nine will be printed. If you enter 7, it will print all the numbers starting from Seven, which in not desirable. The `break` statement has, therefore, been introduced. After printing the value of the number, the `break` statement takes the program to the end of the `switch` statement. The end is just the closing brace corresponding to `switch` after the default `printf()` statement. Therefore, the combination of `switch` and `break` does the trick.

Assuming that a character other than 0 to 9 is entered, none of the cases match. Therefore, default is executed. The program will print 'Illegal Character'. The **default** is optional and even in its absence, when no match is found, the program will come out of the switch statement without any action.

The `case` statements need not be in any specified order. The default can be on top before `case` 0 and the `case` can occur in any order. The program is made to execute, as long as you want, by the `do` statement. If `do` is absent, then the program will be executed only once. The `do...while` is necessary to make it an iterative program.

Every `switch` statement, therefore, contains a condition in the form of an expression. The expression could also be a single variable as in this case. The expression will be evaluated at the time of program execution and must be an integer. Then, depending on the value it goes to a `case` label, which is like a label in a `goto` statement. The label should match with the value of the expression.

Unless the program is made to exit by statements, such as `break` after executing the group of statements corresponding to a particular case, the program will execute all the statements in the program from then on. This should be noted. Therefore, the programmer has to specify where to end. In the case of `if...else`, where to begin and where to end is clear as also in the case of `switch`. The program starts at the beginning of the `case` that meets the condition, but ends at the bottom of the `switch` unless otherwise specified. It will also execute the statements following default. It will of course not bother about the keywords. That is the reason for the `break` statement, since we do not want the program to execute irrelevant statements.

It is a good programming practice to include a break statement after the default as well. If this is not done at the inception, at a later stage when more case statements are added after default, this would lead to problems. Whenever default is executed all the statements following it, even if they belong to some other case, will also be executed if `break` is not included after default.

Now the program can be extended to print the value of the number up to 99 in words. This makes it more complicated since the words are unique up to nineteen. A program is given below for achieving the task. This is made to loop using `do...while`.

```
/*Example 3.9
to print out in words the value
of a number typed in the range 1-99*/
#include <stdio.h>
#include<conio.h>
main()
{
    int num,m,ch='y';
    do
    {
        printf("Type a number 1 to 99\n");
        scanf("%d",&num);
        if ((num >0) && (num <100))
        /*only if the number is within the valid range 0 to 99
the following will be executed*/
        {
            if (num >= 20)
            {
                m=num/10;
                switch(m)
                {
                    case 2:printf("TWENTY ");
                           break;
                    case 3:printf("THIRTY ");
                           break;
                    case 4:printf("FORTY ");
```

NOTES

```
                break;
            case 5:printf("FIFTY ");
                break;
            case 6:printf("SIXTY ");
                break;
            case 7:printf("SEVENTY ");
                break;
            case 8:printf("EIGHTY ");
                break;
            case 9:printf("NINETY ");
                break;
        }
    }
    if (num >20)
        num= num%10;
        switch(num)
        {
            case 1:printf("ONE\n");
                break;
            case 2:printf("TWO\n");
                break;
            case 3:printf("THREE\n");
                break;
            case 4:printf("FOUR\n");
                break;
            case 5:printf("FIVE\n");
                break;
            case 6:printf("SIX\n");
                break;
            case 7:printf("SEVEN\n");
                break;
            case 8:printf("EIGHT\n");
                break;
            case 9:printf("NINE\n");
                break;
            case 10:printf("TEN\n");
                break;
            case 11:printf("ELEVEN\n");
                break;
            case 12:printf("TWELVE\n");
                break;
            case 13:printf("THIRTEEN\n");
                break;
```

NOTES

```
                case 14:printf("FOURTEEN\n");
                      break;
                case 15:printf("FIFTEEN\n");
                      break;
                case 16:printf("SIXTEEN\n");
                      break;
                case 17:printf("SEVENTEEN\n");
                      break;
                case 18:printf("EIGHTEEN\n");
                      break;
                case 19:printf("NINETEEN\n");
                      break;
             }
          }
          else
             printf("number outside range\n");
             printf("\nenter y if you want to continue\n");
             ch=getche();
          if (ch!='y')
             printf("End of session");
          }
       while (ch=='y');
    }
```

Result of the program

```
Type a number 1 to 99
123
number outside range
enter y if you want to continue
yType a number 1 to 99
78
SEVENTY EIGHT
enter y if you want to continue
yType a number 1 to 99
6
SIX
enter y if you want to continue
nEnd of session
```

How does it work ?

The entered number is checked. If it is 20 or above, the following action takes place. For example, assume that a number 64 is entered. It is > = `20`.

Therefore, `m` = `num/10` will give `m` as `6`. Hence, SIXTY will be printed (with a space at the end), corresponding to case 6.

Now since num > 20, the second if condition will be true.

Num = num % 10 will assign the remainder equal to 4 to num. Therefore, case 4 in the second switch will be true and four will be printed.

Assume you have expressed your desire to continue and enter 16.

Since num > = 20 will be false and switch (m) will be ignored. Hence, switch (num) will be executed. Case 16 matches with switch (16). Therefore, SIXTEEN will be printed. The correctness of the program can be checked for any number in the range of 0 – 99.

The expression associated with the switch can be of type char since char can also be considered an integer. For example, the switch statement can be of the form,

```
char m ;
switch (m)
{
   case 'a' : s1 ;
      break ;
   case 'b' : s2 ;
      break ;
}
```

If m evaluates to 'a', s1 will be executed and if it is 'b', s2 will be executed. Remember the label of case has to be a constant expression.

Other characters, such as +, –, etc. can be used as case labels since their integer values are known from the ASCII table.

```
Int op ;
switch (op)
{
   case '+' : s1; break;
   case '/' : s2 ; break;
}
```

Here when op is +, s1 will be executed and when it is /, s2 will be executed.

You can try these by writing programs.

3.4 BREAK, CONTINUE, GOTO STATEMENTS

The keywords while, for and switch test the condition on top, while do...while checks at the bottom for quitting the loop. The break statement helps immediate exit from any part of the loop as demonstrated with the switch statement. It can be used with any other loop construct or anywhere in the program. When the break statement is executed it goes to the bottom of the block. Recall that a block is a group of statements enclosed between an opening brace and the corresponding closing brace.

The **continue statement** is related to break. When continue is executed, it causes the next iteration of the corresponding for, do...while or while loop to begin. Therefore, continue takes the program to the top of the block and in the for loop, it will cause the next increment operation, followed by checking whether the condition is true or false in order to decide the next course of action. This is similar to skipping the current execution and continuing with the next operation after incrementing. The statement continue skips the rest of the statements in the loop for that iteration, whereas break terminates the loop.

Write a program to check whether a given number is positive or negative. If it is zero, the program should terminate after printing the value. If it is a positive integer above zero and <=20000, the value will be printed; if negative, it will go to fetch the next number. If the number is >20000, the program terminates. The program is given below:

```
/*Example 3.10
/*program to demonstrate continue*/
#include <stdio.h>
main()
{
    int a;
    do
    {
       printf("enter a number-enter 0 to end session\n");
       scanf("%d", &a);
       if(a > 20000)
       {
          printf("you entered a high value-going out of
range\n");
             break;
       }
       else
       if(a>=0)
          printf("you entered %d\n", a);
       if (a <0)
       {
          printf("you entered a negative number\n");
       continue;
       }
    }
    while(a !=0);
    printf(" End of session\n");
}
```

Result of the program

```
enter a number-enter 0 to end session
33
you entered 33
enter a number-enter 0 to end session
-60
you entered a negative number
enter a number-enter 0 to end session
45
you entered 45
enter a number-enter 0 to end session
25000
you entered a high value-going out of range
End of session
```

If the number typed > `20000`, or if it is equal to zero, the program comes out of the loop and prints "`End of session`". If the number is negative, `a < 0` and hence, `continue` will be executed. It will go to the top of the loop. The next integer will be received. The program, therefore, terminates when `a = 0` as well as `a > 20000`, but there is a difference. If the number entered is zero, the program checks whether `a > 20000`. Since the condition fails, it checks whether `a > = 0` and since it is true, 0 will be printed and then the `while` condition is checked. The program terminates after the `while` condition is checked.

However, if the number entered is `>20000`, the loop terminates instantly without transacting any business except printing messages as given above.

The `return` statement can appear anywhere in a function and when it is encountered a value is returned to the called function. The `return` may also not return a value in statements as given below:

```
return ;
return (0) ;
```

The `return` statement may appear anywhere in the function and not necessarily at the end of the function. Whenever `return` is executed, the program returns to the function called the current function. The program returns to the place from where it called the function. Thus `return` is also used to suddenly exit from a function or a loop in a function.

Exit Function

The library function **`exit()`** causes the termination of the current program. Note that, `exit()` terminates the execution of the program itself, and not the block. The statement `break` enables coming out of the block or loop in which it is executed but `exit` terminates the program at whatever stage the program may be. `exit` is a powerful function.

3.5 LOOPING STATEMENTS

Quite often we have to perform the same operation a number of times. We may also have to repeat the same operation with one or more of the values changed, which is known as loop or iteration. It is definitely possible to write a program for such situations with the concepts we have learnt so far. However, there are special constructs in any programming language for carrying out repeated calculations.

Iteration Using `if`

Before we look at loop constructs, let us consider an example to see the need for repetitive calculations. Assume that we want to find the sum of the first ten natural numbers, 1 – 10. This can be achieved through successive addition, i.e., first we initialize the sum to 0 and then add 1 to the sum. Next we add 2 to the sum, then 3, and so on till we add 10 to the sum. Thus, by repeated addition 10 times, we have found the sum of first ten natural numbers.

The following algorithm summarizes what we have done.

Step 1: `Sum = 0`

Step 2: `I = 1`

Step 3: `If I < = 10` perform the following operations:

`sum = sum + I; I = I + 1;`

Step 4: Print the sum

Let us analyse the algorithm.

At the beginning, steps 1 and 2 are entered with `sum = 0` and `I = 1`

`since I < = 10`

`Sum` will be equal to `sum + I`,

i.e., `sum = 0 + 1 = 1`

`I = I + 1`, i.e., `I = 2`

Now the program goes to Step 3.

with `I = 2` and `sum = 1`

Since `I < = 10`

`sum = sum + I`

sum was 1 and I is 2

`sum = 1 + 2 = 3`

Next `I` will be incremented to `3`.

Third iteration:

Step 3 is approached with `I = 3` and `sum = 3`.

since `I < = 10`

`sum = sum + I = (1 + 2) + 3`

`I = 4`

Ninth iteration:

Step 3 is approached with `I = 9.`

since `I < = 10`

`sum = (1 + 2 + 3 +............+ 8) + 9`

`I` is incremented to `10.`

Tenth iteration:

Step 3 is approached with `I = 10.`

since `I < = 10`

```
sum  = sum + I
   = (1 + 2 + 3 +........+ 8 + 9) + 10
```

Now I is incremented to 11.

since `I <= 10` is not true, the program does not execute the statements following the `if` and jumps to Step 4.

In Step 4 the sum is printed.

This algorithm is implemented in Example 3.11.

```
/*Example 3.11
Demonstrates the use of if for iteration*/
#include <stdio.h>
main()
{
    int sum=0, i=1; /*declaration and initialization
combined*/
    step3:      /*label- loop starts here*/
    if (i <=10)
       {
          Sum = sum + i;
          i = i + 1;
          goto step3;
       }
    printf("sum of first 10 natural numbers=%d\n", sum);
}
```

Result of the program

```
sum of first 10 natural numbers = 55
```

The program uses `if` and `goto` keywords. According to the algorithm, the program has to go to Step3. Step3 in this program is called a label, which is followed by a colon. The rules for coining a label name are the same as for an identifier. The label can be placed anywhere in the same function where the `goto` statement is found. Usage of `goto` is considered to be bad programming practice since it leads to errors when changes are made in the program and also affects the readability. It is always possible to write a program without using `goto`. This can be done by using a `for` statement.

`For` Statement

The `for` statement is meant for easy implementation of iterations unlike `if`. The syntax of `for` is as follows:

```
for (exp1; exp2; exp3)
{statements;}
```

Note the keyword, the parentheses and semicolons. There is no semicolon after `exp3`. `exp1`, `exp2` and `exp3` are expressions. The usage of the `for` loop is given below:

`exp1`– Contains the initial value of an index or a variable.

`exp3`– Contains the alteration to the index after each iteration of the body of the `for` statement. The body of the statement is either a single statement or a group of statements enclosed within braces.

If a single statement has to be executed, then braces are not required.

`exp2`– Condition that must be satisfied if the body of statements is to be executed.

An example of a `for` loop is given below:

```
for (i = 0;  i < 5; i++)
printf("%d", i);
```

The loop will start with an initial value of `i = 0`. Since `i < 5`, the body of the `for` loop will be executed and it will print `0`. Now the `exp3` will be executed and `i` will be incremented to `1`. Since `i` is less than `5`, body of the loop will again be executed to print `1`. This will continue till `4` is printed. `i` will now be incremented to `5` and since `i` is not less than `5`, the `for` loop will be terminated. This is how `for` is used to carry out repetitive operations.

Let us now write a program for finding the sum of the first ten natural numbers using the `for` statement.

The program is given in Example 3.12

```
/*Example 3.12
Demonstrates the use of the for statement to find the sum of
the first 10 natural numbers*/
#include <stdio.h>
main()
{
   int sum=0, i; /*declaration and initialization combined*/
   for (i=1; i<=10; i++)      /*loop starts here*/
      {
         Sum = sum + i;
      }
   printf("sum of the first 10 natural numbers=%d\n", sum);
}
```

Result of the program

```
sum of first 10 natural numbers = 55
```

Note the difference between Example 3.11 and Example 3.12.

We have eliminated the label Step 3 and the `goto` statement.

The initialization of `i = 1` is carried out as part of the `for` statement.

The incrementing of `i` is also carried out as part of the `for` statement. The program has, therefore, been simplified.

How does the program work?

Step 1: `i = 1`

`i` is checked with `i <= 10`

Since `i` is less than `10`, the `for` loop is executed.

`sum = sum + i = 0 + 1 = 1`

Step 2: `i` is incremented to `2`

`2 is <=10`

Therefore, the `for` loop is executed.

`sum = sum + i = (1) + 2`

Steps 3 to 8: Same logic applies till the condition is met.

Step 9: `i` is incremented to `9`

`9 is < = 10`

Therefore, the `for` loop is executed.

`sum = sum + i = (1 + 2 +........+ 8) + 9`

Step 10: `i` is incremented to `10`

`10 is < = 10`

`sum = sum + I = (1 + 2 +..........+ 9) + 10`

Step 11: `i` is incremented to `11`.

`11 is not < = 10.`

Therefore, the `for` loop is now terminated.

The `printf()` function is now executed automatically.

Let us summarize the operation of the `for` loop.

When a program encounters a `for` loop, it first checks the condition through the expression in the middle. If the condition is satisfied it executes the group of statements. After executing the statements in the body of the loop, the program transfers the execution to the `for` statement and the third expression is executed, which is usually incrementing or decrementing. Then the condition is checked. If the condition is not satisfied, the group of statements will not be executed and the program will skip to the next statement after the statements pertaining to the `for` statement.

By chance if the initial value was typed as `11` instead of `1` in the program, the condition will turn out to be false and the group of statements will not be executed at all.

Three Components of `for`

The three components of a `for` statement are as follows:

`exp1` and `exp3` are assignments or function calls. We will discuss function calls at a later stage; `exp2` is a relational expression. The three expressions may not always be present. However, even if an expression is not present, the associated semicolon should be present.

For example,

```
for (; exp2 ;)
    {s1}
```

Here the initial value is not specified and the incrementing does not take place after every iteration. Presumably, the initial value is assigned elsewhere and incrementing or a similar operation takes place as part of the group of statements following the `for`. However, since `exp2` is present, the loop will terminate.

However, if all three expressions are omitted as given below,

```
for  ( ; ; )
```

the loop will never terminate because the conditional statement is absent. If `exp2` is not present, it is assumed that the condition is true always. Such a statement should not be used.

Instead of incrementing, we can use `i + = 2` as `exp3` when `i` will be incremented by 2 every time.

Let us try to print the list of even numbers up to 50. The program is given below:

```
/*Example 3.13
variation in for statement - to print even numbers*/
#include <stdio.h>
main()
{
    int i=2;
    for (; i<50; i+=2)  /*loop starts here*/
       {
          printf("%i is an even number\n",i);
       }
}
```

Here we initialize `i = 2` before the `for` loop itself. However, the corresponding semicolon is present at the right place.

Result of the program

```
2 is an even number
4 is an even number
6 is an even number
8 is an even number
10 is an even number
```

```
12 is an even number
14 is an even number
16 is an even number
18 is an even number
20 is an even number
22 is an even number
24 is an even number
26 is an even number
28 is an even number
30 is an even number
32 is an even number
34 is an even number
36 is an even number
38 is an even number
40 is an even number
42 is an even number
44 is an even number
46 is an even number
48 is an even number
```

Other forms of `for` Loop

The `for` loops can be nested as follows:

```
for (i = 1; i <= 10; i++)
{
    for (j = 1; j <= 5; j++)
    {
       for (k = 1; k <= 2; k++)
       {
          s1
       }
    }
}
```

The statement `s1` will be executed as follows:

First time	`i = 1,  j = 1,  k = 1`
Second time	`i = 1,  j = 1,  k = 2`
Third time	`i = 1,  j = 2,  k = 1`
	`i = 1,  j = 2,  k = 2`
	`i = 1,  j = 3,  k = 1`
	`i = 1,  j = 3,  k = 2`
Lastly	`i = 10,  j = 5,  k = 2`

`s1` will be executed `2*5*10 = 100` times.

Any level of nesting is acceptable; however, the higher the level of nesting is, the more easy it will be to commit mistakes and more difficult to understand.

Let us look at some more `for` loops.

```
For ( x = -5;  --x >= -10; )
{
 }
```

Here the decrement and conditional statements are combined in `exp2`.

Since decrement is a prefix, the decrement of `x` is carried out first. The condition is then checked in order to decide whether to continue or not. Then the loop is executed. Therefore, the first iteration will be carried out with `x = -6` and the last with `x = -10`.

Another variation of the statement is given as follows:

```
for (y = 100;  y ++<= 200;)
{
    s2
}
```

Here too `exp2` and `exp3` are combined. This is a postfix notation. The following sequence is carried out: condition check, increment and execute the loop. Therefore, the statement `s2` following the `for` loop will be executed the first time with `y = 101` and finally with `y = 201` as well.

The `for` loop is a popular iteration construct not only in 'C' but also in other languages. Here the initial value, the step and the final value are clear and unambiguous and simple to write. There are other loop statements also. In the next section, we will study the `while` loop.

The while Loop

The `while` loop is a subset of the `for` loop. The syntax for the `while` loop is given as follows:

```
while (expression)
    {statements;}
```

This means that the statement(s), which is a single statement or multiple statements, will be executed while the expression is true. When it becomes false, the execution will stop.

The `while` loop is similar to the `for` loop without `exp1` and `exp3`. The `for` loop can be simulated or replaced with the `while` loop as follows.

```
exp1;
    while (exp2)
    {
        statements
        exp3;
    }
```

The programmer can use `while` or `for` at his discretion. The `for` loop is preferred when the initialization and incrementation are simpler.

Let us look at some examples.

Let us write a program for the generation of any multiplication table.

The program is as follows:

```
/*Example 3.14
use of while - You can generate multiplication
tables of your choice using this program.
caution: Don't exceed maximum limits of integer */
#include <stdio.h>
main()
   {
      int a,b,product;
      a=1;
      b=0;
      product=0;
      printf("Enter which table you want");
      scanf("%d",&b);
      while (a <=10)
      {
         product = a*b;
         printf("%2d X %d= %3d\n",a,b,product);
         a++;
      }
   }
```

When the program asks you to enter a table and you type 12, you will get the 12th table. This is shown as follows:

Result of the program

```
Enter which table you want 12
1 X 12 = 12
2 X 12=  24
3 X 12=  36
4 X 12=  48
5 X 12=  60
6 X 12=  72
7 X 12=  84
8 X 12=  96
9 X 12=  108
10 X 12= 120
```

Note here that the condition is (`a < =10`); incrementing is done within the loop in `a++`. Variable `a` is initialized as `1` before entering the `while` loop.

If you want to print the table up to `16*12 = 192` then simply change the condition to `while (a <=16)`. Simple, isn't it?

Let us now write a program to reverse a given number. For example, if 345 is the given number, we should get the reversed number as 543. The program is given below.

```
/*Example 3.15
/*Program to reverse a number*/
#include<stdio.h>
main()
{
    long unsigned n, reverse;
    printf("enter the number to be reversed\n");
    scanf("%lu", &n);
    reverse=0;
    while (n>0)
    {
       reverse=reverse*10+(n%10);
          n=n/10;
    }
    printf("Reversed number=%lu", reverse);
}
```

Result of the program:

```
enter the number to be reversed
4562
Reversed number=2654
```

How does the program work? Let us discuss for `n = 345`.

Before entering the `while` loop, `n = 345`

`reverse = 0`

Since `345 > 0`, the condition is true and the loop is entered.

Iteration 1

```
    reverse  = 0 *10 + (345 % 10)
             = 0 + 5 = 5
          n  = 345 / 10 = 34
```

Iteration 2

```
    reverse  = 5 * 10 + (34 % 10) because n = 34 now
             = 50 + 4 = 54
      n = 34 / 10 =3
```

Iteration 3

```
    reverse  = 54 * 10 + (3 % 10)
             = 540 + 3 = 543
          n  = 3 / 10 = 0
```

NOTES

Now since `n` is not greater than `0`, the program will come out of the loop and will, print.

```
Reversed number = 543
```

Do ... While Loop

The `Do ... While` loop is a modification of the `while` statement. In the `while` statement, before the group of statements following the `while` are executed, the condition associated with the `while` is checked. If the condition is true or fulfilled, then the associated statements are executed. If not, the program skips the statements associated with the `while` loop. This is depicted in Figure 3.1.

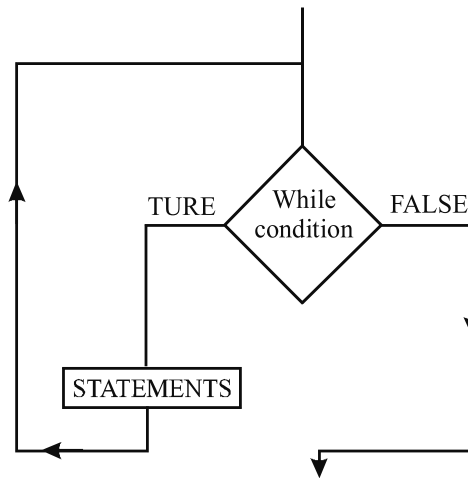


Fig. 3.1 *`while` Loop*

After execution of the statements, the program will check again whether the condition is true and then continue to execute or skip the statements depending on the condition. The statements may not be executed even once if the condition was false at the entry point.

However, the `do...while` loop works differently as shown in Figure 3.2.

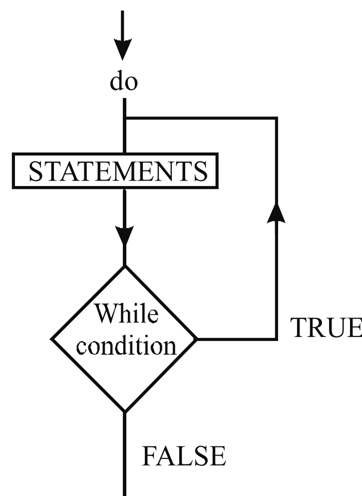


Fig. 2.2 *`do...while` Loop*

Here the statements following do will be executed once before the condition is checked. If it is true, then the statements will be executed again. If not, the program will skip the statements and proceed further. Thus, whatever be the condition, the statements following do will be executed once before checking the condition. This is the essential difference between do...while and while. The while loop tests the condition on top of the loop; but do...while tests at the bottom after executing the statements. The while loop executes the statements after checking the condition; but do...while executes the statements before testing the condition. The syntax of do...while is as follows:

```
do
{
   statements
}
while (expression);
```

The statement do...while is not used as frequently as the while loop.

Example 3.16 converts upper case alphabets to lower case. In case you want to convert more alphabets you will have to execute the program again and again. This can be avoided by the while loop. The program will continue to run as long as you want to convert more and more alphabets. The program has been rewritten with while for converting upper case to lower case. You can convert as long as you want. When you want to stop, enter 1. The program, which uses while for converting an upper case character to a lower case is given in Example 3.16.

```
/*Example 3.16
Conversion of upper case to lower case alphabet*/
#include <stdio.h>
#include<conio.h>
main()
{
    int alpha=0;
    while (alpha!='1')
    {
       printf ("\nenter upper case alphabet- enter 1 to
quit\n");
       alpha=getche();
       if ( alpha >='A' && alpha<='Z')
       {
          alpha= (alpha+32);
          putch(alpha);
       }
       else
       {
          if(alpha!='1')
```

NOTES

```
        printf("\ninvalid entry; retry");
      else
      printf("End of session");
    }
  }
}
```

Output of the program

```
enter upper case alphabet- enter 1 to quit
Pp
enter upper case alphabet- enter 1 to quit
p
invalid entry; retry
enter upper case alphabet- enter 1 to quit
Qq
enter upper case alphabet- enter 1 to quit
1End of session
```

This program works till `1` is pressed. It continues to convert upper case to lower case till `1` is pressed. The program has been designed in such a manner that it will perform at least one iteration of the statements following `while`. This can be rewritten using `do...while`. The rewritten program is given below:

```
/*Example 3.17
Conversion of upper case to a lower case alphabet*/
#include <stdio.h>
#include<conio.h>
main()
{
    int alpha=0;
    do
    {
       printf ("\nenter upper case alphabet- enter 1 to
quit\n");
       alpha=getche();
       if ( alpha >='A' && alpha<='Z')
       {
          alpha=(alpha+32);
          putch(alpha);
       }
       else
       {
          if(alpha=='1')
```

```
            printf("End of Session");
         else
            printf("\ninvalid entry; retry");
      }
   }while(alpha!='1');
}
```

Result of the program

```
enter upper case alphabet- enter 1 to quit
Gg
enter upper case alphabet- enter 1 to quit
o
invalid entry; retry
enter upper case alphabet- enter 1 to quit
Dd
enter upper case alphabet- enter 1 to quit
1End of Session
```

How does it differ? Here too, the program will attempt to convert one character before it can be terminated. Assuming that the first character was 1, the program will still attempt to convert it and print the message "`End of Session`" before it quits.

Suppose the first character is a valid one and a number of characters are converted in succession; when you want to terminate the program, 1 has to be pressed and even then the program will not stop immediately. It will stop only after the statements are executed. Since the problem is the same, a detailed look at both the examples will bring out the similarity in operation between both the constructs. However, there are occasions when it is quite suitable as given in the next section.

Check Your Progress

1. Give the syntax of `if` statement.
2. Which are the logical operators provided by 'C'?
3. How are `continue` and `break` statements related?
4. Define the term `exit()` function.
5. Write the syntax for `for` statement.
6. Define the term `while loop`.

3.6 ARRAYS IN C

An array refers to data structure which holds multiple variables of the same data type. An array uses an indexing system to find each variable stored within it. In C, indexing starts at zero. Arrays, like other variables in C, must be declared before

NOTES

they can be used. Arrays have the following syntax using square brackets to access each indexed value called an element:

```
x[i]
```

In the above declaration, `x[5]` refers to the sixth element in an array called `x`. In C, array elements start with 0. Assigning values to array elements is done by the following statement:

```
x[10] = g;
```

Assigning array elements to a variable is done by the following statement:

```
g = x[10];
```

In the following example, a character based array named word is declared, and each element is assigned a character. The last element is filled with a zero value to signify the end of the character string. A `printf()` function is then used to print out all elements of the array. For this, following C code is required:

```
/* Example 3.18.
Introducing array's */
#include <stdio.h>
int main()
    {
       char word[20];
       word[0] = 'W';
       word[1] = 'E';
       word[2] = 'L';
       word[3] = 'C';
       word[4] = 'O';
     word[5] = 'M';
     word[6] = 'W';
       word[7] = 0;
       printf("The contents of word[] is %s\n", word);
     getch();
    }
```

Result of the program

```
The contents of word[] is WELCOME
```

3.6.1 Declaring Arrays

We can declare an array by specifying its data type, name and the number of elements the array holds between square brackets immediately following the array name. The following syntax is required to declare an array:

```
data_type array_name[size];
```

For example, to declare an integer array which contains 100 elements following statement is required:

```
int a[100];
```

There are some rules on array declaration. The data type can be any valid C data types including structure and union. The array name has to follow the rule of variable and the size of array has to be a positive constant integer. A value stored into an element in the array simply by specifying the array element on the left hand side of the equals sign. The declaration `int values[10];` would reserve enough space for an array called values that could hold up to `10` integer values. Initializing arrays is like a variable in which an array can be initialized. To initialize an array, you provide initializing values which are enclosed within curly braces in the declaration and placed following an equals sign after the array name. The following statement is required to initialize an integer array:

```
int list[5] = {2,1,3,7,8};
```

3.6.2 One Dimensional Array

An array is a vector defined as a simple data structure. It holds a fixed number of equal size data elements of the same data type.

If the array elements are known beforehand, they can be defined right at the beginning. If the ages of the employees are known beforehand, they can be declared as:

```
int emp_age [5] = {40, 32, 45, 22, 27};
```

The data elements are written within braces separated by commas. In this case, when data elements are declared, there is no need to declare the size; we can write:

```
int emp_age [] = { 40, 32, 45, 22, 27 };
```

The latter is advantageous. If the size is not declared, the compiler will count the number of elements and automatically allot the size. On the other hand, if we specify the size and give lesser number of elements, the compiler will assume the other elements to be zero.

For example, if we declare

```
int Marks [ 5 ] = { 100, 70, 80 };
```

In this case, `Marks [0] = 100`

```
Marks [1]   =   70
Marks [2]   =   80
```

What happens to the other elements? The computer will assign 0 to them.

```
Marks [3] = 0
Marks [4] = 0
```

If you declare size 5 and give 7 elements, there will be an error. Therefore, if you know the data elements in advance, you can allow the compiler to calculate the size.

Now let us try some programs.

For example, to get 10 integers, one at a time and print them after they are collected the program will be as follows:

```
/*Example 3.19
  Ten integers of an array are scanned using the
  scan function and printed */
```

NOTES

```
#include <stdio.h>
int main()
{
    int s1[10];
    int i;
    printf("Enter 10 integers \n");
    for (i=0; i<=9; i++)
    {
        scanf("%d",&s1[i]);
    }
    printf("you have entered:\n");
    for (i=0; i<=9; i++)
    {
        printf ("%d\n", s1[i]);
    }
    return 0;
}
```

Result of the program

```
Enter 10 integers
1 2 3 4 5 6 7 8 9 0
you have entered:
1
2
3
4
5
6
7
8
9
0
```

Analyse the program carefully. We declare `s1` as an integer array of size 10.

Next, we use the first `for` loop to scan the entered integers from the keyboard. At the first iteration, `s1[0]` will be received and stored at location `& s1[0]`. This is similar to a simple variable where `'&'` denotes the address of the variable. This is repeated till `s1[9]` is received and stored at `& s1[9]`.

The next `for` loop prints the value of `s1[0]` to `s1[9]`, i.e., 10 integers one at a time and in new line.

Thus, the array is handled the same way and each element has a distinct identification. The elements of an array have the same name with a subscript corresponding to the position, i.e., the array name with the subscript written in square brackets. Elements are indexed for code execution. 'C' does zero(0) based indexing. All the array elements share the common name called array variable, which has been declared as `emp_age [5]` Example 3.19.

Consider another program to understand one dimensional arrays more clearly.

Assume the existence of an array of elements. You want to find out the greatest number in the array and its location. To do that, you set up two other variables known as `max` and `ind`. You initialize them to zero in the above program. Then, you compare each number with `max`. If it is greater than `max`, you note the location in `ind` and the associated value in `max`. When you have checked all the elements in the array, you would have got the greatest number and its location. Since the first element has a subscript 0, you have to add 1 to the subscript to get the position. The program is as follows:

```
/*Example 3.20
   to find the greatest number and
   its position in an array*/
   #include<stdio.h>
   int main()
   {
       int a[5]={1,5,2,6,3};
       int max=0, i, ind=0;
       for(i=0; i<=4; i++)
       {
          if (a[i] > max)
          {
             max =a[i];
             ind=i+1;
          }
       }
       printf("maximum number=%d location=%d\n", max, ind);
       return 0;
   }
```

Result of the program

```
maximum number=6 location=4
```

Let us see how the program works.

```
Iteration 1
    i = 0   max = 0    ind = 0
    a[0] = 1
    since a[0] > max
    max = 1
    ind = 1
Iteration 2
    i = 1   max = 1   ind = 1
    a[1] = 5
    since a[1] > max
    max = 5   ind = 2
Iteration 3
```

```
    i = 2     max = 5      ind = 2
    a[2] = 2
    since a[2] < max no change
Iteration 4
    i = 3   max = 5    ind = 2
    a[3] = 6
    Since a[3] > max
    max = 6    ind = 4
Iteration 5
    i = 4   Max = 6      ind = 4
    a[4] = 3
    since a [4]  < max     no change
```

The program prints:

```
max = 6    location = 4
```

Thus, arrays are very useful for solving real problems encountered every day.

3.6.3 Two Dimensional Array

Multi dimensional arrays operate on the same principle as single dimensional arrays. You have to give the dimensions of the two subscripts or indices in case of a two dimensional array. For example,

```
w [10][5]
```

is a two dimensional array with different subscripts. Here, there will be 50 different elements. The first element can be denoted as `w [0][0]`.

The next element will be `w[0][1]`.

The fifth element will be `w[0][4]`.

The sixth element will be `w[1][0]`.

The last element will be `w[9][4]`.

This can be considered as a row and column representation. There are 10 rows and 5 columns in the above example. When data is stored in the array, the second subscript will change from 0 to 4, one at a time, with the first subscript remaining constant at 0. Then, the first subscript will become 1 and the second subscript will keep increasing from 0 to 4. This is repeated till the first subscript becomes 9 and the second 4. This array can be used to represent the names of 10 persons, with each name containing 5 characters. The first subscript refers to the name of the 0th person, 1st person, 2nd person and so on. The second subscript refers to the 1st character, 2nd character and so on of the name of a person. Thus, 10 such names can be stored in this array.

The dimension of the array can be increased to 3 with 3 square brackets as given below:

```
Marks [ 50 ][ 3 ][3 ];
```

The name of the first element will be `Marks [ 0 ] [ 0 ] [ 0 ]`

The last element will be `Marks [ 49 ] [ 2 ] [2 ]`.

It would be easy to add more dimensions to an array but it would also become more difficult to comprehend under normal circumstances. It may, therefore, be useful to solve complicated scientific applications, however. Now, let us understand the concept of multidimensional arrays using a simple problem.

Assume that we need to write a program to read two arrays (both two dimensional) and multiply the corresponding elements and store them in another two dimensional array. To make the problem simpler, we will use [2][2] arrays.

Let us call the arrays *x*, *y* and *z*.

We have
```
x = {x[0][0] x[0][1] }  y = { y [ 0 ] [ 0 ] y[0][1] }
    {x[1][0]  x[1][1] }     { y [ 1 ] [ 0 ] y[1][1] }
```

We want to multiply `x[0][0]` and `y[0][0]` and store the result in `z[0][0]` and so on.

The values of *x* and *y* are given in the program itself.

```
/*Example 3.21
  /* multiplication of two 2 dimensional arrays*/
  #include <stdio.h>
  int main()
  {
     int i,j;
     int z[2][2];
     int x[2][2]={1, 2, 3, 4};
     int y[2][2]={5, 6, 7, 8};
     for (i=0; i<=1; i++)
     {
        for (j=0; j<=1; j++)
        {
           z[i][j]=x[i][j]*y[i][j];
           printf("z[%d][%d]=%d\n", i, j, z[i][j]);
        }
     }
     return 0;
  }
```

We have declared two arrays `x[2]` and `y[2]` as follows:

```
        x   =   {1  2}                y  = {5  6}
                {3  4}                     {7  8}
x[0][0]  =  1   x[1][1]               =  4
y[0][0]  =  5   y[1][1]               =  8
```

Therefore, after multiplication of the respective elements, we get

```
z = {5 12}
    {21 32}
```

The program prints out the values of the products stored in array `z`.

Result of the program

```
z[0][0]=5
z[0][1]=12
z[1][0]=21
z[1][1]=32
```

Note that the elements are stored row by row contiguously.

In the above example, we have declared elements with a two dimensional array and initialized its one dimensional array as given below:

```
x [2] [2] = {1, 2, 3, 4} ;
```

The system correctly interpreted the same and we get result the correct as product of two matrices. We can actually present the above in another manner as given below:

```
int x [2] [2] = {
          {1, 2},
          {3, 4}
       };
```

In this method, we can indicate the elements closer to a matrix form. Both the above definitions are equivalent. In the latter definition, we can easily visualize a two dimensional array. The first row represents first row of the two dimensional array. The values in the second row represent the second row of the two dimensional array.

To practise the above representation, let us take another example of a two-dimensional array.

```
x [3] [2] = {10, 20, 30, 40, 50, 60} ;
```

The same can be represented in the second form as given below:

```
x [3] [2] = {
      {10, 20},
      {30, 40},
      {50, 60}
   };
```

Note that there is no comma at the end of the last row.

Finding Transpose of a Matrix

Arrays are used to represent matrices. Inter changing rows and columns in a matrix results in transpose of a matrix.

For example,

If $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

Transpose of A = $A^t = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

The algorithm for finding transpose is very simple. After reading the elements of `A [i] [j]`, we get transpose matrix `B [i] [j]` by just assigning each element as follows:

`B[i] [j] = A [j] [i]`

It is so simple.

The algorithm is given below:

NOTES

Algorithm for Finding Transpose of a Matrix

```
Step 1:         int i, j, row, column;
Step 2:         int A[10] [10], B[10] [10];
Step 3:         read row & column of given matrix A
Step 4:         /*getting given matrix row by row*/
                for [i=0; i<row; i++)
                {
                print ("Enter row number of given matrix\n");
                for (j=0; j<column; j++)
                read A[i] [j]
                }
Step 5:         /*transpose of given Matrix */
                for (i=0; i<column; i++)
                for (j=0; j<row; j++)
                  {
                B [i] [j] = A [j] [i];
                }
Step 6:         /*Printing values of Product matrix */
                print ("Elements of transpose matrix are:");
                for (i=0; i<column; i++)
                  {
                 for (j=0; j<row; j++)
                 print (B[i] [j]);
                }
Step 7:         End
```

The program implementing the above algorithm is given below:

```
/*Example 3.22
finding transpose of a matrix*/
#include<stdio.h>
int main()
{
int i,j;
int row, column;
int A[10][10], B[10][10];
```

NOTES

```
printf("Enter number of rows of given matrix\n");
scanf("%d", &row);
printf("Enter number of columns of given matrix\n");
scanf("%d", &column);

/*getting given matrix row by row*/
for(i=0; i<row; i++)
{
printf("Enter row number %d of given matrix\n", i+1);
for(j=0; j<column; j++)
scanf("%d", &A[i][j]);
}

/*transpose of given Matrix */
 for(i=0; i<column; i++)
 for(j=0; j<row; j++)
   {
   B[i][j]=A[j][i];
    }
/*Printing values of Product matrix*/
printf("Elements of transpose matrix are:\n");
for(i=0; i<column; i++)
  {
   for(j=0; j<row; j++)
 printf("%d\t", B[i][j]);
    printf("\n");
     }
    return 0;
   }
```

Result of the program

```
Enter number of rows of given matrix
3
Enter number of columns of given matrix
4
Enter row number 1 of given matrix
1    2    3    4
Enter row number 2 of given matrix
5    6    7    8
Enter row number 3 of given matrix
9    10    11    12
Elements of transpose matrix are:
1    5    9
2    6    10
3    7    11
```

```
4    8    12
```

Notice that we had entered a 3 × 4 matrix. After transposing the elements, it has become a 4 × 3 matrix. Essentially, each row of matrix A has become a column of B. Finding transpose of a matrix is required in many applications.

The input in the above program was a rectangular matrix. What will happen if we input a square matrix in the above program? Let us execute the program and see.

```
Enter number of rows of given matrix
3
Enter number of columns of given matrix
3
Enter row number 1 of given matrix
1    2    3
Enter row number 2 of given matrix
4    5    6
Enter row number 3 of given matrix
7    8    9
Elements of transpose matrix are:
1    4    7
2    5    8
3    6    9
```

During the second time, we wanted to transpose a square matrix. So a 3 × 3 matrix was given as input. The output is a perfect transpose of a given matrix. The columns and rows have interchanged. Thus, the program works correctly both for square matrix as well as the rectangular matrix.

3.6.4 Multidimensional Array

Multi-dimensional arrays operate on the same principle as single dimensional arrays. They are also a user defined data type. We have to give the dimensions of the two subscripts in case of a 2-dimensional array.

For instance, w [10][5]

is a two-dimensional array with different subscripts. Here, there will be 50 different elements. The first element can be denoted as w [0][0].

The next element will be w[0][1].

The fifth element will be w[0][4].

The sixth element will be w[1][0].

The last element will be w [9][4].

This can be considered as a row and column representation. There are 10 rows and 5 columns in this example. When data is stored in the array, the second subscript will change from 0 to 4, one at a time, with the first subscript remaining constant at 0. Then the first subscript will become 1 and the second subscript will keep increasing from 0 to 4. This is repeated till the first subscript becomes 9 and the second 4. This array can be used to represent the names of 10 persons, with

each name containing 5 characters. The first subscript refers to the name of the 0th person, 1st person, 2nd person and so on. The second subscript refers to the 1st character, 2nd character and so on of the name of a person. Thus, 10 such names can be stored in this array.

The dimension of the array can be increased to 3 with 3 square brackets as given below: *Marks* [50][3][3];

The name of the first element will be *Marks* [0] [0] [0]

The last element will be *Marks* [49] [2] [2].

It would be easy to add more dimensions to an array but it would also become more difficult to comprehend under normal circumstances. It may, therefore, be useful to solve complicated scientific applications, however. Now let us focus on the concept of multi-dimensional arrays using a simple problem.

Assume that we need to write a program to read two arrays (both 2 dimensional) and multiply the corresponding elements and store them in another 2-dimensional array. To make the problem simpler, we will use [2][2] arrays.

Let us call the arrays x, y & z.

We have x = {x[0][0] x[0][1] } y = {y[0][0] y[0][1] }

{x[1][0] x[1][1] } {y[1][0] y[1][1] }

We want to multiply x [0][0] and y [0][0] and store the result in z [0][0] and so on.

The values of x and y are given in the program itself.

Example 3.23

```
/* multiplication of corresponding elements of two 2-dimensional
arrays*/
#include <stdio.h>
int main()
{
    int i,j;
    int z[2][2];
    int x[2][2]={1, 2, 3, 4};
    int y[2][2]={5, 6, 7, 8};
    for (i=0; i<=1; i++)
    {
        for (j=0; j<=1; j++)
        {
            z[i][j]=x[i][j]*y[i][j];
            printf("z[%d][%d]=%d\n", i, j, z[i][j]);
        }
    }
}
```

We have declared two arrays x[2] & y[2] as follows:

x = { 1 2 } y = { 5 6 }

{ 3 4 } { 7 8 }

x [0][0] = 1 x [1][1] = 4

y [0][0] = 5 y [1][1] = 8

Therefore, after multiplication of the respective elements, we get

```
z = { 5  12 }
    { 21 32 }
```

The program prints the values of the products stored in array z.

The output of the program appears as follows:

```
z[0][0]=5
z[0][1]=12
z[1][0]=21
z[1][1]=32
```

Note that the elements are stored contiguously, row by row.

In the above example, we have declared elements with a two-dimensional array and initialized its one-dimensional array as given below:

```
x [2] [2] = {1, 2, 3, 4} ;
```

The system correctly interpreted the same and we got the result correctly. We can actually present the above in another manner as given below:

```
int x [2] [2] = {
                  {1, 2},
                  {3, 4}
          };
```

In this method, we are indicating the elements closer to the matrix form. Both the above definitions are equivalent. In the latter definition, we can easily visualize a two-dimensional array. The first row represents the first row of the two-dimensional array. The values in the second row represent the second row of the two-dimensional array.

To practise the above representation, let us take another example of a two-dimensional array.

```
x [3] [2] = {10, 20, 30, 40, 50, 60} ;
```

The same can be represented in the second form as given below:

```
x [3] [2] = {
          {10, 20},
          {30, 40},
          {50, 60}
  };
```

Note that there is no comma at the end of the last row.

Finding Transpose of a Matrix

Arrays can be used to represent matrices. Inter-changing rows and columns in a matrix obtain transpose of a matrix.

NOTES

For instance,

$$\text{If } A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$\text{Transpose of } A = A^t = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

The algorithm for finding transpose is very simple. After reading the elements of A [i] [j], we get transpose matrix B [i] [j] by just assigning each element as follows:

B[i] [j] = A [j] [i]

It is so simple.

The algorithm is given below:

Algorithm for Finding Transpose of a Matrix

```
Matrix_Transpose(int A[i][j])
Step 1:   Declare int i, j, row, column;
Step 2:   Declare int A[10] [10], B[10] [10];
Step 3:   read values of row & column of given matrix A
Step 4:   /*getting given matrix row by row*/
            for [i=0; i<row; i++)
            {
            print("Enter row number of given matrix\n");
            for (j=0; j<column; j++)
            read A[i] [j]
            }
Step 5:   /*transpose of given Matrix */
            for (i=0; i<column; i++)
            for (j=0; j<row; j++)
              {
                    B [i] [j] = A [j] [i];
                            }
Step 6:   /*Printing values of Product matrix */
            print ("Elements of transpose matrix are:");
            for (i=0; i<column; i++)
              {
              for (j=0; j<row; j++)
             print (B[i] [j]);
                    }
Step 7:   End
```

The program implementing the above algorithm is given below:

Example 3.24

```
finding transpose of a matrix*/
#include<stdio.h>
int main()
{
int i,j;
int row, column;
int A[10][10], B[10][10];
printf("Enter number of rows of given matrix\n");
scanf("%d", &row);

printf("Enter number of columns of given matrix\n");
scanf("%d", &column);

/*getting given matrix row by row*/
for(i=0; i<row; i++)
{
printf("Enter row number %d of given matrix\n", i+1);
for(j=0; j<column; j++)
scanf("%d", &A[i][j]);
}

/*transpose of given Matrix */
 for(i=0; i<column; i++)
 for(j=0; j<row; j++)
  {
   B[i][j]=A[j][i];
    }
/*Printing values of Product matrix*/
printf("Elements of transpose matrix are:\n");
for(i=0; i<column; i++)
 {
 printf("\n");
 for(j=0; j<row; j++)
 printf("%d\t", B[i][j]);
   }
  }
```

The result of the program is given below:

```
Enter number of rows of given matrix
3
Enter number of columns of given matrix
4
Enter row number 1 of given matrix
1   2   3   4
Enter row number 2 of given matrix
5   6   7   8
Enter row number 3 of given matrix
9   10   11   12
Elements of transpose matrix are:

1   5   9
2   6   10
3   7   11
4   8   12
```

NOTES

Notice that we had entered a 3x4 matrix. After transpose it has become a 4x3 matrix. Essentially, each row of matrix A has become a column of B. Finding the transpose of a matrix is required in many applications.

The input in the above program was a rectangular matrix. What will happen if we input a square matrix in the above program? Let us execute the program and see.

```
Enter number of rows of given matrix
3
Enter number of columns of given matrix
3
Enter row number 1 of given matrix
1   2   3
Enter row number 2 of given matrix
4   5   6
Enter row number 3 of given matrix
7   8   9
Elements of transpose matrix are:

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

During the second time, we wanted to transpose a square matrix. So a 3x3 matrix was given as input. The output is a perfect transpose of a given matrix. The columns and rows have interchanged. Thus, the program works correctly, both for square matrix as well as the rectangular matrix.

Triangular Matrices

Upper Triangular Matrix

We may recall that a principal diagonal or leading diagonal of a square matrix A contains the elements whose row index and column index are same, i.e., it includes the elements A[1] [1], A[2] [2], A[3] [3], etc. A square matrix in which all the elements below the leading diagonal are zero is called an upper triangular matrix. For instance, the following is an upper triangular matrix.

{10, 20, 30}

{ 0, 50, 60}

{ 0, 0, 40}

Notice that all the elements below the leading diagonal are zero. This can be expressed mathematically as A [i] [j] = 0 for i>j.

Now let us write an algorithm to print the elements of an upper triangular matrix. Essentially, we need not print those elements, which we know will be zero in an upper triangular matrix.

Algorithm for printing elements of upper triangular matrix

Matrix_Upper_Triangular (int A[i][j])

Step 1: int i,j, row;

Step 2: int A[10][10];

Step 3: read row

NOTES

Step 4: /*getting given matrix row by row*/

```
for(i=0; i<row; i++)
{
for(j=0; j<row; j++)
scanf("%d", &A[i][j]);
}
```

Step 5: /*Printing values of matrix*/

```
for(i=0; i<row; i++)
{
for(j=0; j<row; j++)
{if(i>j) continue;
else
print(A[i][j]);
}
}
}
```

Step 6: End

A program implementing the above algorithm is given below:

Example 3.25

```
/* printing elements of upper triangular matrix*/
#include<stdio.h>
int main()
{
int i,j;
int row;
int A[10][10];
printf("Enter number of rows of given matrix\n");
scanf("%d", &row);

/*getting given matrix row by row*/
for(i=0; i<row; i++)
{
printf("Enter row number %d of given matrix\n", i+1);
for(j=0; j<row; j++)
scanf("%d", &A[i][j]);
}
/*Printing values of matrix*/
printf("Elements of upper triangular matrix are:\n");
for(i=0; i<row; i++)
  {
```

```
printf("\n");
for(j=0; j<row; j++)
{if(i>j) continue;
else
 printf("A[%d][%d]=%d\t", i,j, A[i][j]);
  }
 }
 }
```

Let us take a square matrix of size 4x4. However, we will enter all the values of the matrix. The program will print only the non-zero values along with the index. The result of the program is given below:

```
Enter number of rows of given matrix
4
Enter row number 1 of given matrix
1   2   3   4
Enter row number 2 of given matrix
0   5   6   7
Enter row number 3 of given matrix
0   0   8   9
Enter row number 4 of given matrix
0   0   0   10
Elements of upper triangular matrix are:
A[0][0]=1   A[0][1]=2   A[0][2]=3   A[0][3]=4
A[1][1]=5   A[1][2]=6   A[1][3]=7
A[2][2]=8   A[2][3]=9
A[3][3]=10
```

Lower Triangular Matrix

A square matrix in which all the elements above the leading diagonal are zero is called a lower triangular matrix. For instance, the following is a lower triangular matrix.

{10, 0, 0}

{60, 50, 0}

{40, 30, 70}

Notice that all the elements above the leading diagonal are zero. This can be mathematically expressed as A [i] [j] = 0 for i<j.

Multiplication of Matrices

Recall that a matrix multiplication is possible when the number of columns in the first matrix is equal to the number of rows in the second matrix. Let us understand the problem with an example.

Let matrix

```
              [10  20  30]
A [2] [3] =   [40  50  60]
                   [1   2]
B [3] [2] =        [3   4]
                   [5   6]
```

Number of columns in matrix A = 3

Number of columns in matrix B = 3

Hence, multiplication A x B is possible.

Let P be the product matrix. Let us use C notation for subscript, i.e., the first element will have a subscript of zero.

P [0] [0] = [10×1 + 20×3 + 30×5] = 220

P [0] [1] = [10×2 + 20×4 + 30×6] = 280

P [1] [0] = [40×1 + 50×3 + 60×5] = 490

P [1] [1] = [40×2 + 50×4 + 60×6] = 640

Thus, multiplication of 2x3 matrix by 3x2 matrix results in a 2x2 matrix as given below:

[220 280]

P [2] [2] = [490 640]

We can generalize the calculation of each element as given below:

$$P\,[i]\,[j]= \sum_{k=9}^{n-1} A(i,K)K\ B(K,j)$$

where n is the number of columns of the first matrix as well as the number of rows of the second matrix.

i is the number of rows of the first matrix and the product matrix

j is the number of columns of first matrix and the product matrix.

Let us confirm this through an example. For the sake of simplicity, the values are given in the program itself.

Example 3.26

```
/* To demonstrate matrix multiplication*/
#include<stdio.h>
int main()
{
int i,j,k;
int A[2][3]={
      {10, 20, 30},
      {40, 50, 60}
      };
int B[3][2]={
      {1, 2},
      {3, 4},
      {5, 6}
      };
int P[2][2];
 /*Matrix multiplication*/
   for(i=0; i<2; i++)
    for(j=0; j<2; j++)
    {
     P[i][j]=0; /*Initializing product matrix*/
          for(k=0; k<3; k++)
P[i][j]=P[i][j]+A[i][k]*B[k][j];/*calculating elements*/
```

```
        }
    /*Printing values of Product matrix*/
    printf("Elements of Product matrix are:\n");
    for(i=0; i<2; i++)
      for(j=0; j<2; j++)
      {
       printf("P[%d][%d]=%d\n", i,j, P[i][j]);
        }
       }
```

We initialize matrices A & B with actual values. Then, we declare a product matrix P [2] [2]. The dimension of the matrix requires a little explanation.

We have A [2] [3] &
B [3] [2]

Since the number of columns of the first matrix is the same as the number of rows of the second matrix, we can multiply them. The dimensions of the resultant matrix are as given below:

P [number of rows of first matrix] [number of columns of second matrix]

Assume A [3] [2] & B [2] [4]

The product matrix in this case will be P [3] [4]. This has to be understood clearly.

The number of rows of product matrix will be equal to the number of rows of the first matrix and number columns of the product matrix will be equal to the number of columns of the second matrix. Now, look at the section where we carry out matrix multiplication. We initialize the elements of the product matrix. Then each element value is calculated. We are using a nested *for* loops and inside them, another *for* loop. Finally, we print the values. The result of the program is given below:

Example 3.27

```
Elements of Product matrix are:
P[0][0]=220
P[0][1]=280
P[1][0]=490
P[1][1]=640
```

Let us now formulate an algorithm for multiplication of matrices. The algorithm shall be such that it shall be able to accept different sizes of matrices for multiplication. However, multiplication is possible only if the number of columns in the first matrix is equal to the number of columns in the second matrix. The algorithm is given below:

Algorithm for Multiplication of Matrices

Step 1: int I, j, k, row, colrow, column;
/*the first matrix subscripts are the first two and
the second matrix subscripts are the last two.
Colrow gives the number of columns of first matrix
and number of rows of second matrix */

```
Step 2:   int A[10] [10], B[10] [10], P[10] [10];
Step 3:   read row, colrow, column
Step 4:   /*getting first matrix row by row*/
          for (I=0; I<row; I++)
          for (j=0; j<colrow; j++)
          read A[I] [j]
          }
Step 5:   /*getting second matrix row by row*/
          for (I=0; I<colrow; I++)
          for (j=0; j<column; j++)
          read B[I] [j]
          }
Step 6:   /*Matrix multiplication*/
                for (I=0; I<row; I++)
                      for (j=0; j<column; j++)
                      {
          P[I] [j]=0; /*Initializing product matrix*/
          for (k=0; k<colrow; k++)
          P[I] [j] = P[I] [j] + A[I] [k] * B[k] [j];
          /*calculating elements*/
                }
          /*Printing values of Product matrix*/
Step 7:   printf ("Elements of Product matrix are :\n");
          for (I=0; I<row; I++)
                for (j=0; j<column; j++)
                print P[I] [j];

Step 8:   End
```

Let us now generalize the program to carry out matrix multiplication. The generalized program implementing the above algorithm for matrix multiplication is given below:

Example 3.28

```
/* Matrix multiplication generalized*/
#include<stdio.h>
int main()
{
int i,j,k;
int row, colrow, column;
```

/*the first matrix subscripts are first two and

the second matrix subscripts are the last two.

NOTES

colrow gives the number of columns of first matrix and

and number of rows of second matrix*/

int A[10][10], B[10][10], P[10][10];

printf("Enter number of rows of first matrix\n");

```
scanf("%d", &row);
printf("Enter number of columns of first matrix\n");
scanf("%d", &colrow);
printf("Enter number of columns of second matrix\n");
scanf("%d", &column);
/*getting first matrix row by row*/
for(i=0; i<row; i++)
{
printf("Enter row number %d of first matrix\n", i+1);
for(j=0; j<colrow; j++)
scanf("%d", &A[i][j]);
}

/*getting second matrix row by row*/
for(i=0; i<colrow; i++)
{
printf("Enter row number %d of second matrix\n", i+1);
for(j=0; j<column; j++)
scanf("%d", &B[i][j]);
}

/*Matrix multiplication*/
   for(i=0; i<row; i++)
     for(j=0; j<column; j++)
     {
   P[i][j]=0; /*Initializing product matrix*/
   for(k=0; k<colrow; k++)
 P[i][j]=P[i][j]+A[i][k]*B[k][j];/*calculating elements*/

      }
   /*Printing values of Product matrix*/

    printf("Elements of Product matrix are:\n");
    for(i=0; i<row; i++)
      for(j=0; j<column; j++)
      printf("P[%d][%d]=%d\n", i,j, P[i][j]);
      }
```

Look at the program. Since the number of columns of first matrix is same as the number of rows of the second, we call the number as colrow. The row refers to the number of rows in the first matrix and column refers to the number of columns in the second matrix. Thus, we are going to compute

P [row] [column] = A [row] [colrow] x
B [colrow] [column]

After declaration of the three variables, we also declare the three matrices. Since we have to give a constant as the dimension of the matrix, size of 10 has been given. Any large size can also be given. Then, we get the values for the variables row, colrow and column at runtime.

The next section gets the elements of the first matrix A row by row. Thereafter, we get the values of elements of the second matrix B. Then each element of the product matrix is computed. Then the elements of the product matrix are printed.

The interactions with the system and result of first execution follow:

Result of first execution

```
Enter number of rows of first matrix
2
Enter number of columns of first matrix
3
Enter number of columns of second matrix
2
Enter row number 1 of first matrix
10  20  30
Enter row number 2 of first matrix
40  50  60
Enter row number 1 of second matrix
1  2
Enter row number 2 of second matrix
3  4
Enter row number 3 of second matrix
5  6
Elements of Product matrix are:
P[0][0]=220
P[0][1]=280
P[1][0]=490
P[1][1]=640
```

The matrices multiplied are same as in the previous program. The result confirms that our generalized algorithm and the corresponding program work correctly. Now, let us try two different matrices for confirming the correctness of the program.

Result of second execution

```
Enter number of rows of first matrix
4
Enter number of columns of first matrix
3
Enter number of columns of second matrix
2
Enter row number 1 of first matrix
1  2  3
Enter row number 2 of first matrix
4  5  6
Enter row number 3 of first matrix
7  8  9
Enter row number 4 of first matrix
10  20  30
Enter row number 1 of second matrix
40  50
Enter row number 2 of second matrix
```

NOTES

```
60  70
Enter row number 3 of second matrix
80  90
Elements of Product matrix are:
P[0][0]=400
P[0][1]=460
P[1][0]=940
P[1][1]=1090
P[2][0]=1480
P[2][1]=1720
P[3][0]=4000
P[3][1]=4600
```

Let us confirm the correctness of the results manually.

```
        [1   2   3]
A     = [4   5   6]
        [7   8   9]
       [10  20  30]
         [40   50]
B     =  [60   70]
         [80   90]
```

P [0] [0]= [1×40 + 2×60 + 3×80]
= 400

P [0] [1] = [1×50 + 2×70 + 3×90]
= 460

P [3] [1] = [10×50 + 2×70 + 3×90]
= 4600

Thus, the generalized program for matrix multiplication works correctly.

Designated Initializers

The ANSI/ISO standard issued in the year 1990 permitted assignment of values to array elements directly as given in the following program.

Example 3.29

```
#include<stdio.h>
int main()
{
 int i;
 int m[5];
 m[0] = 10;
 m[4] = 20;
for(i=1;i<4;i++)
{
 m[i] = 0;
 }
 printf("the elements of the array array follow \n");
for(i=0;i<5;i++)
{
 printf("%d\n",m[i]);
}
}
```

In the above program, we assign the values for 0th element and 4th element directly as 10 and 20 respectively. The values of the other elements (1st, 2nd and 3rd) are assigned as zero in the for loop.

NOTES

The output of the program is given below:

```
the elements of the array array follow
10
0
0
0
20
```

In the next example, we show the program again but rather than assigning values to the first and last elements of the array directly as in the above program, we initialize them explicitly by subscript, using designated initializers.

Example 3.30

```
#include<stdio.h>
int main()
{ int i;
  int m[5]=
   { [0]=10, // initialize elements with designated
   initializers
    [4]=20
};
 printf("The elements of the array follow \n");
 for(i=0;i<5;i++)
 printf("%d\n",m[i]);
}
```

The result of the program follows

```
The elements of the array follow
10
0
0
0
20
```

Look at the above program carefully. We assign values to the array elements explicitly using designated initializers [0] and [4]. Note, there is no comma after the last assignment. The initialization is carried out within braces and the closing brace is followed by a semicolon. Any element which is not explicitly initialized is implicitly initialized to zero. This type of syntax was not allowed in earlier versions of C90.

The designated initializers are supported for user defined data types, such as arrays, structures, and unions. A designated initializer, or designator, points to a particular element to be initialized. A designator list is a comma-separated list of one or more designators. A designator list followed by an equal sign constitutes a designation.

Designated initializers allow for the following flexibility:

Elements within a user defined data type can be initialized in any order.

The initializer list can omit elements that are declared anywhere in the data type. Elements that are omitted are initialized as if they are static objects: arithmetic types are initialized to 0; pointers are initialized to NULL.

3.7 POINTERS IN C

A pointer is a variable that contains the address of another variable. As you know, any variable has the following four properties:

- Name
- Value
- Address
- Data type

For instance, consider the following declaration of a simple `integer`:

```
int var = 10;
```

The name here is `var` and its value is `10`. Its address is not declared here since you want to give flexibility to the compiler to store it wherever it wants. If you specify an address, then the compiler must store the value at the same address. Specifying actual address is carried out during machine language programming. However, this is not required in High Level Language (HLL) programming and by printing the value of `&var,` you can find out the address of the variable. When the statement to find the address is executed at different times, different addresses will be printed. What is important is that the compiler allocates an address at run time for each variable and retains this till program execution is completed. This is not strictly so in the case of `auto` variables. The compiler forgets the address of a variable when the program comes out of the block in which the variable is declared. At this point, you may also recall that in the case of function declarations in the calling function, the compiler does not allocate memory to the variables in the declaration. That is the reason why the parameters in the declaration part are not recognized in the calling function. It is only a prototype.

The fourth feature of a variable is its data type. In the above example, var is an integer. A pointer has all the four properties of variables. However, the data type of every pointer is always an integer because it is the value of the memory address. Memory addresses are integers. They cannot be floats or any other datatype. They may point to an integer or a float or a character or a function, etc. They have a name. They have a value. For instance, the following is a valid declaration of a pointer to an integer.

```
int * ip;
```

Here, `ip` is the name of a pointer. It points to or contains the address of an integer, which is the value. It will also be stored in another location in memory like any other variable. The pointer itself is an integer even though it is not declared as such.

Let us understand what happens inside the memory in each step. Assume that the variables `a`, `b` and `c` are allotted memory location addresses 100, 104, 108, respectively (we assume 4 bytes are given to an integer). It means `a` is located in 100 to 103, `b` is located in 104 to 107 and `c` is in 108 to 111. For convenience, we will mention only the 1st location for the corresponding variables as its address.

Program source content

```
int b=5, *a, c;
```

Three variables are created. Variable b is given value 5. Variable a is of pointer type.

```
a=&b
```

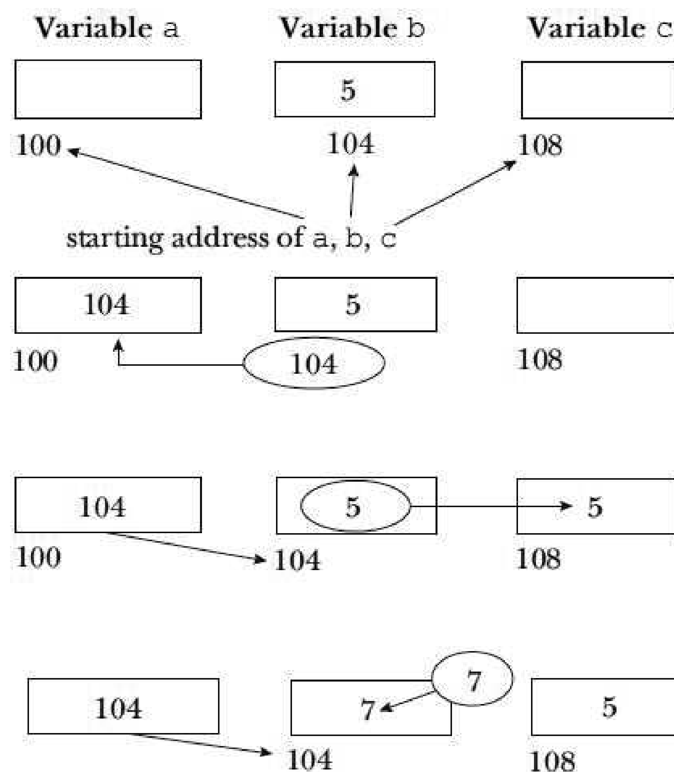
The address of variable b, 104 is assigned to variable a which is a pointer type.

```
c=*a
```

The content of variable a gives the location from where the value 5 is retrieved and assigned to variable c.

```
*a=7
```

Value 7 is put in the address which is the content of variable a.



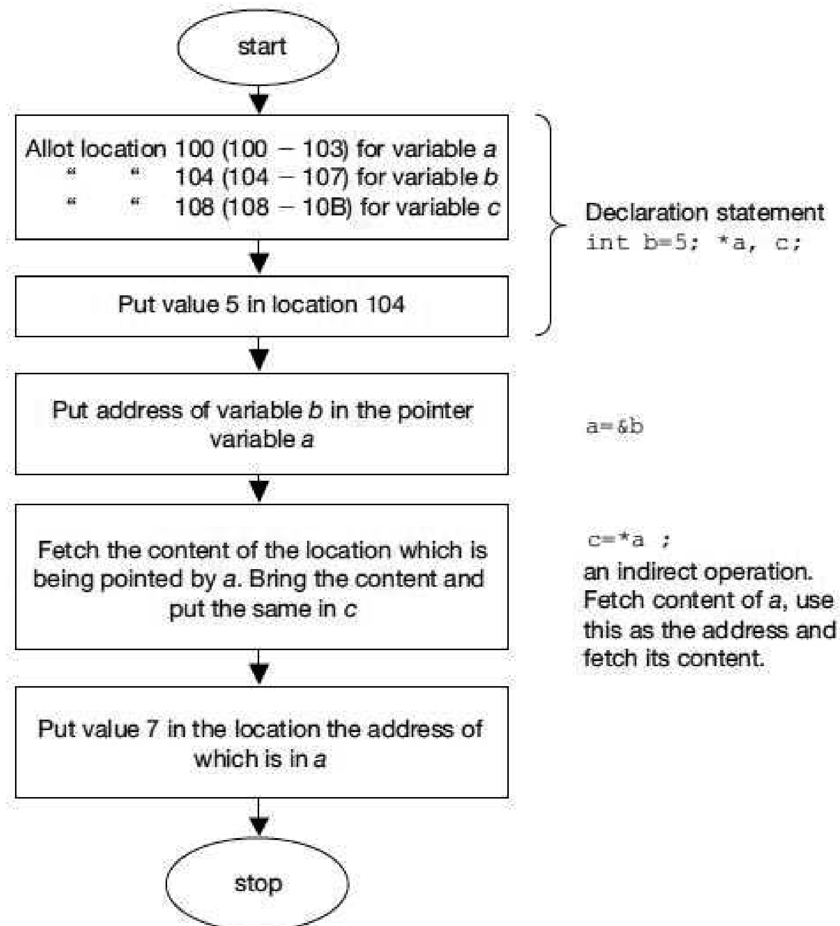
Take a note that *a may appear on both right and left sides of an equal sign in an expression. When *a is used on the right of an equal sign, the value kept in the location, the address of which is the content of the pointer variable a, is picked up and that value takes part in the expression evaluation. If *a appears on the left of an equal sign then the expression is evaluated and the result is stored in the address which is the content of a.

For example, c=*a; statement execution would first fetch the content of a which is assumed to be 104 in our example. It would then fetch the content of 104 because of the indirection operator with a. So, it will evaluate the right of the equal sign as 5. The value 5 will be assigned in the location for c that is 108 in our example.

For example, the statement execution *a=7; would first evaluate the right hand side of the expression, which is a constant integer 7. It would then go to the

address of a, which is 100, pick up its content, which is 104, go to 104 and deposit the right side evaluation result in it. So, 7 will be deposited in address 104, which is actually the address of b. In other words, b is indirectly given a value 7 by using pointer. This is the philosophy of using pointers in C. One or more levels of indirection can be brought into the operations.

The actions taking place internally (by the earlier code) may be shown by the following flowchart.



In the body of the program (other than declaration), the appearance of * and the pointer variable name together (*a) indicates an indirect operation. In indirect operation, the variable appearing with * at its left holds the address.

3.7.1 Pointer to void

A pointer to void can be declared as you declare any other pointer, such as pointer to integer, float, etc.

```
void* void_pointer;
```

Note that void is a keyword and it means 'nothing'. Note also that void_pointer also points to a memory address and hence, it is also an integer like any other pointer. The void_pointer points to nothing. Then what is the use of this pointer?

If you try to assign the address of an integer variable to a float pointer or any other type of pointer other than a pointer to an integer, then there will be an

error. This rule applies to the address of variables of others, such as double variable, char variable, etc. However, there is an exception to the rule and that is where the void pointer is useful. You can assign address of any data type to the pointer to void. For instance, you can declare as follows:

```
void* void_pointer;
int int_var;
void_pointer= & int_var;
```

The above assignment of address of an integer variable to the pointer to void is permissible. Similarly, you can assign a pointer of any type, such as float, double, etc., to a void pointer.

There are times when you write a function but do not know the data type of the returned value. When this is the case, you can use a void pointer. A void pointer is a special type of pointer that has flexibility of pointing to any data type. However, there is one limitation in the use of void pointers as compared to pointers of other types, namely direct dereferencing of void pointer is not permitted. The programmer must change the pointer to void as any other pointer type that points to a valid data type, such as int, char and float and then dereference it. Then, conversion of the pointer to some other valid data type is achieved by using the concept of type casting.

3.7.2 NULL Pointer

The concept of NULL pointer is different from the above concept of void pointer. NULL pointer is a type of pointer of any data type and generally takes a value as zero. This is, however, not mandatory. This denotes that a NULL pointer does not point to any valid memory address.

For example:

```
int* var;
var =0;
```

The above statement denotes var as an integer pointer type that does not point to a valid memory address. This shows that var has a NULL pointer value.

The difference between void pointers and NULL pointers is as follows:

A void pointer is a special type of pointer of void and denotes that it can point to any datatype. NULL pointers can take any pointer type but do not point to any valid reference or memory address. It is important to note that a NULL pointer is different from a pointer that is not initialized.

3.7.3 Declaration of Pointer Variables

Take a look at some examples involving pointers:

```
int dat = 100;
int * var;
var = &dat;
```

Here, dat is an integer variable. Its value is 100; its name is dat; it will be stored in memory in a location with an address.

The next declaration means that var is a pointer to an integer and is a variable. It is an integer and will be stored at a location in memory with an address. The value of var is the address of the integer variable it points to. We do not know as yet which integer it points to. It can however be made to point to any integer we like, by a proper declaration.

Now, look at the next assignment. The variable var is assigned the value of &dat. This means var has the same value as the address of dat. By taking into consideration the previous statements, we can conclude that var is a pointer and it points to dat.

Now, if you specify dat or * var, they point to the same value 100. Similarly, if you specify &dat or var it is the address or to be precise, the starting address of dat or * var.

```
int * var;
* var = 100;
```

The above statements declare var as a pointer to an integer and later propose to assign 100 to the integer variable. We do not know or do not want to make public the name of the variable. However, we can always access the variable as * var. This works well in Borland C++ compilers but could lead to run time errors in other compilers.

Both the statements cannot be combined into one as int * var = 100. This will be flagged as an error even in the Borland C++ compiler. Therefore, it is not possible to combine both the declaration and assignment as a pointer variable and integer constant are concerned. It would be safer to make var point to another variable as given in Example 3.30.

```
int * var;
int a = 100;
var = &a;
```

What will be the value of the output of the following statements after the execution of the following statements?

```
printf ("%d", * var);
printf ("%d", (* var) ++);
printf ("%d", * var);
printf ("%d", var);
```

You can easily guess that the first printf will give the value of * var as 100.

What is the significance of the parentheses and the increment operation, in the second statement? As the bracket or parentheses has precedence over other operators, the value of * var will be printed as 100. After printing, it will be incremented as 101 Because the increment is postfix, the value of * var after execution of the statement will be 101. The next statement will confirm this when it prints 101.

The fourth printf case printed the address of var. It printed 1192 when I executed the program. You are unlikely to get the same address on execution. The

location where the variable is stored will not vary till the execution of the program is completed. If you try the program again, you will get a different address. Don't worry. It does not affect our work. However, you may note down the value and substitute it for the values mentioned here for understanding the concept of pointers.

Remember to enclose the pointer variable within parenthesis as given in the example. The `postfix` of the increment operator, enables increment of the variable after printing.

After execution of the above four `printf` statements, you execute the following statements:

```
printf ("%d", * var ++);
printf ("%d", * case);
```

What happens? The value of `* var`, i.e. 101 is the first to be printed out and then `var` will be incremented, i.e., instead of incrementing the value as desired, the address is incremented and therefore, `var` now points to the next location. Remember `var` was pointing to 1192. Will it go to 1193? No. Since `var` is an `integer`, 1192 and 1193 (2 bytes) are already used. Hence, `var` now points to 1194. The next statement prints the value of `* var`. You had stored 101 in location 1192. You do not know what is contained in 1194. Hence, the next `printf` will print garbage value. Note carefully what happened. You wanted to increment `* var` in the first statement of example 4. However, the compiler has assumed that you wanted to increment thc address, which underlines the importance of parenthesis. Also note that whenever you use a `postfix` notation, the `postfix` operation is effective only after execution of the statement.

Is everything lost now? Can you not go back to address 1192? Yes, you can as the following indicates. Note that the following statements are executed in continuation of all the above statements.

```
printf ("%d", var) ;
printf ("%d", - - var) ;
printf ("%d", * var) ;
```

The first statement in this example prints the address of the location in memory pointed to by `var`. As expected, the pointer is at location 1194. The second statement carries out two operations in the sequence show:

(a) Decrement the Pointer
(b) Print the New Address

Decrementing takes place before printing because it is a `prefix` operator. Now, it prints 1192; the original address is restored. Now the third statement prints the value of `* var` or the value stored in location 1192, i.e., 101.

Note that `prefix` carries out the increment or decrement before printing or any desired operation, whereas `postfix` does that after printing. Note that the fundamentals of pointers are being discussed and that they should be understood clearly before you proceed further.

You now have 101 stored in address 1192 and pointed to by `var`. Let us see what happens on execution of the following statements, in continuation.

NOTES

```
printf ("%d", ++ (* var)) ;
printf ("%d", var) ;
```

These statements are perfectly correct. `* var` is incremented and the new value printed in the first statement. Therefore, 102 will be printed. Has the address been changed? No. Hence, the second statement will print the address as 1192.

Can you increment and decrement addresses? Yes, as the following indicates:

```
printf ("%d", var ++);
printf ("%d", var) ;
```

What will be printed in the first statement above, 1192, or 1194? It will be 1192 because incrementing `var` will take place after the first printf. Obviously, the second statement will print the address incremented after the previous printf viz. 1194. Let us not lose track but get back to the old address and try prefixing increment / decrement operators to the address.

```
var --;                     (a)
printf ("%d", var) ;        (b)
printf ("% d", ++ var) ;    (c)
printf ("% d", -- var) ;          (d)
printf ("% d", var) ;       (e)
printf ("% d", * var)       (f)
```

Before execution of the first statement in this example, you have:

`var` = 1194

location 1192 contains 102.

Now, analyse the execution statement wise:

(a) decrements `var` to 1192
(b) confirms that `var` is 1192 indeed
(c) ++ `var`, increments `var` and then prints as 1194
(d) –– `var` decrements `var` and then prints as 1192
(e) confirms `var` is 1192
(f) The value in `var` is 102
Now, the concepts are becoming clearer. Let us carry out one more example. Assume that all the above statements have been executed and the following are now executed:

```
printf ("%d", * (var ++)) ;          (g)
printf ("%d", * (-- var)) ;          (h)
printf ("%d", var) ;                 (i)
printf ("%d", * var) ;               (j)
```

(g) Here, `* var` is printed as 102 because of the `postfix` operator. Then, `var,` i.e. the address is incremented to 1194.
(h) Here, because of the `prefix`, `var` is decremented to 1192 and then, the value at 1192, i.e., 102 is printed.

The next 2 statements confirm this.

Now you would be familiar with the intricacies of pointers, `prefix`, `suffix` and parenthesis.

NOTES

For your convenience, the program involving all these statements and the output is given below:

```
/*Example 3.31 */
/* pointers*/
#include <stdio.h>
int main()
{
    int * var;
    int a =100;
    var = &a;
    printf("value of * var=%d\n", *var);
    printf("value of (*var)++=%d\n", (*var)++);
    printf("value of * var=%d\n", *var);
    printf("address var=%d\n", var);
    printf("value of *var++=%d\n", *var++);
    printf("value of * var=%d\n", *var);
    printf("address var=%d\n", var);
    printf("original address var again=%d\n", —var);/
*original address restored*/
    printf("value of * var=%d\n", *var);
    printf("value of ++(*var)=%d\n", ++(*var));
    printf("address var=%d\n", var);
    printf("address var++=%d\n", var++);
    printf("address var=%d\n", var);
    var—;
    printf("address var after decrementing=%d\n", var);
    printf("address ++var=%d\n", ++var);
    printf("address --var=%d\n", --var);
    printf("address var=%d\n", var);
    printf("value of * var=%d\n", *var);
    printf("value of *(var++)=%d\n", *(var++));
    printf("value of *(--var) = %d\n", *(--var));
    printf("address var=%d\n", var);
    printf("value of * var=%d\n", *var);
}
```

Result of the program

```
value of * var=100
value of (*var)++=100
value of * var=101
address var=9106
value of *var++=101
value of * var=9108
address var=9108
```

NOTES

```
original address var again=9106
value of * var=101
value of ++(*var)=102
address var=9106
address var++=9106
address var=9108
address var after decrementing=9106
address ++var=9108
address --var=9106
address var=9106
value of * var=102
value of * (var++)=102
value of *(--var) = 102
address var=9106
value of * var=102
```

In the declaration part, you have declared `div` as a function passing two pointer variables and getting back `void` or none. You call `div(&a, &b)`. You do not pass the values but reference to the values. You actually pass the address of `a` and `b` to function `div`.

The function `div` receives the reference, i.e., addresses of `a` and `b`.

```
px = &a;
py = &b;
```

`px` points to `a` and `py` points to `b`. Hence, *`px` gets the value of `a` and *`py` gets the value of `b`. Now, the contents of *`px` and *`py`, i.e., `a` and `b` are copied to `temp1` and `temp2`.

You divide `temp1` by `temp2` and place the result in variable *`px` whose address is known to both the main and the function `div`. In the main function, the address corresponds to `a` and in the function it corresponds to *`px`. Therefore, the address of the quotient is returned to the `main` function indirectly. Similarly, *`py` contains the remainder. It will be stored in b through the reference. Thus, the values of the quotient and remainder are stored in locations `&a` and `&b` and you have indirectly returned 2 values to the main through call by reference. The concept, though, may seem hazy at this point, will become clear as you see more examples.

The function declaration indicates that there is a function `div`, which returns nothing. It passes two pointers to integers. The pointers are declared as `int*p` and `int*q`. You will notice that they are not declared in the `main` function and that, therefore, `p` and `q` have no significance except to indicate that they are pointers to integers. They also indicate that the addresses of two integers are to be passed while calling the function.

3.7.4 Accessing the Address of a Variable

The function call using pointers is known as call by reference. Here, the address of the variable is passed. Note that calling by reference has to be indicated in the function declaration, such as these:

```
fun (int *p, char *cp, float *fp, int *array);
```

This is an indication that the function is to be called by reference for those parameters which are pointers. A mixed declaration could also be used as given below:

```
fun1(int a, char *cp);
```

Here, you are indicating that an integer is passed by value and a character variable is passed by reference. In the above example, while you can either pass a character array (string) or a character through the second declaration, you can only pass one `integer` variable through the first parameter.

The function declarator above the function body has to match the declaration. Hence, when `fun1` is called, an integer followed by an address of character will be passed. In the function `fun1`, you may have a declarator as follows:

```
fun1(int d, char * ch)
```

Here, the value of the integer variable will automatically get assigned to `d` and the `ch` will be assigned the address of the character variable. This means both `ch` and the address of the character variable in the calling function will point to the same location. Thus, both in the calling function and in the called function, the variable is accessible, although under different names. Any modification made to the character variable either in the calling function or the called function affects both.

While calling by reference, you have to pass the address of the variable, if the variable is declared by value, such as `int a`, then you have to pass `&a`. If it was declared as a pointer to say, an integer, such as `int * ip`, then `ip` has to be passed.

Return by Reference

So far, we have seen functions returning only a value, irrespective of whether they were called by value or by reference. Functions can also return references or pointers.

Whenever a function is to return a pointer, this has to be indicated by the following:

Function Declaration

Declare the function as returning pointers. For instance,

```
int * fun1() ;
char * fun2() ;
float * fun3 (int a, float * b, char * c) ;
```

The difference is the insertion of pointer(`*`) between the return data type and the function name.

Function Declarator

This will be in the same format as the prototype or function declaration. For instance,

```
float * fun3 (int a, float * b, char * c);
```

to match the third function declaration in the above example.

NOTES

Function Call

You may call,

```
fun3 (x, &y, z) ;
```

Here, x is a variable and &y is an address. Although z looks like a value, it is in fact, a reference to character since prototype has the declaration char * c.

Return Statements

The program will obviously return an address or pointer.

3.8 POINTER TO ARRAY

At this point, you must note another way of representing the elements of the array.

The address of the 0th element is stored at location p2 or address p2+0. The element with a subscript 1 of the array will be at a location one above or at p2+1. Thus, the address of the nth element of this array *p2 will be at address p2+n. If you know the value of the pointer variable, i.e. the address of pointer, then it would be easy to express its value. The value of the integer stored at the nth location can be represented as * (p2+n) just as the value at address (p2+0) is * (p2 + 0) or * p2. This notation is, therefore, quite handy.

Pointer to Array

If you now declare:

```
int a [ 5 ];
nt * ip;
ip = &a[0];
```

You have defined an array of integer a with 5 elements. When you assign the address of a[0], i.e., the 0th element of a to ip, ip will point to the array. The system will automatically assign addresses for the other elements in the array by noting the datatype of the array and size occupied by each element.

The example below also explains the concept of a function returning a pointer.

```
/* Example 3.32 to find the greatest number in an array*/
#include <stdio.h>
int main()
{
      int array[]= {8, 45, 5, 131, 2};
      int size=5, * max;
      int* fung(int *p1, int size);/*function retuns pointer
to int*/
      max=fung(&array[0], size); /*max is a pointer*/
      printf("%d\n", *max);
}
int * fung(int *p2, int size)
{
```

NOTES

```
        int i, j, maxp=0;
        for (j=0; j<size; j++)
        {
           if (*(p2+j) > maxp)
           {
                 maxp=*(p2+j);
                 i=j;
           }
        }
        return (p2+i); /*pointer returned*/
    }
```

Result of the program

```
131
```

The example above involved arrays and pointers. The usage of a pointer made the passing of an array to a function, a simple task. You define array as an `integer`. However, in order to pass an address, the prototype was defined with a pointer argument `int * p1`. This means an address will be passed to the function while calling it. No distinction was made between a simple `integer` variable and an `integer` array. `int * p1` can be a single valued `integer` or an array. This is possible because arrays are stored contiguously in memory. If the address of the 0th element is known and the datatype is known, it would not be difficult to calculate the address of any element in the array. Therefore, it is enough if the address of the 0th element is passed to a function. *`p1` refers to the address of a variable or to the 0th element of an array. Note the flexibility of arrays in 'C' and how intelligently the language uses pointers.

The called function returns the address of the greatest number in the array. Look at the function declaration. The function returning a pointer is indicated by the following (a star mark between the return datatype and function name):

```
int * fung(...)
```

The address of `array [0]` is received by `fung()` and stored in *`p2`. Or, in other words, `p2` points to the 0th location of the array.

While calling a function, the address has to be passed and this is achieved in the above example by passing `& array[0]`.

In the called program, *`p2` is treated as an array without any additional efforts. By adding the index to `p2,` you get the address of the various elements in the array. Getting value is achieved by placing `*` before the address.

The `if` statement compares `maxp` with `*(p2 + j)` or `p2 [j]`

or `array [j]`. You will easily understand the logic as to how we get the maximum or the greatest number in `* (p2 + j)`. At the end of the iterations, `* (p2 + j)` which is stored at location `p2 + j` contains the maximum value in the array and therefore, we are returning an address or reference to the called function. In this example, `(p2 + 3)` is the address of the greatest number in the array. After `return` from the function, `max` gets the value of `p2 + j`. In the `printf`

*max which is the value stored in max is printed. We have already defined max as a pointer to an integer in the main function. Thus, the function fung returns the address of a value or a pointer and by returning the address; the value is retrieved automatically by the main function.

Now, look at another example using arrays and pointers, as given below:

```
/*Example 3.33 passing an array of integers to function - method2*/
  #include<stdio.h>
  void main()
  {
      int array[]={10, 20, 30, 40, 50};
      int *a;
      void pass(int *a, int k);
      a=&array[0];
      pass(a, 4);
  }
  void pass(int *b, int j)
  {
      int k=0;
      while (k <= j)
      {
         (*b)=(*b)/2;
         printf("value %d @ address %d\n", *b, b);
         k++;
         b++;
      }
  }
```

Result of the program

```
value 5 @ address 8694
value 10 @ address 8696
value 15 @ address 8698
value 20 @ address 8700
value 25 @ address 8702
```

Here, * a has been declared as a pointer to integer. The variable a has been assigned the address of the 0th element of the array. Now, the function call is made using pass (a, 4); again a is the address of the variable array [0].

In the function pass, b received the address of a and the next element of the array is accessed each time by incrementing the address b. Note that the value of elements in the array are divided by 2 in the called function. As you become familiar with pointers, you can write programs very easily.

3.8.1 Multi Dimensional Arrays

You may represent two dimensional array conventionally as a[i][j].

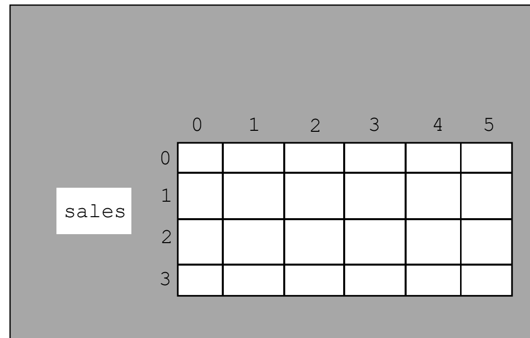
The elements of a two dimensional array of size 3×3 can be represented symbolically as follows:

NOTES

```
a00   a01      a02
a10   a11      a12
a20   a21      a22
```

You can, then visualize this as an array of arrays. Do not get puzzled. Each row is an array and you have three such arrays. Thus, a two dimensional array can be considered as an array of one dimensional arrays. Therefore, you can expect the following type of memory allocation by the system:

```
 0   1   2   0   1   2   0   1   2
a[0][0]    a[1][0]     a[2][0]
```



	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

This means all the elements of the 0th row are stored contiguously. a[0][0] is stored first, followed by all elements of the 0th row. Next, the first row starts with the first element a[1][0]. At the end of the first row, the second row starts with a[2][0]. Try to visualize how the elements of a two dimensional array are stored contiguously.

You know that

a[i] = * (a+i)

where a + i is the address of the ith element.

You also know that

a [0] = *a and its address is stored in location a.

You can now transform a two dimensional array into this convenient form.

You can visualize **b[0], **b[1] and **b[2] to represent the values of row one elements which are stored at locations *b[0], *b[1] and *b[2] as given below:

```
             0 1 2 0 1 2 0 1 2
Address   *b[0] *b[1] *b[2]
```

** indicates pointer to pointer. This is acceptable since the row pointers in turn point to column pointers. If you assume the row one elements as pointers, then the addresses of elements of the two dimensional arrays can be addressed by simple arithmetic.

NOTES

`*b` points to the value in the one dimensional array. Then, `**b` refers to the value in the two dimensional array. All the elements of the two dimensional array are stored contiguously. From the storage pattern we declare the following:

`**b` refers to the value at the 0th row of a two dimensional array. Naturally, the address of the 0th row will be `*b`.

`**(b+1)` refers to the value of the 0th element in the 1st row. Its address will be `*(b+1)`

In general, the 0th element in ith row contains value `**(b+i)` and its address is `*(b+i)`.

You have only talked about the 0th element in each row, i.e., where the row starts conceptually. How do you address the other elements? You know that an element `b[i][j]` is the jth element in the ith row.

You know the address of the 0th element. It is `* (b+i)`. Therefore, the address of the jth element in ith row will be `(* (b+i) + j)`. Note `j` is the offset. Therefore, the value at this location can be expressed as `*(*(b+i)+j)`. You have just added a star outside the address.

Now the address of the 2nd element in the 2nd row will be,

`*(b+2+2),`

Its value will be,

`*(*(b+2)+2)`

What will be the address of the 0th element in the second row?

`(*(b + 2 ))`

What will the value be?

*(* (b + 2))

Note the parentheses and `*`. To understand this clearly, execute the following program.

```
/*Example 3.34 to understand pointers to 2 dimensional array*/
  #include<stdio.h>
  int main()
  {
  int i,j;
  int a[3][3];
      printf("Enter the values of 3x3 matrix\n");
      for (i=0; i<=2; i++)
      for(j=0; j<=2; j++)
         scanf("%d", (*(a+i)+j));
         for (i=0; i<=2; i++)
         {
            for (j=0; j<=2; j++)
            {
               printf("address=%d\n", (*(a+i)+j));
               printf("value=%d\n", *(*(a+i)+j));
```

```
            }
        }
}
```

Result of the program

```
Enter the values of 3x3 matrix
00 01 02
10 11 12
20 21 22
address=9098
value=0
address=9100
value=1
address=9102
value=2
address=9104
value=10
address=9106
value=11
address=9108
value=12
address=9110
value=20
address=9112
value=21
address=9114
value=22
```

This example helps us to understand how a two dimensional array is stored in memory. The array has been declared in the normal way without pointers but the array elements have been received, stored and printed using pointer notation. When you execute the above program, you can type the numbers in any one of the ways given below:

- Enter one integer at a time followed by Return keystroke.
- Enter all integers in a row with spaces between them and finally hit Return keystroke.
- Enter three integers in a row as the result of the program indicates.

Dynamic Allocation of Memory

The dimension or array size declaration of an array is an important subject. Array dimension is to be declared before compiling. For instance, some valid declarations are:

```
char name[25];
int mark[40];
```

You have not come across any problem since you were initializing the arrays and hence, the array size was known. If you were to get the array elements at

runtime, sometimes you could give either a lesser number of elements or more elements. In the former case garbage values will be stored in the empty spaces in the array misleading the user. In the latter case, the elements will be lost. To avoid this problem, you may think that you can specify `marks [n]` and give the value of `n` later at runtime. However, this will not work and the compiler will force you to give the actual dimension. Hence, dynamic memory allocation is useful. `malloc` and **`calloc`** serves to specify the actual dimension at runtime and hence, enable memory allocation dynamically. Next time when you execute the program, you can specify any value for `n`. It will definitely work.

The functions `malloc()` and `calloc()` allocate memory dynamically. The specifications are as follows:

The statement

```
void * malloc (n*size n );
```

returns the pointer to `n` bytes of memory or `NULL` if allotment is not possible. Since it returns a pointer, you have to specify the array as a pointer variable as shown:

```
int *b ; / * array declared * /
b = (int *) malloc (x * 2) ; / * x is the array size * /
```

Similarly, the `calloc` is defined as `void * calloc (sizen, size_size)`. Here, the number of arguments are two. The first one is the array size and the second one is the bytes occupied per element, i.e., the storage space required for the datatype. The function returns a pointer allocating space for an array of `size sizen`, of datatype `size` but the array contents will be initialized to zero. In `malloc`, the initial contents will be garbage values. We can use it as follows:

```
int * b;
b = (int *) calloc (x, 4);
```

Here, `4` indicates the array of type `float` and space is allocated to store `x floats`. When you use `calloc` or `malloc`, you must include `alloc.h`.

Program using Dynamic Memory Allocation

A program to demonstrate `malloc` and `calloc` is given below:

```
/* Example 3.35 to do string concatenation by using dynamic allocation
of memory*/
#include <stdio.h>
#include <alloc.h>
#include <string.h>
main ()
{
    char *newstrcat(char *dest, char *src);
    char *name1, *name2;
    int n1, n2;
    printf("Enter size of 2 names:\n");
    scanf("%d%d", &n1, &n2);
    name1 = (char *) calloc(n1, 1);
```

```
    name2 = (char *) calloc(n2, 1);
    printf("Enter 2 names\n");
    scanf("%s%s", name1, name2);
    printf("New name:%s\n",newstrcat(name1,name2));
}
char *newstrcat(char *dest, char *src)
{
    char *w;
    int i,len,len1,cnt=0;
    len=strlen(dest);
    len1=strlen(src);
    w=(char *)malloc(len+len1+1);
    for(i=0;i<len;i++)
       w[cnt++]=dest[i];
    for(i=0;i<len1;i++)
       w[cnt++]=src[i];
       w[cnt]=0;
    return(w);
}
```

Result of the program

```
Enter size of 2 names:
4 4
Enter 2 names
Rama samy
New name:Ramasamy
```

`calloc` is used to allot memory to `name1` and `name2`. The function `newstrcat` is called while printing. In the called function `malloc` is used to allot space equal to the size of `name1`, `name2 +1`. The additional space is for placing `NULL`. The string is concatenated by copying one character at a time using two `for` statements. Finally, the concatenated string is returned to the `main` function where it is printed.

The memory allocated using `malloc`, `calloc` can also be freed, when it is no longer necessary by using the function `free()`. For instance, in the `main()` of the above example, after the last statement we can add the following statements;

```
free(name1);
free(name2);
```

These statements will deallocate the memory space allocated to them after the job of printing the concatenated string is over. Thus, dynamic allocation of memory as per the exact need and freeing it after use is quite useful for conserving memory. This concept is used by professional programmers when they develop commercial software products. Memory saved is more than money saved in such product development.

NOTES

3.9 POINTER AND FUNCTION

The function call using pointers is known as 'call by reference'. Here, the address of the variable is passed. Note that calling by reference has to be indicated in the function declaration such as these:

fun (int *p, char *cp, float *fp, int *array);

This is an indication that the function is to be called by reference for those parameters which are pointers. A mixed declaration could also be used as given below:

fun1(int a, char *cp);

Here, we are indicating that an integer is passed by value and a character variable is passed by reference. In the above example, while we can either pass a character array (string) or a character through the second declaration, we can only pass one integer variable through the first parameter.

The function declarator above the function body has to match the declaration. Hence when fun1 is called, an integer followed by an address of character will be passed. In the function fun1 we may have a declarator as follows:

fun1(int d, char * ch)

Here, the value of the integer variable will be automatically get assigned *d*, and the *ch* will be assigned the address of the character variable. This means both *ch* and the address of the character variable in the calling function will point to the same location. Thus, both in the calling function and in the called function, the variable is accessible, although under different names. Any modification made to the character variable either in the calling function or the called function affects both.

While calling by reference, we have to pass the address of the variable. If the variable is declared by value such as int *a*, then we have to pass *&a*. If it was declared as a pointer to say, an integer such as int * *ip*, then *ip* has to be passed.

Function 'Return by Reference'

So far we have seen functions returning only a value, irrespective of whether they were called by value or by reference. Functions can also return reference or pointers.

Whenever a function is to return a pointer, this has to be indicated by the following:

Function Declaration

Declare the function as returning pointers. For instance,

int * fun1() ;

char * fun2() ;

float * fun3 (int a, float * b, char * c) ;

The difference is the insertion of pointer(*) between the return data type and the function name.

Function Declarator

This will be in the same format as the prototype or function declaration. For instance,

float * fun3 (int a, float * b, char * c)

to match the third function declaration in above example.

Function Call

We may call

fun3 (x, &y, z);

Here, *x* is a variable and &y is an address. Although *z* looks like a value, it is in fact a reference to character, since prototype has the declaration char * c.

Return Statements

The program will obviously return an address or pointer.

Pointer and one dimensional Array

The example below explains the concept of a function returning a pointer.

Example 3.36 To find the greatest number in an array.

```
#include <stdio.h>
int main()
{
   int array[]={8, 45, 5, 131, 2};
   int size=5, *max;
   int* fung(int *p1, int size);/*function retuns pointer to
      int*/
   max=fung(&array[0], size); /*max is a pointer*/
   printf("%d\n", *max);
}
int * fung(int *p2, int size)
{
   int i, j, maxp=0;
   for (j=0; j<size; j++)
   {
      if (*(p2+j) > maxp)
      {
         maxp=*(p2+j);
         i=j;
      }
   }
   return (p2+i); /*pointer returned*/
}
```

Result of program

```
131
```

The called function returns the address of the greatest number in the array. Look at the function declaration. The function returning a pointer is indicated by the following (a starmark between the return data type and function name).

int * fung(...)

The address of array [0] is received by fung() and stored in **p2*. In other words, *p2* points to the 0th location of the array.

Pointer Notations for Arrays

At this point we must note another way of representing the elements of the array.

The address of the 0th element is stored at the location *p2* or address p2 +0. The element with a subscript 1 of the array will be at one location above or at *p*2 + 1. Thus the address of the nth element of this array **p2* will be at address *p2* + *n*. If we know the value of the pointer variable, i.e., the address of the pointer, then it would be easy to express its value. The value of the integer stored at the nth location can be represented as * (p2 + n) just as the value at address (p2+0) is *(p2 + 0) or * p2. This notation is, therefore, quite handy.

The *if* statement compares *maxp* with **(p2 + j)* or *p2 [j]* or *array [j]*. You will easily understand the logic of getting the maximum or greatest number in * *(p2 + j)*. At the end of the iterations, * *(p2 + j)* which is stored at location p2 + j contains the maximum value in the array, and therefore we are returning an address or reference to the called function. In this example (p2 + 3) is the address of the greatest number in the array. After return from the function, *max* gets the value of p2 + j. In the printf **max* which is the value stored in *max* is printed. We have already defined *max* as a pointer to an integer in the main function. Thus, the function *fung* returns the address of a value or a pointer, and by returning the address, the value is retrieved automatically by the main function.

Arrays and Pointers

Example 3.36 involved arrays and pointers. Use of a pointer made the passing of an array to a function a simple task. We define array as an integer. However, in order to pass an address, the prototype was defined with a pointer argument int * *p1*. This means an address will be passed to the function while calling it. No distinction was made between a simple integer variable and an integer array. int * *p1* can be a single valued integer or an array. This is possible because arrays are stored contiguously in memory. If the address of the 0th element is known, and the data type is known, it would not be difficult to calculate the address of any element in the array. Therefore, it is enough if the address of the 0th element is passed to a function. **p1* refers to the address of a variable or to the 0th element of an array. Note the flexibility of arrays in 'C' and how intelligently the language uses pointers.

While calling, a function, the address has to be passed, and this is achieved in the above example by passing & array[0].

In the called program **p2* is treated as an array without any additional efforts. By adding the index to *p2* we get the address of the various elements in the array. Getting value is achieved by placing * before the address. Let us look at another example using arrays and pointers, as given below:

Example 3.37 Passing an array of integers to function.

```
*method2*/
#include<stdio.h>
int main()
{
   int array[]={10, 20, 30, 40, 50};
   int *a;
   void pass(int *a, int k);
   a=&array[0];
```

NOTES

```
    pass(a, 4);
}
void pass(int *b, int j)
{
    int k=0;
    while (k <= j)
    {
        (*b)=(*b)/2;
        printf("value %d @ address %d\n", *b, b);
        k++;
        b++;
    }
}
```

Result of program

```
value 5 @ address 8694
value 10 @ address 8696
value 15 @ address 8698
value 20 @ address 8700
value 25 @ address 8702
```

Here, * *a* has been declared as a pointer to integer. The variable *a* has been assigned the address of the 0th element of the array. Now the function call is made using pass (a, 4); Again a is the address of the variable array [0].

In the function pass, b received the address of *a*, and the next element of the array is accessed each time by incrementing the address *b*. Note that the values of the elements in the array are divided by 2 in the called function. As we become familiar with pointers we can write programs very easily.

Pointer Notation for Multi-dimensional Arrays

We may represent a two-dimensional array conventionally as a[i][j].

The elements of a two-dimensional array of size 3x3 can be represented symbolically as given below:

a00	a01	a02
a10	a11	a12
a20	a21	a22

We can then visualize this as an array of arrays. Do not get puzzled. Each row is an array, and we have three such arrays. Thus, a two-dimensional array can be considered as an array of one-dimensional arrays. Therefore, we can expect the following type of memory allocation by the system:

0	1	2	0	1	2	0	1	2
a[0][0]			a[1][0]			a[2][0]		

This means all the elements of the 0th row are stored contiguously. a[0][0] is stored first, followed by all elements of the 0th row. Next, the first row starts with the first element a[1][0]. At the end of the first row, the second row starts with a[2][0]. Try to visualize how the elements of a two-dimensional array are stored contiguously.

NOTES

You know that

a[i] = * (air)

where a + i is the address of the element.

You also know that

a [0] = * a and its address is stored in location a.

You can now transform a two dimensional array into this convenient form.

You can visualize **b[0], **b[1] and **b[2] to represent the values of row first elements which are stored at locations *b[0], *b[1] and *b[2] as given below:

	0	1	2	0	1	2	0	1	2
Address	*b[0]			*b[1]			*b[2]		

** indicates pointer to pointer. This is acceptable since the row pointers in turn point to column pointers. If you assume the first row elements as pointers, then the addresses of elements of the two-dimensional arrays can be addressed by simple arithmetic.

**b* points to the value in the one-dimensional array. Then ***b* refers to the value in the two-dimensional array. All elements of the two dimensional array are stored contiguously. From the storage pattern we declare the following:

**b refers to the value at the 0th row of a two-dimensional array. Naturally, the address of the 0th row will be *b.

**(b+1) refers to the value of the 0th element in the 1st row. Its address will be *(b+1)

In general the 0th element in ith row contains the value **(b+i) and its address is *(b+i).

We have only talked about the 0th element in each row, i.e., where the row starts conceptually. How do we address the other elements? You know that an element b[i][j] is the jth element in the ith row,

We know the address of the 0th element. It is * (b+i). Therefore, the address of the jth element in ith row will be (* (b+i) + j). Note, j is the offset. Therefore, the value at this location can be expressed as * (* (b + i) + j). We have just added a star outside the address.

Now the address of the 2nd element in the 2nd row will be

(*(b + 2) + 2)

Its value will be

*(* (b + 2) + 2)

What will be the address of the 0th element in the second row

(*(b + 2))

What will the value be?

*(* (b + 2))

NOTES

Note the parentheses and *. To understand this clearly, execute the following program.

Example 3.38 To understand pointers to 2 dimensional array.

```
#include<stdio.h>
int main()
{
int i,j;
int a[3][3];
   printf("Enter the values of 3x3 matrix\n");
   for (i=0; i<=2; i++)
   for(j=0; j<=2; j++)
      scanf("%d", (*(a+i)+j));
      for (i=0; i<=2; i++)
      {
         for (j=0; j<=2; j++)
         {
            printf("address=%d\n", (*(a+i)+j));
            printf("value=%d\n", *(*(a+i)+j));
         }
      }
}
```

Result of program

```
Enter the values of 3x3 matrix
00  01  02
10  11  12
20  21  22
address=9098
value=0
address=9100
value=1
address=9102
value=2
address=9104
value=10
address=9106
value=11
address=9108
value=12
address=9110
value=20
address=9112
value=21
address=9114
value=22
```

This example helps us to understand how a two-dimensional array is stored in the memory. The array has been declared in the normal way without pointers, but the array elements have been received, stored and printed using pointer notation. When we execute the above program, we can type the numbers in any one of the ways given below:

- Enter one integer at a time followed by Return
- Enter all integers in a row with spaces between them and finally hit Return
- Enter three integers in a row as the result of the program indicates.

NOTES

Since it is a two-dimensional array we would like to input the value in the form of rows and columns and get the output in the same form. It would be better if the computer accepts the inputs at the given places and prompts us to do so, as in the example 3.38

Case Study: Receiving Inputs at Chosen Points

Example 3.39- To accept and print in matrix form.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int a[3][2];
   int i,j;
   printf("Enter values of 3x2 matrix-\n");
   printf("Press Enter after each value");
   for (i=0; i<=2; i++)
   {
      for (j=0; j<=1; j++)
      {
         gotoxy (j*10+10,4+i*2);
         scanf("%d",(*(a+i)+j));
      }
   }
   for (i=0; i<=2; i++)
   {
      for (j=0; j<=1; j++)
      {
         gotoxy (j*10+40, 4+i*2);
         printf("%d", *(*(a+i)+j));
      }
   }
}
```

Result of program

```
Enter values of 3x2 matrix-
Press Enter after each value
    00    01         0    1
    10    11        10    11
    20    21        20    21
```

A library function *gotoxy*(*x*, *y*) has been used in the program. This function makes the cursor go to the point (*x*, *y*) on the screen. At the first iteration when both *j* and *i* are 0, we will call goto(10,4) while scanning the values and call goto(40, 4) while printing the values. Therefore, the first value will be scanned on the 4th line from the top and at the 10th character position. The first value will be printed at the same line and 40th character position as the result of the program indicates. Therefore, we can direct the cursor to any position as we like. Thus, *gotoxy*() will be quite useful in advanced programming.

This has been used both for *scanf()* and *printf()*. The system will scan inputs from those points and print outputs at the points specified. We can specify any point on the screen to take inputs and print outputs. When we execute the program the cursor will go to the specified position. After we have typed one value and entered the return key, the cursor will blink at the next point, giving the impression of inputting a matrix. After the values are given, the output also appears in the form

NOTES

of a matrix. The left pair of numbers is what were entered. Since the same vertical position, but a different horizontal position has been specified in the program, the printout appears on the same lines, but to the right of the input values. Try the program and see for yourself.

Look at the address specification along with the *scanf()*. It is

(* (a + i) + j) which is equivalent to &a [i][j].

Putting a star before this converts this to the value a [i][j].

This program clearly illustrates the way to handle a two-dimensional array with pointers.

Although the array could have been declared as **a, it has been declared with the dimensions of the array. Since it does not know exactly the total number of elements in the former definition, garbage values may be stored in some more locations adjacent to the array elements, which can create problems. Some compilers will cause a run time error, if the array has been declared as **a. Therefore, it would be better to specify the dimension of the array as given in the above two examples.

3.9.1 Chain of Pointers/Array of Pointers

We had discussed that a pointer to a one-dimensional array can be denoted as **b* and two-dimensional array can be denoted as ***b*. In this section, an array of pointers will be discussed.

Case Study: Sorting Character Strings

As you know, names of students in a class can be denoted by a two dimensional array like b[50][20], with the second subscript denoting the width of the names and the first 50 denoting the number of students. There may be insufficient space for names longer than 20 characters, leading to truncation. If the name is short it will lead to wastage of space. By using an array of pointers, we can declare char * name [50] as an array to store the names of 50 students. The number of students in a class is definitely known. Here, name is an array and points to character. Thus, it is an array of pointers. Actually what happens is that 50 addresses are stored in the array name [50], name [0] corresponds to the address of first name, name [1] to the address of second name and so on. Using this concept a program is developed to sort names. It is given below:

Example 3.40 – For sorting character strings.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define N 5 /*5 names sorted*/
int main()
{
   int ret,i,j,p=0;
   char *name[N], t[50];
   for(i=0;i<N;i++)
   {
      printf("Enter name size\n");
      scanf("%d", &p);
      name[i]=(char *)(malloc(p*1));
      printf("Enter name\n");
```

NOTES

```
        scanf("%s",name[i]);
    }
    for(i=0;i<N-1;i++)
    for(j=i+1;j<N;j++)
    {
        ret=strcmp(name[j],name[i]);
        if (ret<0)
        {
            strcpy(t,name[i]);
            strcpy(name[i],name[j]);
            strcpy(name[j],t);
        }
    }
    printf("\nSorted Names Are :\n");
    for(i=0;i<N;i++)
        printf("%s\n",name[i]);
}
```

Result of program

```
Enter name size
5
Enter name
Raman
Enter name size
5
Enter name
Gopal
Enter name size
6
Enter name
Sharma
Enter name size
7
Enter name
Ramanan
Enter name size
4
Enter name
Basu
Sorted Names Are :
Basu
Gopal
Raman
Ramanan
Sharma
```

Here, the user is asked to first enter the number of characters in a name to be entered. After that the name is entered. This helps in allocating the right size to each name, no space more, no space less.

We call *strcmp* and pass references to names being compared. The function *strcmp* returns 0 if strings are equal; <0 if string 1 is less than string 2 in ASCII value; >0 if string 1 is greater than string 2. Thus, names will be exchanged if **name*[*j*] is less than **name*[*i*]. The string comparison starts from the first character of each word. If they are equal it will go to the next character and so on, till they are unequal or *NULL* is reached in either of them. Then the difference in the

NOTES

ASCII values of the characters compared last is returned. This is what we wanted in arranging the names alphabetically. Key in the program and you will find that it works correctly.

3.9.2 Pointers to Functions

We have discussed various aspects of pointers such as pointers to variables, functions returning pointers, etc. What then, is a pointer to a function? Consider the following programs.

```
#include <stdio.h>
int main()
{   char y='a';
    void func ( char z);
    void (*fp) ( char x);
    fp = func ;
    (*fp) (y) ;
    }
    void func ( char x)
    { ...
}
```

A perusal of the above program indicates that the address of *func* is assigned to the function pointer *fp*. While calling the function, (*fp) (y) is used. Thus fp gets the address of *func,* indicating that functions also have addresses. Such function pointers are used in searching and memory resident programs.

3.10 POINTER AND STRUCTURE

You know how to declare pointers to various data types and arrays. Similarly, pointers to structures can also be declared. For example, in the case of the structure `account`, you can declare a pointer as follows:

```
struct account a1 = { 1, "Vasu", 1000 };
struct account * sp;
struct sp = &a1 ;
```

Now, `sp` is a pointer to structure. Therefore, if you assume structure as another basic datatype, declaring it as an array and declaring it as a pointer, etc., follow the same rules. Structure is, in fact, a user defined datatype.

Access to individual elements of a structure defined in the form of a pointer is similar but instead of dot, we use an arrow pointer ->. Arrow pointer is formed by typing minus followed by the greater sign. However, on to the left of the arrow operator, there must be a pointer to a structure. The following example will clarify the point:

```
/*Example 3.41 - structure pointers */
  #include<stdio.h>
  int main()
  {
     struct account
     {
```

```
        unsigned number;
        char name[15];
        int balance;
    }a5;
    static struct account a1={001, "VASU", 1000};
    struct account *sp;
    sp=&a1;
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
    sp->number, sp->name, sp->balance);
    a5=*sp;
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
    a5.number, a5.name, a5.balance);
}
```

Result of the program

```
A/c No:=1 Name:=VASU Balance:=1000
A/c No:=1 Name:=VASU Balance:=1000
```

In this program, `sp` is a pointer to structure `account` and therefore, `sp` is assigned the address of structure `a1`. Then, the contents of structure `*sp` are printed. The elements of `*sp` are copied to `a5` and then, printed (to demonstrate copying of structures). Note the difference between the notations when accessing elements of a structure and a structure pointer.

3.10.1 Passing Structure by Reference

Structures can be passed by reference. Remember, however, that the structure should be defined as a global variable. The following passes structure by reference and withdrawal of money is processed in the called function.

```
/*Example 3.42 - structure pointers & functions*/
#include<stdio.h>
struct account
    {
        unsigned number;
        char name[15];
        int balance;
    };
int main()
{
    static struct account a1={001, "VASU", 1000};
    struct account debit(struct account *sp, int y);
    int deb;
    printf("Enter amount to be withdrawn");
    scanf("%d", &deb);
    debit(&a1, deb);
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
    a1.number, a1.name, a1.balance);
}
```

```
struct account debit(struct account *x, int y)
{
    x->balance-=y;
    return *x;
}
```

Result of the program

```
Enter amount to be withdrawn299
A/c No:=1 Name:=VASU Balance:=701
```

Note that the address of `a1` is passed. The prototype declares passing structure by reference and the amount of debit by value. In the called program, debit is adjusted. Note the pointer notation in subtracting the debit amount from the balance.

Now, look at some more examples to understand structure pointers but before that you must know now to allocate dynamic memory allocation for structures.

Let us say, `struct cycle * cp`; then to allocate memory dynamically, we can write:

```
cp = (struct cycle * ) malloc (size of (struct cycle ) * n)) ;
```

`n` is the number of structure variables of type `struct cycle`.

In the above, you have treated structure similar to other datatype. The size of the structure is not fixed like basic datatype. However, you can use the `sizeof` operator to get the size of the structure variable.

3.10.2 Array of Pointers

It was discussed that a pointer to a one dimensional array can be denoted as `*b` and two dimensional array can be denoted as `**b`. In this section, an array of pointers will be discussed.

As you know, names of students in a class can be denoted by a two dimensional array like `b[50][20]`, with the second subscript denoting the width of the names and the first 50 denoting the number of students. There may be insufficient space for names longer than 20 characters, leading to truncation. If the name is short, it will lead to wastage of space. By using an array of pointers, we can declare

`char * name [50]` as an array to store the names of 50 students. The number of students in a class is definitely known. Here, `name` is an array and points to character. Thus, it is an array of 50 pointers to characters. Actually, what happens is that 50 addresses are stored in the array `name [50]`, `name [0]` corresponds to the address of the first name, `name [1]` to the address of the second name and so on. Using this concept, a program is developed to sort names. It is given below:

```
/* Example 3.43 - for sorting character strings */
#include <stdio.h>
#include <string.h>
#include <alloc.h>
```

NOTES

```
#define N 5 /*5 names sorted*/
int main()
{
    int ret,i,j,p=0;
    char *name[N], t[50];
    for(i=0;i<N;i++)
    {
        printf("Enter name size\n");
        scanf("%d", &p);
        name[i]=(char *)(malloc(p*1));
        printf("Enter name\n");
        scanf("%s",name[i]);
    }
    for(i=0;i<N-1;i++)
    for(j=i+1;j<N;j++)
    {
        ret=strcmp(name[j],name[i]);
        if (ret<0)
        {
            strcpy(t,name[i]);
            strcpy(name[i],name[j]);
            strcpy(name[j],t);
        }
    }
    printf("\nSorted Names Are :\n");
    for(i=0;i<N;i++)
        printf("%s\n",name[i]);
}
```

Result of the program

```
Enter name size
5
Enter name
Raman
Enter name size
5
Enter name
Gopal
Enter name size
6
Enter name
Sharma
Enter name size
7
Enter name
```

```
Ramanan
Enter name size
4
Enter name
Basu
Sorted Names Are :
Basu
Gopal
Raman
Ramanan
Sharma
```

Here, the user is asked to first enter the number of characters in a name to be entered. After that, the name is entered. This helps in allocating the right size to each name, no space more or less.

You call `strcmp` and pass references to names being compared. The function `strcmp` returns 0 if the strings are equal; <0 if string 1 is less than string 2 in ASCII value; >0 if string 1 is greater than string 2. Thus, names will be exchanged if `*name[j]` is less than `*name[i]`. The string comparison starts from the first character of each word. If they are equal, it will go to the next character and so on, till they are unequal or `NULL` is reached in either of them. Then, the difference in the ASCII values of the characters compared last is returned. This is what you wanted in arranging the names alphabetically. Key in the program and you will find that it works correctly.

3.10.3 Pointer to Pointer

You know that pointer is an integer variable that contains the address of any variable of any valid data type. The variable can itself be another pointer. In this case, you have a pointer which contains the address of another pointer. For example, let us declare the following:

```
double dv=3.6;
int *p;
int **pp;
```

Now, we assign the following:

```
P=&dv;
pp=&p;
```

Assume that `p =1000`. Then, address of `dv` denoted by `&dv` will also be 1000. Then, the value of `*pp` will also be 1000.

Assume that the address of `p`, i.e., `&p` is 3000. Then, the address of `p` denoted by `*pp` will also be 3000.

The value of `dv`, denoted by `*(&dv)` equals `3.6` in the above.

The same can also be obtained by stating `*p` which is also `3.6`.

The same can be obtained by double dereferencing of `pp` as `**pp i&` also equal to `3.6.` Thus, `pp` is a pointer to pointer `p`.

NOTES

Check Your Progress

7. State about the array in C.
8. What is void pointer?
9. Define the term `NULL` pointer.
10. What is call by reference?
11. What will the function `goto xy (x, y)` do?

3.11 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The syntax of the `if` statement is given below.

```
if (condition)
 {statements}
```

   If the condition is `True,` then a single statement or group of statements following the `if` will be executed. If more than one statement is to be executed, then the statements are grouped within braces. If it is a single statement, then curly braces are not required.

2. 'C' provides three logical operators for combining more than one condition. These are as follows:

   Logical `and` represented as `&&`

   Logical `or` represented as ||

   Negation or `not` represented as ! (exclamation).

3. The `continue` statement is related to `break`. When `continue` is executed, it causes the next iteration of the corresponding `for`, `do...while` or `while` loop to begin. Therefore, `continue` takes the program to the top of the block and in the `for` loop, it will cause the next increment operation, followed by checking whether the condition is true or false in order to decide the next course of action. This is similar to skipping the current execution and continuing with the next operation after incrementing. The statement `continue` skips the rest of the statements in the loop for that iteration, whereas `break` terminates the loop.

4. There is a library function `exit(),` which causes the termination of the current program. Note that, `exit()` terminates the execution of the program itself, and not the block. The statement `break` enables coming out of the block or loop in which it is executed but `exit()` terminates the program at whatever stage the program may be. The `exit()` is a powerful function.

5. The `for` statement is meant for the easy implementation of iterations unlike `if.` The syntax of `for` is given below:

```
for (exp1; exp2; exp3)
{statements;}
```

6. The `while` loop is a subset of the `for` loop. The syntax for the `while` loop is given below:

```
    while (expression)
  {statements;}
```

7. An array refers to data structure which holds multiple variables of the same data type and uses an indexing system to find each variable stored within it. Like other variables in C, arrays must be declared before they can be used. Arrays can be one, two and multi-dimensional.
8. `void` pointer is a special type of pointer that has the flexibility of pointing to any data type.
9. `NULL` pointer is a type of pointer of any data type and generally takes a value as zero. `NULL` pointer does not point to any valid memory address.
10. The function call using pointers is known as call by reference. Here the address of the variable is passed. Note that calling by reference has to be indicated in the function declaration such as these:

```
fun (int *p, char *cp, float *fp, int *array);
```

11. The function `goto xy (x, y)` makes the cursor go to the point `(x, y)` on the screen.

3.12 SUMMARY

- Real-life application programs do not merely consist of simple multiplication or addition. They call for solving complex problems. Depending on the occurrence of a particular situation, we may follow different paths; the `if` and `else` keywords are quite handy in branching to different segments of the program.
- Switch statements allow clear and easy implementation of multiway decision-making. Assuming that a number is received from the keyboard and that depending on the value, we want to carry out some operations, the `switch` statement can be used effectively in this situation. In simpler situations `if...else` could be used, and in complex situations, `switch` can be used.
- Every `switch` statement, therefore, contains a condition in the form of an expression. The expression could also be a single variable as in this case. The expression will be evaluated at the time of program execution and must be an integer.
- The `break` statement helps immediate exit from any part of the loop as demonstrated with the `switch` statement. It can be used with any other loop construct or anywhere in the program. When the `break` statement is executed it goes to the bottom of the block. Recall that a block is a group of statements enclosed between an opening brace and the corresponding closing brace.

NOTES

- The continue statement is related to `break`. When `continue` is executed, it causes the next iteration of the corresponding `for`, `do...while` or `while` loop to begin.
- The library function `exit()` causes the termination of the current program. Note that, `exit()` terminates the execution of the program itself, and not the block.
- The statement `break` enables coming out of the block or loop in which it is executed but `exit` terminates the program at whatever stage the program may be. `exit` is a powerful function.
- Quite often we have to perform the same operation a number of times. We may also have to repeat the same operation with one or more of the values changed, which is known as loop or iteration.
- The `Do ... While` loop is a modification of the `while` statement. In the `while` statement, before the group of statements following the `while` are executed, the condition associated with the while is checked. If the condition is true or fulfilled, then the associated statements are executed. If not, the program skips the statements associated with the `while` loop.
- An array refers to data structure which holds multiple variables of the same data type and uses an indexing system to find each variable stored within it. Like other variables in C, arrays must be declared before they can be used. Arrays can be one, two and multi-dimensional.
- Multi-dimensional arrays operate on the same principle as single dimensional arrays. They are also a user defined data type.
- Pointer is a variable that contains the address of another variable.
- `void` pointer is a special type of pointer that has the flexibility of pointing to any data type.
- `NULL` pointer is a type of pointer of any data type and generally takes a value as zero. `NULL` pointer does not point to any valid memory address.
- The dimension or array size declaration of an array is an important subject. Array dimension is to be declared before compiling.
- The function call using pointers is known as 'call by reference'. Here, the address of the variable is passed.
- While calling by reference, we have to pass the address of the variable. If the variable is declared by value such as int a, then we have to pass &a. If it was declared as a pointer to say, an integer, such as int * ip, then ip has to be passed.
- We had discussed that a pointer to a one-dimensional array can be denoted as*b and two-dimensional array can be denoted as **b.
- The addresses or pointers can be stepped up or stepped down.
- When we say we are comparing pointers, we are comparing their addresses.

3.13 KEY TERMS

- **Loop or iteration:** It is used to perform the same operation a number of times by repeating the same operation with one or more of the values changed.
- **Switch:** When the switch keyword is encountered, the associated expression is evaluated.
- **Array:** It refers to data structure which holds multiple variables of the same data type and uses an indexing system to find each variable stored within it. Like other variables in C, arrays must be declared before they can be used. Arrays can be one, two and multi-dimensional.
- **Pointer:** It is a variable that contains the address of another variable.
- **`void` pointer:** It is a special type of pointer that has the flexibility of pointing to any data type.
- **`NULL` pointer:** It is a type of pointer of any data type and generally takes a value as zero.
- **Call by reference:** The function call using pointers is known as call by reference. Here, the address of the variable is passed.
- **Gotoxy(x,y):** It is a library function and is used in advanced programming to direct the cursor to go to the point (x,y) on the screen.

3.14 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. State about the conditional statements.
2. Define the term loops in C.
3. What is a `while` loop?
4. How `switch` statement are used?
5. Differentiate between one dimensional and two dimensional array.
6. What is a pointer?
7. How two dimensional array is represented conventionally?
8. Which two functions are frequently used in C language to allocate memory dynamically?
9. Write the syntax for declaring the pointer function.
10. Define the term pointer notation for arrays.
11. Write the representation of pointer to pointer.

NOTES

Long-Answer Questions

1. Briefly explain the conditional statements and branching of C program.
2. Write a program to illustrate the use of `switch` statement.
3. Describe the three component of `for` loop giving appropriate example programs.
4. Discuss the difference between `while` and `do…while` loop.
5. Analyse the multi-dimensional array giving appropriate examples programs.
6. Explain how pointers are declared with the help of relevant examples.
7. Briefly explain the operations using pointer variables. With reference to return by reference, discuss function declaration, function declarator and function call.
8. Explain the relationship between pointers and arrays—one dimensional and multi-dimensional.
9. Describe the role of pointers in dynamic memory allocation giving appropriate examples.
10. Discuss about the pointer and one dimensional array with the help of example.
11. Analyse the chain of pointers giving appropriate example programs.
12. Explain the concepts of pointers to structure, passing structures by reference, array of pointers and pointer to pointer with the help of programs.

3.15 FURTHER READING

Gottfried, Byron S. 1996. *Programming with C*, Schaum's Outline Series. New York: McGraw-Hill.

Jeyapoovan T. 2006. *Computer Programming - Theory & Practice*. New Delhi: Vikas Publishing House Pvt. Ltd.

Khurana, Rohit. 2005. *Object Oriented Programming*. New Delhi: Vikas Publishing House Pvt. Ltd.

Kanetkar, Yashwant. 2003. *Let Us C*. New Delhi: BPB Publication.

Saxena, Sanjay. 2003. *A First Course in Computers*. New Delhi: Vikas Publishing House Pvt. Ltd.

Subburaj, R. 2000. *Programming in C*. New Delhi: Vikas Publishing House Pvt. Ltd.

Ghosh, Smarajit. 2009. *All of C*. New Delhi: PHI Learning Pvt Ltd.

Bronson, Gary J. 2000. *A First Book of ANSI C*, 3rd edition. California: Thomson, Brooks Cole.

UNIT 4 HANDLING OF CHARACTER STRINGS, FUNCTION AND STRUCTURE

Structure

4.0 Introduction
4.1 Objectives
4.2 Strings in C
- 4.2.1 Use of `scanf()` and `printf()` with Strings
- 4.2.2 Reading and Writing Strings: `gets` and `puts`
- 4.2.3 Library Functions for String Handling
- 4.2.4 Two-Dimensional Character Arrays

4.3 User Defined Function Form of C
- 4.3.1 Function Declaration
- 4.3.2 Differences between User Defined Functions and Stored Procedures

4.4 Functions in C
- 4.4.1 Function Call – Passing Arguments to a Function
- 4.4.2 Scope: Rules for Functions
- 4.4.3 Function Parameters
- 4.4.4 Return Values and their Types

4.5 Calling a Function by Value and Reference
4.6 Function Recursion
4.7 Arrays and Functions
4.8 Structures and Functions
- 4.8.1 Declaration
- 4.8.2 Processing a Structure
- 4.8.3 User-Defined Data Type
- 4.8.4 Structure Elements Passing to Functions
- 4.8.5 Structure Passing to Functions
- 4.8.6 Structure Within Structure

4.9 Function with Union
4.10 Answers to 'Check Your Progress'
4.11 Summary
4.12 Key Terms
4.13 Self Assessment Questions and Exercises
4.14 Further Reading

4.0 INTRODUCTION

We know that arrays are user-defined data types containing same data type, such as integers or real numbers. We also discussed one-dimensional and multi-dimensional arrays. We know how to address each element. In C language, string is a special type of array. When a data item, constant or variable consists of array of characters, such arrays are known as strings.

A function, when declared in a program, is a function prototype which may be declared at the beginning of a main program. A function, on execution, is supposed to do something: either return a value as integer, character or float; or perform

some operation. A function consists of two parts, declarator and declaration. A function declaration has the format in which the type of data returned, name of the function and arguments on which the function is to operate, are mentioned. Arguments declared as part of the function prototype are called formal parameters, which are enclosed in a pair of parentheses. A function may not contain any parameter, in which case an empty pair of parentheses should follow the name of the function. A function may not return a value, in which case void is written as the return data type.

A function may be called directly or indirectly by another function. For this, there should be one-to-one correspondence between formal arguments declared and actual arguments sent and should be of the same data type. A function declarator is a replica of a function declaration; the difference lies in the way they are written inside a program body. A declaration in a calling function will end with a semicolon and a declarator in a called function will not end with a semicolon. A structure is a user-defined data type like an array. While an array contains elements of the same data type, a structure contains members of varying data types. A structure declaration ends with a semicolon and the keyword is `struct.` The tag for structure is optional. Structures can be passed by reference and can be nested. `Unions` help in conserving memory as only one of the many variables will be used at a time. The syntax of a union is similar to that of structure.

In this unit, you will study about the handling of character strings, declaring and initializing string variables, string handling function, user defined function form of C, return values and their types, calling a function by value and reference, function recession, function with array, structure and union, structure initialization and array of structures, structure within structure, structure and function.

4.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the handling of character strings
- Define the declaring and initializing string variables
- Analyse the string handling function
- Explain user defined function form of C
- Describe the return values and their types
- Discuss about the calling a function by value and reference
- Understand the significance of function recession
- Define the function with array
- Elaborate the concept of structure and union
- Analyse the structure initialization and array of structures
- Explain the structure within structure in C
- Discuss about the structure and function

4.2 STRINGS IN C

String—An Array of Characters

As we know *char* is a basic data type. A string is formed using an array of the basic data type *char.* We can define a string variable as well as string literal. String literals are words surrounded by double quotation marks. In reality, both of these string types are collections of characters stored contiguously, i.e., next to each other in the memory. The only difference is that we cannot modify string literals, whereas we can modify string variables. We can define a single character by enclosing it within single quotes as given below.

```
char ch = 's';
```

The above declares a *ch* of type *char* and initializes with a character constant s.

A *string* is an array of characters, which means that it may contain zero to many characters. A *string* is enclosed within double quotes.

For instance,

```
char st[1] = "s" ;
```

The variable *st* is a string since it is of type array of characters. It is initialized with constant *s*. When we initialize a string variable as above a NULL character will be appended to the end of the *string* automatically by the system.

The following declaration and initialization create a *string* consisting of the word *'Seven'*. To hold the *null* character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word *'Seven'*.

```
char str1[6] = {'S', 'e', 'v', 'e', 'n', '\0'};
```

We can write the same easily as given below:

```
char str1[ ] = "Seven";
```

Following is the memory presentation of above defined string in C:

S	e	v	e	n	\0

Actually, we do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Note that the ASCII value of NULL, i.e., '\0' is zero, whereas the ASCII value of number zero is 48.

4.2.1 Use of `scanf()` and `printf()` with Strings

One can use `scanf()` to receive strings from the screen. The code for using *scanf* for reading a name is given below:

```
char name [25];
scanf (" %s ", str);
```

Strings can be declared as an array of characters as shown above. In the *scanf* function when we get the array of characters as a string, it is enough to indicate the name of the array without a subscript. In this case the identifier or the name of the string is *str,* When we get a string, there is no need for writing '&'

before *str* in this case in the *scanf* argument list. We can indicate the name of the string variable itself. Let us now write a program to read a *string* with *scanf*() and write with *printf*().

The program is given below:

Example 4.1

```
/* Reading and writing with formatted I/O functions*/
#include<stdio.h>
int main ()
{
char str [10];
printf ("Enter your name\n");
scanf ("%s", str);
printf ("You Entered: %s\n");
}
```

Look at the program. We are declaring a *string str* as a character array of size 10. Look at the way we receive the array. We do not prefix ampersand (&) to the variable name *str* when we use the *scanf* function. We do not even suffix it with square brackets. Then, we print the contents of variable *str*. The format specifier in both the cases is %s. Look at the result of the program.

Result of Example 4.1

```
Enter your name
Rama Krishnan
You Entered: Rama
```

We typed 'Rama Krishnan'. But we got 'Rama' since the white space following the first word was considered to be end of character array. Hence, the computer stopped reading thereafter. This is because we used a formatted input function, namely *scanf* ().

It is to be noted carefully that we cannot exceed the dimension of the character array which is 10 in this case. Also, it cannot receive a multi-word string as the above example demonstrated.

4.2.2 Reading and Writing Strings: *gets* and *puts*

Strings may contain blanks in between. If we use a `scanf()` function to get the string with a space in between such as "Rama Krishnan", Krishnan will not be taken note of since space is an indicator of the end of entry of the input as indicated by the result of the previous example. But *gets* will take all that is entered till the enter key is pressed. Therefore, after entering the full name, the *enter* key can be pressed. Thus, using *gets* is a better way for strings. We can get a string in a much simpler way using *gets ()*. The syntax for *gets* is: *gets (*name*)*;

Similarly, *puts ()* can be used for printing a variable or a constant as given below:

puts (name);

puts ("Enter the word");

However, there is a limitation. *printf* can be used to print more than one variable and *scanf* to get more than one variable at a time, in a single statement. However, *puts* can output only one variable and *gets* can input only one variable

in one statement. In order to input or output more than one variable, separate statements have to be written for each variable. As we know that *gets* and *puts* are unformatted I/O functions, there are no format specifications associated with them.

The example below gives the modified version of the above program.

```
Example 4.2
/* Reading and writing with unformatted I/O functions*/
#include<stdio.h>
int main ()
{
char str [10];
printf (“Enter your name\n”);
gets (str);
printf (“You Entered :”);
puts (str);
}
```

Look at the program. We have declared str as a one-dimensional character array. We read the string as shown below:

```
gets (str);
```

No format needs to be specified. Writing also is similar. The argument to *puts* () as well as *gets* () are *str* of type *string*. Look at the result of the program.

Result of Example 4.2

```
Enter your name
Rama Krishnan
You Entered:Rama Krishnan
```

The entire *string* has been received although there is white space in between. Thus, `gets()` is more suitable for handling strings. But, if we have to receive two strings, we need to call `gets()` twice. Though any number of strings can be received with one `scanf()` statement, it will treat white space as a `string` terminator as in Example 4.1. Similarly, `puts()` can display only one string at a time. Whereas `printf()` can print any number of strings with one statement.

String COPY

Let us write a program to copy one string to another.

```
Example 4.3
/*String copy*/
#include<stdio.h>
int main()
{
char str1[10], str2[10];
int i;
printf(“Enter a string\n”);
gets (str1);
for(i=0; i<10; i++)
str2[i]=str1[i];
printf(“you Entered:\n”);
puts(str1);
printf(“copied string is:\n”);
puts(str2);
}
```

NOTES

Let us analyze the program. Two strings `str1` and `str2` are declared with size 10. Then we get `str1`. In the `for` loop, we assign `str1` to `str2`, one character at a time. Finally, we print both the strings.

The result of the program is given below:

Result of Example 4.3

```
Enter a string
Vinayagar
you Entered:
Vinayagar
copied string is:
Vinayagar
```

We have written a program for copying one string to another. But a library function `strcpy()` is available for this purpose. Let us write a program for copying a string to another using the library function. The program is given below:

Example 4.4

```
/*String copy using library function*/
#include<stdio.h>
#include<string.h>
int main()
{
char str1[]="Subramaniar", str2[11];
strcpy(str2,str1);
printf("you Entered:\n");
puts(str1);
printf("copied string is:\n");
puts(str2);
}
```

Look at the declaration statement. We have declared and initialized `str1` and declared `str2` in one statement. There is no need to give the dimension when we initialize the array as given in the program. The compiler will count the number of elements and add the dimension by itself. Look at the string copy statement, reproduced below:

```
strcpy (str2, str1) ;
```

Contents of the second named string will be copied to first named string. `strcpy()` is a library function and it receives two string variables, the first one is the destination string and the last one the source string. Look at the result of execution of the program.

Result of Example 4.4

```
you Entered:
Subramaniar
copied string is:
Subramaniar
```

The program works correctly. Thus, we can use the library function easily without writing a program for copying strings.

The library function is part of header file `<string.h>`. Therefore, some compilers may require us to include `<string.h>` in the program for using the function.

Finding String Length

Many times we may need to find the length of a given string. Let us now write a program to find the length of a string.

Example 4.5

```
/*Finding length of a string*/
#include<stdio.h>
int main()
{
char str1[]="Shri Rama Jeyam";
int length;
for(length=0; str1[length]!='\0'; length++);
printf("you Entered: ");
puts(str1);
printf("\nits length=% d ", length);
}
```

Look at the program. Look at the following statement carefully.

```
for(length=0; str1[length]!='\0'; length++);
```

The length of the string is computed in this statement only. In the first iteration of the `for` statement, str1 [0] will be compared with NULL ('/0'). Since it is not NULL, length will be incremented to 1. This continues till the last character of the string is compared. After that, we would have reached the end of the string or NULL. Hence, execution of the `for` loop will be terminated. The variable length will thus contain the actual length of the string. It is then printed. Thus, the program works correctly as the result below indicates.

Result of Example 4.5

```
you Entered: Shri Rama Jeyam
its length= 15
```

Even for finding the length of the string there is a library function called `strlen()`. Let us use it and modify the above program. The modified program is given below:

Example 4.6

```
/*String length using library function strlen()*/
#include<stdio.h>
#include<string.h>
int main()
{
char str1[]="Shri Rama Jeyam";
printf("you Entered: ");
puts(str1);
printf("\nits length=%d", strlen(str1));
}
```

Look at the program. We find the length and print it in the last statement reproduced below:

```
printf("\nits length=%d", strlen(str1));
```

When the program execution reaches the line, the cursor will go to the beginning of the next line because of the appearance of \n. Then the text "`its length =` " will be printed. Then, it will look for an integer specified outside the quote, to

print because of the appearance of %d. Here the integer will come out of executing the library function `strlen(str1)`. Thus, we get identical result as given below.

Result of Example 4.6

```
you Entered: Shri Rama Jeyam
its length=15
```

From the above examples, it will be clear that the use of library functions conserves the effort and also shortens the program. We have used a few library functions, which facilitate string manipulation.

4.2.3 Library Functions for String Handling

There are a number of library functions to handle strings. A list of such functions available for string manipulation is given in Table 4.1.

Table 4.1 *String Library Functions*

Function name	**Application**
strrev()	Reverses a string
strlen()	Finds length of a given string
strcat()	String concatenation; appends one string at the end of another
strncat()	Appends first *n* characters of one string at the end of another
strcmp()	Compares two strings
strncmp()	Compares first *n* characters of two strings
strnicmp()	Compares first *n* characters of two strings without regard to case
strmcpi()	Compares two strings without regard to case
strlwr()	Converts the given string to lower case
strupr()	Converts the given string to upper case
strcpy()	Copies a string to another
strncpy()	Copies first *n* characters of one string to another
strdup()	Duplicates a string
strchr()	Finds the first occurrence of a given character in a string
strrchr()	Finds the last occurrence of a given character in a string
strstr()	Finds the first occurrence of a given string in another string
strset()	Sets all characters of a string to a given character
strnset()	Sets first *n* characters of a string to a given character

If these are to be used, `<string.h>` should be included before the main function. Some examples of the uses of the above library functions are discussed below:

`strlen (CS)` – Returns the length of string CS.

`char * strcpy (s, ct)` – Copies string ct to string s, including NULL and return s.

`char * strcat (s, ct)` – Concatenates string ct to end of string s; return s.

`int strcmp (cs, ct)` – Compares string cs to string ct; returns < 0 if cs < ct,

`0 if cs = = ct,` or > 0 if cs > ct.

`char * strchr (cs, c)` — Returns the pointer to the first occurrence of c in cs or NULL if not present.

4.2.4 Two-dimensional Character Arrays

In the above examples we created strings, which are essentially one-dimensional arrays of characters. We can create an array of strings. This will be a two-dimensional array of characters. For instance,

char name[5][10];

declares a two-dimensional array of characters. This can be used to deal with five strings of size 10 each.

Let us now write a program as below to read five names and display them.

Example 4.7

```
/*Two Dimensional array*/
#include<stdio.h>
int main()
{
int i;
char name[5][10];
/*Receiving strings*/
for (i=0; i<5; i++)
 {
 printf("Enter name[%d]: \n", i+1);
   scanf("%s", name[i]);
  }

/*displaying strings*/
for (i=0; i<5; i++)
 {
 printf("name[%d]: %s\n", i+1, name[i]);
  }
  }
```

Look at the program. We declare a two-dimensional character array, called name[5][10].

Then we receive the names. Look at the ease with which we receive it.

```
scanf("%s", name[i]);
```

We do not even give the second dimension. This is possible only in the case of strings. Recall that when *str* was a one-dimensional array, we read it in the `scanf()` function by simply specifying *str* and not its address. But since we are specifying `scanf()`, we cannot give white spaces in between the names.

As we know, the elements of an array will be stored contiguously. In name[i], we are specifying the address of 0th location of the ith row. The array received will be stored thereon continuously. In the next section, we print each string the same way by specifying the first subscript alone. The result of the program is given below:

Result of Example 4.7

```
Enter name[1]:
Ganapathy
Enter name[2]:
Subramani
Enter name[3]:
Narayanan
Enter name[4]:
Joseph
Enter name[5]:
Mohammed
name[1]: Ganapathy
name[2]: Subramani
name[3]: Narayanan
name[4]: Joseph
name[5]: Mohammed
```

The result demonstrates the use of two-dimensional character arrays.

Be cautious not to exceed the size of the array. Although we did not enter 10 characters, the computer recognized the end of the string due to Null character generated by the pressing of Enter key each time.

We will take another interesting example. If a word is a palindrome, we will get the same word when we read the characters from the right to the left as well, as already discussed.

Examples are : nun

malayalam

These words when read from either side give the same name. We will write a program to check whether a word is a palindrome or not.

This program uses a library function called `strlen( )`. The function `strlen (str)` returns the size or length of the given string. Now let us look at the program.

Example 4.8

```
/* to check whether a string is palindrome*/
#include <stdio.h>
#include <string.h>
#define FALSE 0
int main()
{
   int flag=1;
   int right, left, n;
   char w[50]; /* maximum width of string 50*/
   puts("Enter string to be checked for palindrome");
   gets(w);
   n=strlen(w)-1;
   for ((left=0, right=n); left<=n/2; ++left, --right)
   {
      if (w[left]!=w[right])
      {
         flag=FALSE;
         break;
      }
```

NOTES

```
      }
      if (flag)
      {
         puts(w);
         puts("is a palindrome");
      }
      else
         printf("%s is NOT a palidrome");
   }
```

Result of the program

```
Enter string to be checked for palindrome
palap
palap
is a palindrome
```

If `strlen` or *gets* or *puts* are used in a program we have to include `<string.h>` before the `int main()`.

Now let us analyze the functioning of the above example.

We are defining a symbolic constant *FALSE* as 0.

We initialize flag as 1. We define a string *w* as an array of 50 characters.

`gets(w);` returns the word typed and stores it from location &w [0]

Let us assume that we had typed "nun" and go on to analyze what happens in the program.

strlen (w) will return the length of the word typed. In this case *strlen* (w) = 3.

We subtract this by 1 to get the subscript of the extreme right character. The subscript of the extreme left character is obviously 0.

We are initializing the *for* loop with the following:

left = 0 right = n = 2

flag = 1

We check whether left < = n/2 and

since it is so, we check whether w [0]!=w [2].

The condition is false since w[0] = w[2] = 'n'.

Therefore, the group of statements following *if* is skipped: flag remains 1.

Step 2

Now left is incremented to 1 and right is decremented to 1.

Again w[1] ! = w[1] is false, *flag* remains 1

Therefore, control returns to the *for* statement.

Now, left = 2 right = 0

Since left is greater than n/2, the control comes out of the *for* loop. Now, the statement *if* (*flag*) will be executed.

It will check whether *flag* is true. In this case, *flag* is still true.

Therefore, the computer prints that the word is palindrome. But, had the word not been a palindrome what would happen?

To check this, let us assume that we typed "book" and see what happens in the program.

To start with, left = 0 right = 3

w [left] ! = w[right]

Therefore, the statements within the { } will be executed, *flag* will be set to false and then the *break* statement will be executed.

The statement *break* causes immediate exit from the loop. Now *flag* is false. Therefore, the *else* statement is executed to say that the word is NOT a palindrome.

Now let us write a program to reverse a given string.

Example 4.9

```
/* to reverse a string*/
#include <stdio.h>
#include <string.h>
int main()
{
   char w1[]="abcdefghij";
   void rev(char w1[]);
   rev(w1);
}
/*function definition*/
void rev(char w2[])
{
   char w3[]="    "; /*array w3 initialized with 10 blanks*/
   int i=0, j=0;
   for (i= 9, j=0; i>=0, j<=9; i—, j++)
   w3[j]=w2[i];
   printf("original string:\t");
   puts(w2);/*original string printed*/
   printf("Reversed string:\t");
   puts(w3);/*reversed string printed*/
}
```

Result of program

```
original string:   abcdefghij
Reversed string:   jihgfedcba
```

The purpose of the program is to reverse a string. A string is declared as an array of characters and initialized, w1[]="abcdefghij"; and a NULL character will be inserted at the end of the string by the compiler. The programmer need not worry. A function called *rev* is then declared as given below:

```
void rev(char w1[]);
```

The function returns nothing. The function can be called in a simple manner as given below:

```
rev(w1);
```

In the called function, a character array w3 is initialized with 10 blanks. Initialization is important to avoid surprises. The *for* loop reverses the string by copying w2 in the reverse order to w3; W2[0] is copied to w3[9], w2[1] is copied to w3[8] and so on. Then, w2 and w3 are printed one after other.

See how each character is accessed in the *rev* function. Note also the direct initialization of string w1 as an array of characters, in the main function.

4.3 USER DEFINED FUNCTION FORM OF C

User defined functions, and about the function declaration, function call and how to define a function. A function, when declared in a program, is a function prototype which may be declared at the beginning of a main program. A function, on execution, is supposed to do something: either return a value as integer, character or float; or perform some operation. A function consists of two parts, declarator and declaration. A function declaration has the format in which the type of data returned, name of the function and arguments on which the function is to operate, are mentioned. Arguments declared as part of the function prototype are called formal parameters which are enclosed in a pair of parentheses. A function may not contain any parameter, in which case an empty pair of parentheses should follow the name of the function. A function may not return a value, in which case `void` is written as the return data type.

A function may be called directly or indirectly by another function. For this, there should be one to one correspondence between formal arguments declared and actual arguments sent and should be of the same data type. A function declarator is a replica of a function declaration; the difference lies in the way they are written inside a program body. A declaration in a calling function will end with a semicolon and a declarator in a called function will not end with a semicolon.

You will also learn about the scope of a variable and the different storage classes used to define the scope of variables. A variable may be local to a particular function where it is defined, or global for all the functions in that program body if defined before all the functions in that program. There is no restriction in passing any number of values to a function and hence an array can be passed to a function. The restriction, however, lies in the return of values from a function. A function can be called by passing a value to its argument which is 'called by value'. A function can return more than one value when passed by reference. A good understanding of 'Pointers' is required in order to understand this.

There are four types of storage classes: auto, static, extern and register. The default class is '`auto`' if nothing is mentioned about the storage class. All the three types, other than the register, are stored in the memory. The register type is stored in the register of the Central Processing Unit (CPU).

Finally, in this unit, you will learn about functions. A function may call more than one function, and a function may be called by more than one function. A function is recursive when it calls itself either directly or indirectly. Finding the factorial of a number is an example of a recursive function.

4.3.1 Function Declaration

A function in a program consists of three characteristics:

- Function Prototype
- Function Definition
- Function Call

A function prototype is called a function declaration. A function may be declared at the beginning of the main function. Function declaration is of the following type:

```
return data - type  function name  (formal argument 1,
argument 2, ............ );
```

A function after execution may return a value to the function which called it. It may not return a value at all but may perform some operations instead. It may return an integer, character, or float. If it returns a float, we may declare the function as

```
float f1(float arg 1, int arg 2);
```

If it does not return any value we may write the above as

```
void fun2(float arg1, int arg2);   /*void means nothing*.
```

If it returns a character, we may write

```
char fun3(float arg1, int arg2);
```

If no arguments are passed into a function, an empty pair of parentheses must follow the function name. For example,

```
char fun4( );
```

The arguments declared as part of the prototype are also known as formal parameters. The formal arguments indicate the type of data to be transferred from the calling function. This is about function declaration.

Function Definition

Function definition can be written anywhere in the file with a proper declarator, followed by the declaration of local variables and statements. Function definition consists of two parts, namely function declarator or heading and function declarations. The function heading is similar to function declaration but will not terminate with a semicolon.

The use of functions will be demonstrated with simple programs in this unit.

Suppose you wish to get two integers. Pass them to a function `add`. `Add` them in the `add` function. Return the value to the main function and print it. The algorithm for solving the problem will be as follows:

Main Function

Step 1: Define function add

Step 2: Get 2 integers

Step 3: Call add and pass the 2 values

Step 4: Get the sum

Step 5: Print the value

function `add`

Step 1: Get the value

Step 2: Add them

Step 3: Return the value to main

Thus, you have divided the problem. The program is as follows:

```
/*Example 4.10*
  /* use of function*/
  #include <stdio.h>
  int main()
  {
      int a=0, b=0, sum=0;
      int add(int a, int b); /*function declaration*/
      printf("enter 2 integers\n");
      scanf("%d%d", &a, &b);
      sum =add(a, b);   /*function call*/
      printf("sum of %d and %d =%d", a, b, sum);
  }
  /*function definition*/
  int add (int c, int d)   /*function declarator*/
  {
      int e;
      e= c+d;
      return e;
  }
```

Result of the program

```
enter 2 integers
6667  4445
sum of 6667 and 4445 =11112
```

The explanation as to how the program works is given below:

On the fifth statement (seventh line), the declaration of the function `add` is given. Note that the function will return an integer. Hence, the return type is defined as `int`. The formal arguments are defined as `int a` and `int b`. The function name is `add`. You cannot use a variable without declaring it, as also a function without telling the compiler about it. Note also that function declaration ends with a semicolon, similar to the declaration of any other variable. Function declaration should appear at the beginning of the calling function. It hints to the compiler that the function is going to call the function `add`, later in the program. If the calling function is not going to pass any arguments, then empty parentheses are to be written after the function name. The parentheses must be present in the function declaration. This happens when the function is called to perform an operation without passing arguments. In this case, if `a` and `b` are part of the called function `(add)` itself, then we need not pass any parameters. In such a case, the function declaration will be as follows assuming that the called function returns an integer:

```
int add  ( ) ;
```

In Example 4.10, you get the values of `a` and `b`. After that you call the function `add` and assign the value returned by the function to an already defined `int` variable `sum` as follows:

```
sum  =  add ( a , b );
```

Note that `add(a, b)` is the function call or function reference. Here, the return type is not to be given. The type of arguments are also not to be given. It is a simple statement without all the elements of the function declaration. However, the function name and the names of the arguments passed, if any, should be present in the function call. When the program sees a function reference or function call, it looks for and calls the function and transfers the arguments.

The function definition consists of two parts, i.e., the function declarator and function declarations.

The function declarator is a replica of the function declaration. The only difference is that while the declaration in the calling function will end with a semicolon, the declarator in the called function will not end with a semicolon. As in `main()`, the entire functions body will be enclosed within braces. The whole function can be assumed to be one program statement. This means that all the statements within the body will be executed one after another before the program execution returns to the place in the main function from where it was called.

The important points to be noted are as follows:

- The declarator must agree totally with the declaration in the called function, i.e., the return data type, the function name, the argument type should all appear in the same order. The declarator will not end with a semicolon.
- You can also give the same name as in the calling function—in declaration statement or function call—or different names to the arguments in the function declarator. Here, we have given the names `c` and `d`. What is important; however, is that the type of arguments should appear, as it is in the declaration in the calling program. They must also appear in the same order.
- At the time of execution, when the function encounters the closing brace `}`, it returns control to the calling program and returns to the same place at which the function was called.

In this program, you have a specific statement `return (e)` before the closing brace. Therefore, the program will go back to the main function with the value of `e`. This value will be substituted as

```
sum = (returned value)
```

Therefore, `sum` gets the value which is printed in the next statement. This is how the function works.

Assume now that the program gets `a` and `b` values, gets their `sum1`, gets `c` and `d` and gets their `sum2` and then both the sums are passed to the function to get their total. The program for doing this is as follows:

```
/*Example 4.11*
   /* A function called many times */
   #include <stdio.h>
   int main()
   {
       float a, b, c, d, sum1, sum2, sum3;
```

```
    float add(float a, float b); /*function declaration*/
    printf("enter 2 float numbers\n");
    scanf("%f%f", &a, &b);
    sum1 =add(a, b);  /*function call*/
    printf("enter 2 more float numbers\n");
    scanf("%f%f", &c, &d);
    sum2 =add(c, d);  /*function call*/
    sum3 =add(sum1, sum2);  /*function call*/
    printf("sum of %f and %f =%f\n", a, b, sum1);
    printf("sum of %f and %f =%f\n", c, d, sum2);
    printf("sum of %f and %f =%f\n", sum1,sum2, sum3);
}
/*function definition*/
float add (float c, float d)  /*function declarator*/
{
    float e;
    e= c+d;
    return e;
}
```

Result of the program

```
enter 2 float numbers
1.5    3.7
enter 2 more float numbers
5.6    8.9
sum of 1.500000 and 3.700000 =5.200000
sum of 5.600000 and 8.900000 =14.500000
sum of 5.200000 and 14.500000 =19.70000
```

You have defined `sum1, sum2` and `sum3` as `float` variables.

You are calling the function `add` three times with the following assignment statements:

```
sum1 = add( a, b );
sum2 = add( c, d);
sum3 = add( sum1 , sum2 );
```

Thus, the program goes back and forth between `main()` and `add` as given below:

```
int main()
add (a, b)
int main()
add (c, d)
int main()
add (sum 1, sum 2)
int main()
```

Had you not used the function `add`, you would have to write statements pertaining to `add` 3 times in the main program. Such a program would be large and difficult to read. In this method, you have to code for `add` only once and hence, the program size is small. This is one of the reasons for the usage of functions.

In Example 4.11, you could add another function call by `add (10.005, 3.1125)`; This statement will also work perfectly. After the function is executed, the sum will be returned to the `main()` function. Therefore, both variables and constants can be passed to a function by making use of the same function declaration.

You have seen a program, which calls a function thrice. You will now discuss a problem, which calls three functions.

Problem

The user gives a four digit number. If the number is odd, then the number has to be reversed. If it is even, then the number is to be doubled. If it is evenly divisible by three, then the digits are to be added. Now, let us write the algorithm for solving the problem.

Step 1: Get the number.

Step 2: `If` the number is odd call, the `reverse` function.

Step 3: `Else` multiply the number by 2 and hence call `multiply`.

Step 4: `If` the number is evenly divisible by 3, call `add_digit`.

These are the steps. The first one, namely writing a function to multiply by 2 is simple. We will look at the other two steps now.

Reverse

The following steps show the reversing of number.

```
Step 1:  reverse = 0
step 2:  while ( number > 0 )
         reverse = reverse * 10 +  (number % 10 )
         number = number/10
Step 3:  return (reverse).
```

Add-Digits Function

```
Step 1:  sum = 0
Step 2:  while number > 0
         sum = sum + (number % 10)
         number = number / 10
Step 3:  return (sum)
```

Let us see how the above algorithm `add_digit` works.

Let us give 4321 as the number.

```
Step 1:  sum = 0
Step 2:  Iteration 1
         sum = 0 +  modulus of  (4321/10)
             = 0 + 1 = 1
      number = 4321/10 = 432
```

```
Iteration 2
      sum = 1 + modulus of (432/10)
          =  1 + 2
```

After 4 iterations:

```
sum =1+2+3+4
Step 3: sum is returned.
```

The program is given below:

/*Example 4.12*/

```
/*program to demonstrate calling
multiple functions*/
#include<stdio.h>
int main()
{
    long nummul=0;
    long num=0, rev=0, add_digit=0;
       /*good practice to inialize all variables*/
    long reverse(long num);
    long mult(long num);
    int sum_digit(long num);
    printf("enter unsigned number\n");
    scanf("%lu", &num);
    if (num%2)  /*remainder 1*/
    {
       rev = reverse(num);
       printf("number is odd\n");
       printf("number entered=%lu\n number reversed=%lu\n",
num, rev);
    }
    else
    {
       nummul=mult(num);
       printf("number is even\n");
       printf("number=%lu\n its multiple=%lu\n", num,
nummul);
    }
    if (num%3 ==0)
    {
       add_digit= sum_digit(num);
       printf("number evenly divisible by 3\n");
       printf("sum of digits =%lu", add_digit);
    }
}
long reverse(long n)
{
```

NOTES

```
        long r=0;
        while (n>0)
        {
           r=r*10+(n%10);
           n=n/10;
        }
        return r;
    }
    long mult(long p)
    {
        long sq;
        sq=2*p;
        return sq;
    }
    int sum_digit(long num)
    {
        long sum=0;
        while (num >0)
        {
           sum=sum+(num%10);
           num=num/10;
        }
        return sum;
    }
```

Result of the program

```
enter unsigned number
4321
number is odd
number entered=4321
number reversed=1234
```

Look at the program. After getting the number from the user, it evaluates if the remainder of `(num/2) = true`; i.e., if the remainder is 1, then it is true. If the `remainder = 1`, then the number is odd and hence the reverse function is called. The returned value is assigned to `rev` and printed.

If the number is even, the number is doubled. Since the doubled value may exceed the maximum of the unsigned integer, we have declared it as a `long` integer.

Next, we check if the number is evenly divisible by 3.

If it is so, then we add the digits. Thus, the main function of Example 5.3 calls three functions for carrying out specific tasks. All the three functions are supplied with the same arguments but return different values.

4.3.2 Differences between User Defined Functions and Stored Procedures

Functions and stored procedures both are precompiled objects in databases but there is some difference between user defined functions and stored procedure as listed below:

- Function must return at least one value but stored procedure may or may not.
- We can use a function for only data manipulation but stored procedure is used for manipulation as well as modification of data on tables.
- We can not use data manipulation statements inside Function but in stored Procedure, we can use data manipulation statements.
- In stored procedure, we can pass INPUT and OUTPUT parameters, but a function accepts only input parameters and returns a value.

4.4 FUNCTIONS IN C

4.4.1 Function Call – Passing Arguments to a Function

We may call a function either directly or indirectly. When we call the function, we pass the actual arguments or values. Calling a function is also known as function reference.

There must be a one-to-one correspondence between formal arguments declared and the actual arguments sent. They should be of the same data type and in the same order. For example,

```
sum=f1 (20.5, 10); fun4 ();
```

4.4.2 Scope: Rules for Functions

The scope of the variable is local to the function, unless it is a global variable. For instance,

```
int function1 (int I )
{ int j=100;
   double function2 (int j) ;
   function2 (j) ;
}
double function2 (int p)
{  double m;
   return m;
}
```

The variable `j` in `function1` is not known to `function2`. You pass it to `function2` through the argument `j`. This will be assigned as equal to `int p`. Similarly, `m` in `function2` is not known to `function1`. It can be made known to `function1` through the return statement. This makes the scope rules of variables in function quite clear. The scope of variables is local to the

NOTES

function where defined. However, global variables are accessible by all the functions in the program if they are defined above all functions.

/*Example 4.13*

```
/* To demonstrate that the scope of a
variable is local to the function*/
#include <stdio.h>
int main()
{
    float a=100.250, b=200.50;
    void change (float a, float b);
    change(a, b);
    printf("a= %f b= %f\n ", a, b);
    printf("these are the original values");
}
/*function definition*/
void change (float a, float b) /*function declarator*/
{
    a +=1000;
    b-=200.5;
}
```

Result of the program

```
a= 100.250000 b= 200.500000
these are the original values
```

We passed `a = 100.25` and `b = 200.5` to the function. In the function, you modified `a` as `1100.25` and `b` as `zero`. However, when you print `a` and `b` in the main function, you get the same old values. This confirms that variables are local to the function unless otherwise specified.

Notice; however, that in the calling function, the type declaration of formal parameters is symbolic and used only to indicate the format. The `int a` has been declared and assigned a value of 0. This has no relationship with `int a` in the function declaration. You could even omit the variable name and declare as `int add(int, int)`. It will still work. Here `a` and `b` have been given for better readability.

This is the reverse in the case of a called function. In the same program, `int c` and `int d` are explicitly defined in function `add`, in the declarator. The variables are used further in the function `add`. This is not the case with the variables in the declaration statement or prototype of the calling function, which will never be used further.

This method of invoking a function is called call by value, i.e., you call the functions with values as arguments.

4.4.3 Function Parameters

You know now that an argument is a parameter or value. It could be of any of the valid types, such as all forms of integers or a float or char. You come across two types of arguments when you deal with functions:

- Formal Arguments
- Actual Arguments

Formal arguments are defined in the function declaration in the calling function. What is actual argument? Data, which is passed from the calling function to the called function, is called the actual argument. The actual arguments are passed to the called function through a function call.

Each actual argument supplied by the calling function should correspond to the formal arguments in the same order. The new ANSI standard permits declaration of the data types within the function declaration to be followed by the argument name. You have used only this type of declaration as it will help students follow the C++ program easily. This helps in understanding one to one correspondence between the actual arguments supplied and those received in the function and facilitates the compiler to verify that one to one correspondence exists and that the right number of parameters have been passed. It may be noted that formal arguments cannot be used for any other purposes. They only give a prototype for the function. Thus, the names of the formal arguments are dummy and will not be recognized elsewhere, even in the functions in which they are defined.

Although, the types of variables in the function declaration, also known as prototype and function call are to be the same, the names need not be the same. You have already used this concept in Example 4.11 after defining `float a` and `float b` in the functions prototype, you first called `add (a, b)`, `add (c, d)` and then `add (sum1, sum2)`. Thus, the formal arguments defined in the prototype and the actual arguments were not the same in two of the above cases.

When the actual arguments are passed to a function, the function notes the order in which they are received and appropriately stores them in different locations. You must note that even if you use `a` and `b` in the `add` function, they will be stored in different locations. They will have no relationship with `a` and `b` of the `main` function. Therefore, even if `a` and `b` are assigned different values in the called function, the corresponding values in the calling function would not have changed.

Actual and Formal Parameters

Parameters are written in the function prototype and function header of the definition. They are local variables which are assigned values from the arguments when the function is called. When a function is called, the values (expressions) that are passed in the call are called the arguments or actual parameters. At the time of the function call each actual parameter is assigned to the corresponding formal parameter in the function definition. For value parameters (default) the value of the actual parameter is assigned to the formal parameter variable. For reference parameters, the memory address of the actual parameter is assigned to the formal parameter.

NOTES

By default argument values are simply copied to the formal parameter variables at the time of the call. This type of parameter passing is called pass by value. It is the only kind of parameter passing used in C language. Thus, parameters define information that is passed to a function and are of following types:

- The actual parameters are parameters that appear in function calls.
- The formal parameters are parameters that appear in function declarations.

A parameter cannot be both a formal and an actual parameter but both formal parameters and actual parameters can be either value parameters or variable parameters. Following example shows how the actual parameters work with `calc_consumer_bill` function:

```
/*Program Example 4.14*/
#include <stdio.h>
#include <stdlib.h>
int main (void);
int calc_consumer_bill (int, int, int);
int main()
{
  int bill;
  int a = 25;
  int b = 32;
  int c = 27;
  bill = calc_consumer_bill (a, b, c);
  printf("The total bill comes to %d\n", bill);
  exit (0);
}
int calc_consumer_bill (int consumer1, int consumer2,
int consumer3)
{
  int total;
  total = consumer1 + consumer2 + consumer3;
  return total;
}
```

In the function `main()` in the above example `a`, `b`, and `c` are actual parameters in the function call `calc_consumer_bill`. On the other hand, the corresponding variables in `calc_consumer_bill`, namely `consumer1`, `consumer2` and `consumer3` are formal parameters because they are appearing in a function definition.

Formal parameters are those variables which do not mean that they are always variable parameters. You can use numbers, expressions or function calls as actual parameters. Here are some examples of valid actual parameters in the function call `calc_consumer_bill`:

```
bill = calc_consumer_bill (25, 32, 27);
bill = calc_consumer_bill (50+60, 25*2, 100-75);
bill = calc_consumer_bill (a, b, (int) sqrt(25));
```

The last line in above example code will use `math.h` header file because `sqrt() function` is the square root function and returns `double value` so it must be cast into an `int` to be passed to `cal_consumer_bill`.

Let us take an example of C programming in which formal and actual parameters are initialized. Following code shows how formal and actual parameters are defined:

/*Program Example 4.15*/

```
int power(int num, int deg)  //formal parameters
{
   int ans=1,i;
   for(i=0;i<deg;i++)
       ans=ans*num;
   return(ans);
}
void main()
{
      int a=3,b=2,c;
      c=power(a,b);   // a,b are actual parameters
      printf("\n %d^%d=%d",a,b,c);
}
```

In the above coding, a function call `power(a,b)` is used in which `a, b` are known as actual parameters as they are passed and in the statement `int power(int num, int deg)` two parameters, such as `num` and `deg` are considered as the formal parameters.

4.4.4 Return Values and their Types

The return data type is declared in the function declaration in the `main()` function or the calling function and the declarator is indicated in the first line of the function definition. If no value is to be returned, the return data type `void` is specified. `Void` simply means NULL or nothing. Therefore, it does not fall in any other data types, such as `integer` or `float` or `char`.

The return value as you have seen is the result of computation in the called function. You return a value, which is stored in a data type in the called function. The return value means that the value, thus stored in the called function is assigned or copied to a variable in the `main()` or calling function. Therefore, to receive the result, a data type should have been declared and preferably initialized in the calling function.

The return statement can be any of the following types:

```
return (sum);
return V1;
return "true";
return 'Z';
return 0;
return 4.0 + 3.0;
```

In some examples, you have returned variables whose values are known when they are returned and in other examples, you return constants. You can even return expressions. If the return statement is not present, it means the return data type is `void`.

You can also have multiple return statements in a function. However, for every call, only one of the return statements will be active and only one value will be returned.

Check Your Progress

1. Write the drawback of `gets` and `puts` functions.
2. What does `strlen ()` function provides?
3. Define a function in the context of C programming.
4. State the significance of function reference.
5. Define the term actual and formal parameters.

4.5 CALLING A FUNCTION BY VALUE AND REFERENCE

We may call a function either directly or indirectly. When we call the function, we pass the actual arguments or values. Calling a function is also known as function reference.

There must be a one to one correspondence between formal arguments declared and the actual arguments sent. They should be of the same data type and in the same order. For example,

```
sum=f1 (20.5, 10); fun4( );
```

Call by Value

In this section, you have been calling functions by passing values. For example, function calls in some of the above programs are as follows:

```
change(a, b);
rev = reverse(num);
```

The values passed to the function change are a & b which are known. Similarly, while calling function `reverse`, we pass `num`. This is called call by value. When you call functions by value, the called functions can return only one value.

Call by Reference

You can enable a function to return more than one value. One way of accomplishing it is by call by reference.

4.6 FUNCTION RECURSION

NOTES

Basic Concepts

The previous section dealt with the concept of a function calling another function, as well as multiple functions, being called by a number of functions. A function calling itself is called **recursion** and the function may call itself either directly or indirectly. This concept is difficult to understand unless explained through examples. Every program can be written without using recursion but the reverse is not true. Some problems; however, are suitable for recursion. For instance, the factorial problem can be solved using recursion as shown in program below:

```
/* Example 4.16*
   To find the factorial of a given number*/
   #include <stdio.h>
   int main()
   {
      int n;
      long int result;
      long int fact(int n);
      printf("Enter the number whose ");
      printf("factorial is to be found\n");
      scanf("%d", &n);
      result=fact(n);
      printf("result=%ld", result);
   }
   long int fact(int n)
   {
      if (n<1) return 0;
      else
         if (n==1) return 1;
         else
            return (n*fact(n-1));
   }
```

Result of the program

```
Enter the number whose factorial is to be found
10
result=3628800
```

Now, let us analyse how the program proceeds. You get an integer `n` from the keyboard. In order to find `factorial n`, you call `fact(n)`, where `fact` is the function for finding the factorial of number `n`. The recursion takes place in function `fact`. Assume that `n=1`. The main function calls `fact(1)`, which will be assigned to `result` in the main function after return from the function. In the function, since `n` is equal to 1, 1 is returned and printed in `main()`.

Next, assume you want to find out the factorial of say, 2 and `fact(2)` is called. In the function `fact`, since `n` is not equal to `1`, `n * fact(n-1)` is returned, i.e., `2 * fact (1)` is returned to `result`. `Result = 2 * fact(1)`. This intermediate result is stored somewhere and can be called stack. Stack is an array which stores values and gives the last element first. The writing into stack is popularly called push and getting information from stack is called pop. You have not defined any stack and therefore, you can assume that the system does this for you. After pushing the intermediate result into stack, the program calls `fact(1)`, which returns 1. Now, the intermediate result is popped and the value of `fact 1` is substituted to get the factorial of 2 as 2.

Now call factorial 5. You call `fact` and get back,

`result = 5 * fact(4)` [1]

Now, `fact(4)` is called to `get 4 * fact(3.)` Substituting this in equation [1], we get

`result = 5 * 4 * fact(3)`

`fact(3)` again is called to get `3 * fact(2)` and so on till we get `fact(1)` which will be returned as 1. Therefore, we get `factorial 5` as `5 x 4 x 3 x 2 x 1`. Such repetitive calling of the same function is called recursion. The calls are recursive calls. The `result` and function `fact` has been declared to be of type `long` so as to take care of large numbers. If the `fact` function were not called repeatedly, the program size would have become very large. Thus, recursion keeps the program size small but understanding recursion is not easy. If the program can be visualized as recursive, it will result in compact code. Recursive functions can easily become infinite loops. Therefore, the functions should have a statement with `if` so that the program will definitely terminate. In the factorial program, assume for a moment that the first statement in fact function is absent. Then we have to end the program only when `n` becomes 1. What will happen, if by chance, `n` is entered as a negative number? The program will get into an endless loop. Therefore, to avoid such eventualities, you can either have a statement as follows:

```
If (n<1)      exit().
```

Or you could do this by the statement `if (n<1) return` 0, as has been done here.

This will ensure that if either 0 or a negative number is entered, you get the factorial as 0 and the program will terminate gracefully.

You should also note that recursive programs simulate the use of stack. You write to the stack and retrieve information from the stack. Writing to stack is called `push` and retrieving or reading or getting value from the stack is known as `pop`. The feature of stack is last in, first out and therefore, you get the value of the data entered last first, as illustrated in the following example.

You `push` or `pop` only through the top of the stack. Assume that you want to `push a, b & c`, one at a time. You `push 'a'`. It will be on top of the stack. When you push `b, a` will be pushed to the next lower location and `b` will occupy the top of the stack. Next, when you `push 'c', 'c'` will occupy the top of the stack, `'b'` one location before and `'a'` one location before `'b'`.

Thus, you can push data till the stack becomes full. If you pop the stack now, it will eject `'c'`, then `'b'` and so on. Thus, you `pop` on a last-in-first-out basis.

Reconsider the factorial program in which you were storing intermediate results in a stack-like manner and popping LIFO. Take the example of finding the factorial of 4. On the first call, you get `4 * fact(3)`. You push this into the stack and in the next call, you get `3 * fact(2)`. Again, you push the intermediate result to stack and evaluate `fact(2)` to `get 2 * fact(1)`. This is also put to stack. Now, you pass `fact 1` and `get fact(1)` as 1, after which you get the value of `fact(2)` by popping `fact(2)` as `2 * fact(1)`. The popping continues till the result is obtained. Such a conceptualization will help you to understand recursion easily.

Implementation of Euclid's `gcd` Algorithm

Euclid's `gcd` algorithm is quite suitable for recursion. The modified algorithm, which uses recursion, is as follows:

Algorithm using Recursion

Main function

```
Step 1  Read two integers m and n
Step 2  Call function gcd (m, n)
Step 3  Print gcd
```

Function `gcd (m, n)`

```
Step 1  if (n==0) return m;
else
return (gcd(n, m%n));
Step 3  End
```

When two integers are received, the main function calls `gcd` function. In the `gcd` function, it is checked whether `n` equals zero. If so, `m` is the `gcd` if not, the function calls `gcd` again by changing the arguments as follows:

```
m=n
n=m%n
```

See the clarity in the above function.

You can easily observe that by using recursion, even the number of steps in the program has gone down. But, it requires a little skill to convert the program into a recursive function. The program implementing the above algorithm is as follows:

```
/*Example 4.17*
  Euclid's GCD*/
  #include<stdio.h>
  int main()
  {
  int m, n;
  int gcd(int m, int n);
  printf("Enter 2 integers\n");
```

```
scanf("%d %d", &m, &n);
printf("GCD of %d and %d=%d", m, n, gcd(m,n));
}
int gcd(int m, int n)
{
if (n==0) return m;
else
return (gcd(n, m%n));
}
```

The program was executed twice and the result of the program is as follows:

Result of the program

```
First Time
Enter 2 integers
12 256
GCD of 12 and 256=4
Second Time
Enter 2 integers
1225 625
GCD of 1225 and 625=25
```

You can even enter the first number to be lower than the second number as executed during the first time. The program works all right because in one iteration, the numbers get reversed namely the first number is larger than the second number after iteration. Thus, recursion is quite suitable for solving this problem.

4.7 ARRAYS AND FUNCTIONS

There is no restriction in passing any number of values to a function; the restriction is only in the return of values from a function. Therefore, arrays can be passed to a function without any difficulty, one element at a time, as follows:

```
#include <stdio.h>
int main()
{
    int a[]={1,2,3,4,5};
    int j;
    int func(int a);
    for (j=0; j<=4; j++)
       func (a[j]);
    ..........
}
int func(int c)
{
    ......
}
```

Here, `func` has been declared as a function passing a single integer. Note here that the declaration or the prototype gives only the format of the parameters passed. The values are only indicative and are not actual values. They are the formal values. Therefore, the parameters declared inside the parentheses act only as a checklist. They cannot be used in the main function elsewhere without actually declaring them on top of the function. But, for this rule, there would have been a conflict between `a[]` which is an array and `a` which is a simple variable. Here, no conflict arises because `a` is not recognized in the main function. It is only a checklist to see that whenever the function calls `func`, an integer has to be passed. If we try to pass a `float`, the compiler will detect an error. This is not so in the case of variables defined in the function declarator above the functions body, as they are recognized as actual names. In this case, `int c` is declared as a variable in `func`. The initial value will be the same as passed by the calling function. Thus, since `a` is used in the function declaration, only one integer can be passed to the function `func`. Actually, the entire array can be passed to a function irrespective of its size, by suitable declaration, as the following example indicates.

```
/*Example 4.18*/
  /* To find the greatest number in an array*/
  #include <stdio.h>
  int main()
  {
      int array[]= {8, 45, 5, 911, 2};
      int size=5, max;
      int fung(int array[], int size);
      max=fung(array, size);
      printf("%d\n", max);
  }
  int fung(int a1[], int size)
  {
      int i, j, maxp=0;
      for (j=0; j<size; j++)
      {
         if (a1[j] > maxp)
         {
            maxp=a1[j];
         }
      }
      return maxp;
  }
```

Result of the program

```
911
```

The objective of Example 8.5 is to find the greatest number in an array. In the program, an array called array is initialized with 5 values as given below:

`int array[]= {8, 45, 5, 911, 2}`; size is declared as 5 and a function called `fung` has been declared. It will pass an array and an integer to the called function. The array size has been kept open and the called function will return an integer. The next statement calls `fung` and passes all elements of the array and an integer 5 equal to `size`. The function gets the actual values and `size=5`. The maximum value in the array is found in the `for` loop and stored in `maxp`. The value `maxp` is returned to the main function and printed there. Thus, the function is called by value.

4.8 STRUCTURES AND FUNCTIONS

A structure is synonymous with a record. A **structure**, similar to a record, contains a number of fields or variables. The variables can be of any of the valid data types. In order to use structures, you have to first define a unique structure. The definition of the record `book`, which you call a structure, is as follows:

```
struct book
{
    char title [25];
    char author [15];
    char publisher [25];
    float price ;
    unsigned year;
};
```

`struct` is a keyword or a reserved word of 'C'. A structure tag or name follows which is `book` in this case. This is not a must but giving a tag to structure will improve the reader's understanding. The beginning of the structure is indicated by an opening brace. Thereafter, the fields of the record or data elements are declared one by one. The variables or fields declared are also called members of the structure. Structure consists of different types of data elements which is different from an array. Let us now look at the members of `struct book`.

The `title` of the book is declared as a string with width 25; similarly, the `author` and `publisher` are arrays of characters or strings of the specified width. The price is defined as a `float` to take care of the fractional part of the currency. The `year` is defined as an `unsigned integer`.

Note carefully the appearance of a semicolon after the closing brace. This is unlike other blocks. The semicolon indicates the end of the declaration of the structure. Thus, you have to understand the following when you want to declare a structure:

(a) `struct` is the header of a structure definition.

(b) It can be followed by an optional name for the structure.

(c) Then the members of the structure are declared one by one within a block.

(d) The block starts with an opening brace but ends with a closing brace followed by a semicolon.

(e) The members can be of any data type.

(f) The members have a relation with the structure; i.e., all of them belong to the defined structure and they have identity only as members of the structure and not; otherwise. Therefore, if you assign a name to the author, it will not be accepted. You can only assign values to `book.author`.

4.8.1 Declaration

The structure definition given above is similar to the prototype in a function in so far as memory allocation is considered. The system does not allocate memory as soon as it finds the structure definition, which is for information and checking consistency later on. The allocation of memory takes place only when structure variables are declared. What is a structure variable? It is similar to other variables. For example, `int I` means that `I` is an integer variable. Similarly, the following is a structure variable declaration:

```
struct book s1;
```

Here `s1` is a variable of type structure book. Suppose, you define,

```
struct book s1, s2 ;
```

This means that there are two variables `s1` and `s2` of type `struct book`. These variables can hold different values for their members.

Another point to be noted is that the structure declaration appears above all other declarations. An example which does nothing but defines structure and declares structure variables is as follows:

```
main ()
{
    struct book
    {
    char title [25];
    char author [15];
    char publisher [25];
    float price;
    unsigned year;
    };
struct book s1, s2, s3 ;
}
```

If you want to define a large number of books, then how will you modify the structure variable declaration? It will be as follows:

```
struct book s[1000];
```

This will allocate space for storing 1000 structures or records of books. However, how much storage space will be allocated for each element of the array? It will be the sum of storage spaces required for each member. In `struct book`, the storage space required will be as follows:

```
title 25 + 1 (for NULL to indicate end of string)
author 15 + 1
publisher 25 + 1
price 4
year 2
```

Therefore, the system allots space for 1000 structure variables each with the above requirement. Space is allocated only after seeing the structure variable declaration.

Look at another example to make the concept clear. You know that the bank account of each account holder is a record. Now, define a structure for it.

```
struct account
{
unsigned number;
char name [15];
int balance ;
} a1, a2;
```

Instead of declaring separate structure variables, such as `struct account a1, a2`; we can use coding as in the example given above. Here, the variables are declared just after the closing brace of the structure declaration and terminated with a semicolon. This is perfectly correct. The declaration of the members of the structure is clear; the balance has been declared as an integer instead of a `float` to make it simple. This means that the minimum transaction is a rupee.

4.8.2 Processing a Structure

The structure variable declaration is of no use unless the variables are assigned values. Here, each member has to be specifically accessed for each structure variable. For example, to assign the account number for variable `a1`, you have to specify as follows:

```
a1. number = 0001 ;
```

There is a dot operator between the structure variable name and the member name or tag.

Suppose, you then want to assign account no.2 to `a2`, it can be assigned as follows:

```
a2. number = 2;
```

If you want to know the address where `a2` number is stored, you can use

```
printf (" %u " , & a2 . number) ;
```

This is similar to other data types. The structure is a complex data type and therefore, you have to indicate which structure variable to which the member belongs as; otherwise, the number is common to all the structure variables, such as `a1, a2, a3`, etc., Therefore, it is necessary to be specific. Assuming that you want to get the value from the keyboard, you can use `scanf()` as follows:

```
scanf ( " % u ", & a1 . number ) ;
```

You can also assign initial values directly as follows:

```
struct account a1 = { 0001, "Vasu", 1000};
struct account a2 = { 0002, "Ram", 1500 };
```

All the members are specified. This is similar to the declaration of initial values for arrays. However, note the semicolon after the closing brace. The `struct a1` will, therefore, receive the values for the members in the order in which they appear. Therefore, you must give the values in the right order and they will be accepted automatically as follows:

```
a1 . number = 0001
a1 . name   = Vasu
a1 . balance = 1000
```

Note too that if the initial values are assigned as above, inside a function, they will be treated as `static` variables. If they are declared before main, they will be treated as global variables.

Let us write a program to create a structure account, open 2 accounts with initial deposits in the accounts. Deposit Rs 1000 to Vasu's account, withdraw 500 from Ram's account and print the balance. The following example demonstrates the above.

```
/*Example 4.19 - structures*
#include<stdio.h>
int main()
{
    struct account
    {
       unsigned number;
       char name[15];
       int balance;
    };
    static struct account a1= {001, "VASU", 1000};
    static struct account a2= {002, "RAM", 2000};
    a1.balance+=1000;
    a2.balance-=500;
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
    a1.number, a1.name, a1.balance);
    printf("A/c No:=%u\t Name:=%s\t balance:=%d\n",
    a2.number, a2.name, a2.balance);
}
```

Result of the program

```
A/c No:=1    Name:=VASU   Balance:=2000
A/c No:=2    Name:=RAM   balance:=1500
```

A simple program was written for a bank transaction. For a deposit, we write

```
a1 . balance = a1 . balance + 1000 ;
```

Therefore, the balance is updated. Similarly, when an amount is withdrawn, the balance is adjusted. However, in practice, the user cannot write a program for each credit and deposit. You will develop a program soon which will not require the user to do programming.

4.8.3 User-Defined Data Type

A structure is also a data type. It is not a basic type defined in C language like `int`, `float`, etc. But, it is defined by the programmer. Hence, it is called as user-defined data type.

Array of Structures

Let us now create an array of structures for the account. This is nothing but an array of accounts, declared with size. Let us restrict the size to 5. The records will be created by using keyboard entry. The program is given below:

```
/*Example 4.20 - to demonstrate structures*
    #include<stdio.h>
    int main()
    {
        struct account
      {
           unsigned number;
           char name[15];
           int balance;
      }a[5];
        int i;
        for(i=0; i<=4; i++)
        {
           printf("A/c No:=\t Name:=\t Balance:=\n");
           scanf("%u%s%d", &a[i].number, a[i].name,
    &a[i].balance);
        }
        for(i=0; i<=4; i++)
        {
           printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
           a[i].number, a[i].name, a[i].balance);
        }
    }
```

Result of the program

```
    A/c No:=   Name:=          Balance:=
    1          suresh          5000
    A/c No:=   Name:=          Balance:=
```

```
2         Lesley         3000
A/c No:=  Name:=         Balance:=
3         Ahmed          5500
A/c No:=  Name:=         Balance:=
4         Lakshmi        10900
A/c No:=  Name:=         Balance:=
5         Thomas         29000
A/c No:=1   Name:=suresh  Balance:=5000
A/c No:=2   Name:=Lesley  Balance:=3000
A/c No:=3   Name:=Ahmed  Balance:=5500
A/c No:=4   Name:=Lakshmi Balance:=10900
A/c No:=5   Name:=Thomas  Balance:=29000
```

The structure array has been declared as a part of structure declaration as `a[5]`. The individual elements of the 5 accounts are scanned and printed in the same order. Note that when you scan a name, you do not give the address but the actual name of the variable as in `a[I].name`, since it is a string variable. Remember this uniqueness. This program basically gets the 5 structures or records pertaining to 5 account holders. Thereafter, the details of the 5 accounts are printed using the `for` statement. The first half of the result was typed by the user and the last 5 lines are the output of the program.

4.8.4 Structure Elements Passing to Functions

Structures can be copied individually, member wise as well as at one go.

For example, let `a3` and `a1` be `struct account`. The following are valid:

```
a3.number = a1.number;
a3.balance = a1.balance;
```

Here, the members of `a1` are copied into `a3`, one by one.

You can also write `a3=a1`; when all the elements of a1 will be copied to `a3`. The latter coding can be used if all the elements are to be copied and the former, if some members are to be copied selectively.

Note that structures cannot be compared as: for example if `(a4 == a2)`. This is not a valid operation.

Let us pass a structure into a function, element by element. This is implemented and shown in Example 4.21.

```
/*Example 4.21 - passing structure element
      to function*/
      #include<stdio.h>
      #include<string.h>
      int main()
      {
          struct account
```

NOTES

```
    {
       unsigned number;
       char name[15];
       int balance;
    };
    static struct account a1= {001, "VASU", 1000};
    int credit(unsigned a, char *n, int d);
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
    a1.number, a1.name, a1.balance);
    a1.balance=credit(a1.number, a1.name, a1.balance);
    printf("A/c No:=%u\t Name:=%s\t new balance:=%d\n",
    a1.number, a1.name, a1.balance);
}
int credit(unsigned a, char *name, int b)
{
    int d;
    unsigned num;
    char *client;
    printf("Enter account number\n");
    scanf("%u", &num);
    if(a==num)
  {
       printf("Enter name in caps\n");
       scanf("%s", client);
       if(strcmp(name, client)== 0)
       {
          printf("enter deposit made\n");
          scanf("%d", &d);
          b+=d;
          return b;
       }
       else
       {
          printf("name does not match\n");
          return b;
       }
    }
  else
  {
       printf("account number does not match\n");
```

```
        return b;
    }
}
```

Result of the program

```
A/c No:=1   Name:=VASU   Balance:=1000
Enter account number
1
Enter name in caps
VASU
enter deposit made
4600
A/c No:=1   Name:=VASU   new balance:=5600
```

Now, look at the program carefully. A function prototype has been declared in `main()` as given below:

```
int credit(unsigned a, char *n, int d);
```

Here, there is no reference to structure at all. A structure `a1` is passed to function `credit` by passing individual elements of a structure. In function `credit`, these values are received. Then, the deposit is entered and added to the balance after checking the correctness of the details of the account. The new balance is returned to `main` and stored in `a1`.

4.8.5 Structure Passing to Functions

Passing each member of the structure is a tedious job. The entire structure can instead be passed to the function making for easy handling. The above example can be altered by passing an entire structure, as follows:

`/*Example 4.22 - passing entire structure to function*/`

```
#include<stdio.h>
struct account
{
  unsigned number;
  char name[15];
  int balance;
};
int main()
{
    static struct account a1= {001, "Vasu", 1000};
    struct account credit(struct account x);
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
    a1.number, a1.name, a1.balance);
    a1=credit(a1);
    printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n",
```

```
    a1.number, a1.name, a1.balance);
}
struct account credit (struct account y)
{
    int x;
    printf ("enter deposit made");
    scanf ("%d", &x);
    y.balance+=x;
    return y;
}
```

Result of the program

```
A/c No:=1    Name:=Vasu    Balance:=1000
enter deposit made 6700
A/c No:=1    Name:=Vasu    Balance:=7700
```

If you want to pass a structure, the called function should also know the structure and hence, the structure has to be declared before the main function. Therefore, structure `account` has been declared as a global structure. The function `credit` is declared with return data type `structure` as follows:

```
struct account credit (struct account x);
```

Thus, you pass and `return struct account`. Then, credit is called by simply passing structure `a1`. In the called program, the deposit is added to the balance and updated. This is returned to the `main()` where the updated record is printed.

4.8.6 Structure Within Structure

You have been nesting `if` statement and loops so far. You can now create structures within structures. Here, a structure defined earlier can become a member of another structure. For example, you can create a structure called deposit using other data types and structure account. The declaration of the basic structure should precede the desired structure as follows:

```
struct account
{
   Unsigned number ;
   char name [15];
   int balance ;
};
struct deposit
{
   struct account ac ;
   int amount;
   int years;
};
```

You can write a program to demonstrate the concept.

/*Example 4.23 - structure within structure*/

```
#include<stdio.h>
int main()
{
    struct account
    {
        unsigned number;
        char name[15];
        int balance;
    };
    struct deposit
    {
        struct account ac;
        unsigned amount;
        int years;
    }d2;
    static struct deposit d1={001, "VASU", 1000, 50000,
3};
    d2=d1; /*structure copy*/
    printf("A/c No:=%u\t Name:=%s\t
Balance:=%d\tdeposit=%u\tterm=%d\n",
    d2.ac.number, d2.ac.name, d2.ac.balance, d2.amount,
d2.years);
}
A/c No:=1   Name:=VASU  Balance:=1000 deposit=50000
term=3
```

You have created the structure `account`. Then, you have created another structure `deposit`. In the deposit structure, `struct account ac` is one of the members and 2 more members, amount and years have been declared.

Next structure `deposit d1` is initialized. The first 3 elements pertain to the members of `struct account` and the last two for `amount` and `years`, respectively.

Now `d1` is copied to `d2` in a simple manner and the deposit details of `d2` are printed.

Whenever members of included structures are accessed, you will find two dots, instead of the usual one dot. This is essential since `d1.name` is invalid. Since name is in `ac`, you have to access it as `d1.ac.name`.

However, nesting can be up to any level. You can create one more level of nesting as shown:

NOTES

```
struct account
{
    unsigned number ;
    char name [15];
    int balance ;
};
struct deposit
{
    struct account ac;
    int amount ;
    int years ;
};
struct loan
{
    struct deposit dep ;
    int amount;
    char date [10];
};
```

You see that `struct deposit` is included as a member of loan.

Let us declare:

```
struct loan loan1;
```

Now, to access loan amount, we have to specify:

```
loan1 . amount
```

To access deposit amount, we have to specify:

```
oan1 . dep . amount
```

To access the balance, we have to specify:

```
loan1 . dep . ac . balance
```

Therefore, usage of the same variable name `amount` has not resulted in a conflict since it has to be seen in which context it is defined. Thus, structures can be used within structures without difficulty.

Structure Containing Arrays

You saw some members of structures being arrays. You used them to represent an array of characters. You can also have integer, `float` arrays as members of structures. For example,

```
struct fixed_deposit
{
unsigned Ac_Number;
char name[25];
double deposit[4];
}rama;
```

Here, you have defined a structure `fixed_deposit` and declared a variable `rama` of the same type. In the structure definition, you have a member `deposit` as an array of double of size 4. This means, you can store 4 deposits of `rama`. This is called array within structure.

You can access the first deposit of `rama` by the following:

```
rama.deposit[0]
```

The last deposit of `rama` can be accessed by:

```
rama.deposit[3]
```

Application of Structures

Structures, as you know, can be used for database management. They can be used in several applications, such as libraries, departmental stores, banking, etc. Structures are also used in C++. The syntax of structures is similar to classes in C++. Structures contain data but classes contain data and functions.

Structures are also used in a variety of other applications, such as:

- Graphics
- Formatting Floppy Discs
- Mouse Movement
- Payroll

Thus, structure is a very useful construct.

4.9 FUNCTION WITH UNION

Union is a variable which holds at a common assigned area different data types of varying sizes at different points in time. Assume that a program, at different points in time of execution uses a `double`, a `float`, an `integer` and a `string`. In the normal course, you would have to allocate memory space for each data type. Assuming that you want to use only one of them at any time and if you do not mind losing the values, you can save a lot of memory space by declaring a common area for storing them. If you use dedicated memory for each variable, the space would remain unutilized most of the time during program execution. This common storage area can be declared as a `union` as shown here:

```
union uname
   {
      double d;
      float f;
      char s[ ];
      int i ;
   } un1 ;
```

See the resemblance between `structure` declaration and `union` declaration. The common properties are :

(i) They can have a name optionally, such as `uname`.

(ii) They can contain members of varying data types.

(iii) The declarations end with a semicolon.

(iv) These `union` members can be accessed in the same way as structure members, as shown:

`union_name.member` or `union_pointer->member`

(v) A `union` can be assigned to another `union`, such as

`un2 = un1;`

where the structure of `un1` along with its members are copied to `un2`.

However, the difference is:

(i) The memory size of the structure variable is the sum of the sizes of its members, whereas in `union`, it is the largest size of its members.

(ii) It is the programmer's responsibility to keep track of which type is currently in use unlike in structure where no member is lost.

(iii) In structures, all members can be initialized, whereas a `union` can only be initialized with a value of the type of its first member.

A program using `union` to store either an `int` value or `float` value is given as follows:

/* Example 4.24 – Demonstrate union */

```
#include <stdio.h>
#include <conio.h>
union sel
{   int n;
    float f;
};
int main()
{   union sel m1;
    void printval(union sel *m1,char type);
    char type ='p';
    char cont ='y';
    while(cont=='y')
    {
      printf("\nWant to enter Integer Or Float (i/f):");
      type=getche();
      if(type=='i')
      {
         printf("\nEnter Integer Value :");
         scanf("%d",&m1.n);
      }
      else
      {
         printf("\nEnter Float Value :");
```

```
            scanf("%f",&m1.f);
         }
      printval(&m1,type);
      printf("\nWant to continue- enter (y/n):");
      cont=getche();
    }
}
void printval(union sel *m1,char type)
{
   if(type=='i')
      printf("Integer Value Is %d\n",m1->n);
   else
      printf("Float Value Is %f\n",m1->f);
}
```

Output of the program

```
Want to enter Integer Or Float (i/f):i
Enter Integer Value :456
Integer Value Is 456
Want to continue- enter (y/n):y
Want to enter Integer Or Float (i/f):3
Enter Float Value :300.30
Float Value Is 300.299988
Want to continue- enter (y/n):n
```

The `union sel` is declared before `main`, with two members `int n` and `float f`. A function `printval` is also declared. Depending upon whether the user wants an `integer` value or `float` value, type is set to `i` or `f`. If type is `i`, integer value is received and if it is `f`, a `float` value is received. The value received is printed in the function `printval`. Note carefully how the pointer to `union` is declared in the function prototype and header. You have to declare `union` whenever you pass a `union` to a function. It is declared as `union sel * m1`. As you would have declared `int * i`, the type here is `union sel`.

If you have perused the result, you would find that the program asks for `float` value, although 3 has been typed instead of `f`. It is because of the design of the program. It will ask for `float` value when a character other than `i` is typed. You can take this as an exercise to correct the program to ask for float value only when `'f'` is typed.

Now, consider another example.

In the case of an employee database, you would like to store either the father's name (in the case of men and unmarried women), the husband's name in the case of married women or the guardian's name. This can be represented as a `union` as shown below:

NOTES

```
Union guardian
{
   char father [10];
   char husband [10];
   char guardian [10];
} e ;
```

You have defined a `union guardian` of employees. By using this, you can save memory space. The memory required will be 10 bytes. Had you considered this as a structure, you would have used 30 bytes and all three variables will be used. This is not required since only one value will be present at any time and `union` is useful in such cases.

You can now define a structure for an employee database with `union` embedded. You have to declare `union` above structure since `union` will be one of the members of the structure. You can define it as follows :

```
union guardian
{  char father [10];
   char husband [10];
   char guardian [10];
} u;
struct employee
{  char name [15];
   float basic;
   char birthdate [10];
   union guardian u;
} emp [2] ;
```

You know how to refer to members of structures, for example, you can refer to the name of the employee's father as: `emp[0].u.father`.

Now look at Example 4.25.

```
/*Example 4.25/
   union within structure*/
   #include <stdio.h>
   int main()
   {
      union guardian
      {
         char *father;
         char *husband;
         char *guardian;
      }u;
      struct employee
      {
```

```
        char *name;
        float basic;
        char *birthdate;
        union guardian u;
    }emp[2];
    int i;
    emp[0].name="RAM";
    emp[0].basic= 20000.00;
    emp[0].birthdate= "19/11/1948";
    emp[0].u.father= "SWAMY";
    emp[1].name="SITA";
    emp[1].basic= 12000.00;
    emp[1].birthdate= "19/11/1958";
    emp[1].u.husband="RAM";
    for(i=0;i<2;i++)
    {
        if( emp[i].basic ==12000)

printf("Name:%s\nbirthdate:%s\nguardian:%s\n",
        emp[i].name, emp[i].birthdate, emp[i].u.husband);
    }
}
```

Result of the program

```
Name:SITA
birthdate:19/11/1958
guardian:RAM
```

The employee details are given in the form of `struct emp` which also contains `union` for `guardian`. All the string variables have been given as pointers to `char`. The `struct` is defined as an array of size 2 and the initial values of `emp[0]` and `emp[1]` are assigned for each member. Next, the data of employees whose basic equals 12000 are printed. Note how the `union` is handled. You will note that in such cases when only one of the three will be entered, there is no need to reserve memory for storing 3 variables. It is enough to store only one of them. A `union` provides a methodology for achieving this.

Check Your Progress

6. What is recursion?
7. Differentiate between a structure and an array.
8. Which keyword is used to declare a structure?
9. What do you understand by members of a structure?
10. What is union? How is it declared?

NOTES

4.10 ANSWERS TO 'CHECK YOUR PROGRESS'

1. `puts` can output only one variable and `gets` can input only one variable in one statement.
2. `Strlen ()` function provides the length of a given string.
3. A function, when declared in a program, is a function prototype which may be declared at the beginning of a main program. A function, on execution, is supposed to do something: either return a value as integer, character or float; or perform some operation. A function consists of two parts, declarator and declaration. A function declaration has the format in which the type of data returned, name of the function and arguments on which the function is to operate.
4. We may call a function either directly or indirectly. When we call the function, we pass the actual arguments or values. Calling a function is also known as function reference.
5. Parameters are written in the function prototype and function header of the definition. They are local variables which are assigned values from the arguments when the function is called. When a function is called, the values (expressions) that are passed in the call are called the arguments or actual parameters. At the time of the function call each actual parameter is assigned to the corresponding formal parameter in the function definition. For value parameters (default) the value of the actual parameter is assigned to the formal parameter variable.
6. When a function calls itself, directly or indirectly, it is known as recursion.
7. A structure contains different data elements whereas an array contains same data elements.
8. Keyword `struct` is used to declare a structure.
9. A structure is like a record that contains fields as name of variables. These variables are known as members.
10. A `union` is a variable that holds a common assigned area for different data types but only one is used at a time. It is declared with a keyword union followed by a name (which is optional) above the opening brace and defining members as done in case of structure.

4.11 SUMMARY

- A string is formed using an array of the basic data type *char.* We can define a string variable as well as string literal. String literals are words surrounded by double quotation marks.
- A *string* is an array of characters, which means that it may contain zero to many characters. A *string* is enclosed within double quotes.
- One can use `scanf` () to receive strings from the screen.

- Strings may contain blanks in between. If we use a `scanf` () function to get the string with a space in between such as "Rama Krishnan", Krishnan will not be taken note of since space is an indicator of the end of entry of the input.
- `printf` can be used to print more than one variable and `scanf` to get more than one variable at a time, in a single statement. However, puts can output only one variable and gets can input only one variable in one statement.
- A function, when declared in a program, is a function prototype which may be declared at the beginning of a main program. A function, on execution, is supposed to do something: either return a value as integer, character or float; or perform some operation.
- A function consists of two parts, declarator and declaration. A function declaration has the format in which the type of data returned, name of the function and arguments on which the function is to operate
- Function definition can be written anywhere in the file with a proper declarator, followed by the declaration of local variables and statements.
- Function definition consists of two parts, namely function declarator or heading and function declarations. The function heading is similar to function declaration but will not terminate with a semicolon.
- We may call a function either directly or indirectly. When we call the function, we pass the actual arguments or values. Calling a function is also known as function reference.
- Parameters are written in the function prototype and function header of the definition. They are local variables which are assigned values from the arguments when the function is called. When a function is called, the values (expressions) that are passed in the call are called the arguments or actual parameters. At the time of the function call each actual parameter is assigned to the corresponding formal parameter in the function definition. For value parameters (default) the value of the actual parameter is assigned to the formal parameter variable.
- When a function calls itself, directly or indirectly, it is known as recursion.
- A structure is synonymous with a record. A structure, similar to a record, contains a number of fields or variables. The variables can be of any of the valid data types.
- A structure contains different data elements whereas an array contains same data elements.
- The allocation of memory takes place only when structure variables are declared.
- Structures find application in departmental stores, graphics, formatting floppy discs and mouse movement, among others.
- Structures can be passed by reference after defining it as a global variable.
- Union is a variable which holds at a common assigned area different data types of varying sizes at different instances of time.

4.12 KEY TERMS

- **`gets:`** This function is used to scan or read a line of text from a standard input (stdin) device, and store it in the string variable.
- **`Strrev():`** It is a function that reverse the given string.
- **Function:** Upon execution, a function either returns a value as integer, character or float; or performs some operation. A function consists of two parts: declarator and declaration.
- **Parameter:** In programming, it is a value passed to a subroutine or function for processing.
- **Actual parameters:** These are the parameters that appear in function calls.
- **Formal parameters:** These are the parameters that appear in function declaration.
- **Recursion:** When a function calls itself, directly or indirectly, it is known as recursion.
- **Structure**: It is the collection of variables of different types under a single name for better handling.
- **Union:** In C they are related to structures and are defined as objects that may hold objects of different types and sizes. They are analogous to various records in other programming languages.

4.13 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Define the term string.
2. Write a program to illustrate the use of gets and puts.
3. State about the function definition.
4. Differentiate between user defined functions and stored procedures.
5. What is parameter?
6. Define the term return values and types.
7. Differentiate between call by value and call by reference.
8. State the significance of arrays and functions in C.
9. Write the rules to declare a structure.
10. How you will access a structure?
11. What are the applications of structures?

Long-Answer Questions

1. Briefly explain the String with the help of examples programs.
2. Discuss briefly how you will declare a string array.

3. Write program to convert the given string to lower case.
4. Explain the concept of function definition. Write a program to get two integers, pass them to a function add, add them in the add function, return the value to the main function and print it.
5. Describe the concept of calling a function. Mention the two methods by which a function is called.
6. Analyse the rules for function giving appropriate relevant examples.
7. Discuss about the parameter and its types with the help of example program.
8. Explain the recursive Euclid's `gcd` algorithm with the help of example programs.
9. Describe the structure and functions giving appropriate relevant examples.
10. Discuss briefly the function with union with the help of example programs.

4.14 FURTHER READING

Gottfried, Byron S. 1996. *Programming with C*, Schaum's Outline Series. New York: McGraw-Hill.

Jeyapoovan T. 2006. *Computer Programming - Theory & Practice*. New Delhi: Vikas Publishing House Pvt. Ltd.

Khurana, Rohit. 2005. *Object Oriented Programming*. New Delhi: Vikas Publishing House Pvt. Ltd.

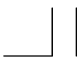
Kanetkar, Yashwant. 2003. *Let Us C*. New Delhi: BPB Publication.

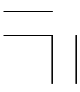
Saxena, Sanjay. 2003. *A First Course in Computers*. New Delhi: Vikas Publishing House Pvt. Ltd.

Subburaj, R. 2000. *Programming in C*. New Delhi: Vikas Publishing House Pvt. Ltd.

Ghosh, Smarajit. 2009. *All of C*. New Delhi: PHI Learning Pvt Ltd.

Bronson, Gary J. 2000. *A First Book of ANSI C*, 3rd edition. California: Thomson, Brooks Cole.





5.0 Introduction
5.1 Objectives
5.2 Introduction to Files in C
5.3 Opening and Closing a File in C
5.4 Input / Output (I/O) Operations on File
5.5 Error Handling and I/O Operator
5.6 Character I/O
5.7 File I/O and FEOF
  5.7.1 FEOF
5.8 STDIN, STDOUT and STDERR
5.9 Random Access to the Files
5.10 Command Line Arguments
5.11 C Preprocessors
5.12 ANSI Editions
5.13 Answers to 'Check Your Progress'
5.14 Summary
5.15 Key Terms
5.16 Self Assessment Questions and Exercises
5.17 Further Reading

In C programming, you include `<stdio.h>`: a file essential for any program to read from a standard input device or to write to a standard output device. It has declaration pointers to three files, namely `stdin, stdout` and `stderr` which means that the contents of these files are added to the program, when the program executes. Hard disk drive or the floppy disk drive as the input/output medium. In day-to-day usage of large applications, the standard input/output is neither convenient nor adequate to handle large volumes of data and hence, the disk drives only serve as Input/Output (I/O) devices. Files for storing data, popularly known as data files. Data files stored in the secondary or auxiliary storage devices, such as hard disk drives or floppy disks, are permanent unless deleted. In contrast, what is written to the monitor is only for immediate use. The data stored in disk drives can be accessed later and modified, if necessary.

The library functions are included in the program during 'Preprocessing'. The word 'Pre-processor' means that processing is carried out before the compilation of the program written by us. We have quite often used `#include` and `#define`. The former includes the contents of files, such as `<stdio.h>` during compilation, while `#define` replaces the symbolic name with actual values before actual compilation. Thus preprocessing is nothing but the operations or processing carried out before actually processing. Let us see in details the various pre-processor directives used in 'C'.

In this unit, you will study about the introduction of file in C, opening and closing a file in C, input/output operations on file, error handling i/o operator, random access to the file, command line argument, preprocessors, macro substitution, ANSI editions and computer control directives.

After going through this unit, you will be able to:

- Understand the basic of file in C
- Explain the opening and closing a file in C
- Define the input/output operations on file
- Analyse the error handling I/O operator
- Describe the random access to the file
- Discuss about the command line argument
- Understand the basic of C preprocessors
- Define the concept of macro substitution
- Elaborate on the ANSI editions
- Explain the computer control directives

In "C", we come across two types of files:

1. Stream Oriented
2. System Oriented

System oriented files or low level files are more closely related to the operating system and hence require more complex programming skills to use them. They may be found to be more efficient than the former in some cases, but we will not discuss them further because of their complexity. Instead, we will discuss stream oriented files only in this chapter.

Stream oriented files are also called standard files. Data can be stored in standard files in two ways as given below:

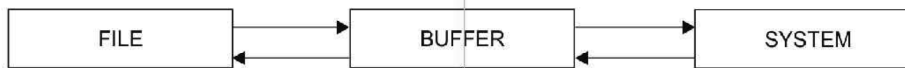
- Storing characters or numerals consecutively. Each character is interpreted as an individual data item.
- The data items are arranged in blocks in an unformatted manner. Each block may be an array or a structure.

Let us see how disk I/O is organized. If the file is stored in a floppy or hard disk drive, the following actions are involved in reading from the file:

- Finding out where the data is
- Positioning the head over the correct location on the disk

- Reading the content
- Transmitting to the main memory.

Similar activities are involved in writing to a disk as well. If the computer, or more specifically the operating system which handles files in a computer, reads or writes one character at a time comprising of the 4 steps listed above, then it will be uninteresting, and the response will be too slow. It may cause wear out of the storage system quickly. Therefore it would be better to receive large volumes of data or characters to a buffer in the computer system and then perform whatever actions are dictated by the program. Similarly, all characters to be written can be collected in a buffer and written on to the disk, either after the buffer is full, or after the operation is completed. This will minimize the overheads required for read or write operations. The buffer is also memory which is used to store data temporarily without the knowledge of the user. In fact we created a buffer and stored values into it before printing them using the *sprintf* function. The concept is similar here also. This is a good practice. Therefore the characters are read or written through a buffer assigned by the system. The operations are essentially performed as depicted pictorially below:



What is a file pointer? It is a pointer to a file, just like other pointers to arrays, structures etc. It points to a structure that contains information about the file. The information connected with a file is as given below:

- Location of the buffer.
- The position in the file of the character currently being pointed to.
- Whether the file is being read or written.
- Whether an error has occurred or the end of the file has been reached.

We do not need to know the details of these because `stdio.h` handles it elegantly. There is a structure with `typedef FILE` in `stdio.h` which handles all file related tasks as above, whether it is in the floppy or the hard disk drive. Therefore in order to use a file without difficulty, we have to include `stdio.h` and declare a file pointer which points to `FILE` as given below:

```
FILE  * fp;
```

Therefore the file pointer points to a structure which contains information about the file management functions. When we open a file, and the opening of the file is successful, the file pointer will point to the first character in the file. In other words, the file gets opened and loaded to the buffer. NULL is a macro defined in `stdio.h` which indicates that file open has failed. Therefore, when file open is successful the file pointer will point to the address of the buffer which will be a non-zero integer value. If not, the file pointer will get a value of NULL which is 0.

The file pointer will point to the next character after the first one is fetched or copied on to the system. The structure `FILE` keeps track of where the pointer remains at any point in time after opening the file. It keeps track of which files are

being used. It also knows whether the file is empty, the end of the file has been reached or an error has occurred. We don't have to worry about file management tasks once a file pointer has been declared in our program to point to `FILE`. Since `FILE` is known to `<stdio.h>`, we do not have to bother about it. This declaration of structure `FILE` has relieved the programmer from most of the mundane jobs.

---

are formats that save data for future use. Any file has to be opened for any further processing, such as reading, writing or appending, i.e., writing at the end of the file. The characters will be written or read one after another from the beginning to the end, unless otherwise specified. You have to open the file and assign the file pointer to take care of further operations. Hence, you can declare,

```
FILE * fp;
fp = fopen ("filename", "r");
```

`Filename` is the name of the file, which you want to open. You must give the path name correctly so that the file can be opened. `'r'` indicates that the file has to be opened for reading purposes.

`fp = fopen ("Ex1.C", "r");` will enable opening file `Ex1.C`.

Therefore, the arguments to `fopen()` are the name of the file and the mode character. Obviously `w` is for write, `a` for append, i.e., adding at the end of the file. If these characters are specified, the operations as indicated can be performed after opening the file. It is, therefore, essential to indicate the operations to be performed before opening the file. When the file is opened in the `'w'` mode, the data will be written to the file from the beginning. This means that if the named file is already in existence, the previous contents will be lost due to overwriting. If the file does not exist, then a file with the assigned name will be opened. When the append mode is specified, the writing will start after the last entry, or in other words, previous contents of the file will be preserved.

`FILE` provides the link between the operating system and the program currently being executed. `FILE` is a structure containing information about the corresponding files, including information such as:

- The Location of the File
- The Location of the Buffer
- The Size of the File

After the command is executed in the read mode, the file will be loaded into the buffer if it is present. If the file is absent, or the file specification is not correct, then the file will not be opened. If the opening of the file is successful, the pointer will point to the first character in the file, and if not, NULL is returned, meaning that the access is not successful. The `fopen()` function returns a pointer to the starting address of the buffer area associated with the file and assigns it to the file pointer, `fp` in this case.

After the operations are completed, the file has to be closed. The syntax for closing file is given below:

```
fclose(filepointer);
```

`fclose()` also flushes or empties the buffer. The function `fputc()` performs putting one character into a file. If for every `fputc()`, the computer prints a character to a file, then it will get tired. Therefore, it collects all the characters to be written onto a file in the buffer. When the buffer is full or when `fclose()` is executed, the buffer is emptied by writing to the hard disc drive in the assigned file name.

You can open files in the text mode or the binary mode. In the binary, data will be stored in the binary form and the storage space will be equal to the number of bytes required for the storage of various data types. In the text mode, they will be stored as alphanumeric characters. If you require to use the file in the binary mode, you must use `'rb'` for reading, `'wb'` for writing, and `'ab'` for appending. If you want to store data in the text mode, you have to append *t* to the mode character as `'rt'`, `'wt'`, `'at'`, etc. Since the default is in the text mode, `t` will be assumed if nothing is specified after the mode character. Therefore, mode `'w'` means opening a text file for writing.

The difference between opening files in the binary mode and the text mode are given in Table 5.1.

Table 5.1 *Difference between Binary Mode and Text Mode Operations*

Text Mode	Binary Mode
New line character (`\n`) is converted to CR|LF combination while writing to file.	No such conversion.
While reading, CR|LF is converted back to `\n`.	Does not arise.
A special character is inserted at the end of the file. While reading the file, EOF is detected.	There is no such arrangement.
Text mode needs more than the 2 bytes for storing an integer, since it treats each digit as a character. e.g., 30,000 needs 5 bytes.	In binary mode the numbers will be stored in the specified width. 30000 needs 2 bytes only.

Therefore, binary files and text files are to be processed taking into account their properties as above, although the file could be opened in any mode. The file I/O functions, such as `fgetc`, `fputc`, `fscanf`, `fprintf`, `fgets`, `fputs` are applicable to the operations in any of the modes.

The files can be used to store employee records using structures in a payroll program. Book records can be stored in a file in a library database. Inventories

can be stored in a file. However, storing all these in the text mode will consume more space on the file. Hence, the binary mode can be used to create the files. Some files cannot be stored in the text mode at all, such as executable files.

The C programming language provides many standard library functions for file input and output operations. These functions define the C standard library header `<stdio.h>`. The functionality descends from a 'Portable I/O Package' written by Mike Lesk at Bell Labs in the early 1970s, which officially became part of the UNIX operating system in Version 7.

The I/O (Input/Output) functionality of C is low-level by modern standards; C abstracts all file operations into operations on streams of bytes, which may be 'Input Streams' or 'Output Streams'. In addition, C has no direct support for random access data files to read from a record in the middle of a file, hence the programmer has to create a stream to find from the middle of the file, and then read bytes in sequence from the stream.

In addition, the C library uses streams to operate with physical devices, such as keyboards, printers, terminals or with any other type of files supported by the system. Streams are an abstraction to interact with these in a uniform way. All streams have similar properties independent of the individual characteristics of the physical media they are associated with.

Most of the C file Input/Output functions are defined in `<stdio.h>`, which contains the standard C functionality but in the `std` namespace. C language provide support for opening a file, closing a file, reading data from a file and writing data to a file through a set of standard library functions.

`fopen` Opens a file with a non-Unicode filename on Windows and possible UTF-8 filename on Linux.

`freopen` Opens a different file with an existing stream.

`fflush` Synchronizes an output stream with the actual file.

`fclose` Closes a file.

`setbuf` Sets the buffer for a file stream.

`setvbuf` Sets the buffer and its size for a file stream.

`fwide` Switches a file stream between wide-character I/O and narrow-character I/O.

`fread` Reads from a file.

`fwrite` Writes to a file.

clearer Clears errors.

feof Checks for the End-Of-File (EOF).

ferror Checks for a file error.

remove Erases a file.

rename Renames a file.

tmpfile Returns a pointer to a temporary file.

tmpnam Returns a unique filename.

Fundamentally, a file is considered as a container in computer storage devices which is typically used for storing data. The file handling in C is done using the standard I/O functions, such as `fprintf()`, `fscanf()`, `fread()`, `fwrite()`, `fseek()`, etc. For reading and writing to a text file, we use the functions `fprintf()` and `fscanf()`.

Let us look at a program to write numbers to a binary file using `fprintf()` and then read from the file using `fscanf()`.

Example 5.1 - Writing digits to a binary file and then reading.

```
#include <stdio.h>
int main()
{
    int alpha,i;
    FILE *fp;
    fp=fopen("ss.doc", "wb");
    if(fp==NULL)
       printf("could not open file\n");
    else
    {
       for (i=0; i<=99; i++)
          fprintf(fp," %d", i);
          fclose(fp);
          /*now read the contents*/
          fp=fopen("ss.doc", "rb");
       for (i=0; i<100; i++)
       {
          fscanf(fp,"%d", &alpha);
          printf(" %d", alpha);
       }
       fclose (fp);
    }
}
```

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99

(a) The file `ss.doc` is opened in the binary mode for writing. If the opening of the file was not successful, the message will be displayed and program execution will stop. If successful, the program will enter the `else` block. Numbers 0 to 99 are generated one after another and written then and there to the file using the `fprintf()` function. There should be space before `%d` as shown in `fprintf()`, otherwise the program may not work.

(b) The file is closed using `fclose()`.

(c) Now the same file is opened for reading in the binary mode.

(d) Next the text is scanned using `fscanf()`, one at a time, and written on the monitor using simple `printf()`. The difference between `scanf()` and `fscanf()` is the specification of the file pointer before the format specifier.

(e) After reading, the file is closed.

The result of the program is read from the file `ss.doc` and printed on the monitor.

In all the programs involving files, a similar check to see that file opening was successful should be made. For the sake of improved readability, this statement has been skipped in the rest of the programs.

Let us look at one more example of writing, appending and then reading one integer at a time with the help of the `for` loop. Look at the following program.

Example 5.2 - writing, then appending digits to a file and then reading.

```
#include <stdio.h>
int main()
{
    int alpha,i;
    FILE *fp;
    fp=fopen("ss.doc", "wb");
    for (i=0; i<20; i++)
        fprintf(fp," %d", i);
        fclose(fp);
        fp=fopen("ss.doc", "ab");
    for (i=20; i<100; i++)
        fprintf(fp," %d", i);
        fclose(fp);
    /*now read the contents*/
    fp=fopen("ss.doc", "rb");
```

```
    for (i=0; i<100; i++)
    {
       fscanf(fp,"%d", &alpha);
       printf(" %d", alpha);
    }
    fclose (fp);
}
```

The result will be same as Example 5.1.

A binary file is opened in the write mode, and digits from 0 to 19 are written on to the file. The file is then closed using `fclose()`. The same file is opened in the append mode again, and numbers from 20 to 99 are appended to the file. After the file is closed, the file is opened in the read mode. The contents of the file are read using `fscanf()` and written to the monitor. Remember to leave a space before `%d` in `fprintf()` as otherwise you may have a problem. The file is closed again. We have used the same file pointer, since at any time only one file is in use. If more than one file is to be kept open simultaneously, it may call for multiple pointers.

After having worked with the formatted I/O, let us now look at the unformatted I/O. If you want to read a character from the file, you can use the `getc()` or `fgetc()` functions. If `alpha` is the name of the character variable, you can write,

```
    alpha = fgetc (fp);
```

This means the character pointed to by `fp` is read and assigned to `alpha`. You can go to the help screen of the `'C'` language system to get more details as well as search for help on any of the library functions. The help screen gives the syntax of the functions and also provides examples in which the function or command is used. Even after reading this book or any other book on 'C', you will not be able to use all the functions. Hence, the best way is to take the help from the help screen whenever other functions are to be used.

`fgetc()` reads the character pointed to by `fp`. It then increments `fp` so that `fp` points to the next character. We can keep on incrementing `fp` till the end of file, i.e., end of data is reached. When a file is created in the text mode, the system inserts a special character at the end of the text. Therefore, while reading a file, when the last character has been read and the end of the file is reached, EOF is returned by the file pointer. The following program reads one character at a time till EOF is reached from an already created text file, `ss.doc`. The program is implemented using the `do...while` statement.

Example 5.3 - reading characters from a file

```
#include <stdio.h>
int main()
{
```

```
    int alpha;
    FILE *fp;
    fp=fopen("ss.doc", "r");
    do
    {
       alpha=fgetc(fp);
       putchar(alpha);
    } while(alpha!=EOF);
    fclose (fp);
}
```

Since file `ss.doc` is read, the output will be same as Example 13.1, if no change has been made in the file. If we were to read from a binary file, EOF may not be recognized. Therefore, a counter can be set up to read a predefined number of characters as given in the previous examples.

File copy can be achieved by reading one character at a time and writing to another file either in the write mode or the append mode. Here it is proposed to read from a file and write to two different files, one in the write mode and another in the append mode. This means you have to open three files in the following manner:

```
FILE * fr, *fw; *fa;
```

You can assign three file pointers as given above. Three files are then opened. You can use any name for the file pointers and there may be as many file pointers as the number of files to be used. The program is as follows:

Example 5.4 - reading from file ss, writing to file ws and appending to file as, all at a time.

```
#include <stdio.h>
int main()
{
    int alpha;
    FILE *fr,*fw, *fa;
    fr=fopen("ss.doc", "r");
    fw=fopen("ws.doc", "w");
    fa=fopen("as.doc", "a");
    do
    {
       alpha=fgetc(fr);
       fputc(alpha, fw);
       fputc(alpha, fa);
       putchar(alpha);
    }while(alpha!=EOF);
    fclose (fr);
```

```
        fclose (fw);
        fclose (fa);
    }
```

After opening the three files, `alpha()` gets the character, which is written to both the files using `fputc()`, and the character is also displayed on the screen. This is continued till EOF is received in `alpha` from `ss.doc`, the source file.

Finally, the files are closed. Verify that your program has worked alright.

Since, you are also writing to the monitor, in addition to writing and appending to files, the program output appears as follows.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99
```

There are some more mode specifiers with `fopen` like `r+`, `w+` and `a+`, which are given in the table below:

`r+`	Opens an already existing file for reading and writing.
`w+`	Opens a new file for writing as well as reading.
`a+`	Opens an already existing file for appending and reading.

You have discussed writing to and reading from a file, one character at a time, using both the unformatted and formatted I/O (Input/Output) for the purpose. You can also read one line at a time. This is enabled by the `fgets()` function. This is a standard library function with the following syntax:

```
Char * fgets (char * buf, int max line, FILE * fp);
```

`fgets()` reads the next line from the file pointed to by `fp` into the character array `buf`. The line means characters up to `maxline` –1, i.e., if `maxline` is 80, each execution of the function will permit reading of up to 79 characters in the next line. Here 79 is the maximum, but you can even read 10 characters at a time, if it is specified.

```
fgets(alpha, 10, fr);
```

Here `alpha` is the name of the buffer from where 10 characters are to be read at a time. The file pointer `fr` points to the file from which the line is read, and the line read is terminated with NULL. Therefore, it returns a line if available and NULL if the file is empty or an error occurs in opening the file or reading the file.

The complementary function to `fgets()` is `fputs()`. Obviously `fputs()` will write a line of text into the file. The syntax is as follows:

```
int fputs (char * buf , file * fp );
```

The contents of array `buf` are written onto the file pointed to by `fp`. It returns EOF on error and zero otherwise. Note that the execution of `fgets()` returns a line and `fputs()` returns zero after a normal operation.

The functions `gets()` and `puts()` were used with `stdio`, whereas `fgets()` and `fputs()` operate on files.

We can write a program to transfer two lines of text from the buffer to a file and then read the contents of the file to the monitor. This is shown in Example 5.5.

Example 5.5 - writing and reading lines on files

```
#include <stdio.h>
#include<string.h>
int main()
{
    int i;
    char alpha[80];
    FILE *fr,*fw;
    fw=fopen("ws.doc", "wb");
    for(i=0; i<2; i++)
    {
        printf("Enter a line up to 80 characters\n");
        gets(alpha);
        fputs(alpha, fw);
    }
    fclose(fw);
    fr=fopen("ws.doc", "rb");
    while
    (   fgets(alpha,20, fr)!=NULL)
        puts(alpha);
        fclose (fr);
}
```

Note carefully the `fgets()` statement. Here, `alpha` is the buffer with a width of 80 characters. Each line can be up to 80 characters and two lines are entered through `alpha` to `ws.doc`. Later on, 20 characters are read into `alpha` at a time from same file till NULL is returned.

```
Enter a line upto 80 characters
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Enter a line upto 80 characters
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
aaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaabbbb
bbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb
bb
```

More than 20 numbers of `a & b` were written on to the file. However, since we have specified reading 20 characters at a time. The output appears in 6 lines. Had we specified reading more characters at a time, the number of reads would have reduced. You can try this yourself.

Thus, you can read and write one line at a time.

When an error is encountered during file processing, the program execution will be terminated and error messages will be displayed. Following measures may be adopted to identify or avoid errors during file processing.

(i) `ferror()` function can be used to detect any error during file accessing. This function will return a zero when there is no error, otherwise it will return a non-zero number.

```
FILE *fptr;
if(ferror(fptr) == 0)
   printf("\n The file is available for processing");
else
   printf("\n Error in accessing the file!!!");
```

(ii) To verify whether a file exists in the disk, the following definitions will help.

```
if(fptr == NULL)
   printf("\n No content or file does not exist");
```

(iii) When data in a file may be exhausted, end of file condition will be encountered. Use the following definition to display end of file condition.

```
if(feof(fptr) == EOF)
   printf("\n End of file is reached");
```

Note that EOF is a macro which specifies the end of file condition.

Entering an invalid file name or trying to write data to a write protected disk are some of the other reasons for errors during file processing.

1. What are the two types of files?
2. Write the two ways of storing data in standard files.
3. Write the syntax for creating and opening a file.
4. What are the two modes of opening a file?
5. How is a file copied?
6. What happened when an error occur during file processing?

Character I/O (Input/Output) refers to the file handling concept in which file is read and written once it is opened. You can open a file for writing, close it and reopen it

for reading, then close it, and open it again for appending, etc. Each time you open the file, you could use the same file pointer or you could use a different one. The file pointer is simply a tool that you use to point to a file and you decide what file it will point to. The two prime operations are handled in character I/O and they are described as follows:

Once the file has been opened for reading using `fopen()` function in C language, the content of file is brought into memory and a pointer points to the very first character. To read the file's contents from memory, a function called `fgetc()` is coded in the following way:

```
ch = fgetc(fp) ;
```

The C function `fgetc( )` reads the character from current pointer position, advances the pointer position so that it now points to the next character and returns the character that is read, which is collected in the variable `ch`. Once the file has been opened, you refer to the file through the file pointer `fp`. The function `fgetc()` reads character until it reaches to the end of file, which is signified by a special character in the file, while the time of creating the file. This character can also be generated from the keyboard by typing [CTRL+ Z] simultaneously. While reading from the file, when `fgetc()` function encounters this special character, instead of returning the character that it has read, it returns the macro `EOF`. The `EOF` macro is defined in the file `<stdio.h>`. In place of the function `fgetc()`, the macro `getc()` is used with the same effect. The file is opened by using the function `fopen()` and if file is opened in `"r"` mode, it reads the content of file. Similarly, while opening the file for writing, `fopen()` sometimes fail to open due to a number of reasons. One of the reasons could be that disk space sometimes might be insufficient to open a new file or the disk may be 'write–protected'. Therefore, it is important for any program that accesses disk files to check whether a file has been opened successfully before trying to read or write to the file. If the file opening fails due to any of the several reasons mentioned above, the `fopen()` function returns a value `NULL`, which is defined in `"stdio.h"` as `#define NULL 0`. The following program shows how file is opened in read mode with the help of `"r"` mode and pointer:

```
#include "stdio.h"
main()
{
        FILE *fp ;
        fp =fopen("PR1.C", "r") ;
        if(fp == NULL)
        {
              puts("Cannot open file") ;
              exit() ;
        }
```

Various modes are available in C for file handling operations. The `"r"` mode is one of the several modes in which you can open a file for reading purpose. The following piece of code shows how various functions are used to read a character in a specified file:

```
#include <stdio.h>
/* character input */
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
```

In the above piece of code, the next character is obtained from the input stream in each case. It is treated as an `unsigned char` and then converted to an `int`, which is the return value. The constant `EOF` is returned, and the end-of-file indicator is set for the associated stream. On error, `EOF` is returned, and the error indicator is set for the associated stream. Successive calls will obtain characters sequentially. The functions, if implemented as macros, might evaluate their stream argument more than once. There is also the supporting `ungetc` routine, which is used to push back a character on to a stream, causing it to become the next character to be read. This is not an output operation and can never cause the external contents of a file to be changed. The functions, such as `fflush, fseek` or `rewind` operation on the stream work between the pushback and the read operations and hence will cause the pushback to be forgotten. Only one character of pushback is guaranteed and attempts to pushback `EOF` are ignored. Pushing back a number of characters then read or even discard them leaves the file position indicator unchanged. The file position indicator is decremented by every successful call to `ungetc` for a binary stream, but unspecified for a text stream or a binary stream which is positioned at the beginning of the file. Table 5.1 displays the list of all possible modes in which a file can be opened:

Table 5.1 *List of Modes in File Handling*

`r`	This mode is used to read the content of a file. If the file exists, loads it into memory and sets up a pointer, which points to the first character in it. If the file does not exit it returns NULL.
`w`	This mode is used to write to the file. If the file exists, its contents are overwritten. If the file does not exit, a new file is created. It returns NULL, if file is unable to open.
`a`	This mode is used to append the content at the end of the file. If the file exists, loads it into memory and sets up a pointer which points to the first character in it. If the file does not exist, a new file is created. It returns NULL, if file is not opened.
`r+`	This mode is used to read the existing content of a file and also modify the content of the file that already exists in the file. If it exists, loads it into memory and sets up a pointer which points to the first character in it. If file doesn't exist it returns NULL.
`w+`	This mode is used to write new content of a file, read and modify the existing content of a file. If the file exists, its contents are destroyed. If the file does not exist a new file is created. It returns NULL, if unable to open file.
`a+`	This mode is used to read and append both with the existing content of a file. It also appends new contents to end of file, but it can not modify the existing contents of the file. If the file exists, loads it in to memory and sets up a pointer which points to the first character in it. If the file does not exist, a new file is created. Returns NULL, if unable to open file.

After knowing all types of mode, you are able to write a C program in which you can read the charter of specified file. When you run the following program, you will not get any output to the monitor because it does not generate output. After running it, look at your directory for a file named TEXT_FILE.TXT and type it, i.e., where your output will be. The following program reads characters from the file:

```
#include <stdio.h>
void main( )
   {
     FILE *fp;
     char c;
    fp=fopen("TEXT_FILE.TXT", "r");
     //File is
     if (fp == NULL)
       printf("File does not exist\n");
else
   {
do
{
    c = getc(fp);  /* get one character from the file */
     putchar(c);    /* display it on the monitor */
}while(c != EOF);   /* repeat until EOF (end of file)*/
   }
  fclose(fp);
}
```

In the above program, it is checked whether the specified file exists or not, and if it does, the main body of the program is executed. If the file does not exist a message is printed and then loop come out from the main function. It is also possible that if the file does not exist, the system will set the pointer equal to NULL which we can test. The main body of the program is limited within 'do … while' loop in which a single character is read from the file and output to the display unit until an End Of File (EOF) is detected from the input file. The file is then closed and the program is terminated. The variable returned from the getc function is a character, so you can use a char variable for this purpose. There is a problem that could develop here if you use an unsigned char because C usually returns a minus one for an EOF, i.e., an unsigned char type variable and is also not capable of containing the content. An unsigned char type variable can only have the values of 0 to 255, so it will return a 255 for a minus one in C. After compiling and running the program name of TEXT_FILE.TXT can be changed and run the program again to see that the NULL test actually works as stated in the program. Reading character in a specified file, the data is entered through the keyboard and the program writes it character by character, to the file input until it reaches to EOF.

The fputc() function is similar to the putch() function, in the sense that both output the characters. The putch() function always writes to the visual display

unit, whereas `fputc()` writes to the file. Here, file is the same file, which is signified by `ft` pointer. The writing process continues till all characters from the source file have been written to the target file, following which the while loop terminates. The function `fgetc()` reads characters from a file. The function `fputc()` also writes characters to a file. The usage of these character I/O functions is completed in the way so that you can copy the contents of one file into another, as declared in the following program. The following program takes the contents of a text file and copies them into another text file, character by character:

```
#include <stdio.h>
Void main()
{
        FILE *fs, *ft ;
        char ch ;

        fs = fopen("pr1.c", "r");
        if (fs == NULL)
        {
              puts("Cannot open source file");
               exit( ) ;
        }
        ft = fopen ("pr2.c", "w") ;
        if(ft == NULL)
        {
              puts("Cannot open target file");
               fclose(fs);
               exit() ;
        }
        while (1)
        {
               ch = fgetc(fs);

               if(ch == EOF)
                  break;
               else
                  fputc ( ch, ft )  ;
        }
        fclose(fs) ;
        fclose(t);
}
```

Function `fclose` is used in the above coding because once you have finished reading the content of a specified file, you need to close the file in the following way:

```
fclose(fp);
```

The `fclose()` function deactivates the file and hence it can no longer be accessed using `getc()`. The following program shows how file is opened for reading and writing operations simultaneously:

```
#include <stdio.h>
void main()
{
file *f1;
printf("Data input output");
    f1=fopen("Input", "w"); /*Opening file in writing
mode*/
 while((c=getchar())!=EOF)
/*get a character from key board*/
      putc(c,f1); /*write a character to input*/
      fclose(f1); /*close the file input*/
      printf("nData output\n");
  f1=fopen("INPUT", "r"); /*Reopening the file in reading
mode*/
  while((c=getc(f1))!=EOF)
      printf("%c",c);
      fclose(f1);
}
```

We are familiar with reading and writing. So far we were reading from and writing to standard input/output. Therefore, we used functions for the formatted I/O (Input/ Output) with `stdio`, such as `scanf()` and `printf()`. We also used unformatted I/O such as `getch()`, `putch()` and other statements. When dealing with files, there are similar functions for I/O. The functions `getc()`, `fgetc()`, `fputc()` and `putc()` are unformatted file I/O functions similar to `getch()` and `putch()`. We will consider the formatted file operations in this section. When it pertains to standard input or output, we use `scanf()` and `printf()`. To handle formatted I/O with files, we have to use `fscanf()` and `fprintf()`.

We can write numbers, characters, etc. to the file using `fprintf()`. This helps in writing to the file neatly with a proper format. In fact, any output can be directed to a file instead of the monitor. However, we have to indicate which file we are writing to by giving the file pointer. The following syntax has to be followed for `fprintf()`:

```
    fprintf (filepointer, "format specifier", variable
names);
```

We are only adding the file pointer as one of the parameters before the format specifier. This is similar to `sprintf()`, which helps in writing to a buffer. In the case of `sprintf()`, buffer was a pointer to a string variable. Here, instead of a pointer to a string variable, a pointer to a file is given in the

fprintf() statement. Like the string pointer in sprintf(), the file pointer should have been declared in the function and should be pointing to the file.

Before writing to a file, the file must be opened in the write mode. You can declare the following:

```
FILE * fp ;
fp = fopen ("filename", "wb");
```

You have to write wb within double quotes for opening a file for writing in the binary mode. Therefore, fopen() searches the named file. If the file is found, it starts writing. Obviously the previous contents will be lost. If a file is not found, a new file will be created. If unable to open a file for writing, NULL will be returned.

We can also append data to the file after the existing contents. In this manner, we will be able to preserve the contents of a file. However, when you open the file in the append mode, and the file is not present, a new file will be opened. If a file is present, then writing is carried out from the current end of the file. After writing is completed either in the write mode or the append mode, a special character will be automatically included at the end of the text in case of text files. In case of binary files, no special character will be appended. This can be read back as EOF. Usually it is – 1, but it is implementation-dependent. Hence, it is safer to use EOF to check the end of the text files.

The function feof() returns a non–zero value if the end of the given file stream has been reached. The syntax for feof() function is written as follows:

```
#include <stdio.h>
int feof( FILE *stream );
```

Function feof() checks for end of file on the indicated file. It is implemented as a macro. When the data is being read from a terminal, the terminal user can generate an end of file condition by entering a line consisting of [CTRL-\] and here you need to hold down the CTRL key and then press the backslash, followed immediately by a carriage return. Table 5.3 how feof() function is used in various ways:

Table 5.3 *Usage of feof() Function*

int	feof (FILE *stream)
int	feof_unlocked (FILE *stream)
int	ferror (FILE *stream)
int	ferror_unlocked (FILE *stream)
int	fileno (FILE *stream)
int	fileno_unlocked (FILE *stream)

In Table 5.3 function clearerr clears the end-of-file and error indicators for the stream pointed to by stream. The function feof tests the end-of-file indicator for the stream pointed to by stream, returning non-zero if it is set. The end-of-file indicator can only be cleared by the function clearerr. The function ferror tests the error indicator for the stream pointed to by stream, returning non-zero if it is set. The error indicator can only be reset by the clearerr function.

The function `fileno` examines the argument `stream` and returns its integer descriptor. The `clearerr_unlocked`, `feof_unlocked`, `ferror_unlocked` and `fileno_unlocked` functions are equivalent to `clearerr`, `feof`, `ferror` and `fileno` respectively, except that the caller is responsible for locking the stream with `flockfile` before calling them. These functions may be used to avoid the overhead of locking the stream and to prevent races when multiple threads are operating on the same stream. The `feof` function has been used to test for EOF. The following program is used to open the specified text file '`Text_File.txt`' in the current directory and read each character in it until the EOF symbol is encountered.

```
#include <stdio.h>
int main()
{
  FILE *in;
  if (in = fopen("Text_File.txt", "rt"))
   {
    for (char c; !feof(in); fscanf(in, "%c", &c));
    fclose(in);
   }
  return 0;
}
```

The result of the above program is that it opens '`Text_File.txt`' file and checks the content until it reaches EOF. This function should not fail and do not set the external variable `errno`.

STDIN, STDOUT STDERR

We have been including program files in our programs. We have been including <stdio.h> ritually in every file. This file is essential for any program to read from standard input device or to write to the standard output device. The file `<stdio.h>` has declarations to the pointers to three files, namely `stdin`, `stdout` and `stderr`. It means that the contents of these files are added to the program when the program executes. Each of the files performs an essential task as given below:

(a) `stdin` facilitates usage of the standard input device for program execution, and normally points to the keyboard which is the default input device.

(b) `stdout` facilitates the usage of a standard output device where program output is displayed, and points to the video monitor.

(c) `stderr` facilitates sending error messages to the standard device that is again the monitor.

The `stdin`, `stdout` and `stderr` are pointers or file pointers and are declared in `<stdio.h>`. So far we have been using `stdin` and `stdout` for input and output. In this unit we will learn to use either the hard disk drive or

the floppy disk drive as the input/output medium. In day to day usage of large applications the standard input/output is neither convenient nor adequate to handle large volumes of data and hence the disk drives only serve as I/O devices. We will discuss the usage of files for storing data, popularly known as data files. Data files stored in secondary or auxiliary storage devices such as hard disk drives or floppy disks are permanent unless deleted. In contrast, what is written to monitor is only for immediate use. The data stored in disk drives can be accessed later and modified, if necessary.

The standard streams are for input, output and error output. By default, standard input is read from the keyboard, while standard output and standard error are printed to the screen. The following Table 5.4 shows the stream pointers that are available to access the standard streams:

Table 5.4 *Stream Pointers*

Pointer	Stream
`stdin`	Standard input
`stdout`	Standard output
`stderr`	Standard error

These pointers can be used as arguments to functions. Some functions, such as `getchar` and `putchar, stdin` and `stdout` are useful. These pointers are constants and cannot be assigned new values. The `freopen` function can be used to redirect the streams to disk files or to other devices. The operating system allows you to redirect a program's standard input and output at the command level. The following code shows how `stdin, stdout` and `stderr` used in a C program:

```
/* FILENO.C: This program uses _fileno to obtain
 * the file handle for some standard C streams.
 */
#include <stdio.h>
int main()
{
   printf("The file handle for stdin is %d\n", _fileno(
stdin) );
   printf("The file handle for stdout is %d\n", _fileno(
stdout) );
   printf("The file handle for stderr is %d\n", _fileno(
stderr) );
return 0;
}
```

After executing the above program, the following result appears on the screen:

```
The file handle for stdin is 0
The file handle for stdout is 1
The file handle for stderr is 2
```

The output stream, called `stderr`, is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` normally appears on the screen even if the standard output is redirected.

```
#include <stdio.h>
   /* cat:  concatenate files */
   main(int argc, char *argv[])
    {
        FILE *fp;
        void filecopy(FILE *, FILE *);
        char *prog = argv[0];  /* program name for errors
*/
        if (argc == 1 ) /* no args; copy standard input */
            filecopy(stdin, stdout);
        else
            while (—argc > 0)
  if ((fp = fopen(*++argv, "r")) == NULL)
   {
           fprintf(stderr,  "%s:  can  not  open  %s\n",
prog, *argv);
                   exit(1);
               }
  else
               {
                   filecopy(fp, stdout);
                   fclose(fp);
               }
        if (ferror(stdout))
 {
          fprintf(stderr, "%s: error writing stdout\n",
prog);
            exit(2);
 }
        exit(0);
 }
```

The above coding signals errors in two ways. First, the diagnostic output produced by `fprintf` goes to `stderr`, so it finds its way to the screen instead of disappearing down a pipeline or into an output file. If you include `argv[0]` in the message, the source of an error is identified. Second, the program uses the standard library function `exit` that terminates program execution when it is called. The argument of `exit` is available to whatever process called this one, so the success or failure of the program can be tested by another program that uses this one as a sub-process. Conventionally, a return value of `0` signals that all is well; non-zero values usually signal abnormal situations. The `exit` function calls `fclose` function for each open output file to flush out any buffered output. The function `ferror` returns

non-zero if an error occurred on the stream `fp`. The code written for `ferror` is as follows:

```
int ferror(FILE *fp)
```

Files are the formats that are required for saving data for future use. Random Access Memory (RAM) holds data temporarily. are used to save data permanently. The C program files can be stored at the user specified location. Figure 5.1 shows the file sequence hierarchy in C language.

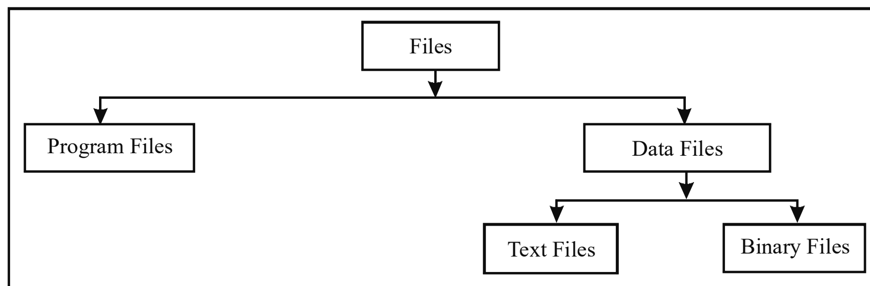


Fig. 5.1 *File Sequence Hierarchy*

True random access file handling only accesses the file at the point at which the data should be read or written rather than having to process it sequentially. A different approach is also possible whereby a part of the file is used for sequential access to locate something in the random access portion of the file in the same way as a File Allocation Table (FAT) works. The three main functions that seek forward and backward with file handling are as follows:

(i) `rewind()` – This function returns the file pointer to the beginning.

(ii) `fseek()` – This function returns the position of the file pointer.

(iii) `ftell()` – This function returns the current offset of the file pointer.

Each of these functions operates on the C file pointer, which is just the offset from the start of the file, and can be positioned at will. All read/write operations take place at the current position of the file pointer.

`rewind()`

The `rewind()` function can be used in sequential or random access C file programming. It simply tells the file system to position the file pointer at the start of the file. Any error flags are also cleared and no value is returned.

`fseek()` **`ftell()`**

The `fseek()` function is very useful in random access files where either the record or block size is known or there is an allocation system that denotes the start and end positions of records in an index portion of the file. The `fseek()` function takes three parameters:

(i) `FILE * f` – It represents the file pointer;

(ii) `long offset` – It represents the position offset;

(iii) `int origin` – It represents the point from which the offset is applied.

The `origin` parameter can be one of the following three values:

(i) `SEEK_SET` – It works from the start.

(ii) `SEEK_CUR` – It works from the current position.

(iii) `SEEK_END` – It works from the end of the file.

There are two fundamental types of files, text and binary. Of these, binary is generally simpler to deal with. Random access means you can move to any part of a file and read or write data from it without having to read through the entire file. Basically, seeking forward and backward pointer in file handling is done with `fseek` function. The general format of `fseek` function is as follows:

```
fseek(file * Fileptr, offset, position);
```

This function is used to move the file position to a desired location within the file. `Fileptr` is a pointer to the file concerned. `offset` is a number or variable of type long and `position` is an integer number. `offset` specifies the number of positions (in bytes) to be moved from the location specified at the `position`.

Input and output are normally sequential in which each read or write takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without reading or writing any data:

```
long lseek(int fd, long offset, int origin);
```

This function sets the current position in the file whose descriptor is `fd` to `offset` which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. The `origin` can be `0`, `1` or `2` to specify that `offset` is to be measured from the beginning, from the current position or from the end of the file, respectively. With `lseek` it is possible to treat files more or less like arrays at the price of slower access. For example, the following function reads any number of bytes from any arbitrary place in a file. It returns the number `read` or `-1` on error.

```
#include "syscalls.h"
/*get:  read n bytes from position pos */
int get(int fd, long pos, char *buf, int n)
   {
       if (lseek(fd, pos, 0) >= 0) /* get to pos */
           return read(fd, buf, n);
       else
           return -1;
   }
```

The return value from `lseek` is a `long` integer type that gives the new position in the file which depend on the setting conditions, such as seeking forward or seeking backward or `-1` if an error occurs. The standard library function `fseek` is similar to `lseek` except that the first argument is a `FILE *` and the return

value is non-zero if an error occurs. To determine whether the functions `fseek()` or `lseek()` operates correctly check its return value.

This can be used to copy a file to another file. Assume that the first named file is to be copied to the second named file. You may write a program and convert it into an executable file, specifying the argument in the DOS command line.

You may specify as follows at the C> prompt:

```
C > prgname . exe f1.cpp f2.cpp.
```

This means that you want to copy the contents of `f1.cpp` to `f2.cpp`. Here, the number of arguments are 3, and therefore `argc` will contain 3.

```
*agrv[0] = prgname.exe
*agrv[1] = f1.cpp - source to copy from
*agrv[2] = f2.cpp - file where to be copied
```

A character at a time is to be fetched from `f1.cpp` and put into `f2.cpp`.

A menu-based program to create employee records on file and calculate the age of any employee on date is given below:

Example 5.6
Create a Personal File for Employees & calculate the age of any employee ON DATE.

```
#include <stdio.h>
#include <dos.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
typedef struct
{
    char name[40];
    char code[5];
    char dob[9];
    char qual[40];
}employee;
    FILE *fp;
    struct date today;
int main()
{
    int create_emp();
    int calc_age();
    int ret,ch,onscrn=1;
    getdate(&today);
```

```
    printf("Today's Date Is %d/%d/%d\nIs It O.K :",
       today.da_day,today.da_mon,today.da_year);
    scanf("%c",&ch);
    onscrn=1;
    while(onscrn)
    {
       clrscr();
       printf("1: Create Employee Data File\n");
       printf("2: Calculate Age Of Employee\n");
       printf("3: Exit From Program\nEnter Your Choice :");
       scanf("%d",&ch);
       switch(ch)
       {
          case 1:
             create_emp();
             break;
          case 2:
             calc_age();
             break;
          case 3:
             onscrn=0;
             break;
       }
    }
    fclose(fp);
}
int create_emp()
{
    employee emp1;
    int i,n;
    fp=fopen("emp.dat","a");
    clrscr();
    printf("How Many Employees :");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
       clrscr();
       printf("Employee %d Details :\n",i+1);
       printf("\n\nEmployee Name :");
       scanf("%s",&emp1.name);
       printf("Employee Code :");
       scanf("%s",&emp1.code);
       printf("Date Of Birth :(dd/mm/yy)");
       scanf("%s",&emp1.dob);
```

```
        printf("Qualification :");
        scanf("%s",&emp1.qual);
        fprintf(fp, "%40s%5s%9s%40s\n", emp1.name,
emp1.code,emp1.dob, emp1.qual);
    }
    fclose(fp);
    return(0);
}
int calc_age()
{
    int ret,nyob,age,llfound=0,onscrn=1;
    employee emp1;
    char nam[40],*sear,*ori;
    char yob[5];
    fp=fopen("emp.dat","r");
    clrscr();
    printf("Employee Name To Search :");
    scanf("%s",nam);
    sear =strlwr(nam);
    while(onscrn)
    {
        ret=fscanf(fp, "%40s%5s%9s%40s\n", emp1.name,
emp1.code, emp1.dob, emp1.qual);
        if(ret==EOF)
        {
            onscrn=0;
            continue;
        }
        ori=strlwr(emp1.name);
        if(strcmp(sear,ori)==0)
        {
            clrscr();
            printf("Employee Name :%s\n",emp1.name);
            printf("Employee Code :%s\n",emp1.code);
            printf("Date of Birth :%s\n",emp1.dob);
            printf("Qualification :%s\n",emp1.qual);
            strcpy(yob,"19");
            strncat(yob,emp1.dob+6,2);
            yob[4]=0;
            nyob=atoi(yob);
            age = today.da_year - nyob;
            printf("Age of Employee :%d\n",age);
            getch();
            onscrn=0;
```

```
            llfound=1;
        }
    }
    fclose(fp);
    if(!llfound)
    {
        printf("%s Not found in emp.dat\n",nam);
        getch();
    }
    return(0);
}
```

```
Employee Name     :   saravanan
Employee Code     :   06
Date of Birth     :   02/06/63
Qualification     :   MBA
Age of Employee   :   36
```

You should be able to understand the program by reading the following:

The function `<dos.h>` is included. Look at the online help and see what it does. You will find that it defines various constants and declarations needed for DOS and 8086 specific calls. We can use it to get the system date to calculate the age of the employee. After the system date is confirmed, the menu appears. If you choose 1, it calls `create_emp` and asks for the number of employees. Then it accepts the records of a specified number of employees. The employee record is a structure.

After creating the records, you can opt to calculate the employee's ages by entering 2. This invokes the function `calc_age`. The function asks for the name of employee it must search for. If the name matches, the age will be calculated and displayed. The records are written in the append mode, so you will not lose the records. Note that the structures are written to the file using `fprintf()` and read from the file using `fscanf()`. Note also that `date` is a structure with three members `da_day, da_mon, da_year.`

This example has demonstrated that structures can be written to a file.

The library functions are included in the program during "pre-processing". The word 'Preprocessor' means that processing is carried out before the compilation of the program written by us. We have quite often used `#include` and `#define`. The former includes the contents of files such as `<stdio.h>` during compilation, while `#define` replaces the symbolic name with actual values before actual compilation. Thus pre-processing is nothing but the operations or processing carried out before actually processing. Let us see in details the various pre-processor directives used in 'C'.

The files following the statement are to be included in the program. For instance, when we use standard library function `printf()` we include `<stdio.h>`. The file `stdio.h` contains the declarations or the prototype for the `printf()` function. Now that you know function and function prototypes it is easy for you to visualise. The library functions may be elsewhere. The header file contains prototypes for one or more library functions. For instance, `stdio` contains prototypes for `scanf()`, `printf`, `getchar`, `putchar` etc., with one function corresponding to each one of them. But for the `include` statement, the compiler would not have recognized `printf`. Therefore the contents of the `printf()` function are to be included in our program file before actual compilation. We can write this in the following forms:

```
#include <filename>
#include "filename"
```

Such lines will be replaced by the contents of the file before compilation. The file may be in the user area or may be in the header file. If the right specifications for the filename are given the system will find the file and include it. If we write the include statement as `#include <stdio.h>` or `#include <prg.cpp>`, the system looks for the file only in the specified directories, namely h and cpp respectively. On the other hand, if we specify `#include "stdio.h"` or `#include "prg.cpp"`, the system would look for the file in the current directory as well as the specified list of directories.

The statement `#include` is not only applicable to library functions and their prototypes, but also applicable to programmer developed files. If there is a large program file, it can be divided function-wise and stored in different files. All these files are to be included in the file containing `main()` by using `#include`. `#include` is a directive or pre-processor directive to the compiler. Programmers can also combine all function declarations, definitions, variable declaration etc. in case of a large program file in a separate file, which can be included in every file containing part of the program.

We have used this kind of statement a number of times to increase the flexibility of programming. The syntax is as given below:

```
#define  symbolic_name   replacement_constant
#define MAX 10
```

Wherever *MAX* is found, the compiler will replace it as 10. This may also happen before the program is actually compiled, and such statements are also called macros or macro definitions. *MAX* is called a macro template and 10, its corresponding macro expansion. We know the rules of macro definition:

- No semicolon at the end of the statement
- A macro template is usually written in capital letters for ease of identification
- No commas in between.

The advantages of such macro definitions are clear :

- No accidental change of constants
- If a constant occurs at a number of places like the size of an array, the rate of interest etc, and if you want to increase the size of the array or in the other case, the rate of interest later, then you would need to makes change at a number of places in the program code. Using macros, simplifies the procedure, as a change carried at the top would be reflected throughout the program.

We can use macros for substituting complex statements such as those given below:

```
#define INPUT(a) scanf("%f" , &a);
# define OUTPUT printf("Enter a value");
# define AREACYL(r, h) (2*3.14 * r *( r+h))
# define volsp(r) ( 3.14 * r * r * r * 4.0/3.0 )
```

Note the last two statements carefully. We are passing arguments. For instance *AREACYL* refers to the area of the cylinder, *r* is the radius and *h* is the height. Whenever you want to calculate the area of the cylinder, you can specify the symbolic constant *AREACYL* which is the macro with the arguments namely radius and height.

Eg. area = AREACYL(3.0, 2,0);

Now *AREACYL* will be replaced by 2.0 * 3.14 * 3.0 * (3.0 + 2.0). Of course you should have defined *area* as a `float` in the function. Thus you have used the macro like a function. Similarly the *#define volsp* calculates the volume of a sphere of radius r. You can give any radius. It will substitute the r with the given radius. For instance, you can call

x = volsp(4.0);

Remember however, that there should not be spaces between the macro name and the argument. Spaces left there, if any, will be treated as the argument and hence as replacement text. Whenever we define a macro in the form of a function, the entire macro should be enclosed with parenthesis.

Now look at the example given below:

```
/*Example 5.7 - to demonstrate macros*/
#include<stdio.h>
#define INPUT(a) scanf("%f" , &a);
#define OUTPUT printf("Enter radius of sphere\n");
#define volsp(r) ( 3.14 * r * r * r * 4.0/3.0 )
#define AREACYL(r,h) (2.0*3.14*r*(r+h))
main()
{
    float a, v, r, h;
    OUTPUT
    INPUT(a)
    v=volsp(a);
    printf("voume of sphere of radius %f = %f\n", a,v);
    printf("Enter radius and after a space height of
cylinder\n");
```

```
    INPUT(r)
    INPUT(h)
    printf("area of cylinder=%f\n", AREACYL(r,h));
}
```

Result of program

```
Enter radius of sphere
2.0
voume of sphere of radius 2.000000 = 33.493332
Enter radius and after a space height of cylinder
4
5
area of cylinder=226.080000
```

Four macros have been defined in the program. In the main(), OUTPUT will be replaced by - printf('Enter radius of sphere\n'); -wherever it is found, at the time of pre-processing, but before compilation. Similarly the other 3 macros will also be substituted at the time of pre-processing. Parameters are passed in the macros *AREACYL(r,h)* and *volsp(r)*. This is similar to passing values in a function. The program prints the volume of the sphere and then the area of the cylinder for the dimensions given at run time.

There are, however, differences between macros and functions, although a macro resembles a function in the above examples. As you have guessed right, the executable code of a macro based program will be larger than a function based program if the macro is used more than once. This is due to the reason that macro will be substituted wherever it appears, whereas actual code of the corresponding function will be at one place and not substituted whenever called. Macros are faster than functions, as they do not require to be called with arguments and return values, unlike functions. This calling and return does not arise with macros, since macros are substituted at every place of occurrence. Therefore depending on the context, a macro or a function could be used i.e. if speed is the criteria, a macro could to be used, and if program size is the criteria, a function could be used.

By now you know how a program gets converted into executable code. The program gets converted to executable code in the following manner. The program statements you write are the source code, which is made complete during pre-processing, and then compiled. These two operations take place when you say compile, and the the object code is produced after compilation with a file name.OBJ extension. The linker in the system links all files and gives you an executable file. The code in turn is called executable code and stored with file name.EXE extension. When you say run, the .EXE file is executed. Thus you may have 4 files in the same name with different extensions as given below:

```
name.c
name.bak /*back up file*/
name.OBJ
name.EXE
```

Whenever you want to execute the program, use the executable file.

"C" is a flexible language. We have defined certain macros. We can always undefine them at some point later in the program. The syntax is given below:

```
e.g.  #undef  INPUT
      #undef  AREACYL
```

Whenever the compiler comes across such `#undef` directives it will cease to recognise the corresponding macro definition from that point onwards.

if

We have discussed the execution of statements at run time depending on certain conditions. 'C' allows even conditional compilation. In 'C', a set of statements will be included for compilation, depending on certain conditions. In the former case conditions were checked at the time of program execution. In the latter case, the conditions are checked at the time of compilation itself. Therefore the check is carried out during pre-processing.

For instance

```
# if (constant integer expression)
{ s1
}
# endif or #elif  or  # else
```

The group of statements *s1* will be included if the constant integer expression is true, i.e., non zero. The statements are those following `#if` and upto `#endif` or `#elif` or `#else`. Let us assume that the I/O bus of the computer varies and that it could be 1 byte, 2 bytes, 4 bytes and 8 bytes. The following code segment defines the size of short integer and integer depending on the I/O bus. This is a hypothetical example to illustrate the concept.

```
main()
{
    #if (DATA == 1)
          SHRT-MAX = + 127;
          SHRT-MIN  = -  128;
          INT-MAX   =  + 127;
          INT-MIN   =  -  128;
    #else
          #if (DATA == 2)
                SHRT-MAX = + 127;
                SHRT-MIN = - 128;
                INT-MAX   = + 32767;
                INT-MIN   = - 32768;
```

```
          #elif (data == 4)/* Similar to else if */
                SHRT-MAX = + 32767;
                SHRT-MIN = - 32768;
                INT-MAX  = + 2147483647;
                INT-MIN   = - 247483648;
                #elif
          .......
    #endif
 }
```

The first group of statements are clearly marked by #if and #else. These will be compiled if DATA== 1. Similarly the other groups will be compiled depending on the value of DATA. Although in practice, this can be achieved using the general branch constructs, DATA will be only one of the various types for a particular system. Therefore all the blocks need not be compiled. If the irrelevant statements are not included, you can reduce the size of the code, then program execution will be faster. If we had used general branching, the entire code will be compiled, although the other conditions will never be used. Conditional compilation thus helps in improving performance. Since the compiler has to work with varying configurations, conditional compilation provides the flexibility to compile the appropriate statements depending upon the type of hardware. Therefore a single program will suffice, as otherwise different programs would be rwquired for different configurations.

`#ifdef`

There are other types of conditional compilation statements. We define macros with `#define` and make them inactive using `#undef`. We can execute certain statements if the macro has been defined using the `#ifdef` statement. For instance,

```
#ifdef INPUT
    s1;  s2;
#endif.
```

When the compiler goes over to the `#ifdef` *INPUT* statement, it will check whether *INPUT* remains defined. If so, *s1* and *s2* will be compiled. If not, the compiler will skip the statements. This statement gives flexibility to the programmer when the requirements of the program change frequently. It is also useful when the software is to be supplied for different hardware configurations and therefore software suitable for all configurations must be written. Only one of them will be true in any situation and hence one `#ifdef` will be true. The corresponding code segment will be compiled. If the same thing is executed through switch case, then the whole code, applicable for all conditions will have to be compiled, whereas here only one segment of code will be compiled. This is the essential difference.

The opposite of `#ifdef` is `#ifndef`. Here a set of statements will be compiled if a macro has not been defined.

Eg.

```
main()
# ifndef Volsp
    execute statements for AREACYL
#endif
#ifndef AREACYL
    calculate volsp
#endif
```

'C' programming has a number of features which make it interesting. The various library functions can be included in the programs so as to reduce the coding effort as well as to improve the quality of programs. However, the compiler should know the library functions used, and therefore the declarations of the functions are made known so that through the declarations in the included file the entire function can be called at the time of compiling the program. The header files are included through the *#include* statement. When the file name is indicated like `<stdio.h>`, the system looks for the file in the current directory. On the other hand if indicated within quotes like '`stdio.h`', the system looks for the file in the current directory, as well as in the other specified directories.

Whenever some segments of code or variable names are repeatedly used in a program, these could be defined as macros. The macro name (instead of the whole code) can be given at locations when these are required. This eliminates errors due to wrong typing and improves readability. Even multi-line programs can be treated as macros and arguments passed as in a function. The essential difference between a macro and a function are :

- A macro will be substituted at all places increasing the code size at the time of execution.
- A function will be included only once, keeping the code size to the minimum.
- A macro will facilitate faster execution whereas a function can be slow.

'C' also provides for compilation of specific code segments depending on conditions. This is achieved through the `#if`, `#endif`, `#elif` and `#else` statements. Conditional compilation can also be carried out depending on whether a particular macro has been defined or not. The 'C' statements, which provide this facility, are:

```
#ifdef   #ifndef   and   #endif
```

Thus in "C" programming, there is a stage in between coding and compilation, which is known as pre-processing. At this stage the macros have to be substituted, the included files actually brought to the source code file, and then the entire code presented to the "C" language compiler. Even in compilation we can have flexibility through the use of conditional compilation directives. Thus the same version of the software can be executed in varying hardware configurations through the use of conditional compilation statements.

ANSI C, ISO C, and Standard C are defined as the successive standards for the C programming language published by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO). Fundamentally, these names are referred specifically to the original and best-supported version of the standard (known as C89 or C90). Software developers are inspired to develop the programs in C by conforming to the standards which also supports portability between the compilers.

The first standard for C was published by ANSI. Although this document was subsequently adopted by International Organization for Standardization (ISO) and subsequent revisions published by ISO have been adopted by ANSI, "ANSI C" is still used to refer to the standard.

ANSI C is a set of successive standards which were published by the American National Standards Institute (ANSI) for the C programming language. The ANSI specifies the syntax and semantics of programs written in C.

In 1983, the American National Standards Institute (ANSI) formed a committee, X3J11, to establish a standard specification of C. In 1985, the first Standard Draft was released, sometimes referred to as C85. In 1986, another Draft Standard was released, sometimes referred to as C86. The prerelease Standard C was published in 1988, and sometimes referred to as C88.

The ANSI standard was completed in 1989 and ratified as ANSI X3.159-1989 "Programming Language C". This version of the language is often referred to as "ANSI C". Later the label "C89" was used to distinguish it from "C90" but using the same labelling method.

The same standard as "C89" was ratified by the International Organization for Standardization as ISO/IEC 9899:1990, with only formatting changes which is sometimes referred to as "C90". Therefore, the terms "C89" and "C90" refer to essentially the same language.

This standard has been withdrawn by both ANSI/INCITS and ISO/IEC.

In 1995, the ISO published an extension, called Amendment 1, for the ANSI-C standard. Its full name finally was ISO/IEC 9899:1990/AMD1:1995 or nicknamed "C95". Aside from error correction there were further changes to the language capabilities, such as:

- Improved multi-byte and wide character support in the standard library, introducing `<wchar.h>` and `<wctype.h>` as well as multi-byte I/O.
- Addition of digraphs to the language.

- Specification of standard macros for the alternative specification of operators, for example `and` for `&&`.
- Specification of the standard macro `__STDC_VERSION__`.

In addition to the amendment, the following two technical corrigenda were published by ISO for C90:

- ISO/IEC 9899:1990/Cor 1:1994 TCOR1 in 1994.
- ISO/IEC 9899:1990/Cor 2:1996 in 1996.

C11 was officially ratified and published on December 8, 2011. Notable features include improved Unicode support, type-generic expressions using the new `_Generic` keyword, a cross-platform multi-threading API or Application Program Interface (`threads.h`) and atomic types which support in both core language and the library (`stdatomic.h`).

Following technical corrigendum has been published by ISO for C11:

- ISO/IEC 9899:2011/Cor 1:2012

As of October 2018, "C17" is the current standard for the C programming language.

"C17" addresses defects in "C11" without introducing new language features.

C2x is an informal name for the next (after C17) major C language standard revision.

ANSI C is now supported by almost all the widely used compilers. GCC (GNU Compiler Collection) and Clang are two major C compilers popular today, both are based on the C11 with updates including changes from later specifications, such as C17 and C18. Any source code written only in standard C and without any hardware dependent assumptions is virtually guaranteed to compile correctly on any platform with a conforming C implementation. Without such precautions, most programs may compile only on a certain platform or with a particular compiler, due, for example, to the use of non-standard libraries, such as Graphical User Interface or GUI libraries, or to the reliance on compiler- or platform-specific attributes, such as the exact size of certain data types and byte endianness.

C was originally developed by Dennis Ritchie at AT&T Bell Labs between 1969 and 1973. It has a free-format program source code. C is a general-purpose programming language. C is one of the oldest currently used programming languages and is one of the most widely used programming languages. ANSI C is a set of successive standards which were published by the American National Standards

Institute (ANSI) for the C programming language. The ANSI specifies the syntax and semantics of programs written in C.

Some key differences between C and ANSI C include the following:

- ANSI C allows the inclusion of a function prototype which gives the type of the function and the type of each parameter before defining the function.
- In C, function declarations are assumed by default to be of type `int`. Hence, integer type functions need not declared at all.
- In C, function headers have different syntax.
- Function prototypes must be declared without a list of arguments and types, and consist of the type, function name and an empty set of parentheses.
- C converts all float types in an expression to double precision types.
- The type `signed char` is not available in C, but is in ANSI C.
- In C, type `void` is not available.
- In C, functions are assumed to return integer if they return nothing.
- Many old C compilers expect that the first character of a preprocessor directive line is a `#`. No leading white space is allowed.
- In C, the signed `type` qualifier is not available.
- In C, the unsigned qualifier can be used to qualify integer types only.
- In C, the unary positive sign is not allowed.
- In C, the `type long double` is not available.
- In C, the `const` qualifier is not available.
- In C, enumeration type is not available.
- Automatic arrays cannot be initialized in declarations in C. Only external and static arrays can be initialized.
- Automatic structures and arrays of structures cannot be initialized in declarations in C.
- In C, some old compilers may not allow references to entire structures, requiring the use of structure pointers or individual structure members.

7. Define the term character I/O.
8. What is the purpose of `feof` function?
9. State use of the rewind () function.
10. Define the term command line.
11. What are preprocessors in C?
12. Give the definition of ANSI C.

1. In “C”, we come across two types of files:
   - Stream Oriented
   - System Oriented
2. Data can be stored in standard files in two ways as given below:
   - Storing characters or numerals consecutively. Each character is interpreted as an individual data item.
   - The data items are arranged in blocks in an unformatted manner. Each block may be an array or a structure.
3. You have to open the file and assign the file pointer to take care of further operations. Hence, you can declare,

```
FILE * fp;
   fp = fopen (“filename”, “r”);
```

4. Text and binary are the two modes of opening a file.
5. File copy can be achieved by reading one character at a time and writing to another file either in the write mode or the append mode.
6. When an error is encountered during file processing, the program execution will be terminated and error messages will be displayed.
7. Character I/O (Input/Output) refers to the file handling concept in which file is read and written once it is opened. You can open a file for writing, close it and reopen it for reading, then close it, and open it again for appending, etc.
8. The function `feof()` returns a nonzero value if the end of the given file stream has been reached.
9. The `rewind()` function can be used in sequential or random access C file programming. It simply tells the file system to position the file pointer at the start of the file. Any error flags are also cleared and no value is returned.
10. Command line can be used to copy a file to another file. Assume that the first named file is to be copied to the second named file. You may write a program and convert it into an executable file, specifying the argument in the DOS command line.
11. The library functions are included in the program during ‘Preprocessing’. The word ‘Preprocessor’ means that processing is carried out before the compilation of the program written by us. We have quite often used `#include` and `#define`. The former includes the contents of files, such as `<stdio.h>` during compilation, while `#define` replaces the symbolic name with actual values before actual compilation. Thus preprocessing is nothing but the operations or processing carried out before actually processing. Let us see in details the various preprocessor directives used in ‘C’.

12. ANSI C is a set of successive standards which were published by the American National Standards Institute (ANSI) for the C programming language. The ANSI specifies the syntax and semantics of programs written in C.

- System oriented files or low level files are more closely related to the operating system and hence require more complex programming skills to use them. They may be found to be more efficient than the former in some cases, but we will not discuss them further because of their complexity.
- The buffer is also memory which is used to store data temporarily without the knowledge of the user. In fact we created a buffer and stored values into it before printing them using the *sprintf* function.
- Files are formats that save data for future use. Any file has to be opened for any further processing, such as reading, writing or appending, i.e., writing at the end of the file. The characters will be written or read one after another from the beginning to the end, unless otherwise specified. You have to open the file and assign the file pointer to take care of further operations.
- When an error is encountered during file processing, the program execution will be terminated and error messages will be displayed.
- Character I/O (Input/Output) refers to the file handling concept in which file is read and written once it is opened. You can open a file for writing, close it and reopen it for reading, then close it, and open it again for appending, etc.
- You are familiar with reading and writing. So far you were reading from and writing to standard input/output. Therefore, you used functions for the formatted I/O with `stdio`, such as `scanf()` and `printf()`. We also used unformatted I/O, such as `getch()`, `putch()` and other statements.
- The function `feof()` returns a nonzero value if the end of the given file stream has been reached.
- The file <*stdio.h*> has declarations to the pointers to three files, namely *stdin*, *stdout* and *stderr*.
- *stdin* facilitates usage of the standard input device for program execution, and normally points to the keyboard which is the default input device.
- *stdout* facilitates the usage of a standard output device where program output is displayed, and points to the video monitor.
- *stderr* facilitates sending error messages to the standard device that is again the monitor.
- Files are the formats that are required for saving data for future use. Random Access Memory (RAM) holds data temporarily. File structures are used to save data permanently.

- The `rewind()` function can be used in sequential or random access C file programming. It simply tells the file system to position the file pointer at the start of the file. Any error flags are also cleared and no value is returned.
- True random access file handling only accesses the file at the point at which the data should be read or written rather than having to process it sequentially. A different approach is also possible whereby a part of the file is used for sequential access to locate something in the random access portion of the file in the same way as a File Allocation Table (FAT) works.
- Command line can be used to copy a file to another file. Assume that the first named file is to be copied to the second named file. You may write a program and convert it into an executable file, specifying the argument in the DOS command line.
- The library functions are included in the program during 'Preprocessing'. The word 'Preprocessor' means that processing is carried out before the compilation of the program written by us. We have quite often used `#include` and `#define`. The former includes the contents of files, such as `<stdio.h>` during compilation, while `#define` replaces the symbolic name with actual values before actual compilation. Thus preprocessing is nothing but the operations or processing carried out before actually processing. Let us see in details the various preprocessor directives used in 'C'.
- Four macros have been defined in the program. In the main(), OUTPUT will be replaced by - printf("Enter radius of sphere\n"); -wherever it is found, at the time of preprocessing, but before compilation. Similarly the other 3 macros will also be substituted at the time of preprocessing. Parameters are passed in the macros *AREACYL(r,h)* and *volsp(r)*.
- Thus in 'C' programming, there is a stage in between coding and compilation, which is known as preprocessing.
- ANSI C, ISO C, and Standard C are defined as the successive standards for the C programming language published by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO).
- ANSI C is a set of successive standards which were published by the American National Standards Institute (ANSI) for the C programming language. The ANSI specifies the syntax and semantics of programs written in C.

- These are closely related to the operating system and require complex programming.
- **`ferror ()`** It is a function to detect errors which return a zero when there is no error, otherwise it will return a non-zero number.

- Character I/O refers to the file handling concept in which file is read and written once it is opened.
- **`stdin`** It facilitates usage of the standard input device for program execution.
- **`stdout`** It facilitates the usage of a standard output device where the program output to be displayed.
- **`stderr`** It sends error messages to the standard output device.
- **`fseek()`** It is a function which is very useful in random access files where either the record or block size is known or there is an allocation system that denotes the start and end positions of records in an index portion of the file.
- In 'C' programming, there is a stage in between coding and compilation, which is known as preprocessing.

---

1. How is a data file opened and closed?
2. Differentiate between text mode and binary mode.
3. What is file copy?
4. Define the term error handling and I/O operator.
5. What does character I/O refer to?
6. Write the syntax of `feof()` function.
7. Differentiate between `stdin` and `stdout` function.
8. Define the random access files.
9. State the concept of command line arguments.
10. Give the definition of macro.
11. What is ANSI C Edition?

1. Explain the formatted I/O with files giving appropriate example programs.
2. Discuss about the opening and closing a file in and also write a program to write and read a data file.
3. Analyse the input/output operation on file and also write a program to copy content of one file to another.
4. Describe the various modes used in file handling with the help of relevant examples.
5. Briefly explain the `feof` and write a program to check the EOF condition using `feof ()` function.

6. Analyse the stream pointers giving appropriate example programs.
7. Discuss briefly random access files with the help of relevant examples.
8. Describe the command line arguments with the help of relevant examples.
9. Briefly explain the macro in C giving appropriate example programs.
10. Discuss briefly conditional compilation directives with the help of relevant examples.
11. Briefly explain the difference between C and ANSI C.

Gottfried, Byron S. 1996. *Programming with C*, Schaum's Outline Series. New York: McGraw-Hill.

Jeyapoovan T. 2006. *Computer Programming - Theory & Practice*. New Delhi: Vikas Publishing House Pvt. Ltd.

Khurana, Rohit. 2005. *Object Oriented Programming*. New Delhi: Vikas Publishing House Pvt. Ltd.

Kanetkar, Yashwant. 2003. *Let Us C*. New Delhi: BPB Publication.

Saxena, Sanjay. 2003. *A First Course in Computers*. New Delhi: Vikas Publishing House Pvt. Ltd.

Subburaj, R. 2000. *Programming in C*. New Delhi: Vikas Publishing House Pvt. Ltd.

Ghosh, Smarajit. 2009. *All of C*. New Delhi: PHI Learning Pvt Ltd.

Bronson, Gary J. 2000. *A First Book of ANSI C*, 3rd edition. California: Thomson, Brooks Cole.