As per the
Revised
Syllabus Effective
August 2007

C Programming and Data Structures

Fourth Edition

About the Author

E Balagurusamy, former Vice Chancellor, Anna University, Chennai, is currently Member, Union Public Service Commission, New Delhi. He is a teacher, trainer, and consultant in the fields of Information Technology and Management. He holds an ME (Hons) in Electrical Engineering and Ph.D in Systems Engineering from the Indian Institute of Technology, Roorkee. His areas of interest include Object-Oriented Software Engineering, Electronic Business, Technology Management, Business Process Re-engineering, and Total Quality Management.

A prolific writer, he has authored a large number of research papers and several books. His best selling books, among others include:

- *Programming in C#, 2/e*
- *Programming in Java, 3/e*
- *Object-Oriented Programming with C++, 4/e*
- *Programming in BASIC, 3/e*
- *Programming in ANSI C, 4/e*
- *Numerical Methods*
- *Reliability Engineering*

A recipient of numerous honours and awards, he has been listed in the Directory of Who's Who of Intellectuals and in the Directory of Distinguished Leaders in Education.

As per the Revised Syllabus Effective August 2007

C Programming and Data Structures

Fourth Edition

E Balagurusamy

Member, UPSC
New Delhi

Tata McGraw-Hill Publishing Company Limited
NEW DELHI

McGraw-Hill Offices
New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

Tata McGraw-Hill

Published by the Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008.

ISBN-13: 978-0-07-0084759

ISBN-10: 0-07-0084750

Managing Director—*Ajay Shukla*
General Manager: Publishing—SEM & Tech Ed: *Vibha Mahajan*
Sponsoring Editor—*Shalini Jha*
Senior Copy Editor—*Dipika Dey*
Junior Manager—Production, *Anjali Razdan*

Marketing General Manager—Higher Education & Sales: *Michael J. Cruz*
Product Manager—SEM & Tech Ed: *Biju Ganesan*

Controller—Production: *Rajender P. Ghansela*
Asst. General Manager— Production: *B. L. Dogra*

Typeset at Script Makers, 19, A1-B, DDA Market, Paschim Vihar, New Delhi 110 063, and printed at Shakti Packers, 5C/9, New Rohtak Road, Karol Bagh, New Delhi-110 005

Cover: SDR Printers

DALZCRBFRAALX

The McGraw·Hill Companies

Contents

Foreword *xi*
Preface *xiii*
Road Map to the Syllabus *xv*
The C99 Standard *xvii*

UNIT I

1. Algorithms **1.3–1.15**

1.1 Introduction 1.3
1.1.1 Algorithm/pseudocode 1.3
1.1.2 Flow Chart 1.4
1.2 Three Basic Operations 1.5
1.2.1 Sequence 1.5
1.2.2 Selection 1.6
1.2.3 Iteration 1.13
Review Questions and Exercises 1.14

2. Overview of C **2.1–2.12**

2.1 Introduction 2.1
2.2 Importance of C 2.1
2.3 Sample C Programs 2.2
2.4 Program Development Steps 2.5
2.5 Structure of a C Program 2.9
Review Questions and Exercises 2.11

3. Constants, Variables, and Data Types **3.1–3.26**

3.1 Introduction 3.1
3.2 Character Set 3.1
3.3 C Tokens 3.1
3.4 Keywords and Identifiers 3.2
3.5 Constants 3.3
3.6 Variables 3.7
3.7 Basic Data Types and Sizes 3.8
3.8 Declaration of Variables 3.11
3.9 Assigning Values to Variables 3.14
3.10 Defining Symbolic Constants 3.21
Review Questions and Exercises 3.24

4. Operators and Expressions 4.1–4.22

4.1 Introduction 4.1
4.2 Arithmetic Operators 4.1
4.3 Relational Operators 4.4
4.4 Logical Operators 4.5
4.5 Assignment Operators 4.5
4.6 Increment and Decrement Operators 4.7
4.7 Conditional Operator 4.8
4.8 Bitwise Operators 4.8
4.9 Special Operators 4.9
4.10 Arithmetic Expressions 4.11
4.11 Evaluation of Expressions 4.11
4.12 Precedence of Arithmetic Operators 4.12
4.13 Some Computational Problems 4.14
4.14 Type Conversions in Expressions 4.15
4.15 Operator Precedence and Associativity 4.18
Case Studies 4.19
Review Questions and Exercises 4.20

5. Managing Input and Output Operations 5.1–5.25

5.1 Introduction 5.1
5.1.1 Input/Output Statements and Header Files 5.1
5.2 Reading a Character 5.2
5.3 Writing a Character 5.4
5.4 Formatted Input 5.5
5.5 Formatted Output 5.14
Case Studies 5.21
Review Questions and Exercises 5.24

6. Decision Making and Branching 6.1–6.29

6.1 Introduction 6.1
6.2 Decision Making with If Statement 6.1
6.3 Simple If Statement 6.2
6.4 The If … Else Statement 6.5
6.5 Nesting of If ... Else Statements 6.7
6.6 The Else If Ladder 6.11
6.7 The Switch and Break Statements 6.13
6.8 The ? : Operator 6.17
6.9 The Goto Statement 6.20
6.9.1 Label 6.20
Case Studies 6.22
Review Questions and Exercises 6.26

7. Decision Making and Looping 7.1–7.30

7.1 Introduction 7.1
7.2 The While Statement 7.3
7.3 The Do Statement 7.4

7.4 The For Statement 7.6
7.5 Jumps in Loops 7.14
7.6 Structured Programming 7.20
Case Studies 7.21
Review Questions and Exercises 7.27

UNIT II

8. Arrays **8.3–8.31**
8.1 Introduction 8.3
8.2 One-Dimensional Arrays 8.3
8.3 Two-Dimensional Arrays 8.8
8.4 Initializing Two-Dimensional Arrays 8.13
8.5 Multidimensional Arrays 8.15
Case Studies 8.17
Review Questions and Exercises 8.28

9. Handling of Character Strings **9.1–9.23**
9.1 Introduction 9.1
9.2 Declaring and Initializing String Variables 9.2
9.3 Reading Strings from Terminal 9.2
9.4 Writing Strings to Screen 9.5
9.5 Arithmetic Operations on Characters 9.8
9.6 Putting Strings Together 9.10
9.7 Comparison of Two Strings 9.12
9.8 String-Handling Functions 9.13
9.9 Table of Strings 9.15
Case Studies 9.18
Review Questions and Exercises 9.22

10. User-Defined Functions **10.1–10.50**
10.1 Introduction 10.1
10.1.1 Standard Library Functions 10.1
10.2 Need for User-Defined Functions 10.4
10.3 A Multi-Function Program 10.5
10.4 The Form of C Functions 10.7
10.5 Return Values and their Types 10.8
10.6 Calling a Function 10.9
10.7 Category of Functions 10.10
10.8 No Arguments and no Return Values 10.10
10.9 Arguments but no Return Values 10.12
10.10 Arguments with Return Values 10.14
10.11 Handling of Non-Integer Functions 10.18
10.12 Nesting of Functions 10.21
10.13 Recursion 10.22
10.14 Functions with Arrays 10.23
10.15 The Scope and Lifetime of Variables in Functions—Storage Classes 10.26

10.16 ANSI C Functions 10.36
10.17 The Preprocessor 10.39
10.18 ANSI Additions 10.42
Case Study 10.45
Review Questions and Exercises 10.49

UNIT III

11. Pointers **11.3–11.32**

11.1 Introduction 11.3
11.2 Understanding Pointers 11.3
11.3 Accessing the Address of a Variable 11.5
11.4 Declaring and Initializing Pointers 11.6
11.5 Accessing a Variable Through its Pointer 11.7
11.6 Pointer Expressions 11.9
11.7 Pointer Increments and Scale Factor—Address Arithmetic 11.11
11.8 Pointers and Arrays 11.12
11.9 Pointers and Character Strings 11.15
11.10 Pointers and Functions 11.17
11.11 Pointers and Structures 11.23
Case Studies 11.25
Review Questions and Exercises 11.31

UNIT IV

12. Structures and Unions **12.3–12.25**

12.1 Introduction 12.3
12.2 Structure Definition—Declaring Structures 12.3
12.3 Giving Values to Members 12.5
12.4 Structure Initialization 12.6
12.5 Comparison of Structure Variables 12.8
12.6 Arrays of Structures 12.8
12.7 Arrays within Structures 12.11
12.8 Structures within Structures—Nested Structures 12.12
12.9 Structures and Functions 12.15
12.10 Unions 12.17
12.11 Size of Structures 12.19
12.12 Bit Fields 12.19
Case Study 12.21
Review Questions and Exercises 12.24

UNIT V

13. File Management in C **13.3–13.22**

13.1 Introduction 13.3
13.2 Defining and Opening a File 13.7

13.3 Closing a File 13.8
13.4 Input/Output Operations on Files 13.9
13.5 Error Handling During I/O Operations 13.12
13.6 Random Access to Files 13.14
13.7 Command Line Arguments 13.17
Case Study 13.19
Review Questions and Exercises 13.21

UNIT VI

14. Data Structures **14.3–14.21**

14.1 Introduction to Data Structures 14.3
14.2 Stacks 14.3
14.3 Queues 14.6
14.4 Circular Queues 14.10
14.5 Applications of Stacks 14.11
Review Questions and Exercises 14.21

UNIT VII

15. Dynamic Memory Allocation and Linked Lists **15.3–15.47**

15.1 Introduction 15.3
15.2 Dynamic Memory Allocation 15.3
15.3 Linked Lists 15.9
15.3.1 Self-Referential Structure 15.9
15.3.2 Singly Linked List 15.10
15.4 Advantages of Linked Lists 15.12
15.5 Types of Linked Lists 15.12
15.6 Pointers Revisited 15.14
15.7 Basic List Operations 15.15
15.8 Application of Linked Lists 15.24
15.9 Circular Linked Lists 15.25
15.10 Doubly Linked Lists 15.25
15.11 Doubly Linked List Operations 15.26
15.12 Doubly Linked Circular Lists 15.35
15.13 Stacks and Queues using Linked List 15.36
Case Studies 15.40
Review Questions and Exercises 15.46

16. Binary Trees and Graphs **16.1–16.25**

16.1 Binary Trees—Representation and Terminology 16.1
16.2 Binary Tree Traversal 16.2
16.3 Graphs 16.8
16.4 Graph Representation in C 16.9
16.5 Graph Traversal 16.13
Review Questions and Exercises 16.24

UNIT VIII

17. Sorting and Searching Techniques **17.3–17.25**

17.1 Introduction 17.3
17.2 Sorting 17.3
17.2.1 Sorting Efficiency 17.4
17.2.2 Exchange Sorting—Bubble Sort 17.5
17.2.3 Exchange Sorting—Quick Sort 17.7
17.2.4 Selection Sort 17.11
17.2.5 Merge Sort 17.13
17.2.6 Simple Insertion sort 17.15
17.2.7 Shell Sort 17.17
17.3 Searching 17.18
17.3.1 Linear Search 17.18
17.3.2 Binary Search 17.20
17.3.3 Indexed Sequential Search 17.23
Review Questions 17.24

Appendix A
ASCII Values of Characters *A.1*

Appendix B
Multiple Choice Questions *B.1–B.9*

Appendix C
Solved Question Papers – C Programming and Data Structures (May/June 2008) *C.1–C.76*

Bibliography *B.1*

Foreword

It gives me great pleasure to introduce *C Programming and Data Structures* by *Dr E Balagurusamy*, publication of which heralds the completion of a book that caters completely and effectively to the students of JNTU.

The need for a good textbook for this subject can be easily understood. Numerous books are available to the students for the subject, but almost none of them have the right combination of simplicity, rigour, pedagogy and syllabus compatibility. These books usually do not address one or more of the specific problems faced by students of this subject in JNTU. There has always been a need for a good book relevant to the requirements of the students and dealing with all aspects of the course. I am sure that the present book will be able to fill this void.

The book has been organized and executed with lot of care and dedication. The author has been an outstanding teacher and a pioneer of IT education in India. A conscious attempt has been made to simplify concepts to facilitate better understanding of the subject.

Dr Balagurusamy deserves our praise and thanks for accomplishing this trying task. McGraw-Hill Education, a prestigious publishing house, also deserves a pat on the back for doing an excellent job.

Dr K. Rajagopal

Vice-Chancellor
Jawaharlal Nehru Technological University
Hyderabad

Preface

C is a general-purpose structured programming language that is powerful, efficient and compact. C combines the features of high-level language with the elements of the assembler and is thus close to both man and machine. The growth of C during the last few years has been phenomenal. It has emerged as the language of choice for most applications due to speed, portability and compactness of code. It has now been implemented on virtually every sort of computer, from micro to mainframe.

For many years, the *de facto* standard for implementing the language has been the original *C Reference Manual* by Kernighan and Ritchie published in 1978. During these years of growth and popularity, C has undergone many changes. Numerous different features and facilities have been developed and marketed. This has resulted in minor problems in terms of portability of programs. Consequently, the American National Standard Institute (ANSI) constituted a committee to look into the language features and produce a more comprehensive and unambiguous definition of C. The result is ANSI C. Most compilers have already adopted ANSI standards.

This book incorporates all the features of ANSI C that are essential for a C programmer. The ANSI standards are explained in detail in appropriate places. For the convenience of readers, these places have been noted in the text by a special mention. A major highlight of this revised edition is the inclusion of updated information on compiler C99 standard, with relevant theory and programs. Another important addition in this edition is Appendix C containing 2008 solved question papers (4 sets).

The concept of 'learning by example' has been stressed throughout the book. Every important feature of the language is treated in depth followed by a complete program example to illustrate its use. Case studies at the end of the chapters not only describe the common ways in which C features are put together but also show real-life applications. Wherever necessary, pictorial descriptions of concepts are included to facilitate better understanding.

This book contains more than 100 examples and programs. All the programs have been tested using compilers compatible to both UNIX and MS-DOS operating systems and, wherever appropriate, the nature of output has been discussed. These programs also demonstrate the general principles of a good programming style. 200 multiple choice questions are given to help the students test their conceptual understanding of the subject.

The book also offers detailed discussions on Data Structures, Standard Library Functions, Bit Fields, Linked Lists, Doubly Linked Lists, Queues, Stacks and Graphs with sample codes and algorithms. Chapter 17 provides comprehensive information on Sorting and Searching techniques. Both these topics are explained with sample codes. The various pros and cons of each of these techniques are also covered.

This edition comes with a CD which provides

More than 600 objective/review/debugging questions (8 units)

100 Programming exercises

Model question paper

2006 and 2007 solved question paper

Computer programs for lab

The objective of the supplementry CD is to make the students learn the programming language and enable them to write their own programs using C and data structures.

The author is grateful to Mr A Rama Rao, Layola Institute of Technology & Management, Guntur and Mr P Chenna Reddy, JNTU College of Engineering, Pulivendula for their useful comments and suggestions.

E Balagurusamy

Road Map to the Syllabus

(Effective from August 2007)

Jawaharlal Nehru Technological University Hyderabad

I Year B.Tech	**T**	**P**	**C**
	4	**0**	**6**

C Programming and Data Structures

Objectives

- To make the student learn a programming language.
- To teach the student to write programs in C to solve the problems.
- To introduce the student to simple linear and non-liner data structures such as lists, stacks, queues, trees and graphs.

Syllabus

Unit I

Algorithm / pseudo code, flowchart, program development steps, structure of C program, a simple C program, identifiers, basic data types and sizes, constants, variables, arithmetic, relational and logical operators, increment and decrement operators, conditional operator, bit-wise operators, assignment operators, expressions, type conversions, conditional expressions, precedence and order of evaluation.

Input-output statements, statements and blocks, if and switch statements, loops- while, do-while and for statements, break, continue, goto and labels, programming examples.

Go To

CHAPTER 1 – ALGORITHMS
CHAPTER 2 – OVERVIEW OF C
CHAPTER 3 – CONSTANTS, VARIABLES, AND DATA TYPES
CHAPTER 4 – OPERATORS AND EXPRESSIONS
CHAPTER 5 – MANAGING INPUT AND OUTPUT OPERATIONS
CHAPTER 6 – DECISION MAKING AND BRANCHING
CHAPTER 7 – DECISION MAKING AND LOOPING

Unit II

Designing structured programs, functions, basics, parameter passing, storage classes- extern, auto, register, static, scope rules, block structure, user-defined functions, standard library functions, recursive functions, header files, C preprocessor, C program examples.

Go To

CHAPTER 7 – DECISION MAKING AND LOOPING
CHAPTER 10 – USER-DEFINED FUNCTIONS

Unit III

Arrays- concepts, declaration, definition, accessing elements, storing elements, arrays and functions, two-dimensional and multi-dimensional arrays, applications of arrays. pointers- concepts, initialization of pointer variables, pointers and function arguments, address arithmetic, character pointers and functions, pointers to pointers, pointers and multidimensional arrays, dynamic memory managements functions, command line arguments, C program examples.

Go To

CHAPTER 8 – ARRAYS
CHAPTER 9 – HANDLING OF CHARACTER STRINGS
CHAPTER 11 – POINTERS

Unit IV

Derived types- structures- declaration, definition and initialization of structures, accessing structures, nested structures, arrays of structures, structures and functions, pointers to structures, self-referential structures, unions, typedef, bitfields, C program examples.

Go To

CHAPTER 12 – STRUCTURES AND UNIONS
CHAPTER 15 – DYNAMIC MEMORY ALLOCATION AND LINKED LISTS

Unit V

Input and output – concept of a file, text files and binary files, streams, standard I/o, formatted I/o, file I/o operations, error handling, C program examples.

Go To CHAPTER 13 – FILE MANAGEMENT IN C

Unit VI

Searching – linear and binary search methods, sorting – bubble sort, selection sort, insertion sort, quick sort, merge sort.

Go To

CHAPTER 17 – SORTING AND SEARCHING TECHNIQUES

Unit VII

Introduction to data structures, singly linked lists, doubly linked lists, circular list, representing stacks and queues in C using arrays and linked lists, infix to post fix conversion, postfix expression evaluation.

Go To

CHAPTER 14 – DATA STRUCTURES
CHAPTER 15 – DYNAMIC MEMORY ALLOCATION AND LINKED LISTS

Unit VIII

Trees- binary trees, terminology, representation, traversals, graphs—terminology, representation, graph traversals (dfs and bfs)

Go To CHAPTER 16 – BINARY TREES AND GRAPHS

The C99 Standard

Computer languages are not static; they evolve, reacting to changes in methodologies, applications generally accepted practices, and hardware. C is no exception. In the case of C, two evolutionary paths were set in motion. The first is the continuing development of the C language. The second is C++, for which C provided the starting point. While most of the focus of the past several years has been on C++, the refinement of C has continued unabated.

For example, reacting to the internationalization of the computing environment, the original C89 standard was amended in 1995 to include various wide-character and multibyte functions. Once the 1995 amendment was complete, work began on updating the language, in general. The end result is, of course, C99. In the course of creating the 1999 standard, each element of the C language was thoroughly reexamined, usage patterns were analyzed, and future demands were anticipated. As expected, C's relationship to C++ provided a backdrop for the entire process. The resulting C99 standard is a testimonial to the strengths of the original. Very few of the key elements of C were altered. For the most part, the changes consist of a small number of carefully selected additions to the language and the inclusion of several new library functions. Thus C is still C!

Here we will examine those features added by C99, and the few differences between C99 and C89.

C99

Perhaps the greatest cause for concern that accompanies the release of a new language standard is the issue of compatibility with its predecessor. Does the new specification render old programs obsolete? Have important constructs been altered? Do I have to change the way that I write code? The answers to these types of questions often determine the degree to which the new standard is accepted and, in the longer term, the viability of the language itself. Fortunately, the creation of C99 was a controlled, even-handed process that reflects the fact that several experienced pilots were at the controls. Put simply: If you liked C the way it was, you will like the version of C defined by C99. What many programmers think of as the world's most elegant programming language, still is! In this chapter we will examine the changes and additions made to C by the 1999 standard. Many of these changes were mentioned in passing in Part One. Here they are examined in closer detail. Keep in mind, however, that as of this writing, there are no widely used compilers that support many of C99’s new features. Thus, you may need to wait a while before you can ‘test drive’ such exciting new constructs as variable-length arrays, restricted pointers, and the long long data type.

C89 vs. C99: AN OVERVIEW

There are three general categories of changes between C89 and C99:

- Features added to C89
- Features removed from C89
- Features that have been changed or enhanced

Many of the differences between C89 and C99 are quite small, and clarify nuances of the C language. This book will concentrate on the larger changes that affect the way programs are written.

Features Added

Perhaps the most important features added by C99 are the new keywords:

- inline
- restrict
  - _Bool
  - _Complex
  - _Imaginary

Other major additions include

- Variable-length arrays
- Support for complex arithmetic
- The **long long int** data type
- The //comment
- The ability to intersperse code and data
- Additions to the preprocessor
- Variable declarations inside the for statement
- Compound literals
- Flexible array structure members
- Designated initializers
- Changes to the **printf( )** and **scanf( )** family of functions
- The _ _**func**_ _ predefined identifier
- New libraries and headers

Most of the features added by C99 are innovations created by the standardization committee, of which many were based on language extensions offered by a variety of C implementations. In a few cases, however, features were borrowed from C++. The **inline** keyword and // style comments are examples. It is important to understand that C99 does not add C++-style classes, inheritance, or member functions. The consensus of the committee was to keep C as C.

Features Removed

The single most important feature removed by C99 is the ''implicit **int**' rule. In C89, in many cases when no explicit type specifier is present, the type **int** is assumed. This is not allowed by C99. Also removed is implicit function declaration. In C89, if a function was not declared before it is used, an implicit declaration is assumed. This is not supported by C99. Both of these changes may require existing code to be rewritten if compatibility with C99 is desired.

Features Changed

C99 incorporates several changes to existing features. For the most part, these changes expand features or clarify their meaning. In a few cases, the changes restrict or narrow the applicability of a feature. Many such changes are small, but a few are quite important, including:

- Increased translation limits
- Extended integer types
- Expanded integer type promotion rules
- Tightening of the **return** statement

As it affects existing programs, the change to **return** has the most significant effect because it might require that code be rewritten slightly.

Throughout the remainder of this chapter, we will examine the major differences between C89 and C99.

RESTRICT-QUALIFIED POINTERS

One of the most important innovations in C99 is the **restrict** type qualifier. This qualifier applies only to pointers. A pointer qualified by **restrict** is initially the only means by which the object it points to can be accessed. Access to the object by another pointer can occur only if the second pointer is based on the first. Thus, access to the object is restricted to expressions based on the **restrict**-qualified pointer. Pointers qualified by **restrict** are primarily used as function parameters, or to point to memory allocated via **malloc( )**. The **restrict** qualifier does not change the semantics of a program. By qualifying a pointer with **restrict**, the compiler is better able to optimize certain types of routines by making the assumption that the **restrict**-qualified pointer is the sole means of access to the object. For example, if a function specifies two **restrict**-qualified pointer parameters, the compiler can assume that the pointers point to different (that is, non-overlapping) objects. For example, consider what has become the classic example of **restrict:** the **memcpy( )** function.

In C89, it is prototyped as shown here:

```
void *memcpy(void *str1, const void *str2, size_t size);
```

The description for **memcpy( )** states that if the objects pointed to by *str*1 and *str2* overlap, the behaviour is undefined. Thus, **memcpy( )** is guaranteed to work for only non-overlapping objects. In C99, **restrict** can be used to explicitly state in **memcpy( )**'s prototype what C89 must explain with words.

Here is the C99 prototype for **memcpy( ):**

```
void *memcpy
(void * restrict str1, const void * restrict str2, size_t size);
```

By qualifying *str1* and *str2* with **restrict**, the prototype explicitly asserts that they point to non-overlapping objects. Because of the potential benefits that result from using **restrict**, C99 has added it to the prototypes for many of the library functions originally defined by C89.

Inline

C99 adds the keyword **inline**, which applies to functions. By preceding a function declaration with **inline**, you are telling the compiler to optimize calls to the function. Typically, this means that the

function's code will be expanded in line, rather than called. However, **inline** is only a request to the compiler, and can be ignored. Specifically, C99 states that using **inline** 'suggests that calls to the function be as fast as possible.' The **inline** specifier is also supported by C++, and the C99 syntax for **inline** is compatible with C++.

To create an in-line function, precede its definition with the inline keyword. For example, in the following program, calls to the function **max**( ) are optimized:

```
#include <stdio.h>
inline int max(int a, int b)
{
return a > b ? a : b;
}
int main(void)
{
int x=5, y=10;
printf("Max of %d and %d is: %d\n", x, y, max(x, y));
return 0;
}
```

For a typical implementation of **inline**, the preceding program is equivalent to this one:

```
#include <stdio.h>
int main(void)
{
int x=5, y=10;
printf("Max of %d and %d is: %d\n", x, y, (x>y ? x : y));
return 0;
}
```

The reason that **inline** functions are important is that they help you create more efficient code while maintaining a structured, function-based approach. As you probably know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time.

However, when a function is expanded in line, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to **inline** only very small functions. Further, it is also a good idea to **inline** only those functions that will have significant impact on the performance of your program.

Remember: Although **inline** typically causes a function's code to be expanded in line, the compiler can ignore this request or use some other means to optimize calls to the function.

NEW BUILT-IN DATA TYPES

C99 adds several new built-in data types. Each is examined here.

_Bool

C99 adds the **_Bool** data type, which is capable of storing the values 1 and 0 (true and false). **_Bool** is an integer type. As many readers know, C++ defines the keyword **bool**, which is different from **_Bool**. Thus, C99 and C++ are incompatible on this point. Also, C++ defines the built-in Boolean constants **true** and **false**, but C99 does not. However, C99 adds the header **<stdbool.h>**, which defines the macros **bool**, **true**, and **false**. Thus, code that is compatible with C/C++ can be easily created.

The reason that **_Bool** rather than **bool** is specified as a keyword is that many existing C programs have already defined their own custom versions of **bool**. By defining the Boolean type as **_Bool**, C99 avoids breaking this preexisting code. However, for new programs, it is best to include **<stdbool.h>** and then use the **bool** macro.

_Complex and _Imaginary

C99 adds support for complex arithmetic, which includes the keywords **_Complex** and **_Imaginary**, additional headers, and several new library functions. However, no implementation is required to implement imaginary types, and freestanding implementations (those without operating systems) do not have to support complex types. Complex arithmetic was added to C99 to provide better support for numerical programming.

The following complex types are defined:

- float _Complex
- float _Imaginary
- double _Complex
- double _Imaginary
- long double _Complex
- long double _Imaginary

The reason that **_Complex** and **_Imaginary**, rather than **complex** and **imaginary**, are specified as keywords, is that many existing C programs have already defined their own custom complex data types using the names **complex** and **imaginary**. By defining the keywords **_Complex** and **_Imaginary**, C99 avoids breaking this preexisting code.

The header **<complex.h>** defines (among other things) the macros **complex** and **imaginary**, which expand to **Complex** and **_Imaginary**. Thus, for new programs, it is best to include **<complex.h>** and then use the **complex** and **imaginary** macros.

The long long Integer Types

C99 adds the **long long int** and **unsigned long long int** data types. A **long long int** has a range of at least–(2_{63}–1) to 2_{63}–1. An **unsigned long long int** has a minimal range of 0 to 2_{64}–1. The **long long** types allow 64-bit integers to be supported as a built-in type.

Array Enhancements

C99 has added two important features to arrays: variable length and the ability to include type qualifiers in their declarations.

Variable-Length Arrays

In C89, array dimensions must be declared using integer constant expressions, and the size of an array is fixed at compile time. C99 changes this for certain circumstances. In C99, you can declare an array whose dimensions are specified by any valid integer expression, including those whose value is known only at run time. This is called a *variable-length array* (VLA). However, only local arrays (that is, those with block scope or prototype scope) can be of variable length.

Here is an example of a variable-length array:

```
void f(int dim1, int dim2)
{
int matrix[dim1] [dim2]; /* a variable-length, 2-D array */
/* . . . */
}
```

Here, the size of **matrix** is determined by the values passed to **f( )** in **dim1** and **dim2**. Thus, each call to f( ) can result in matrix being created with different dimensions.

It is important to understand that variable-length arrays do not change their dimensions during their lifetime. (That is, they are not dynamic arrays.) Rather, a variable-length array can be created with a different size each time its declaration is encountered.

You can specify a variable-length array of an unspecified size by using * as the size.
The inclusion of variable-length arrays causes a small change in the **sizeof** operator. In general, **sizeof** is a compile-time operator. That is, it is normally translated into an integer constant whose value is equal to the size of the type or object when a program is compiled. However, when it is applied to a variable-length array, **sizeof** is evaluated at run time. This change is necessary because the size of a variable-length array cannot be known until run time.

One of the major reasons for the addition of variable-length arrays to C99 is to support numeric processing. Of course, it is a feature that has widespread applicability. But remember, variable-length arrays are not supported by C89 (or by C++).

Use of Type Qualifiers in an Array Declaration

In C99 you can use the keyword **static** inside the brackets of an array declaration when that declaration is for a function parameter. It tells the compiler that the array pointed to by the parameter will always contain at least the specified number of elements. Here is an example:

```
int f(char str [static  80])
{
// here, str is always a pointer to an 80-element array
// . . .
}
```

In this example, **str** is guaranteed to point to the start of an array of **chars** that contains at least 80 elements.

You can also use the keywords **restrict**, **volatile**, and **const** inside the brackets, but only for function parameters. Using **restrict** specifies that the pointer is the sole initial means of access to the object.

Using **const** states that the same array is always pointed to (that is, the pointer always points to the same object). The use of **volatile** is allowed, but meaningless.

Single-Line Comments

C99 adds the single-line comment to C. This type of comment begins with // and runs to the end of the line.

For example:

```
// This is a comment
int i; // this is another comment
```

Single-line comments are also supported by C++. They are convenient when only brief, single-line remarks are needed. Many programmers use C's traditional multiline comments for longer descriptions, reserving single-line comments for "play-by-play" explanations.

Interspersed Code and Declarations In C89, within a block, all declarations must precede the first code statement. This rule does not apply for C99.

For example:

```
#include <stdio.h>
int main(void)
{
int i;
i = 10;
int j; // wrong for C89; OK for C99 and C++
j = i;
printf("%d %d", i, j);
return 0;
}
```

Here, the statement `i = 10;` comes between the declaration of **i** and the declaration of **j**. This is not allowed by C89. It is allowed by C99 (and by C++). The ability to intersperse declarations and code is widely used in C++.

Adding this feature to C makes it easier to write code that will be used in both environments.

PREPROCESSOR CHANGES

C99 makes a number of small changes to the preprocessor.

Variable Argument Lists

Perhaps the most important change to the preprocessor is the ability to create macros that take a variable number of arguments. This is indicated by an ellipsis (. . .) in the definition of the macro. The built-in preprocessing identifier **__VA_ARGS__** determines where the arguments will be substituted.

For example, given this definition

```
#define MyMax(. . .) max(__VA_ARGS__)
```

this statement

```
MyMax(a, b);
```

is transformed into

```
max(a, b);
```

There can be other arguments prior to the variable ones. For example, given

```
#define compare(compfunc, . . .) compfunc(__VA_ARGS__)
```

this statement

```
compare(strcmp, "one", "two");
```

is transformed into

```
strcmp("one", "two");
```

As the example shows, **__VA_ARGS__** is replaced by all of the remaining arguments.

The _Pragma Operator

C99 includes another way to specify a pragma in a program: the **_Pragma** operator. It has the following general form:

_Pragma ("directive")

Here, *directive* is the pragma being invoked. The addition of the **_Pragma** operator allows pragmas to participate in macro replacement.

Built-in Pragmas

C99 defines the following built-in pragmas:

Pragma	Meaning
STDC FP_CONTRACT ON/OFF/DEFAULT	When on, floating-point expressions are treated as indivisible units that are handled by hardware-based methods. The default state is implementation defined.
STDC FENV_ACCESS ON/OFF/DEFAULT	Tells the compiler that the floating-point environment might be accessed. The default state is implementation defined.
STDC CX_LIMITED_RANGE ON/OFF/DEFAULT	When on, tells the compiler that certain formulas involving complex values are safe. The default state is off.

You should refer to your compiler's documentation for details concerning these pragmas.

Additional Built-in Macros

C99 adds the following macros to those already supported by C89:

_ _STDC_HOSTED_ _	1 if an operating system is present.
_ _STDC_VERSION_ _	199901L or greater. Represents version of C.
_ _STDC_IEC_559_ _	1 if IEC 60559 floating-point arithmetic is supported.
_ _STDC_IEC_599_COMPLEX_ _	1 if IEC 60559 complex arithmetic is supported.
_ _STDC_ISO_10646_ _	A value of the form *yyyymmL* that states the year and month of the ISO/IEC 10646 specification supported by the compiler.

Declaring Variables within a for Loop

C99 enhances the **for** loop by allowing one or more variables to be declared within the initialization portion of the loop. A variable declared in this way has its scope limited to the block of code controlled by that statement. That is, a variable declared within a **for** loop will be local to that loop. This feature has been included in C because often the variable that controls a **for** loop is needed only by that loop. By localizing this variable to the loop, unwanted side effects can be avoided.

Here is an example that declares a variable within the initialization portion of a **for** loop:

```
#include <stdio.h>
int main(void)
{
// declare i within for
for(int i=0; i < 10; i++)
printf("%d ", i);
return 0;
}
```

Here, **i** is declared within the **for** loop, rather than prior to it.

As mentioned, a variable declared within a **for** is local to that loop. Consider the following program. Notice that the variable **i** is declared twice: at the start of **main( )** and inside the **for** loop.

```
#include <stdio.h>
int main(void)
{
int i = -99;
// declare i within for
for(int i=0; i < 10; i++)
printf("%d", i);
printf("\n");
```

```
printf("Value of i is: %d", i); // displays-99
return 0;
}
```

This program displays the following:

```
0 1 2 3 4 5 6 7 8 9
Value of i is: -99
```

As the output shows, once the **for** loop ends, the scope of the i declared within that loop ends. Thus, the final **printf( )** statement displays **–99**, the value of the **i** declared at the start of **main( )**.
The ability to declare a loop-control variable inside the **for** has been available in C++ for quite some time, and is widely used. It is expected that most C programmers will do the same.

Compound Literals

C99 allows you to define *compound literals*, which are array, structure, or union expressions designating objects of the given type. A compound literal is created by specifying a parenthesized type name, which is then followed by an initialization list, which must be enclosed between curly braces. When the type name is an array, its size must not be specified. The object created is unnamed.

Here is an example of a compound literal:

```
double *fp = (double[]) {1.0, 2.0, 3.0};
```

This creates a pointer to **double**, called **fp**, which points to the first of a three-element array of **double** values.

A compound literal created at file scope exists throughout the lifetime of the program. A compound literal created within a block is a local object that is destroyed when the block is left.

Flexible Array Structure Members

C99 allows you to specify an unsized array as the last member of a structure. (The structure must have at least one other member prior to the flexible array member.) This is referred to as a *flexible array member*. It allows a structure to contain an array of variable size. The size of such a structure returned by **sizeof** does not include memory for the flexible array.

Typically, memory to hold a structure containing a flexible array member is allocated dynamically, using **malloc( )**. Extra memory must be allocated beyond the size of the structure to accommodate the desired size of the flexible array.
For example, given

```
struct mystruct {
int a;
int b;
float fa[]; // flexible array
};
```

the following statement allocates room for a 10-element array:

```
struct mystruct *p;
p = (struct mystruct *) malloc(sizeof(struct mystruct) + 10 *
sizeof(float));
```

Since **sizeof(struct mystruct)** yields a value that does not include any memory for **fa**, room for the 10-element array of **floats** is added by the expression

```
10 * sizeof(float)
```

when **malloc( )** is called.

Designated Initializers

A new feature of C99 that will be especially helpful to those programmers working with sparse arrays is *designated initializers*. Designators take two forms: one for arrays and one for structures and unions. For arrays, the following form is used,

[index] = val

where *index* specifies the element being initialized to the value *val*. For example:

```
int a[10] = { [0]  = 100, [3] = 200 };
```

Here, only elements 0 and 3 are initialized.
For structure or union members, the form used is:

. member-name

Using a designator with a structure allows an easy means of initializing only selected members of a structure.

For example:

```
struct mystruct {
int a;
int b;
int c;
} ob = { .c = 30, .a = 10 };
```

Here, **b** is uninitialized.

Using designators also allows you to initialize a structure without knowing the order of its members. This is useful for predefined structures, such as **div_t**, or for structures defined by some third party.

Additions to the printf( ) and scanf( ) Family of Functions

C99 adds to the **printf( )** and **scanf( )** family of functions the ability to handle the **long long int** and **unsigned long long int** data types. The format modifier for **long long** is **ll**.

For example, the following fragment shows how to output a **long long int** and an **unsigned long long int**:

```
long long int val;
unsigned long long int u_val;
printf("%lld %llu", val, val2);
```

The **ll** can be applied to the **d**, **i**, **o**, **u**, and **x** format specifiers for both **printf( )** and **scanf( )**.

C99 adds the **hh** modifier, which is used to specify a **char** argument when using the **d**, **i**, **o**, **u**, or **x** format specifiers. Both the **ll** and **hh** specifiers can also be applied to the **n** specifier.

The format specifiers **a** and **A**, which were added to **printf( )**, cause a floating-point value to be output in a hexadecimal format. The format of the value is

[-]0xh.hhhhp+d

When **A** is used, the x and the p are uppercase. The format specifiers **a** and **A** were also added to **scanf( )**, and read a floating-point value.

In a call to **printf( )**, C99 allows the 1 modifier to be added to the **%f** specifier (as in **%lf**), but it has no effect. In C89, **%lf** is undefined for **printf( )**.

NEW LIBRARIES IN C99

C99 adds several new libraries and headers. They are shown here:

Header	Purpose
<complex.h>	Supports complex arithmetic.
<fenv.h>	Gives access to the floating-point status flags and other aspects of the floating-point environment.
<inttypes.h>	Defines a standard, portable set of integer type names. Also supports functions that handle greatest-width integers.
<iso646.h>	Added in 1995 by Amendment 1. Defines macros that correspond to various operators, such as **&&** and ^.
<stdbool.h>	Supports Boolean data types. Defines the macros **bool**, **true**, and **false**, which help with C++ compatibility.
<stdint.h>	Defines a standard, portable set of integer type names. This header is included by **<inttypes.h>**.
<tgmath.h>	Defines type-generic floating-point macros.
<wchar.h>	Added in 1995 by Amendment 1. Supports multibyte and widecharacter functions.
<wctype.h>	Added in 1995 by Amendment 1. Supports multibyte and widecharacter classification functions.

The _ _func_ _ Predefined Identifier

C99 defines **_ _func_ _**, which specifies the name (as a string literal) of the function in which **_ _func_ _** occurs.

For example:

```
void StrUpper(char *str)
```

```
{
static int i = 0;
i++;
printf("%s has been called %d time(s).\n", __func__, i);
while(*str) {
*str = toupper(*str);
str++;
}
}
```

When called the first time, **StrUpper( )** will display this output:

```
StrUpper has been called 1 time(s).
```

INCREASED TRANSLATION LIMITS

The term 'translation limits' refers to the minimum number of various elements that a C compiler must be able to handle. These include such things as the length of identifiers, levels of nesting, number of **case** statements, and number of members allowed in a structure or union. C99 has increased several of these limits beyond the already generous ones specified by C89.

Here are some examples:

Limit	C89	C99
Nesting levels of blocks	15	127
Nesting levels of conditional inclusion	8	63
Significant characters in an internal identifier	31	63
Significant characters in an external identifier	6	31
Members of a structure or union	127	1023
Arguments in a function call	31	127

Implicit int No Longer Supported

Several years ago, C++ dropped the implicit **int** rule, and with the advent of C99, C follows suit. In C89, the implicit **int** rule states that in the absence of an explicit type specifier, the type **int** is assumed. The most common use of the implicit **int** rule was in the return type of functions. In the past, C programmers often omitted the **int** when declaring functions that returned an **int** value.

For example, in the early days of C, **main( )** was often written like this:

```
main ()
{
/* . . . */
}
```

In this approach, the return type was simply allowed to default to **int**. In C99 (and in C++) this default no longer occurs, and the **int** must be explicitly specified, as it is for all of the programs in this book.

Here is another example. In the past a function such as

```
int isEven(int val)
{
return !(val%2);
}
```

would often have been written like this:

```
/* use integer default */
isEven (int val)
{
return !(val%2);
}
```

In the first instance, the return type of **int** is explicitly specified. In the second, it is assumed by default. The implicit **int** rule does not apply only to function return values (although that was its most common use).

For example, for C89 and earlier, the **isEven( )** function could also be written like this:

```
isEven(const val)
{
return ! (val%2);
}
```

Here, the parameter **val** also defaults to **int—**in this case, **const int**. Again, this default to **int** is not supported by C99.

> **Note** *Technically, a C99-compatible compiler can accept code containing implied **ints** after reporting a warning error. This allows old code to be compiled. However, there is no requirement that a C99-compatible compiler accept such code.*

Implicit Function Declarations Have Been Removed

In C89, if a function is called without a prior explicit declaration, then an implicit declaration of that function is created. This implicit declaration has the following form:

extern int *name*( );

Implicit function declarations are no longer supported by C99.

> **Note** *Technically, a C99-compatible compiler can accept code containing implied function declarations after reporting a warning error. This allows old code to be compiled. However, there is no requirement that a C99-compatible compiler accept such code.*

Restrictions on return

In C89, a function that has a non-**void** return type (that is, a function that supposedly returns a value) could use a **return** statement that did not include a value. Although this creates undefined behaviour, it was not technically illegal. In C99, a non-**void** function *must* use a **return** statement that returns a value. That is, in C99, if a function is specified as returning a value, any **return** statement within it must have a value associated with it. Thus, the following function is technically valid for C89, but invalid for C99:

```
int f(void)
{
/* . . . */
return ; // in C99, this statement must return a value
}
```

Extended Integer Types

C99 defines several extended integer types in **<stdint.h>**. Extended types include exact-width, minimum-width, maximum-width, and fastest integer types.
Here is a sampling:

Extended Type Meaning

int16_t An integer consisting of exactly 16 bits
int_least16_t An integer consisting of at least 16 bits
int_fast32_t Fastest integer type that has at least 32 bits
intmax_t Largest integer type
uintmax_t Largest unsigned integer type
The extended types make it easier for you to write portable code.

For loop changes In a for-loop, the first expression may be a declaration, with a scope encompassing only the loop.

```
for (decl; pred; inc)
  stmt;
```

is equivalent to:

```
{
  decl;
  for (; pred; inc)
    stmt;
}
```

Type specifiers or data types Type specifiers: new combinations added for:

- _Bool
- float _Complex, double _Complex, long double _Complex
- signed and unsigned long long int.

New type long long (signed and unsigned), at least 64 bits wide.

> **Note:** It seems that these type specifiers may occur in any order, e.g, `_Complex double long or signed long int long would be legal.`

The implementation of the complex types is defined by the standard to use cartesian coordinates (real and imaginary part), i.e. forbids an implementation using polar coordinates (distance from [0,0] and an angle). Furthermore, the same paragraph also specifies that a complex type has the same alignment requirements as an array of two elements of the corresponding floating types, the first must be the real part and the second the imaginary part.

Objects of the new boolean type `_Bool` may have one of the two values zero or one.

Format Modifiers for Printf( ) Added by C99

C99 adds several format modifiers to **printf( )**: **hh**, **ll**, **j**, **z**, and **t**. The **hh** modifier can be applied to **d**, **i**, **o**, **u**, **x**, **X**, or **n**. It specifies that the corresponding argument is a **signed** or **unsigned char** value or, in the case of **n**, a pointer to a **signed char** variable. The **ll** modifier also can be applied to **d**, **i**, **o**, **u**, **x**, **X**, or **n**. It specifies that the corresponding argument is a **signed** or **unsigned long long int** value or, in the case of **n**, a pointer to a **long long int**. C99 also allows the **l** to be applied to the floating-point specifiers **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G**, but it has no effect. The **j** format modifier, which applies to **d**, **i**, **o**, **u**, **x**, **X**, or **n**, specifies that the matching argument is of type **intmax_t** or **uintmax_t**. These types are declared in **<stdint.h>** and specify greatest-width integers. The **z** format modifier, which applies to **d**, **i**, **o**, **u**, **x**, **X**, or **n**, specifies that the matching argument is of type **size_t**. This type is declared in **<stddef.h>** and specifies the result of **sizeof**.

The **t** format modifier, which applies to **d**, **i**, **o**, **u**, **x**, **X**, or **n**, specifies that the matching argument is of type **ptrdiff_t**. This type is declared in **<stddef.h>** and specifies the difference between two pointers.

Sample code

```
#include <stdio.h>
int main(void)
{
/* This prints ''this is a test" left justified
in 20 character field.
*/
printf("%-20s", "this is a test");
/* This prints a float with 3 decimal places in a 10
character field. The output will be " 12.235".
*/
printf("%10.3f", 12.234657);
return 0;
}
```

Changes to the Integer Promotion Rules

C99 enhances the integer promotion rules. In C89, a value of type **char**, **short int** , or an **int** bit-field can be used in place of an **int** or **unsigned int** in an expression. If the promoted value can be held in an **int**, the promotion is made to **int**; otherwise, the original value is promoted to **unsigned int**.

In C99, each of the integer types is assigned a *rank*. For example, the rank of **long long int** is greater than **int**, which is greater than **char**, and so on. In an expression, any integer type that has a rank less than **int** or **unsigned int** can be used in place of an **int** or **unsigned int**.

THE PRINTF( ) FORMAT SPECIFIERS

Code	Format
%a	Hexadecimal output in the form 0x*h.hhhhp+d* (C99 only).
%A	Hexadecimal output in the form 0X*h.hhhhP+d* (C99 only).
%c	Character.
%d	Signed decimal integers.
%i	Signed decimal integers.
%e	Scientific notation (lowercase e).
%E	Scientific notation (uppercase E).
%f	Decimal floating point.
%F	Decimal floating point (C99 only; produces uppercase INF, INFINITY, or NAN when applied to infinity or a value that is not a number. The **%f** specifier produces lowercase equivalents.)
%g	Uses **%e** or **%f**, whichever is shorter.
%G	Uses **%E** or **%F**, whichever is shorter.
%o	Unsigned octal.
%s	String of characters.
%u	Unsigned decimal integers.
%x	Unsigned hexadecimal (lowercase letters).
%X	Unsigned hexadecimal (uppercase letters).
%p	Displays a pointer.
%n	The associated argument must be a pointer to an integer. This specifier causes the number of characters written (up to the point at which the **%n** is encountered) to be stored in that integer.
%%	Prints a percent sign.

RESTRICT-QUALIFIED POINTERS

The C99 standard has added a new type qualifier that applies only to pointers: **restrict**. A pointer qualified by **restrict** is initially the only means by which the object it points to is accessed. Access to the object by another pointer can occur only if the second pointer is based on the first. Thus, access to the object is restricted to expressions based on the **restrict**-qualified pointer. Pointers qualified by **restrict** are primarily used as function parameters or to point to memory allocated via **malloc( )**. By qualifying a pointer with **restrict**, the compiler is better able to optimize certain types of routines. For example, if a function specifies two **restrict**-qualified pointer parameters, then the compiler can assume that the pointers point to different (that is, non-overlapping) objects. The **restrict** qualifier does not change the semantics of a program.

1. **/* Write a program to test the given string is palindrome or not. implement the program by using _Bool datatype */**

```
#include<stdio.h>
#include<string.h>

enum _Bool{false,true};  // _Bool DataType which is capable of
storing the vales 0 and 1
enum _Bool IsPalindrome(char string[])  // _Bool datatype
{
 int left,right,len=strlen(string);
enum _Bool matched=true;   // _Bool datatype
 if(len==0)
  return 0;
  left=0;
  right=len-1;
  /* Compare the first and last letter,second & second last & so
on */
  while(left<right&&matched)
  {
   if(string[left]!=string[right])
   matched=false;
   else
   {
    left++;
    right--;
   }
  }
  return matched;
 }

int main()
{
  char string[40];
  clrscr();
   printf("****Program to test if the given string is a
palindrome****\n");
  printf("Enter a string:");
  scanf("%s",string);
  if(IsPalindrome(string))
  printf("The given string %s is a palindrome\n",string);
  else
  printf("The given string %s is not a palindrome\n",string);
  getch();
  return 0;
}
```

2. **/* In C89, within a block, all declarations must precede the first code statement. This rule does not apply for C99 */**

```
#include<stdio.h>
int main(void)
{
int i;
i=10;
int j; //wrong for c89; ok for c99
j=i;
clrscr();
printf("%d %d",i,j);
return 0;
getch();
}
```

3. **/* Write a program to print the signed and unsigned numbers by using C99 standards*/**

```
#include<stdio.h>
int main(void)
{
int i= -99;
{
for(int i=0; i<10; i++) //declare i within for
printf("%d",i);
printf("\n");
}
printf("Value of i is: %d",i); //display-99
return 0;
}
```

4. **/* Write a program to open the file by using C99 standards */**

```
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
FILE *fp;
if((fp=fopen("test","wb"))==NULL);
{
printf("cannot open file.\n");
exit(1);
fprintf(fp,"this is a test %d %f", 10,20.01); // fprintf
fclose(fp);
return 0;
}
}
```

5. /* Write a C program that uses functions to perform the following operations by using C99 standards(_Complex, _Imaginary):
(i) Reading a complex number
(ii) Writing a complex number
(iii) Addition of two complex numbers
(iv) Multiplication of two complex numbers
(Note: represent complex number using a structure.) */

```
#include<stdio.h>
#include<math.h>
#include<complex.h>

void _Complex(int opern);

struct comp
{
     double realpart;
     double _Imaginary;
};

void main()
{
int opern;
clrscr();
printf("\n\n \t\t\t***** MAIN MENU *****");
printf("\n\n Select your option: \n1 : ADD\n2 : MULTIPLY\n0 :
EXIT \n\n\t\t Enter your Option [  ]\b\b");

scanf("%d",&opern);

switch(opern)
{
 case 0:
   exit(0);
 case 1:
 case 2:
   _Complex(opern);
 default:
 main();
     }

}

void _Complex(int opern)
```

```
{
     struct comp w1, w2, w;

     printf("\n Enter two Complex Numbers (x+iy):\n Real Part of
      First  Number:");
     scanf("%lf",&w1.realpart);
     printf("\n Imaginary Part of First Number:");
     scanf("%lf",&w1._Imaginary);
     printf("\n Real Part of Second Number:");
     scanf("%lf",&w2.realpart);
     printf("\n Imaginary Part of Second Number:");
     scanf("%lf",&w2._Imaginary);

     switch(opern)
     {

     /*addition of complex number*/
     case 1:
   w.realpart = w1.realpart+w2.realpart;
   w._Imaginary = w1._Imaginary+w2._Imaginary;
   break;

     /*multiplication of complex number*/
     case 2:
                          w.realpart=(w1.realpart*w2.realpart)-
(w1._Imaginary*w2._Imaginary);

 w._Imaginary=(w1.realpart*w2._Imaginary)+(w1._Imaginary*w2.realpart);
   break;
     }

     if (w._Imaginary>0)
   printf("\n Answer = %lf+%lfi",w.realpart,w._Imaginary);
     else
   printf("\n Answer = %lf%lfi",w.realpart,w._Imaginary);
     getch();
     main();
}
```

6. /* Write a program to find the maximum number by using C99 standards(inline keyword) */

```
#include <stdio.h>
inline int max(int a, int b)  // inline keyword
{
return a > b ? a : b;
```

```
}
int main(void)
{
int x=5, y=10;
printf("Max of %d and %d is: %d\n", x,y,max(x,y));

return 0;
}
```

7. /* Write a C program to generate Pascal's Triangle by using C99 standards */

```
#include<stdio.h>
#include<conio.h>

void main()
{
 int bin,q,r;
 clrscr();
 bin=1;
 q=0;

 printf("Rows you want to input:");
 scanf("%d",&r);

 printf("\nPascal's Triangle:\n");

 while(q<r)
 {
  for(int p=40-3*q;p>0;--p) // declare p with in for
  printf(" ");
  for(int x=0;x<=q;++x)    // declare x with in for
  {
    if((x==0)||(q==0))
      bin=1;
    else
      bin=(bin*(q-x+1))/x;
      printf("%6d",bin);
  }

 printf("\n");
 ++q;
 }
getch();
}
```

8. /* Write a C program to generate all the prime numbers between 1 and n, where n is a value supplied by the user by using C99 standards.(long long int,declare variables with in for loop) */

```
#include <stdio.h>

void main()
{
long long int no,check;   // long long int
clrscr();
printf("<----------------------PRIME NO. SERIES-------------
----------->");
printf("\n\n\n\t\t\tINPUT THE VALUE OF N: ");
scanf("%d",&no);
printf("\n\nTHE PRIME NO. SERIES B/W 1 TO %lld : \n\n",no);


for(int counter = 1; counter <= no; counter++)    // declare
counter with in for
{
 check = 0;
 //THIS LOOP WILL CHECK A NO TO BE PRIME NO. OR NOT.

 for(int counter1 = counter-1; counter1 > 1 ; counter1--)
   if(counter%counter1 == 0)
   {
    check++;        // INCREMENT CHECK IF NO. IS NOT A PRIME NO.
    break;
   }
   if(check == 0)
   printf("%d\t",counter);
}
getch();
}
```

9. /* Write a program to print text and numbers by using C99 standards */

```
#include<stdio.h>
int main(void)
{
printf("%-20s","this is a test"); // left justified
printf("%10.3f",12.234657);
return 0;
}
```

10. /* Write a program for addition of matrices by using the restrict keyword */

```
void fadd(double a[static restrict 10],
const double b[static restrict 10]) // restrict keyword, restricrts a and b
{
int i;
for(i=0; i<10; i++)
{
if(a[i] < 0.0)
return;
a[i] += b[i];
}
return;
}
```

11. /* Write a C program to find the roots of a quadratic equation by using format modifiers in C99. */

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
float a,b,c,root1,root2;
clrscr();
printf("\n Enter values of a,b,c for finding roots of a qua-
dratic eq:\n");
scanf("%f%f%f",&a,&b,&c);
/*checking condition*/
if(b*b>4*a*c)
{
 root1=-b+sqrt(b*b-4*a*c)/2*a;
 root2=-b-sqrt(b*b-4*a*c)/2*a;
 printf("\n*****ROOTS ARE*****\n");
  printf("\nroot1=%1.3f\nroot2=%1.2f",root1,root2);  //format
modifiers for printf()
}
else
 printf("\n Imaginary Roots.");
 getch();
}
```

12. /* Write a C program to find the sum of individual digits of a positive integer by using long long int in C99 standards*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
  long long int num, k=1, sum=0; // long long int
  clrscr();
  printf("Enter the number whose digits are to be added:");
  scanf("%lld",&num);
while(num!=0)
{
  k=num%10;
  sum=sum+k;
  k=num/10;
  num=k;
}
printf("Sum of the digits:%lld",sum);
getch();
}
```

UNIT I

Chapter 1

Algorithms

1.1 INTRODUCTION

There is a tremendous similarity between the human actions and the way a computer performs various functions. Nicholas Wirth—the inventor of a computer language—'Pascal' used to say "A program is equal to algorithm + data". In the next few chapters, we will explore the meaning of these terms; first learning about and clearly analyzing human actions and then showing how computers also act similarly, thereby demystifying the complexity that surrounds computers.

We perform hundreds of activities during a day. An activity could be anything from brushing teeth to making tea and from going to work to having dinner. There are two basic ways to describe in detail, any of these activities:

1. By describing the process step-by-step—called as **algorithm**.
2. By representing the various steps in the form of a diagram—called as **flow chart**.

1.1.1 Algorithm/pseudocode

The term algorithm is very popular in the computer literature. An algorithm is also called **pseudocode.** At first, it might sound very complex. However, it is very simple to understand. In fact, we all perform hundreds of algorithms in our daily life without even realizing it!

Algorithm is the step-by-step method of performing any task. When we break up a big task into smaller steps, what we actually do is to create an algorithm. In other words, we perform hundreds of algorithms. Brushing teeth, making tea, getting ready for school or office, are all different algorithms.

For example, when making a cup of tea, we follow the algorithm as shown in Fig. 1.1 (with a few variations, of course!).

An interesting observation: Many steps can be further sub-divided into still smaller sub-steps. For example, we can write an algorithm for Step 1 (Boil water) by writing detailed steps for boiling water. However, it is up to an individual as to how detailed steps one should describe. Each step can be called as an **instruction**. Also, we can notice that steps 4 and 5 can be executed in parallel, i.e. we can boil milk while we wait. In actual life, there is a tremendous amount of parallelism in many of our actions. We normally hear, see and feel simultaneously to comprehend a situation. Within this also, when we see

a picture, for example, actually we see millions of spots of the picture to generate an image on our retina while we compare all dots in parallel with already stored millions of images to identify an object instantaneously! Therefore, **parallel computing** was developed to speed up things and ultimately try to imitate human actions in **artificial intelligence**. We need not discuss these in detail in later chapters, but continue with serial algorithms for now.

Begin

1. Boil water.
2. Put tea powder in the kettle.
3. Pour boiled water in the kettle.
4. Wait for three minutes.
5. Boil milk.
6. Put boiled milk in a cup.
7. Add sugar to the cup.
8. Empty the kettle in the cup.
9. Stir the cup with a spoon.

End

Fig. 1.1 *Algorithm for making tea*

1.1.2 Flow Chart

There is another way to write the steps involved in any process. This is by making use of various symbols. The symbols form a diagram that represents the steps in a pictorial fashion similar to an algorithm. This is also very easy to understand. Such a diagram is called **flow chart**. In its simplest form, a flow chart consists of a series of rectangles connected to each other by arrows. The rectangles represent the various steps and the arrows designate the flow. A flow chart for our tea-making example could be drawn as in Fig. 1.2.

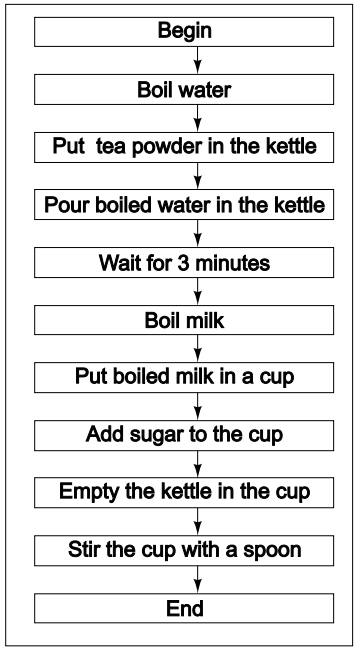


Fig. 1.2 *Flow chart for making tea*

As can be seen, these are very primary concepts that we learn since childhood. We learn to make tea by observing someone making tea: just like any other activity. Thus, the algorithm for making tea was recorded in our brain long back somewhere deep—in our memory. There is another part of our memory which stores the details of our current activity or thoughts. Whenever we want to make tea, somehow this algorithm is brought back from our deep memory into the current memory (i.e. it is 'remembered') and then it is actually executed. This concept of storing something in memory, retrieving it back in the current memory (i.e. 'remembering' it) whenever needed and actually performing a task is very primitive, yet extremely important, as we shall see, due to its similarity with computers.

1.2 THREE BASIC OPERATIONS

The tea-making algorithm and flow chart discussed earlier were quite simple. One step followed another in a sequential fashion. However, things are not so simple in real life! There are so many *if*s and *but*s. For example, consider the following statements:

- If it is 9 am, I must go to the office.
- If it is raining, take your umbrella with you.
- Read each line and say it loudly until the end of this page.

How do we take care of such things in an algorithm and a flow chart? And how many different kinds of situations we must cater to? This section attempts to answer these questions.

In general, the steps in an algorithm can be divided in three basic categories as listed below:

- **Sequence**—A series of steps that we perform one after the other
- **Selection**—Making a choice from multiple available options
- **Iteration**—Performing repetitive tasks

These three basic categories of activities combined in different ways can form the basis for describing *any* algorithm. It might sound surprising initially. But it is true. Think of any situation in our daily life and try to fit it in one of the three categories: it works!

Let us now look at each of the categories in more detail.

1.2.1 Sequence

A sequence is a series of steps that we follow in any algorithm without any break, i.e. unconditionally. The algorithm for making tea described in Fig. 1.1 belongs to this category. Figure 1.3 describes another algorithm in this category for 'boiling water'. What this means is that we have *exploded* further the step or instruction 1 in the algorithm for making tea given in Fig. 1.1. We can explode all such steps in Fig. 1.1 in the following way.

Begin

1. Take a utensil.
2. Take the utensil to the water tap.
3. Open the tap.
4. Fill the utensil with water.
5. Close the tap.
6. Put the utensil on the gas or in the oven.
7. Start the gas or the oven.
8. Wait until the water boils.
9. Switch off the gas or the oven.

End

Fig. 1.3 *Algorithm for boiling water*

We will not draw the flow chart for boiling water. It should be clear by now that it is a matter of writing all of the above 9 steps in rectangles one after the other; each connected to its successor by an arrow.

1.2.2 Selection

We will quickly realize that only 'sequence' is not good enough to express an algorithm. Quite a few of our actions depend on certain events. Thus, there is a need to be able to make a choice from many available options. Therefore, there is a process of selection. A selection statement generally takes the form as shown in Fig. 1.4.

1. **If** …
2. **Then**…
3. **End-if**

Fig. 1.4 *Selection*

We take so many decisions, small and big, in our daily life without even realizing it. For example, if it is hot, we switch on the fan. We can depict this selection process as shown in Fig. 1.5.

1. **If** it is hot
2. **Then** switch on the fan
3. **End-if**

Fig. 1.5 *Example of a selection*

Note that *End-if* is an additional step that indicates the end of the selection process. A question may be asked: is *End-if* really necessary? Let us illustrate this by the following selection process that does not have an *End-if*. Refer to Fig. 1.6.

1. **If** the guest wants tea
2. **Then** make tea
3. Offer biscuits

Fig. 1.6 *Importance of End-if*

What do we do if the guest does not want tea? Do we offer him biscuits? It is not clear from the selection process described above. It can be argued and proved either way. That is, we are not sure whether the *Offer biscuits* portion is a part of our *If* condition or not. A miserly person would always say that he considers *Offer biscuits* as a part of the condition, and hence would only offer biscuits to someone who wants tea! Hence, it is always better to end a selection process with an *End-if* to avoid any confusion at least for our algorithms. Different computer programming languages have different conventions.

The position and placement of *End-if* instructions can change the meaning of the entire algorithm as shown in Fig. 1.7 (a) and 1.7 (b). Let us study the algorithm shown in Fig. 1.7 (a). If the guest wants tea, the algorithm will execute step 2 after step 1 (i.e. make tea), then fall through step 3 and then step 4 (i.e. offer biscuits). Thus, if the guest wants tea, he gets the biscuits, too! Now, let us trace the algorithm if the guest does not want tea. In this case, the algorithm will follow step 3 after step 1 (i.e. skip the tea) and then fall through step 4 (i.e. offer biscuits). Thus, biscuits will be offered regardless of tea. If we study algorithm shown in Fig. 1.7 (b), we will notice that biscuits are offered only with tea. If the guest

does not want tea, the algorithm will follow step 1 and directly step 4. The semicolon at the end of step 2 indicates that tea and biscuits are offered together. You will notice that the positioning of *End-if* (at step 3 or 4) has made all the difference! The Figs 1.8 (a) and 1.8 (b) show the corresponding flow charts.

Begin
1. **If** the guest wants tea
2. **Then** make tea
3. **End-if**
4. Offer biscuits

End

Fig. 1.7(a) *Offer biscuits to all guests*

Begin
1. **If** the guest wants tea
2. **Then** make tea;
3. offer biscuits
4. **End-if**

End

Fig. 1.7(b) *Offer biscuits only to guests who want tea*

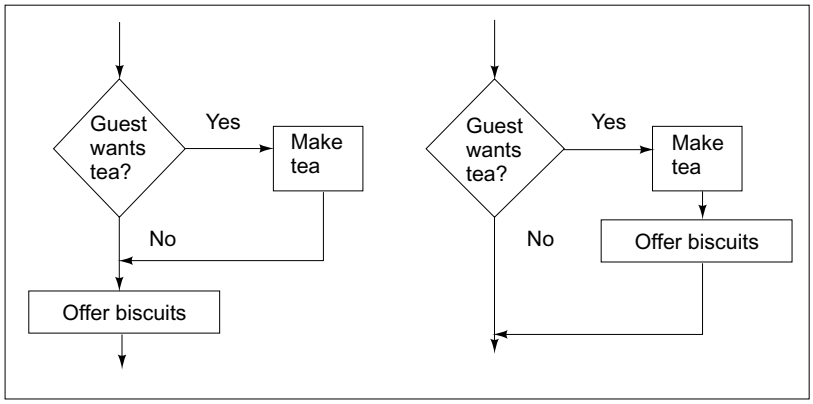


Fig. 1.8(a) **Fig. 1.8(b)**

In both of these cases (a and b), we have not specified the action if the guest wants coffee. Let us modify our algorithm and also the flow chart to accommodate that possibility. Note that we are offering coffee as an alternative to tea. We will be generous enough to offer biscuits in either case. Since we want to offer the biscuits irrespective of the acceptance/rejection of tea/coffee offer, we should consider only the first algorithm for modification, viz. Fig. 1.7(a). The modified algorithm and its corresponding flow chart are shown in Figs 1.9 and 1.10 respectively.

Begin
1. **If** the guest wants tea
2. **Then** make tea
3. **End-if**
4. **If** the guest wants coffee
5. **Then** make coffee
6. End-if
7. Offer biscuits

End

Fig. 1.9 *Algorithm for offering tea/coffee and biscuits*

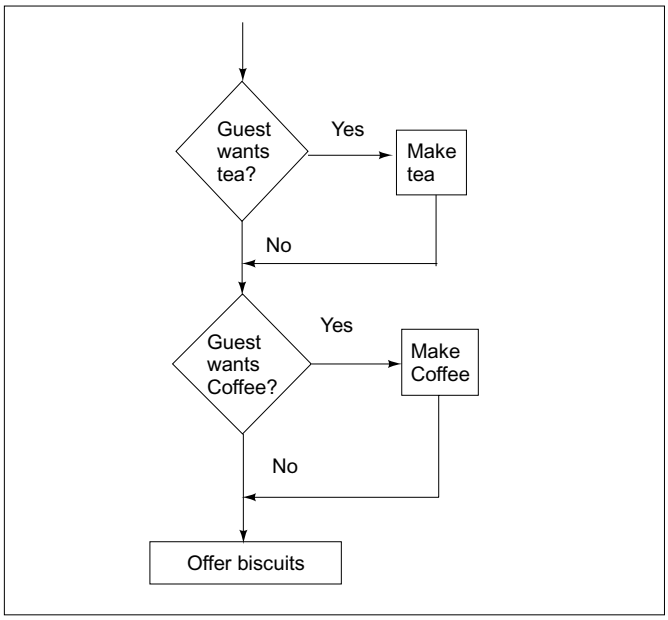


Fig. 1.10 *Flow chart for offering tea/coffee and biscuits*

Note that we offer biscuits regardless of the acceptance of the tea/coffee offer. That is exactly what we had wanted. Consider the following algorithm as shown in Fig. 1.11(a). Figure 1.11(b) shows the corresponding flow chart.

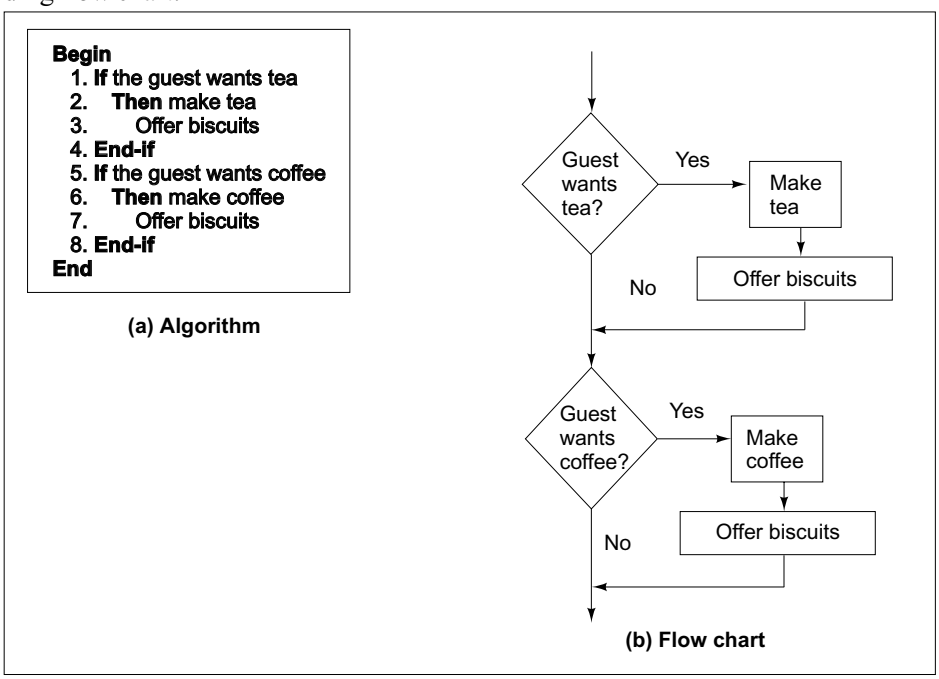


Fig. 1.11 *Offering biscuits only if guest wants tea /coffee*

Can it achieve the same objective as the algorithm shown in Fig. 1.9? A close examination will reveal that it is *not* the same. For instance, if the guest rejects both tea and coffee, the algorithm shown in Fig. 1.9 offers the guest biscuits regardless. However, the algorithm shown in Fig. 1.11(a) does not offer anything! How easily things that seem similar can mislead us, as they actually are different. These are the things that one needs to be careful of while writing an algorithm. Otherwise, it leads to the programming errors called as **bugs**, which programmers refer to, all the time.

We will notice some problem in the algorithm and flow charts depicted in Figs 1.9–1.11. After a guest chooses to have tea, the algorithm still asks if the guest wants coffee. Unless a guest wants both simultaneously, this is completely wasteful. We can avoid this by introducing a **goto** instruction. Using this, we can rewrite the algorithm of Fig. 1.9 as shown in Figs 1.12 (a) and 1.12 (b).

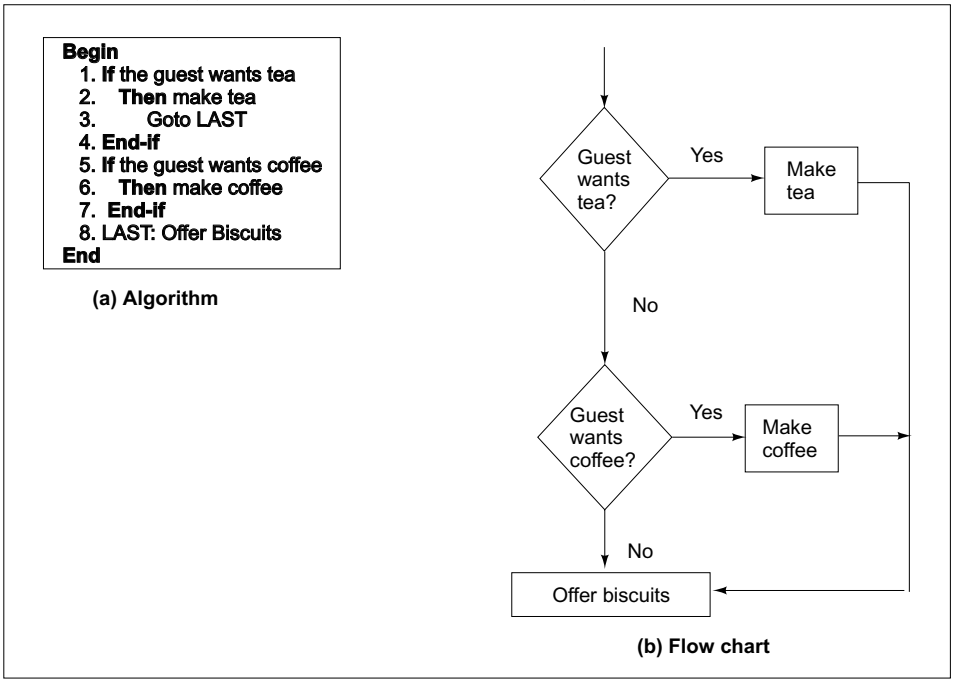


Fig. 1.12 *Improved algorithm to avoid unnecessary step*

In this algorithm, *LAST* is called **label** for an instruction. *Offer biscuits* (step 8). This allows the *goto* instruction to branch to a specific instruction. However, the *goto* makes a program difficult to understand and therefore, modify. So, generally, *goto*s are avoided. If you avoid such a *goto* in a program, you can read the program starting at the top and ending at the bottom without branching up or down. This is called as **goto-less** or **top-down** or **structured** programming method. Such programs are easy to understand and therefore, easy to maintain. We will see how the same algorithm can be written in a structured fashion later.

1.2.2.1 The compound conditions

It is very easy to group different conditions into one. We generally use words like *and* and/or *or* to join sentences. In the same way, we can combine two or more conditions into a single **compound condition**. For example, take a look at the algorithm shown in Fig. 1.13.

```
1. If it is a weekday
2.     and it is 7 am or more
3.     and you are feeling ok
4.   Then
5.       Take breakfast
6.       Go to work
7. End-if
```

Fig. 1.13 *Compound conditions with and*

In this algorithm, you are doing two things together, viz. taking breakfast and going to work. However, you perform these only if all the three conditions are met, viz. it has to be a weekday, it has to be at least 7 am and you have to be feeling good. If any of these is not valid, you will not perform both of these actions. For instance, if it is 7 am, and you are feeling ok, but if it is a weekend, you will not perform both the actions. But then if you take the algorithm very seriously in real life, you will notice a bug. You will not go to work on a weekend all right, but you will also have to skip your breakfast! So, be careful!

Similarly, the algorithm in Fig. 1.14 illustrates the use of *or* word to join two or more conditions. (Note that we are now assuming implicit *Begin* and *End* statements for our algorithms.)

```
1. If it is a weekday
2.       or work is pending
3.       or boss is angry
4.   Then
5.       Take breakfast
6.       Go to work
7. End-if
```

Fig. 1.14 *Compound conditions with or*

In this case, even if any of the conditions is met, you have breakfast and go to work. For instance, if your boss is angry, then even if it is a holiday and there is actually no pending work, you will still perform these two actions. Only if all the three conditions are not met, you skip having breakfast and going to work.

1.2.2.2 The nested conditions

Sometimes, things are not very easy to express. With a variety of situations to worry about, it becomes really complicated. For example, if something depends on still something else to happen (or otherwise), there will be an *if* within another *if*. We will call it a **nested condition**. Let us look at an example.

```
1. If you are feeling ok
2.   Then go to work
3.   Else
4.     If you have fever
5.       Then go to the doctor
6.       Else
7.           Just relax
8.     End-if
9. End-if
```

Fig. 1.15 *Nested conditions*

You will notice that there is an *If* at the beginning (step 1) and corresponding *End-if* at the bottom (step 9), which denotes the end of that *If* condition. Within this, you will notice another *If* and *End-if* pair

written in the same columns (steps 4 and 8). The *Else* provides an alternative path of execution. Let us trace the algorithm if you are ok. In this case, the algorithm executes step 2, skips steps 3 through 8 and goes directly to step 9. If you are not ok but you do not have fever, the algorithm goes through steps 1, 3, 4, 6, 7, 8 and 9. You will not go to work and you will just relax, but you will not consult a doctor even if you have stomach pain. This is a bug!

If you are not ok and if you have fever, you will go through steps 1, 3, 4, 5, 8 and 9. In this case, you will not go to work, you will not consult the doctor, but you will not relax. So, there is another bug. Try removing it!

Let us rewrite the first algorithm of Fig. 1.6 using nested conditions. This will make the benefit of *else* clearer.

```
1. If the guest wants tea
2.     Then make tea
3.     Else
4.         If the guest wants coffee
5.             Then make coffee
6.         End-if
7. End-if
8. Offer biscuits
```

Fig. 1.16 *The 'tea-coffee' algorithm using nested conditions*

As can be seen, if the guest accepts the offer for tea, we would simply make tea and offer biscuits without bothering if he wants coffee (unless the guest wants both!). This does not happen in Fig. 1.6. It might not make a big difference if we have just one or two conditions to check. However, as the number of conditions increases, it is desired that the algorithm be as compact and efficient as possible.

As a further illustration, let us write an algorithm to find the largest of any three given numbers. It is very easy to write an algorithm to find the larger of two numbers. Assuming that the two numbers are a and b, we simply need to compare the two. However, when there are three numbers, things are a bit complicated. The algorithm is as shown in Fig. 1.17.

```
 1. If a > b
 2.   Then If a > c
 3.             Then choose a
 4.             Else choose c
 5.          End-if
 6.   Else If b > c
 7.             Then choose b
 8.             Else choose c
 9.          End-if
10. End-if
```

Fig. 1.17 *Selecting the largest of three numbers*

To ascertain that the algorithm indeed works as expected, we might assign values as 15, 2 and 21 to a, b and c respectively. When we imagine these numbers in place of a, b and c in the above algorithm and check if the algorithm works correctly, it is called as a **pencil run** of our algorithm. It is also termed as a **walkthrough** of an algorithm. It is always a good practice to do a walkthrough of an algorithm with a variety of values. That is, we should take a, b and c such that once a is the greatest of them all (e.g. a = 7, b = 2 and c = 1), then b (e.g. a = 13, b = 20, c = 10) and then c (e.g. a = 18, b = 24, c = 49). This will make sure that our algorithm gives the correct result in each case. When we are doing a walkthrough of our algorithm in this fashion, we are actually checking if it works fine.

1.2.2.3 Testing and debugging

What the above discussion means is, we are checking to see if our algorithm gives the desired result. Hence, we are **testing** our algorithm. A set of values of a, b and c that we use for testing (e.g. a = 18, b = 24, c = 49) is called as a **test case** as we test our algorithm for a possibility (or case) at a given time. A group of test cases makes up the **test data**.

If for any one of such conditions, our algorithm gives an incorrect result, we say that there is a **bug** in the algorithm—it is another name for an error. It is necessary then to correct this bug, that is, change the algorithm accordingly and do a walkthrough again. Now the algorithm should give the desired result. This process of removing a bug from an algorithm is called as **debugging**.

To understand this, let us deliberately introduce a bug in our algorithm. The modified algorithm to find out the largest of three numbers is shown in Fig. 1.18. The statements in Italics are interchanged from Fig. 1.17.

```
1. If a > b
2.   Then If a > c
3.           Then choose a
4.           Else choose c
5.        End-if
6.   Else If b > c
7.           Then choose c
8.           Else choose b
9.        End-if
10. End-if
```

Fig. 1.18 *A bug in an algorithm*

Let us test our modified algorithm as shown in Fig. 1.18 with values as a=1, b=5 and c=9. You will realize that our algorithm gives a result that b is the largest of the three. This is, of course, incorrect. This is how debugging helps us in identifying and correcting mistakes.

1.2.2.4 Indentation and Algorithm maintenance

A question: why write *end-if* exactly below *if* ? Why to start some sentences at the beginning and some after leaving some spaces? The algorithm could as well be written as shown in Fig. 1.19.

```
1. If the guest wants tea
2. Then make tea
3. Else
4. If the guest wants coffee
5. Then make coffee
6. End-if
7. End-if
8. Offer biscuits
```

Fig. 1.19 *Indentation*

Well, the guest might end up having to drink coffee when he actually wanted tea! The obvious problem with this style of writing is that it is not easy to follow. That is, it is not **indented**. Further, if someone is to change it (add another drink, for example), it is not easy to see immediately where to make changes. That, in software terms means an algorithm that is difficult to **maintain**.

1.2.3 Iteration

Let us imagine a different situation. Suppose that I have asked my friend to wait for me until I arrive. Then we two would go to see a movie. How would my friend write an algorithm for this situation? At first, it might appear that one should be able to do this with the help of (a series of) If-Then-Else statements. Let us say the friend decides to check my arrival every two minutes. So, the algorithm in Fig. 1.20 should suffice.

```
1. If I arrive
2.   Then we go for a movie
3.   Else wait for two minutes
4.     If I arrive
5.       Then we go for a movie
6.       Else wait for two minutes
7.         If I arrive
8.           Then…
```

Fig. 1.20 *Need for iteration*

Wait a minute. How many times should this be written? If I arrive two hours hence, for example, the above instructions would have to be written 60 times! But if I come after four hours from now, they would be repeated 120 times!! This is like imagining what if the clocks were not round. Think that a clock to be a straight line. After 12, it is 1 all right, but the hourly clock hand moves straight (for simplicity, disregard the seconds and minutes). So, we have:

1 2 3 4 5 6 7 8 9 10 11 12 1 2 3 4 5 6 7 8 9 10 11 12

As can be seen, just for one day, we would need 24 sets of 1-12. We have the same problems with our algorithm:

- It is too long and repetitive
- We do not have any control over it as we are not sure when it will end

An iteration algorithm would solve these problems. There are many ways to write this algorithm. Figure 1.21 shows all of them. Let us see draw the corresponding flow chart also.

```
1. Repeat
2.     Wait for two minutes
3.  until I arrive
4. Go to movie
```

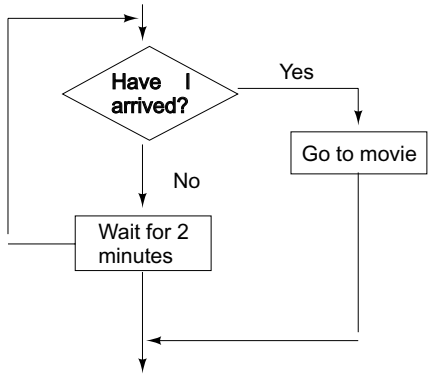
(a)

```
1. Do
2.    Wait for two minutes
3. While I do not arrive
4. Go to movie
```

(b)

```
1. While I do not arrive
2.    Wait for two minutes
3.  End-while
4. Go to movie
```

(c)



Flow chart for case

Fig. 1.21 *Iteration*

Repeat-until and Do-while (cases a and b) are essentially the same. Both check the success/failure using common terms like *until* and *while*. However, they differ significantly from (c) (While-End-while). We shall now see, why. What happens if I arrive before the algorithm was executed – that is, at the beginning of the algorithm? In (a) and (b), my friend will still wait for two minutes! The algorithms first ask my friend to wait for two minutes and then check if I have arrived. On the other hand, case (c) first checks to see if I have arrived. Only then does it ask my friend to wait for two minutes. Note that the flow chart is drawn to suit style (c). We could as well draw it for the first two (wait first and then check). It must be pointed out that algorithms (a) and (b) are not bad at all. In some situations we could prefer case (a) or (b). In others, we would opt for (c).

To summarize, sequence, selection and iteration form the building blocks for writing any algorithm. These three basic types of instructions can deal with virtually any situation and can put it in words. There are many different ways to write the same algorithm. Our aim should be to make it as compact and understandable as possible without any loss of meaning.

POINTS TO REMEMBER

1. Describing the process step-by-step is called as **algorithm**.
2. Representing the various steps in the form of a diagram is called as **flow chart**.
3. **Parallel computing** was developed to speed up things and ultimately try to imitate human actions in **artificial intelligence**.
4. Sequence involves a series of steps that we perform one after the other.
5. Selection involves making a choice from multiple available options.
6. Iteration involves performing repetitive tasks.
7. The above three basic categories of activities combined in different ways can form the basis for describing *any* algorithm.
8. Programs without go to statements are easy to understand and therefore, easy to maintain.
9. We can combine two or more conditions into a single **compound condition**.
10. Putting an *if* within another *if* forms a **nested condition.**
11. It is always a good practice to do a walkthrough of an algorithm with a variety of values.
12. The process of removing a bug from an algorithm is called as **debugging.**

Review Questions and Exercises

1.1 Point down the differences between an algorithm and a flowchart.
1.2 Write an algorithm for withdrawing Rs. 1000 from the bank.
1.3 Draw a flowchart for the above.
1.4 Describe in detail the steps involved in testing.
1.5 Discuss sequence, selection and iteration in detail.

State whether the following statements are True or False

(a) Describing the process step by step is called as flowchart.
(b) Algorithm involves very complex process.
(c) When we break up a big task into smaller steps, what we actually do is to create an algorithm.

(d) Each step in an algorithm can be called as an instruction.
(e) Parallel computing slows down the things.
(f) In general, the steps in an algorithm can be divided in five basic categories.
(g) Making a choice from multiple available options is called as a sequence.
(h) Performing repetitive tasks is called as iteration.
(i) The positioning of *End-if* can change the meaning in a process.
(j) Avoiding a *goto* in a program, makes it a goto-less or top-down or structured.
(k) goto-less programs are difficult to understand.
(l) We can combine two or more conditions into a single compound condition.
(m) If there is an *if* within another *if* then it is called as a compound condition.
(n) Checking to see if our algorithm gives the desired result is called as a test case.
(o) A group of test cases makes up the test data.
(p) Sequence, selection and iteration form the building blocks for writing any algorithm.

Answers to True and False Questions

(a) False	(b) False	(c) True	(d) True
(e) False	(f) False	(g) False	(h) True
(i) True	(j) True	(k) True	(l) True
(m) True	(n) True	(o) True	(p) True

Chapter 2

Overview of C

2.1 INTRODUCTION

'C' seems a strange name for a programming language. But this strange sounding language is one of the most popular computer languages today. C was an offspring of the 'Basic Combined Programming Language' (BCPL) called B, developed in the 1960s at Cambridge University. B language was modified by Dennis Ritchie and was implemented at Bell Laboratories in 1972. The new language was named C. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell Laboratories, was coded almost entirely in C.

For many years, C was used mainly in academic environments, but eventually with the release of C compilers for commercial use and the increasing popularity of UNIX, it began to gain widespread support among computer professionals. Today, C is running under a number of operating systems including MS-DOS. Since MS-DOS is a dominant operating system for microcomputers, it is natural that C has begun to influence the microcomputer community at large.

This book describes the features of C that are generally supported by most compilers. It also includes all the essential features of ANSI C.

2.2 IMPORTANCE OF C

The increasing popularity of C is probably due to its many desirable qualities. It is a robust language whose rich set of built-in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assembly language with the features of a high-level language and therefore it is well suited for writing both system software and business packages. In fact, many of the C compilers available in the market are written in C.

Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators. It is many times faster than BASIC. For example, a program to increment a variable from 0 to 15000 takes about one second in C while it takes more than 50 seconds in an interpreter BASIC.

There are only 32 keywords and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs.

C is highly portable. This means that C programs written for one computer can be run on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.

C language is well suited for structured programming, thus requiring the user to think of a problem in terms of function modules or blocks. A proper collection of these modules would make a complete program. This modular structure makes program debugging, testing and maintenance easier.

Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to the C library. With the availability of a large number of functions, the programming task becomes simple.

2.3 SAMPLE C PROGRAMS

Before discussing any specific features of C, we shall look at some sample C programs and analyse and understand how they work.

Sample Program 1: Printing a Message

Consider a very simple program given in Fig. 2.1. This program when executed, will produce the following output:

```
I see, I remember
```

```
main( )
{
/* .....printing begins..... */
printf("I see, I remember");
/* .....printing ends..... */
}
```

Fig. 2.1 *A program to print one line of text*

Let us have a close look at the program. The first line informs the system that the name of the program is **main** and the execution begins at this line. The **main( )** is a special function used by the C system to tell the computer where the program starts. Every program must have *exactly one* **main** function. If we use more than one **main** function, the compiler cannot tell which one marks the beginning of the program.

The empty pair of parentheses immediately following **main** indicates that the function **main** has no *arguments* (or parameters). The concept of arguments will be discussed in detail later when we discuss functions (in Chapter 10).

The opening brace '{' in the second line marks the beginning of the function **main** and the closing brace '}' in the last line indicates the end of the function. In this case, the closing brace also marks the end of the program. All the statements between these two braces form the *function body*. The function body contains a set of instructions to perform the given task.

In this case, the function body contains three statements, out of which only the **printf** line is an executable statement. The lines beginning with /* and ending with */ are known as *comment* lines. These are used in a program to enhance its readability and understanding. Comment lines are not execut-

able statements and therefore anything between /* and */ is ignored by the compiler. In general, a comment can be inserted anywhere blank spaces can occur—at the beginning, middle, or end of a line, but never in the middle of a word.

Because comments do not affect the execution speed and the size of a program, we should use them liberally in our porgrams. They help the programmers and other users in understanding the various functions and operations of a program and serve as an aid to debugging and testing. We shall see the use of comment lines more in the examples that follow.

Let us now look at the **printf( )** function, the only executable statement of the program.

```
printf("I see, I remember");
```

printf is a predefined, standard C function for printing output. *Predefined* means that it is a function that has already been written and compiled, and linked together with our program at the time of linking. The concepts of compilation and linking are explained later in this chapter (Section 2.4). The **printf** function causes everything between the starting and the ending quotation marks to be printed out. In this case, the output will be:

```
I see, I remember
```

Note that the print line ends with a semicolon. *Every statement in C should end with a semicolon* (;) mark.

Suppose we want to print the above quotation in two lines as

```
I see,
I remember !
```

This can be achieved by adding *another* **printf** function as shown below:

```
printf("I see, \n");
printf("I remember !");
```

The information contained between the parentheses is called the *argument* of the function. The argument of the first **printf** function is "I see,\n" and the second is "I remember !". These arguments are simply strings of characters to be printed out.

Notice that the argument of the first **printf** contains a combination of two characters \ and **n** at the end of the string. This combination is collectively called the *newline* character. A newline character instructs the computer to go to the next (new) line. It is similar in concept to the carriage return key on a typewriter. After printing the character comma (,) the presence of the newline character **\n** causes the string "I remember" to be printed on the next line. No space is allowed between \ and n.

If we omit the newline character from the first **printf** statement, then the output will again be a single line as shown below.

```
I see, I remember !
```

This is similar to the output of the program in Fig. 2.1. However, note that there is no space between , and I.

It is also possible to produce two or more lines of output by one **printf** statement with the use of newline characters at appropriate places. For example, the statement

```
printf("I see,\n I remember !");
```

will output

```
I see,
I remember !
```

while the statement

```
printf("I\n.see,\n.....I\n.....remember !");
```

will print out

```
I
. See,
.....I
.....remember !
```

Before we proceed to discuss further examples, we must note one important point. C does make a distinction between *uppercase* and *lowercase* letters. For example, **printf** and **PRINTF** are not the same. In C, everything is written in lowercase letters. However, uppercase letters are used for symbolic names representing constants. We may also use uppercase letters in output strings like "I SEE" and "I REMEMBER".

The above exam[illegible]**r** is one of the simplest programs. Figure 2.2 highlights the gener[illegible] All C programs need a **main** function.

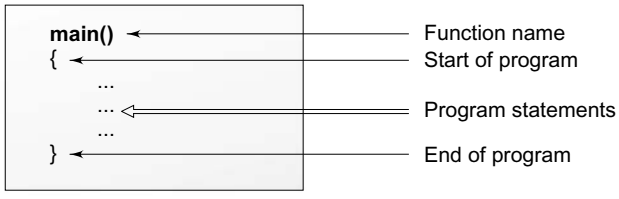


Fig. 2.2 *Format of simple C programs*

Sample Program 2: Adding Two Numbers

Consider another program which performs addition on two numbers and displays the result. The complete program is shown in Fig. 2.3.

```
/* Program ADDITION                          line-1   */
/* Written by EBG                            line-2   */
main()                                       /* line-3   */
{                                            /* line-4   */
        int number;                          /* line-5   */
        float amount;                        /* line-6   */
                                             /* line-7   */
        number = 100;                        /* line-8   */
                                             /* line-9   */
        amount = 30.75 + 75.35;              /* line-10  */
        printf("%d\n", number);              /* line-11  */
        printf("%5.2f", amount);             /* line-12  */
}                                            /* line-13  */
```

Fig. 2.3 *Program to add two numbers*

This program when executed will produce the following output:

```
100
106.10
```

The first two lines of the program are comment lines. It is a good practice to use comment lines in the beginning to give information such as name of the program, author, date, etc. Comment characters are also used in other lines to indicate line numbers.

The words **number** and **amount** are *variable names* that are used to store numeric data. The numeric data may be either in *integer* form or in *real* form. In C, *all variables should be declared*, to tell the compiler what the variable names are and *what type of data* they hold. The variables must be declared before they are used. In lines 5 and 6, the declarations

```
int number;
float amount;
```

tell the compiler that **number** is an integer (**int** is the abbreviation for integer) and **amount** is a floating (**float**) point number. Declaration statements must appear at the beginning of the function as shown in Fig. 2.3. All declaration statements end with a semicolon. C supports many other data types and they are discussed in detail in Chapter 3.

The words such as **int** and **float** are called the *keywords* and cannot be used as variable names. A list of keywords is given in Chapter 3.

Data is stored in a variable by *assigning* a data value to it. This is done in lines 8 and 10. In line-8, an integer value 100 is assigned to the integer variable **number** and in line-10, the result of addition of two real numbers 30.75 and 75.35 is assigned to the floating point variable **amount**. The statements

```
number = 100;
amount = 30.75 + 75.35;
```

are called the *assignment* statements. Every assignment statement must have a semicolon at the end.

The next statement is an output statement that prints the value of **number**. The print statement

```
printf("%d\n", number);
```

contains two arguments. The first argument "%d" tells the compiler that the value of the second argument **number** should be printed as a *decimal integer*. Note that these arguments are separated by a comma. The newline character \ n causes the next output to appear on a new line.

The last statement of the program

```
printf("%5.2f", amount);
```

prints out the value of **amount** in floating point format. The format specification %5.2f tells the compiler that the output must be in floating point, with five places in all and two places to the right of the decimal point.

2.4 PROGRAM DEVELOPMENT STEPS

Executing a program written in C involves a series of steps. These are:

1. Creating the program.
2. Compiling the program.

3. Linking the program with functions that are needed from the C library.
4. Executing the program.

Figure 2.4 illustrates the process of creating, compiling and executing a C program. Although these steps remain the same irrespective of the *operating system*, system commands for implementing the steps and conventions for naming *files* may differ on different systems.

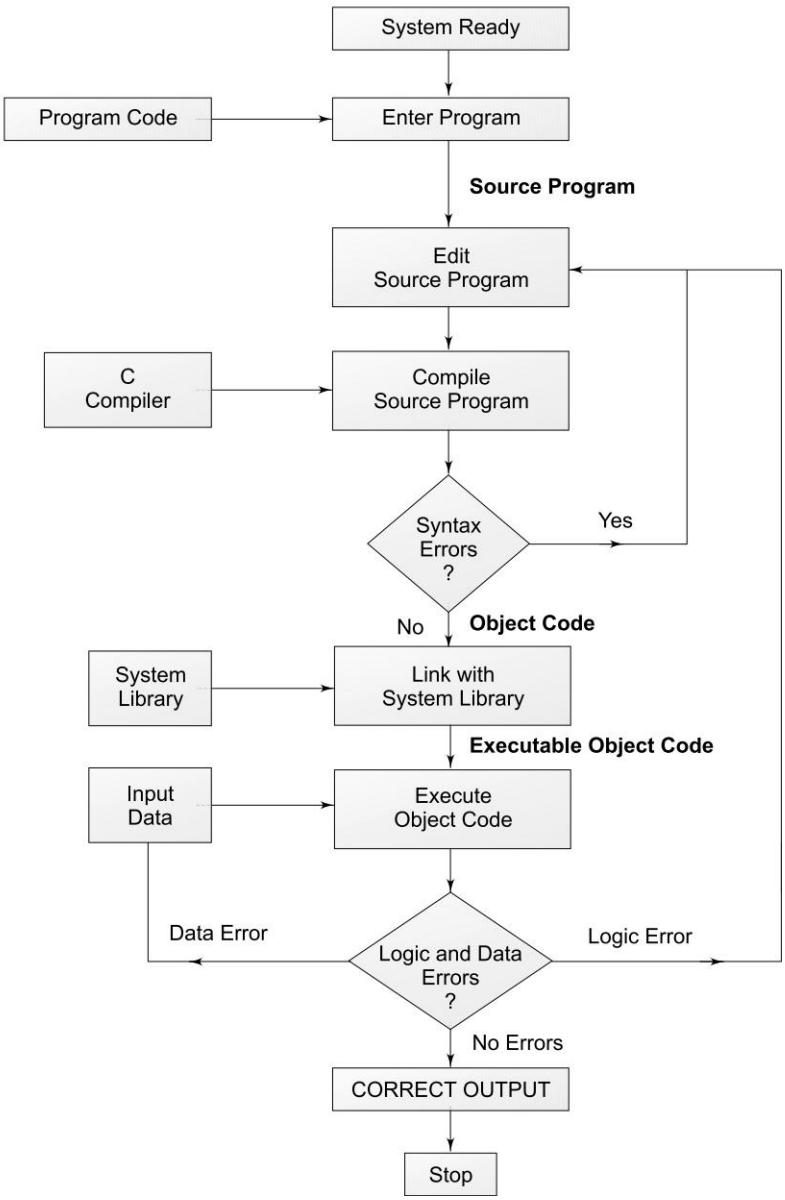


Fig. 2.4 *Process of compiling and running a C program*

An operating system is a program that controls the entire operation of a computer system. All input/output operations are channelled through the operating system. The operating system, which is an interface between the hardware and the user, handles the execution of user programs.

The two most popular operating systems today are UNIX (for minicomputers) and MS-DOS (for microcomputers). We shall discuss briefly the procedure to be followed in executing C programs under both these operating systems.

UNIX System

Creating the program

Once we load the UNIX Operating system into the memory, the computer is ready to receive the program. The program must be entered into a *file*. The file name can consist of letters, digits and special characters, followed by a *dot* and a letter **c**. Examples of valid file names are:

hello.c
program.c
ebg1.c

The file is created with the help of a text *editor*, either **ed** or **vi**. The command for calling the editor and creating the file is

ed *filename*

If the file existed before, it is loaded. If it does not yet exist, the file has to be created so that it is ready to receive the new program. Any corrections in the program are done under the editor. (The name of your system's editor may be different. Check your system manual.)

When the editing is over, the file is saved on disk. It can than be referenced any time later by its file name. The program that is entered into the file is known as the *source program*, since it represents the original form of the program.

Compiling and Linking

Let us assume that the program has been created in a file named *ebg1.c*. Now the program is ready for compilation. The compilation command to achieve this task under UNIX is

cc *ebg1.c*

The source program instructions are now translated into a form that is suitable for execution by the computer. The translation is done after examining each instruction for its correctness. If everything is alright, the compilation proceeds silently and the translated program is stored on another file with the name *ebg1.o*. This program is known as *object code*.

Linking is the process of putting together other program files and functions that are required by the program. For example, if the program is using **exp()** function, then the object code of this function should be brought from the **math library** of the system and linked to the main program. Under UNIX, the linking is automatically done (if no errors are detected) when the **cc** command is used.

If mistakes in the *syntax* and *semantics* of the language are discovered, they are listed out and the compilation process ends right there. The errors should be corrected in the source program with the help of the editor and the compilation is done again.

The compiled and linked program is called the *executable object code* and is stored automatically in another file named **a.out**.

Note that some systems use different compilation command for linking mathematical functions.

cc *filename* **-lm**

is the command under UNIPLUS SYSTEM V operating system.

Executing the Program

Execution is a simple task. The command

a.out

would load the executable object code into the computer memory and execute the instructions. During execution, the program may request for some data to be entered through the keyboard. Sometimes the program does not produced the desired results. Perhaps, something is wrong with the program *logic* or *data*. Then it would be necessary to correct the source program or the data. In case the source program is modified, the entire process of compiling, linking and executing the program should be repeated.

Creating Your Own Executable File

Note that the linker always assigns the same name **a.out**. When we compile another program, this file will be over written by the executable object code of the new program. If we want to prevent from happening, we should rename the file immediately by using the command

mv a.out *name*

We may also achieve this by specifying an option in the **cc** command as follows:

cc -o *name source-file*

This will store the executable object code in the file name and prevent the old file **a.out** from being destroyed.

Multiple Source Files

To compile and link multiple source program files, we must append all the file names to the **cc** command.

cc *filename-1.c .... filename-n.c*

These files will be separately compiled into object files called

filename-i.o

and then linked to produce an executable program file **a.out** as shown in Fig. 2.5.

It is also possible to compile each file separately and link them later. For example, the commands

cc -c *mod1.c*
cc -c *mod2.c*

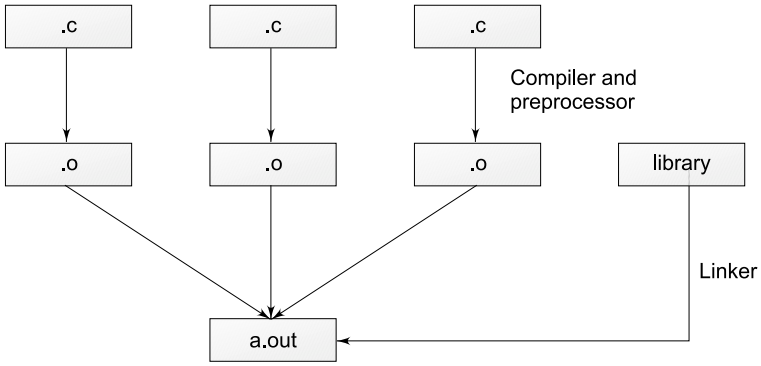


Fig. 2.5 *Compilation of multiple files*

will compile the source files *med1.c* and *mod.2c* into object files *mod1.o* and *mod2.o*.They can be linked together by the command

cc *mod1.o mod2.o*

We may also combine the source files and object files as follows:

cc *mod1.c mod2.o*

Only *mod1.c* is compiled and then linked with the object file *mod2.o*. This approach is useful when one of the multiple source files need to be changed and recompiled or an already existing object file is to be used along with the program to be compiled.

MS-DOS System

The program can be created using any word processing software in non-document mode. The file name should end with the characters ".c", like **program.c, pay.c,** etc. Then the command

MSC pay.c

under MS-DOS operating system would load the program stored in the file **pay.c** and generate the **object code**. This code is stored in another file under the name **pay.obj**. In case any language errors are found, the compilation is not completed. The program should then be corrected and compiled again.

The linking is done by the command

LINK pay.obj

which generates the executable code with the filename **pay.exe**. Now the command

pay

would execute the program and give the results.

2.5 STRUCTURE OF A C PROGRAM

The examples discussed so far illustrate that a C program can be viewed as a group of building blocks called *functions*. A function is a subroutine that may include one or more *statements* designed to perform

a *specific task*. To write a C program, we first create functions and then put them together. A C program may contain one or more sections as shown in Fig. 2.6.

The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later. The link section provides instructions to the compiler to link functions from the system library. The definition section defines all symbolic constants.

There are some variables that are used in more than one function. Such variables are called *global* variables and are declared in the *global* declaration section that is outside of all the functions. This section also declares all the user-defined functions.

Every C program must have one **main()** function section. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon(;).

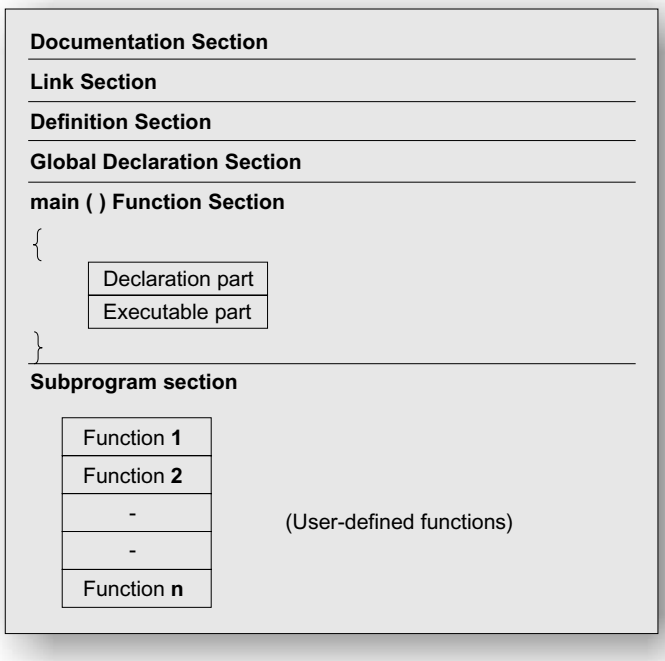


Fig. 2.6 *An overview of a C program*

The subprogram section contains all the user-defined functions that are called in the **main** function. User-defined functions are generally placed immediately after the **main** function, although they may appear in any order.

All sections, except the main function section, may be absent when they are not required.

POINTS TO REMEMBER

1. Every C program requires a **main( )** function (use of more than one **main( )** is illegal.). The place **main** is where the program execution begins.
2. The execution of a function begins at the opening brace of the function and ends at the corresponding closing brace.
3. C programs are written in lowercase letters. However, uppercase letters are used for symbolic names and output strings.
4. All the words in a program line must be separated from each other by at least one space, or a tab, or a punctuation mark.
5. Every program statement in a C program must end with a semicolon.
6. All variables must be declared for their types before they are used in the program.
7. We must make sure to include **header files** using **#include** directive when the program refers to special names and functions that it does not define.
8. Compiler directives such as **define** and **include** are special instructions to the compiler to help it compile a program. They do not end with a semicolon.
9. The sign # of compiler directives must appear in the first column of the line.
10. When braces are used to group statements, make sure that the opening brace has a corresponding closing brace.
11. **C** is a free-form language and therefore a proper form of indentation of various sections would improve legibility of the program.
12. A comment can be inserted almost anywhere a space can appear. Use of appropriate comments in proper places increases readability and understandability of the program and helps users in debugging and testing. Remember to match the symbols /* and */ appropriately.

Review Questions and Exercises

2.1 If you have access to a computer system with the C compiler, find out the details of the operating system and special features of the compiler.

2.2 Compile and run all the programs discussed in this chapter. Compare the outputs with those presented with the programs.

2.3 Remove the semicolon at the end of the **printf** statement in the program of Fig. 2.1 and try to execute it. What is the output of the program?

2.4 In the Sample Program 2, delete line-5 and execute the program. How helpful is the error message?

2.5 Modify the Sample Program 3 to print the following output:

Year	Amount
1	5550.00
2	6160.00
.	.
.	.
.	.
10	14197.11

Note that the modified output contains only 10 values (for Year = 1,...., 10).

2.6 Identify syntax errors in the following program:

```
Include <math.h>
main { }
(
        FLOAT X;
/*** ASSIGNMENT SECTION***
        X = 2.5,
        Y = exp (x);
/*** PRINTING SECTION ***/
        print(x, y)
)
```

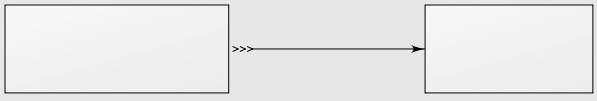
2.7 Write a program that will print your mailing address in the following form:

```
First line: Name
Second line: Door No, Street
Third line: City, Pin code
```

2.8 Write a program to output the following multiplication table:

```
5 × 1 =   5
5 × 2 =  10
5 × 3 =  15
   .
   .
   .
5 × 20 = 100
```

2.9 Write a program that will print the following figure:



2.10 State whether the following statements are TRUE or FALSE:

(a) Every line in a C program should end with a semicolon.
(b) In C language, lowercase letters are significant.
(c) Every C program ends with an END word.
(d) main( ) is where the program begins its execution.
(e) A line in a C program may have more than one statement.
(f) A printf( ) statement can generate *only one* line of output.
(g) The closing brace of the main( ) in a program is the logical end of the program.
(h) Syntax errors will be detected by the compiler.

Chapter 3

Constants, Variables, and Data Types

3.1 INTRODUCTION

A programming language is designed to help process certain kinds of *data* consisting of numbers, characters and strings and to provide useful output known as *information*. The task of processing of data is accomplished by executing a sequence of precise instructions called a *program*. These instructions are formed using certain symbols and words according to some rigid rules known as *syntax rules* (or *grammar*). Every program instruction must confirm precisely to the syntax rules of the language.

Like any other language, C has its own vocabulary and grammar. In this chapter, we will discuss the concepts of constants and variables and their types as they relate to C programming language.

3.2 CHARACTER SET

The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. However, a subset of characters is available that can be used on most personal, micro, mini and mainframe computers. The characters in C are grouped into the following categories:

1. Letters
2. Digits
3. Special characters
4. White spaces

The compiler ignores white spaces unless they are a part of string constant. White spaces may be used to separate words, but are prohibited between the characters of keywords and identifiers.

3.3 C TOKENS

In a passage of text, individual words and punctuation marks are called *tokens*. Similarly, in a C program the smallest individual units are known as C tokens. C has six types of tokens as shown in Fig. 3.1. C programs are written using these tokens and the syntax of the language.

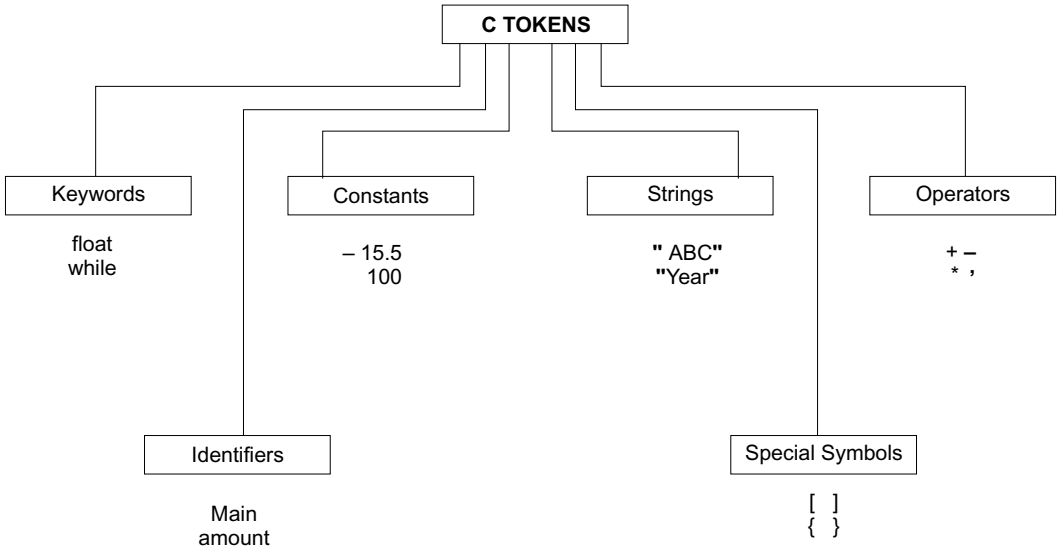


Fig. 3.1 *C tokens and examples*

3.4 KEYWORDS AND IDENTIFIERS

Every C word is classified as either a *keyword* or an *identifier*. All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. The list of all keywords in ANSI C are listed in Table 3.1. All keywords must be written in lowercase. Some compilers may use additional keywords that must be identified from the C manual.

Table 3.1 ANSI C Keywords

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 refer to the names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identifiers. It is usually used as a link between two words in long identifiers.

3.5 CONSTANTS

Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants as illustrated in Fig. 3.2.

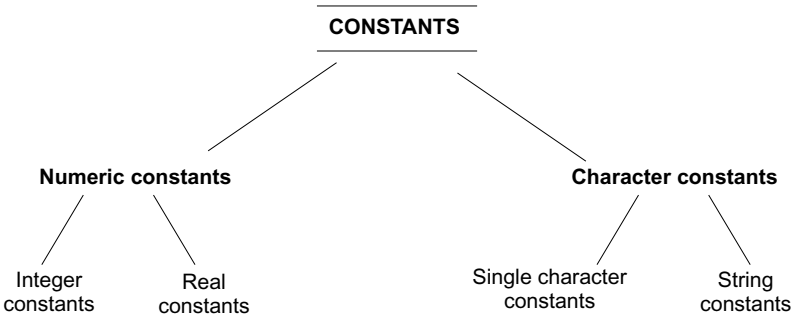


Fig. 3.2 *Basic types of C constants*

Integer Constants

An *integer* constant refers to a sequence of digits. There are three types of integers, namely, *decimal*, *octal* and *hexadecimal*.

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or + sign. Valid examples of decimal integer constants are:

123
–321
0
654321
+78

Embedded spaces, commas, and non-digit characters are not permitted between digits. For example,

15 750
20,000
$1000

are illegal numbers. Note that ANSI C supports *unary plus* which was not defined earlier.

An *octal* integer constant consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integers are:

037
0
0435
0551

A sequence of digits preceded by 0x and 0X is considered as *hexadecimal* integer. They may also include alphabets A through F or a through f. The letters A through F represent the numbers 10 through 15. Following are the examples of valid hex integers.

```
0X2
0x9F
0Xbcd
0x
```

We rarely use octal and hexadecimal numbers in programming.

The largest integer value that can be stored is machine-dependent. It is 32767 on 16-bit machines and 2,147,483,647 on 32-bit machines. It is also possible to store larger integer constants on these machines by appending *qualifiers* such as U, L and UL to the constants. For example:

56789U	or	56789u	(unsigned integer)
987612347UL	or	98761234ul	(unsigned long integer)
9876543L	or	98765431	(long integer)

The concept of unsigned and long integers are discussed in detail in Sec. 3.7.

Example 3.1

Representation of integer constants on a 16-bit computer.
The output in Fig. 3.3 shows that the integer values larger than 32767 are not properly stored on a 16-bit machine. However, when they are qualified as long integer (by appending L), the values are correctly stored.

Program

```
/ ********************************************************************** /
/*          INTEGER NUMBER ON 16-BIT MACHINE                          */
/ ********************************************************************** /

main()
{
    printf("Integer values\n\n");
    printf("%d %d %d\n", 32767,32767+1,32767+10);
    printf("\n");
    printf("Long integer values\n\n");
    printf("%ld %ld %ld\n", 32767L,32767L+1L,32767L+10L);
}
```

Output

```
Integer values

32767–32768–32759

Long integer values

32767 32768 32777
```

Fig. 3.3 *Representation of integer constants*

Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices, and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called *real* (or *floating point*) constants. Further examples of real constants are:

```
  0.0083
 –0.75
 435.36
+247.0
```

These numbers are shown in *decimal notation*, having a whole number followed by a decimal point and the fractional part. It is possible to omit digits before the decimal point, or digits after the decimal point. That is,

```
215.

 .95
 –.71
 +.5
```

are all valid real numbers.

A real number may also be expressed in *exponential* (or *scientific*) *notation*. For example, the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10^2. The general form is:

mantissa **e** exponent

The *mantissa* is either a real number expressed in *decimal notation* or an integer. The *exponent* is an integer number with an optional *plus* or *minus* sign. The letter **e** separating the mantissa and the exponent can be written in either lowercase or uppercase. Since the exponent causes the decimal point to "float", this notation is said to represent a real number in *floating point form*. Examples of legal floating point constants are:

```
 0.65e4
 12e–2
 1.5e+5
 3.18E3
– 1.2E–1
```

Embedded white space is not allowed.

Exponential notation is useful for representing numbers that are either very large or very small in magnitude. For example, 7500000000 may be written as 7.5E9 or 75E8. Similarly, –0.000000368 is equivalent to –3.68E–7.

Floating point constants are normally represented as double-precision quantities. However, the suffixes f or F may be used to force single-precision and l or L to extend double-precision further.

Some examples of valid and invalid numeric constants are given in Table 3.2.

Table 3.2 Examples of Numeric Constants

Constant	Valid ?	Remarks
698354L	Yes	Represents long integer
25,000	No	Comma is not allowed
+5.0E3	Yes	(ANSI C supports unary plus)
3.5e–5	Yes	
7.1e 4	No	No white space is permitted
–4.5e–2	Yes	
1.5E + 2.5	No	Exponent must be an integer
$255	No	$ symbol is not permitted
0X7B	Yes	Hexadecimal integer

Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of *single* quote marks. Examples of character constants are:

'5' 'X' ';' ' '

Note that the character constant '5' is not the same as the *number* 5. The last constant is a blank space.

Character constants have integer values known as ASCII values. For example, the statement

```
printf("%d", 'a');
```

would print the number 97, the ASCII value of the letter a. Similarly, the statement

```
printf("%c", 97);
```

would output the letter 'a'. ASCII values for all characters are given in Appendix II.

Since each character constant represent an integer value it is also possible to perform arithmetic operations on character constants. These are discussed in Chapter 8.

String Constants

A string constant is a sequence of character enclosed in *double* quotes. The characters may be letters, numbers, special characters and blank space. Examples are:

"Hello!"
"1987"
"WELL DONE"
"?...!"
"5+3"
"X"

Remember that a character constant (e.g., 'X') is not equivalent to the single character string constant (e.g., "X"). Further, a single character string constant does not have an equivalent integer while a character constant has an integer value. Character strings are often used in programs to build meaningful programs. Manipulation of character strings are considered in detail in Chapter 9.

Backslash Character Constants

C supports some special backslash character constants that are used in output functions. For example, the symbol '\n' stands for newline character. A list of such backslash character constants is given in Table 3.3. Note that each one of them represents one character, although they consist of two characters. These character combinations are known as *escape sequences*.

Table 3.3 Backslash Character Constants

Constant	Meaning
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\" '	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

3.6 VARIABLES

A *variable* is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution. In Chapter 2, we used several variables. For instance, we used the variable **amount** in Sample Program 3 to store the value of money at the end of each year (after adding the interest earned during that year).

A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program. Some examples of such names are:

```
Average
height
Total
Counter_1
class_strength
```

As mentioned earlier, variable names may consist of letters, digits, and the underscore(_) character, subject to the following conditions:

1. They must begin with a letter. Some systems permit underscore as the first character.
2. ANSI standard recognizes a length of 31 characters. However, the length should not be normally more than eight characters, since only the first eight characters are treated as significant by many compilers.

3. Uppercase and lowercase are significant. That is, the variable **Total** is not the same as **total** or **TOTAL**.
4. The variable name should not be a keyword.
5. White space is not allowed.

Some examples of valid variable names are:

John	Value	T_raise
Delhi	x1	ph_value
mark	sum1	distance

Invalid examples include:

123	(area)
%	25th

Further examples of variable names and their correctness are given in Table 3.4.

Table 3.4 Examples of Variable Names

Variable name	Valid?	Remark
First_tag	Valid	
char	Not valid	char is a keyword
Price$	Not valid	Dollar sign is illegal
group one	Not valid	Blank space is not permitted
average_number	Valid	First eight characters are significant
int_type	Valid	Keyword may be part of a name

If only the first eight characters are recognized by a compiler, then the two names

average_height
average_weight

mean the same thing to the computer. Such names can be rewritten as

avg_height and **avg_weight**

or

ht_average and **wt_average**

without changing their meanings.

3.7 BASIC DATA TYPES AND SIZES

C language is rich in its *data types*. Storage representations and machine instructions to handle constants differ from machine to machine. The variety of data types available allow the programmer to select the type appropriate to the needs to the application as well as the machine.

ANSI C supports four classes of data types:

1. Primary (or fundamental) data types
2. User-defined data types
3. Derived data types
4. Empty data set

The primary data types and their extensions are discussed in this section. The user-defined data types are defined in the next section while the derived data types such as arrays, functions, structures and pointers are discussed as and when they are encountered. The empty data set is discussed in the chapter on functions.

All C compilers support four fundamental data types, namely integer (**int**), character (**char**), floating point (**float**), and double-precision floating point (**double**). Many of them also offer extended data types such as **long int** and **long double**. Various data types and the terminology used to describe them are given in Fig. 3.4. The range of the basic four types are given in Table 3.5. We discuss briefly each one of them in this section.

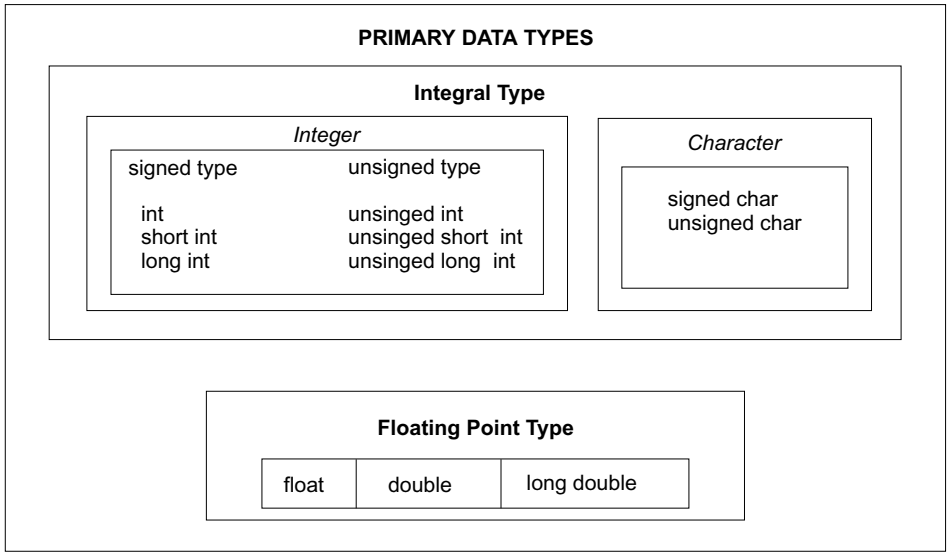


Fig. 3.4 *Primary data types in C*

Table 3.5 Size and Range of Basic Data Types

Data type	Range of values
char	–128 to 127
int	–32,768 to 32,767
float	3.4e–38 to 3.4e+38
double	1.7e–308 to 1.7e+308

Integer Types

Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. If we use a 16 bit word length, the size of the integer value is limited to the range –32768 to +32767 (that is, -2^{15} to $+2^{15}-1$). A signed integer

uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from –2, 147, 483, 648 to 2, 147, 483, 647.

In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely **short int, int**, and **long int**, in both **signed** and **unsigned** forms. For example, **short int** represents fairly small integer values and requires half the amount of storage as a regular **int** number uses. Unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always positive. Therefore, for a 16 bit machine, the range of unsigned integer numbers will be form 0 to 65,535.

We declare **long** and **unsigned** integers to increase the range of values. The use of qualifier **signed** on integers is optional because the default declaration assumes a signed number. Table 3.6 shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

Table 3.6 Size and Range of Data Types on a 16-bit Machine

Type	Size (bits)	Range
char or signed char	8	– 128 to 127
unsigned char	8	0 to 255
int or signed int	16	– 32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	– 128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	– 2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E-38 to 3.4E+38
double	64	1.7E–308 to 1.7E+308
long double	80	3.4E–4932 to 1.1E + 4932

Floating Point Types

Floating point (or real) numbers are stored in 32 bits (on all 16 bits and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword **float**. When the accuracy provided by a **float** number is not sufficient, the type **double** can be used to define the number. A **double** data type number uses 64 bits giving a precision of 14 digits. These are known as *double precision* numbers. Remember that **double** type represents the same data type that **float** represents, but with a greater precision. To extend the precision further, we may use **long double** which uses 80 bits.

Character Types

A single character can be defined as a character(**char**) type data. Characters are usually stored in 8 bits (one byte) of internal storage. The qualifier **signed** or **unsigned** may be explicitly applied to **char**. While **unsigned chars** have values between 0 to 255, **signed chars** have values from –128 to 127.

3.8 DECLARATION OF VARIABLES

After designing suitable variable names, we must declare them to the compiler. Declaration does two things:

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

The declaration of variables must be done before they are used in the program.

Primary Type Declaration

A variable can be used to store a value of any data type. That is, the name has nothing to do with its type. The syntax for declaring a variable is as follows:

data-type v1, v2, … vn;

v1, v2, … vn are the names of variables. Variables are separated by commas. A declaration statement must end with a semicolon. For example, valid declarations are:

```
int count;
int number, total;
double ratio;
```

int and **double** are the keywords to represent integer type and real type data values respectively. Table 3.7 shows various data types and their keyword equivalents.

Table 3.7 Data Types and Their Keywords

Data type	Keyword equivalent
Character	char
Unsigned character	unsigned char
Signed character	signed char
Signed integer	signed int (or int)
Signed short integer	signed short int (or short int or short)
Signed long integer	signed long int (or long int or long)
Unsigned integer	unsigned int (or unsigned)
Unsigned short integer	unsigned short int (or unsigned short)
Unsigned long integer	unsigned long int (or unsigned long)
Floating point	float
Double-precision floating point	double
Extended double-precision floating point	long double

The program segment given in Fig. 3.5 illustrates declaration of variables.**main**() is the beginning of the program. The opening brace { signals the execution of the program. Declaration of variables is usually done immediately after the opening brace of the program. The variables can also be declared outside (either before or after) the **main** function. The importance of place of declaration will be dealt in detail later while discussing functions.

```
main()/* ..... Program Name ..... */
{
/* ..... Declaration ..... */
    float                   x, y;
    int                     code;
    short int               count;
    long int                amount;
    double                  deviation;
    unsigned                n;
    char                    c;
    /* ..... Computation ..... */
            .
          . . . .
          . . . .
          . . . .
} /* ..... Program ends ..... */
```

Fig. 3.5 *Declaration of Variables*

When an adjective (qualifier) **short, long, or unsigned** is used without a basic data type specifier, C compilers treat the data type as an **int**. If we want to declare a character variables as unsigned, then we must do so using both the terms like **unsigned char**.

User-Defined Type Declaration

C supports a feature known as "type definition" that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables. It takes the general form:

typedef *type* identifier;

Where *type* refers to an existing data type and "identifier" refers to the "new" name given to the data type. The existing data type may belong to any class of type, including the user-defined ones. Remember that the new type is 'new' only in name, but not the data type. **typedef** cannot create a new type. Some examples of type definition are:

```
typedef int units;
typedef float marks;
```

Here, **units** symbolizes **int** and **marks** symbolizes **float**. They can be later used to declare variables as follows:

```
units batch1, batch2;
marks name1[50], name2[50];
```

batch1 and batch2 are declared as **int** variable and name 1[50] and name 2[50] are declared as 50 element floating point array variables. The main advantage of **typedef** is that we can create meaningful data type names for increasing the readability of the program.

Another user-defined data type is enumerated data type provided by ANSI standard. It is defined as follows:

enum identifier {*value1*, *value2*, … *valuen*};

The "identifier" is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as *enumeration constants*). After this definition, we can declare variables to be of this 'new' type as below:

enum identifier v1, v2, … vn;

The enumerated variables v1, v2, … vn can only have one of the values *value1*, *value2*, … *valuen*. The assignments of the following types are valid:

```
v1 = value3;
v5 = value1;
```

An example:

```
enum day {Monday,Tuesday, … Sunday};
enum day week_st, week_end;

week_st = Monday;
week_end = Friday;
if(week_st == Tuesday)
week_end = Saturday;
```

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant **value1** is assigned0, **value2** is assigned 1, and so on. However, the automatic assignments can be overridden by assigning values explicitly to the enumeration constants. For example:

enum day {Monday = 1, Tuesday, … Sunday};

Here, the constant Monday is assigned the value of 1. The remaining constants are assigned values that increase successively by 1.

The definition and declaration of enumerated variables can be combined in one statement. Example:

enum day {Monday, … Sunday} week_st, week_end;

Declaration of Storage Class

Variables in C can have not only data type but also *storage class* that provides information about their location and visibility. The storage class decides the portion of the program within which the variables are recognized. Consider the following example:

```
/* Example of storage classes */
int m;
main()
{
    int i;
    float balance;
    ....
```

```
    ....
    function1();
}
function1()
{
    int i;
    float sum;
    ....
    ....
}
```

The variable **m** which has been declared before the **main** is called *global* variable. It can be used in all the functions in the program. It need not be declared in other functions. A global variable is also known as an *external* variable.

The variables **i**, **balance** and **sum** are called *local* variables because they are declared inside a function. Local variables are visible and meaningful only inside the functions in which they are declared. They are not known to other functions. Note that the variable **i** has been declared in both the functions. Any change in the value of **i** in one function does not affect its value in the other.

C provides a variety of storage class specifiers that can be used to declare explicitly the scope and lifetime of variables. The concepts of scope and lifetime are important only in multifunction and multiple file programs and therefore the storage classes are considered in detail later when functions are discussed. For now, remember that there are four storage class specifiers (**auto**, **register**, **static**, and **extern**) whose meanings are given in Table 3.8.

Table 3.8 Storage Classes and Their Meaning

Storage class	Meaning
auto	Local variable known to only to the function in which it is declared. Default is *auto*.
static	Local variable which exists and retains its value even after the control is transferred to the calling function.
extern	Global variable known to all functions in the file.
register	Local variable which is stored in the register.

The storage class is another qualifier (like **long** or **unsigned**) that can be added to a variable declaration as shown below:

```
auto int count;
register char ch;
static int x;
extern long total;
```

Static and external (**extern**) variables are automatically initialized to zero. Automatic (**auto**) variables contain undefined values (known as 'garbage') unless they are initialized explicitly.

3.9 ASSIGNING VALUES TO VARIABLES

Variables are created for use in program statements such as

```
value = amount + inrate * amount;
while (year <= PERIOD)
{
    ....
    ....
    year = year + 1;
}
```

In the first statement, the numeric value stored in the variable **inrate** is multiplied by the value stored in **amount** and the product is added to **amount**. The result is stored in the variable **value**. This process is possible only if the variables **amount** and **inrate** have already been given values. The variable **value** is called the *target variable*. While all the variables are declared for their type, the variables that are used in expressions (on the right side of equal (=) sign of a computational statement) *must* be assigned values before they are encountered in the program. Similarly, the variable **year** and the symbolic constant **PERIOD** in the **while** statement must be assigned values before this statement is encountered.

Assignment Statement

Values can be assigned to variables using the assignment operator = as follows:

```
variable_name = constant;
```

We have already used such statements in Chapter 2. Further examples are:

```
initial_value    =     0;
final_value      =   100;
balance          =  75.84;
yes              =   'x';
```

C permits multiple assignments in one line. For example

```
initial_value = 0; final_value = 100;
```

are valid statements.

An assignment statement implies that the value of the variable on the left of the 'equal sign' is set equal to the value of the quantity (or the expression) on the right. The statement

```
year = year + 1;
```

means that the 'new value' of **year** is equal to the 'old value' of **year** plus 1.

During assignment operation, C converts the type of value on the right-hand side to the type on the left. This may involve truncation when real value is converted to an integer.

It is also possible to assign a value to a variable at the time the variable is declared. This takes the following form:

```
data-type variable_name = constant;
```

Some examples are:

```
int final_value = 100;
char yes = 'x';
double balance = 75.84;
```

The process of giving initial values to variables is called *initialization*. C permits the initialization of more than one variables in one statement using multiple assignment operators. For example the statements

```
p = q = s = 0;
x = y = z = MAX;
```

are valid. The first statement initializes the variables **p**, **q**, and **s** to zero while the second initializes **x**, **y**, and **z** with **MAX**. Note that **MAX** is a symbolic constant defined at the beginning.

Remember that external and static variables are initialized to zero by *default*. Automatic variables that are not initialized explicitly will contain garbage.

Example 3.2

Program in Fig. 3.6 shows typical declarations, assignments and values stored in various types of variables.

The variables **x** and **p** have been declared as floating-point variables. Note that the way the value of 1.234567890000 that we assigned to **x** is displayed under different output formats. The value of **x** is displayed as 1.234567880630 under %. 12 lf format, while the actual value assigned is 1.234567890000. This is because the variable **x** has been declared as a **float** that can store values only upto six decimal places.

Program

```
/ ************************************************************************** /
/*                EXAMPLES OF ASSIGNMENTS                                   */

/ ************************************************************************** /

main()
{
/* ................ DECLARATIONS ..................................... */
    float       x, p ;
    double      y, q ;
    unsigned    k ;

/* ......... DECLARATION AND ASSIGNMENTS ............... */

    int         m = 54321 ;
    long int    n = 1234567890 ;

/* .......... ASSIGNMENTS ..................................... */

    x = 1.234567890000;
    y = 9.87654321;
    k = 54321 ;
    p = q = 1.0 ;

/* ........ PRINTING ..................................... */

    printf("m = %d\n", m) ;
    printf("n = %ld\n", n) ;
    print("x = %.12lf\n", x) ;
    printf("x = %f\n", x) ;
    printf("y = %. 12lf\n", y) ;
```

```
    printf("y = %lf\n", y) ;
    printf("k = %u p = %f q = %. 12lf\n", k, p, q) ;
}

Output

m = –11215
n = 1234567890
x = 1.234567880630
x = 1.234568
y = 9.87653210000
y = 9.876543
k = 54321  p = 1.000000  q = 1.000000000000
```

Fig. 3.6 *Examples of assignments*

The variable **m** that has been declared as **int** is not able to store the value 54321 correctly. Instead, it contains some garbage. Since this program was run on a 16-bit machine, the maximum value that an **int** variable can store is only 32767. However, the variable **k** (declared as **unsigned**) has stored the value 54321 correctly. Similarly, the **long int** variable **n** has stored the value 1234567890 correctly.

The value 9.87654321 assigned to **y** declared as double has been stored correctly but the value is printed as 9.876543 under %lf format. Note that unless specified otherwise, the **printf** function will always display a **float** or **double** value to six decimal places. We will discuss later the output formats for displaying numbers.

Declaring a Variable as Constant

We may like the value of certain variables to remain constant during the execution of a program. We can achieve this by declaring the variable with the qualifier **const** at the time of initialization. Example:

```
const int class_size = 40;
```

const is a new data type qualifier defined by ANSI standard. This tells the compiler that the value of the **int** variable **class_size** must not be modified by the program. However, it can be used on the right_hand side of an assignment statement like any other variable.

Declaring a Variable as Volatile

ANSI standard defines another qualifier **volatile** that could be used to tell explicitly the compiler that a variable's value may be changed at any time by some external sources (from outside the program). For example:

```
volatile int date;
```

The value of **date** may be altered by some external factors even if it does not appear on the left-hand side of an assignment statement. When we declare a variable as **volatile**, the compiler will examine the value of the variable each time it is encountered to see whether any external alteration has changed the value.

Remember that the value of a variable declared as **volatile** can be modified by its own program as well. If we wish that the value must not be modified by the program while it may be altered by some other process, then we may declare the variable as both **const** and **volatile** as shown below:

```
volatile const int location = 100;
```

Overflow and Underflow of Data

Problem of data overflow occurs when the value of a variable is either too big or too small for the data type to hold. The largest value that a variable can hold also depends on the machine. Since floating point values are rounded off to the number of significant digits allowed (or specified), an overflow normally results in the largest possible real value, whereas an underflow results in zero.

Integers are always exact within the limits of the range of the integral data types used. However, an overflow which is a serious problem may occur if the data type does not match the value of the constant. C does not provide any warning or indication of integer overflow. It simply gives incorrect results. (Overflow normally produces a negative number.) We should therefore exercise a greater care to define correct data types for handling the input/output values.

Reading Data from Keyboard

Another way of giving values to variables is to input data through keyboard using the **scanf** function. It is a general input function available in C and is very similar in concept to the **printf** function. It works much like an INPUT statement in BASIC. The general format of **scanf** is as follows:

```
scanf("control string", &variable1, &variable2,....);
```

The control string contains the format of data being received. The ampersand symbol **&** before each variable name is an operator that specifies the variable name's *address*. We must always use this operator, otherwise unexpected results may occur. Let us look at an example:

```
scanf("%d", &number);
```

When this statement is encountered by the computer, the execution stops and waits for the value of the variable **number** to be typed in. Since the control string "%d" specifies that an integer value is to be read from the terminal, we have to type in the value in integer form. Once the number is typed in the 'Return' Key is pressed, the computer then proceeds to the next statement. Thus, the use of **scanf** provides an interactive feature and makes the program 'user friendly'.

Example 3.3
The program in Fig. 3.7 illustrates the use of scanf function.

The first executable statement in the program is a **printf**, requesting the user to enter an integer number. This is known as "prompt message" and appears on the screen like

```
Enter an integer number
```

As soon as the user types in an integer number, the computer proceeds to compare the value with 100. If the value typed in is less than 100, then a message

```
Your number is smaller than 100
```

is printed on the screen. Otherwise, the message

Your number contains more than two digits

is printed. Outputs of the program, run for two different inputs are also shown in Fig. 3.7.

Program

```
/ ************************************************************************************/
/*            INTERACTIVE COMPUTING USING scanf FUNCTION                            */
/ ************************************************************************************/

main()
{
    int number;

    printf("Enter an integer number\n");
    scanf ("%d", &number);

    if (number < 100)
        printf("Your number is smaller than 100\n\n");
    else
        printf("Your number contains more than two digits\n");
}
```

Output

```
Enter an integer number
54
Your number is smaller than 100

Enter an integer number
108
Your number contains more than two digits
```

Fig. 3.7 *Use of **scanf** function*

Some compilers permit the use of the 'prompt message' as a part of the control string in scanf, like

scanf("Enter a number %d", &number);

We discuss more about **scanf** in Chapter 5.

In Fig. 3.7 we have used a decision statement **if**…**else** to decide whether the number is less than 100. Decision statements are discussed in depth in Chapter 6.

Example 3.4

Sample Program 3 discussed in Chapter 2 can be converted into a more flexible interactive program using **scanf** as shown in Fig. 3.8.

In this case, computer requests the user to input the values of the amount to be invested, interest rate rate and period of investment by printing a prompt message

Input amount, interest rate, and period

and then waits for input values. As soon as we finish entering the three values corresponding to the three variables **amount**, **inrate**, and **period**, the computer begins to calculate the amount at the end of each year, upto 'period' and produces output as shown in Fig. 3.8.

Program

```
/ ********************************************************************** /
/*            INTERACTIVE INVESTMENT PROGRAM                          */
/ ********************************************************************** /

main()
{

    int year, period ;
    float amount, inrate, value ;

    printf("Input amount, interest rate, and period\n\n") ;
    scanf("%f %f %d", &amount, &inrate, &period) ;
    printf("\n") ;
    year = 1 ;

    while (year <= period)
    {
        value = amount + inrate * amount ;
        printf("%2d Rs %8.2f\n", year, value) ;
        amount = value ;
        year = year + 1 ;
    }
}
```

Output

```
Input amount, interest rate, and period

10000 0.14 5

1   Rs 11400.00
2   Rs 12996.00
3   Rs 14815.44
4   Rs 16889.60
5   Rs 19254.15

Input amount, interest rate, and period
20000 0.12 7
1   Rs 22400.00
2   Rs 25088.00
3   Rs 28098.56
4   Rs 31470.39
5   Rs 35246.84
6   Rs 39476.46
7   Rs 44213.63
```

Fig. 3.8 *Interactive investment program*

Note that the **scanf** function contains three variables. In such cases, care should be exercised to see that the values entered match the *order* and *type* of the variables in the list. Any mismatch might lead to unexpected results. The compiler may not detect such errors.

3.10 DEFINING SYMBOLIC CONSTANTS

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. One example of such a constant is 3.142, representing the value of the mathematical constant "**pi**". Another example is the total number of students whose marksheets are analysed by a 'test analysis program'. The number of students, say 50, may be used for calculating the class total, class average, standard deviation, etc. We face two problems in the subsequent use of such programs.

1. Problem in modification of the program.
2. Problem in understanding the program.

Modifiability

We may like to change the value of "pi" from 3.142 to 3.14159 to improve the accuracy of calculations or the number 50 to 100 to process the test results of another class. In both the cases, we will have to search throughout the program and explicitly change the value of the constant wherever it has been used. If any value is left unchanged, the program may produce disastrous outputs.

Understandability

When a numeric value appears in a program, its use is not always clear, especially when the same value means different things in different places. For example, the number 50 may mean the number of students at one place and the 'pass marks' at another place of the same program. We may forget what a certain number meant, when we read the program some days later.

Assignment of such constants to a *symbolic name* frees us from these problems. For example, we may use the name **STRENGTH** to define the number of students and **PASS_MARK** to define the pass marks required in a subject. Constant values are assigned to these names at the beginning of the program. Subsequent use of the names **STRENGTH** and **PASS_MARK** in the program has the effect of causing their defined values to be automatically substituted at the appropriate points. A constant is defined as follows:

```
#define  symbolic-name  value of constant
```

Valid examples of constant definitions are:

```
#define STRENGTH      100
#define PASS_MARK      50
#define MAX           200
#define PI            3.14159
```

Symbolic names are sometimes called *constant identifiers*. Since the symbolic names are constants (not variables), they do not appear in declarations. The following rules apply to a **#define** statement which define a symbolic constant.

1. Symbolic names have the same form as variable names. (Symbolic names are written in CAPITALS to visually distinguish them from the normal variable names which are written in lowercase letters. This is only a convention, not a rule.)
2. No blank space between the pound sign '#' and the word **define** is permitted.

3. '#' must be the first character in the line.
4. A blank space is required between **#define** and *symbolic name* and between the *symbolic name* and the *constant*.
5. **#define** statements must not end with a semicolon.
6. After definition, *the symbolic name* should not be assigned any other value within the program by using an assignment statement. For example, STRENGTH = 200; is illegal.
7. Symbolic names are NOT declared for data types. Its data type depends on the type of constant.
8. **#define** statements may appear *anywhere* in the program but before it is referenced in the program (the usual practice is to place them in the beginning of the program).

#define statement is a *preprocessor* compiler directive and is much more powerful than what has been mentioned here. More advanced types of definitions will be discussed later. Table 3.11 illustrates some invalid statements of **#define**.

Table 3.11 Examples of Invalid #define Statements

Statement	Validity	Remark
#define X = 2.5	Invalid	'=' sign is not allowed
#define MAX 10	Invalid	No white space between # and define
#define N 25;	Invalid	No semicolon at the end
#define N 5, M 10	Invalid	A statement can define only one name.
#Define ARRAY 11	Invalid	define should be in a lowercase letters
#define PRICE$ 100	Invalid	$ symbol is not permitted in name

CASE STUDIES

1. Calculation of Average of Numbers

A program to calculate the average of a set of N numbers is given in Fig. 3.9.

Program

```
/ ********************************************************************************* /
/*                          AVERAGE OF n VALUES                                    */
/ ********************************************************************************* /

#define    N   10                              /* SYMBOLIC CONSTANT */

main()
{
    int      count;                              /* DECLARATION OF */
    float    sum, average, number ;                    /* VARIABLES */

    sum = 0;                                     /* INITIALIZATION */
    count = 0;                                   /* OF VARIABLES */

    while (count < N)
    {
```

```
        scanf("%f", &number) ;
        sum = sum + number ;
        count = count + 1 ;
    }

    average = sum/N ;
    printf("N = %d Sum = %f", N, sum);
    Printf("Average = %f", average);
}
```

Output

```
1
2.3
4.67
1.42
7
3.67
4.08
2.2
4.25
8.21
N = 10 Sum = 38.799999 Average = 3.880000
```

Fig. 3.9 *Average of N numbers*

The variable **number** is declared as **float** and therefore it can take both integer and real numbers. Since the symbolic constant **N** is assigned the value of 10 using the **#define** statement, the program accepts ten values and calculates their sum using the **while** loop. The variable **count** counts the number of values and as soon as it becomes 11, the **while** loop is exited and then the average is calculated.

Notice that the actual value of sum is 38.8 but the value displayed is 38.799999. In fact, the actual value that is displayed is quite dependent on the computer system. Such an inaccuracy is due to the way the floating point numbers are internally represented inside the computer.

2. Temperature Conversion Problem

The program presented in Fig. 3.10 converts the given temperature in fahrenheit to celsius using the following conversion formula:

$$C = \frac{F - 32}{1.8}$$

Program

```
/ ************************************************************************** /
/*              FAHRENHEIT—CELSIUS CONVERSION TABLE          */
/ ************************************************************************** /

#define   F_LOW   0                    /* ************************* */
#define   F_MAX   250                  /* SYMBOLIC CONSTANTS */
#define   STEP    25                   /* ************************* */
```

```
main()
{
    typedef float REAL;                          /* TYPE DEFINITION */
    REAL fahrenheit, celsius;                    /* DECLARATION */

    fahrenheit = F_Low;                          /* INITIALIZATION */
    printf{"Fahrenheit     Celsius\n\n") ;
    while (fahrenheit <= F_MAX)
    {
        celsius = (fahrenheit – 32.0) / 1.8 ;
        printf("%5.1f       %7.2f\n", fahrenheit, celsius);
        fahrenheit = fahrenheit + STEP ;
    }
}
```

Output

```
Fahrenheit          Celsius

     0.0             –17.78
    25.0              –3.89
    50.0              10.00
    75.0              23.89
   100.0              37.78
   125.0              51.67
   150.0              65.56
   175.0              79.44
   200.0              93.33
   225.0             107.22
   250.0             121.11
```

Fig. 3.10 *Temperature conversion*

The program prints a conversion table for reading temperature in celsius, given the fahrenheit values. The minimum and maximum values and step size are defined as symbolic constants. These values can be changed by redefining the **#define** statements. A user-defined data type name **REAL** is used to declare the variables **fahrenheit** and **celsius**.

The formation specifications %5.1f and %7.2 in the second **printf** statement produces two column output as shown.

Review Questions and Exercises

3.1 What are trigraph characters? How are they useful?
3.2 Describe the four basic data types. How could we extend the range of values they represent?
3.3 What is an unsigned integer constant? What is the significance of declaring a constant unsigned?
3.4 Describe the characteristics and purpose of escape sequence characters.
3.5 What is a variable and what is meant by the "value"of a variable?
3.6 How do variables and symbolic names differ?
3.7 State the differences between the declaration of a variable and the definition of a symbolic name.

3.8 What is initialization? Why is it important?
3.9 What are the qualifiers that an int can have at a time?
3.10 A programmer would like to use the word DPR to declare all the double-precision floating point values in his program. How could he achieve this?
3.11 What are enumeration variables? How are they declared? What is the advantage of using them in a program?
3.12 Describe the purpose of the qualifiers **const** and **volatile**.
3.13 When dealing with very small or very large numbers, what steps would you take to improve the accuracy of the calculations?
3.14 Which of the following are invalid constants and why?

0.0001	5x1.5	99999
+100	75.45 E–2	“15.75”
–45.6	–1.79 e + 4	0.00001234

3.15 Which of the following are invalid variable names and why?

Minimum	First.name	n1+n2	&name
doubles	3rd_row	n$	Row1
float	Sum Total	Row Total	Column-total

3.16 Find errors, if any, in the following declaration statements.

```
Int x;
float letter, DIGIT;
double = p,q
exponent alpha, beta;
m, n, z: INTEGER
short char c;
long int m; count;
long float temp;
```

3.17 What would be the value of x after execution of the following statements?

```
int x, y = 10;
char z = ‘a’;

x = y + z;
```

3.18 Identify syntax errors in the following program. After corrections, what output would you except when you execute it?

```
#define PI  3.14159
main()
{
        int R,C;                /* R-Radius of circle */
        float perimeter;        /* Circumference of circle */
        float area;             /* area of circle */
        C = PI
        R = 5;
        Perimeter = 2.0 * C * R;
        Area = C*R*R;
        printf(“%f”, “%d”,&perimeter,&area)}
```

3.19 Write a program to determine and print the sum of the following harmonic series for a given value of n:

$$1 + 1/2 + 1/3 + \ldots + 1/n$$

The value of n should be given interactively through the terminal.

3.20 Write a program to read the price of an item in decimal form (like 15.95) and print the output in paise (like 1595 paise).

Chapter 4

Operators and Expressions

4.1 INTRODUCTION

C supports a rich set of operators. We have already used several of them, such as = , +, –, *, & and <. An *operator* is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical of logical *expressions*.

C operators can be classified into a number of categories. They include:

1. Arithmetic operators.
2. Relational operators.
3. Logical operators.
4. Assignment operators.
5. Increment and decrement operators.
6. Conditional operators.
7. Bitwise operators.
8. Special operators.

4.2 ARITHMETIC OPERATORS

C provides all the basic arithmetic operators. They are listed in Table 4.1. The operators +, –, *, and / all work the same way as they do in other languages. These can operate on any built-in data type allowed in C. The unary minus operator, in effect, multiplies its single operand by –1. Therefore, a number preceded by a minus sign changes its sign.

Table 4.1 Arithmetic Operators

Operator	Meaning
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Integer division truncates any fractional part. The modulo division produces the remainder of an integer division. Examples of arithmetic operators are:

```
a – b          a + b
a * b          a / b
a % b        – a * b
```

Here **a** and **b** are variables and are known as operands. The modulo division operator % cannot be used on floating point data.

Note that C does not have an operator for *exponentiation*. Older versions of C does not support unary plus but ANSI C supports it.

Integer Arithmetic

When both the operands in a single arithmetic expression such as a + b are integers, the expression is called an *integer expression*, and the operation is called *integer arithmetic*. Integer arithmetic always yields an integer value. The largest integer value depends on the machine, as pointed out earlier. In the above examples, if **a** and **b** are integers, then for **a** = 14 and **b** = 4 we have the following results:

```
a – b = 10
a + b = 18
a * b = 56
a / b = 3 (decimal part truncated)
a % b = 2 (remainder of division)
```

During integer division, if both the operands are of the same sign, the result is truncated towards zero. If one of them is negative, the direction of truncation is implementation dependent. That is,

```
6/7 = 0 and –6/–7 = 0
```

but –6/7 may be zero or – 1. (Machine dependent)

Similarly, during modulo division, the sign of the result is always the sign of the first operand (the dividend). This is

```
–14  %   3 =  –2
–14  %  –3 =  –2
 14  %  –3 =   2
```

Example 4.1

The program in Fig. 4.1 shows the use of integer arithmetic to convert a given number of days into months and days.

The variables **months** and **days** are declared as integers. Therefore, the statement

```
months = days/30;
```

truncates the decimal part and assigns the integer part to **months**. Similarly, the statement

```
days = days%30;
```

assigns the remainder part of the division to **days**. Thus the given number of days is converted into an equivalent number of months and days and the result is printed as shown in the output.

Program

```
/ ************************************************************************** /
/*          PROGRAM TO CONVERT DAYS TO MONTHS AND DAYS                    */
/ ************************************************************************** /
main()
{
      int     months, days;

      printf("Enter days\n");
      scanf("%d", &days);

      months  =  days / 30;
      days    =  days % 30;

      printf("Months  =  %d  Days  =  %d", months, days);
}
```

Output

```
Enter days
265
Months  =  8          Days  =  25
Enter days
364
Months  =  12         Days  =  4
Enter days
45
Months  =  1          Days  =  15
```

Fig. 4.1 *Illustration of integer arithmetic*

Real Arithmetic

An arithmetic operation involving only real operands is called *real arithmetic*. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result. If **x**, **y**, and **z** are **floats**, then we will have:

```
x =    6.0/7.0   =   0.857143
y =    1.0/3.0   =   0.333333
z =  – 2.0/3.0   =  –0.666667
```

The operator % cannot be used with real operands.

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a *mixed-mode arithmetic* expression. If either operand is of the real type, then only the real operation is performed and the result is always a real number. Thus

```
15/10.0 = 1.5
```

where as

```
15/10 = 1
```

More about mixed operations will be discussed later when we deal with the evaluation of expressions.

4.3 RELATIONAL OPERATORS

We often compare two quantities, and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of *relational operators*. We have already used the symbol '<', meaning 'less than'. An expression such as

a < b or 1 < 20

containing a relational operator is termed as a *relational expression*. The value of a relational expression is either *one* or *zero*. It is one if the specified relation is *true* and zero if the relation is *false*. For example

10 < 20 is *true*

but

20 < 10 is *false*

C supports six relational operators in all. These operators and their meanings are shown in Table 4.2.

Table 4.2 Relational Operators

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

A simple relational expression contains only one relational operator and takes the following form:

ae–1 relational operator *ae–2*

ae–1 and *ae–2* are arithmetic expressions, which may be simple constants, variables or combination of them. Given below are some examples of simple relational expressions and their values:

4.5	<=	10	TRUE	
4.5	<	–10	FALSE	
–35	>=	0	FALSE	
10	<	7+5	TRUE	
a+b	==	c+d	TRUE	only if the sum of values of *a* and *b* is equal to the sum of values of *c* and *d*.

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators.

Relational expressions are used in *decision statements* such as, **if** and **while** to decide the course of action of a running program. We have already used the **while** statement in Chapter 2. Decision statements are discussed in detail in Chapters 6 and 7.

4.4 LOGICAL OPERATORS

In addition to the relational operators, C has the following three *logical operators*.

&& meaning logical **AND**
|| meaning logical **OR**
! meaning logical **NOT**

The logical operators **&&** and || are used when we want to test more than one condition and make decisions. An example is:

```
a > b && x == 10
```

An expression of this kind which combines two or more relational expressions is termed as a *logical expression* or a *compound relational expression*. Like the simple relational expressions, a logical expression also yields a value of *one or zero*, according to the *truth table* shown in Table 4.3. The logical expression given above is true only if **a** > **b** is *true* and **x** == 10 is *true*. If either (or both) of them are false, the expression is *false*.

Table 4.3 Truth Table

op–1	op–2	Value of the expression	
		op–1 && op–2	op–1 || op–2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Some examples of the usage of logical expressions are:

1. **if** (age > 55 && salary < 1000)
2. **if** (number < 0 || number > 100)

We shall see more of them when we discuss decision statements.

4.5 ASSIGNMENT OPERATORS

Assignment operators are used to assign the result of an expression to a variable. We have seen the usual assignment operator, '='. In addition, C has a set of 'shorthand' assignment operators of the form

v **op** = *exp*;

Where *v* is a variable, *exp* is an expression and *op* is a C binary arithmetic operator. The operator **op** = is known as the *shorthand* assignment operator.
The assignment statement

v op = exp;

is equivalent to

```
v = v op (exp);
```

with **v** evaluated only once. Consider an example

```
x += y+1;
```

This is same as the statement

```
x = x + (y+1);
```

The shorthand operator + = means 'add y + 1 to x' or 'increment x by y + 1'. For y = 2, the above statement becomes

```
x += 3;
```

and when this statement is executed, 3 is added to x. If the old value of x is, say 5, then the new value of x is 8. Some of the commonly used shorthand assignment operators are illustrated in Table 4.4.

Table 4.4 Shorthand Assignment Operators

Statement with simple assignment operator	Statement with shorthand operator
a = a + 1	a += 1
a = a – 1	a –= 1
a = a * (n + 1)	a *= n + 1
a = a / (n + 1)	a /= n + 1
a = a % b	a %= b

The use of shorthand assignment operators has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

These advantages may be appreciated if we consider a slightly more involved statement, like

```
value(5*j–2) = value(5*j–2) + delta;
```

With the help of the + = operator, this can be written as follows:

```
value(5*j–2) += delta;
```

It is easier to read and understand, and is more efficient because the expression 5*j–2 is evaluated only once.

Example 4.2

Program of Fig. 4.2 prints a sequence of squares of numbers. Note the use of the shorthand operator *=.

The program attempts to print a sequence of squares of numbers starting from 2. The statement

```
a *= a;
```

which is identical to

```
a = a*a;
```

replaces the current value of **a** by its square. When the value of **a** becomes equal or greater than N(= 100) the **while** is terminated. Note that the output contains only three values 2, 4 and 16.

```
Program
/ ********************************************************************************** /
/*       PROGRAM TO SHOW USE OF SHORTHAND OPERATORS        */
/ ********************************************************************************** /

#define              N          100
#define              A            2

main()
{
      int        a;

      a    =    A;
      while( a < N )
      {
            printf("%d\n", a);
            a *= a;
      }
}

Output

2
4
16
```

Fig. 4.2 *Use of shorthand operator *=*

4.6 INCREMENT AND DECREMENT OPERATORS

C has two very useful operators not generally found in other languages. These are the *increment* and *decrement* operators:

++ and −−

The operator++ add 1 to the operand while −− subtracts 1. Both are unary operators and take the following form:

++ m; or m ++;
−−m; or m−−;
++ m; is equivalent to m = m + 1; (or m += 1;)
−−m; is equivalent to m = m−1; (or m −= 1;)

We use the increment and decrement statements in **for** and **while** loops extensively.

While ++ m and m ++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following:

```
m = 5;
y = ++m;
```

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statements as

```
m = 5;
y  = m++;
```

then, the value of y would be 5 and m would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left then increments the operand.

Similar is the case, when we use ++ (or – –) in subscripted variables. That is, the statement

```
a[i++] = 10;
```

is equivalent to

```
a[i] = 10;
i = i + 1;
```

The increment and decrement operators can be used in complex statements. Example:

```
m = n++ –j+10;
```

Old value of **n** is used in evaluating the expression. **n** is incremented after the evaluation. Some compilers require a space on either side of n++ or ++n.

4.7 CONDITIONAL OPERATOR

A ternary operator pair "? :" is available in C to construct conditional expressions of the form

exp1 ? *exp2* : *exp3*;

where *exp1*, *exp2*, and *exp3* are expressions.

The operator ? : works as follows: *exp1* is evaluated first. If it is nonzero (true), then the expression *exp2* is evaluated and becomes the value of the expression. If *exp1* is false, *exp3* is evaluated and its value becomes the value of the expression. Note that only one of the expressions (either *exp2* or *exp3*) is evaluated. For example, consider the following statements.

```
a = 10;
b = 15;
x = (a > b)? a : b;
```

In this example, x will be assigned the value of b. This can be achieved using the if..else statements as follows:

```
if (a > b)
    x = a;
else
    x = b;
```

4.8 BITWISE OPERATORS

C has a distinction of supporting special operators known as *bitwise operators* for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to **float** or **double**. Table 3.5 lists the bitwise operators and their meanings. They are discussed in detail in Appendix I.

Table 4.5 Bitwise Operators

Operator	Meaning
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right
~	One's complement

4.9 SPECIAL OPERATORS

C supports some special operators of interest such as comma operator, **sizeof** operator, pointer operators (& and *) and member selection operators (. and –>). The comma and **sizeof** operators are discussed in this section while the pointer operators are discussed in Chapter 12. Member selection operators which are used to select members of a structure are discussed in Chapters 11 and 12. ANSI committee has introduced two preprocessor operators known as "string-izing" and "token-pasting" operators (# and ##). They will be discussed in Chapter 14.

The Comma Operator

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated *left to right* and the value of *right-most* expression is the value of the combined expression. For example, the statement

```
value = (x = 10, y = 5, x+y);
```

first assigns the value 10 to **x**, then assigns 5 to **y**, and finally assigns 15 (i.e. 10 + 5) to **value**. Since comma operator has the lowest precedence of all operators, the parentheses are necessary. Some applications of comma operator are:
In **for** loops:

```
for (n = 1, m = 10; n <= m; n++, m++)
```

In **while** loops:

```
while(c = getchar(), c != '10')
```

Exchanging values:

```
t = x, x = y, y = t;
```

The sizeof Operator

The **sizeof** is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples:

```
m = sizeof(sum);
n = sizeof(long int);
k = sizeof(235L);
```

The **sizeof** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

Example 4.3

In Fig. 4.3, the program employs different kinds of operators. The results of their evaluation are also shown for comparison.

Notice the way the increment operator ++ works when used in an expression. In the statement

```
c = ++a – b;
```

new value of **a** (= 16) is used thus giving the value 6 to c. That is, a is incremented by 1 before it is used in the expression. However, in the statement

```
d = b++ + a;
```

the old value of **b** (= 10) is used in the expression. Here, **b** is incremented by 1 after it is used in the expression.

We can print the character % by placing it immediately after another % character in the control string. This illustrated by the statement

```
printf("a%%b = %d\n", a%b);
```

The program also illustrates that the expression

```
c > d ? 1 : 0
```

assumes the value 0 when c is less than d and 1 when c is greater than d.

Program

```
/ ********************************************************************** /
/*                ILLUSTRATION OF OPERATORS                           */
/ ********************************************************************** /
main ()
{
    int a, b, c, d;
    a = 15;
    b = 10;
    c = ++a – b;

    printf("a = %d b = %d c = %d\n", a, b, c);
    d = b + + + a;

    printf("a = %d b = %d d = %d\n",a, b, d);
    printf("a/b = %d\n", a/b);
    printf("a%%b = %d\n", a%b);
    printf("a *= b = %d\n", a*=b);
    printf("%d\n", (c > d) ? 1 : 0);
    printf("%d\n", (c < d) ? 1 : 0);
}
```

Output

```
a = 16 b = 10 c = 6
a = 16 b = 11 d = 26
a/b = 1
a%b = 5
a *= b = 176
0
1
```

Fig. 4.3 *Further illustration of arithmetic operators*

4.10 ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. We have used a number of simple expressions in the examples discussed so far. C can handle any complex mathematical expressions. Some of the examples of C expressions are shown in Table 3.6. Remember that C does not have an operator for exponentiation.

4.11 EVALUATION OF EXPRESSIONS

Expressions are evaluated using an assignment statement of the form

variable = *expression*;

Table 4.6 Expressions

Algebraic expression	C expression
a x b – c	a * b – c
(m + n) (x + y)	(m + n) * (x + y)
$\frac{ab}{c}$	a * b/c
$3x^2 + 2x + 1$	3 * x * x + 2 * x + 1
$\frac{x}{y} + c$	x/y + c

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted. Examples of evaluation statements are

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

The blank space around an operator is optional and adds only to improve readability. When these statements are used in a program, the variables **a**, **b**, **c**, and **d** must be defined before they are used in the expressions.

Example 4.4
The program in Fig. 4.4 illustrates the use of variables in expressions and their evaluation.

Output of the program also illustrates the effect of presence of parentheses in expressions. This is discussed in the next section.

Rules for Evaluation of Expression

- First, parenthesized sub expression from left to right are evaluated.
- If parentheses are 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 associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expressions within parentheses assume highest priority.

4.12 PRECEDENCE OF ARITHMETIC OPERATORS

An arithmetic expression without parentheses will be evaluated from *left to right* using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C

High priority	* / %
Low priority	+ –

The basic evaluation procedure includes two left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered. During the second pass, the low priority operators (if any) are applied as they are encountered. Consider the following evaluation statement that has been used in the program of Fig. 4.4.

Program

```
/ ************************************************************************** /
/*                    EVALUATION OF EXPRESSIONS                           */
/ ************************************************************************** /

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 = %f\n", x);
    printf("y = %f\n", y);
    printf("z = %f\n", z);
}
```

Output

```
x = 10.000000
y = 7.000000
z = 4.000000
```

Fig. 4.4 *Illustrations of evaluation of expressions*

```
x = a – b/3 + c*2–1
```

When a = 9, b = 12, and c = 3, the statement becomes

```
x = 9 – 12/3 + 3*2–1
```

and is evaluated as follows

First pass

```
Step1: x  =  9 – 4 + 3*2 – 1
Step2: x  =  9 – 4 + 6 – 1
```

Second pass

```
Step3: x  =  5 + 6 – 1
Step4: x  =  11 – 1
Step5: x  =  10
```

These steps are illustrated in Fig. 4.5. The numbers inside parentheses refer to step numbers.

However, the order of evaluation can be changed by introducing parentheses into an expression. Consider the same expression with parentheses as shown below:

```
9 – 12/(3 + 3)*(2 – 1)
```

Whenever parentheses are used, the expressions within parentheses assume highest priority. If two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the new steps.

First pass

```
Step1:  9 – 12/6 * (2 – 1)
Step2:  9 – 12/6 * 1
```

Second pass

```
Step3:  9 – 2 * 1
Step4:  9 – 2
```

Third pass

```
Step5: 7
```

This time, the procedure consists of three left-to-right passes. However, the number of evaluation steps remain the same as 5 (i.e. equal to the number of arithmetic operators).

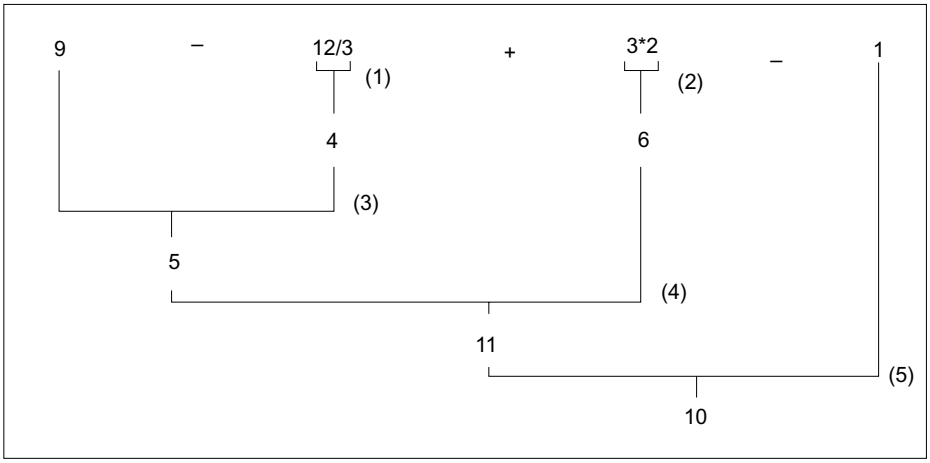


Fig. 4.5 *Illustration of hierarchy of operations*

Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses. Just make sure that every opening parenthesis has a matching closing one. For example

$$9 - (12/(3+3) * 2) - 1 = 4$$

whereas

$$9 - ((12/3) + 3 * 2) - 1 = -2$$

While parentheses allow us to change the order of priority, we may also use them to improve understandability of the program. When in doubt, we can always add an extra pair just to make sure that the priority assumed is the one we require.

4.13 SOME COMPUTATIONAL PROBLEMS

When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors. We know that the computer gives approximate values for real numbers and the errors due to such approximations may lead to serious problems. For example, consider the following statements:

```
a = 1.0/3.0;
b = a * 3.0;
```

We know that (1.0/3.0) 3.0 is equal to 1. But there is no guarantee that the value of **b** computed in a program will equal 1.

Another problem is division by zero. On most computers, any attempt to divide a number by zero will result in abnormal termination of the program. In some cases such a division may produce meaningless results. Care should be taken to test the denominator that is likely to assume zero value and avoid any division by zero.

The third problem is to avoid overflow or underflow errors. It is our responsibility to guarantee that operands are of the correct type and range, and the result may not produce any overflow or underflow.

Example 4.5

Output of the program in Fig. 4.6 shows round-off errors that can occur in computation of floating point numbers.

Program

```
/ ********************************************************************************** /
/*              PROGRAM SHOWING ROUND-OFF ERRORS                                    */
/*                          Sum of n terms of 1/n                                   */
/ ********************************************************************************** /

main ()
{
    float   sum, n, term;
    int     count = 1;
    sum     = 0;
    printf("Enter value of n\n");
        scanf("%f", &n);
    term = 1.0/n;
    while ( count <= n )
    {
        sum = sum + term;
        count++;
    }
    printf("Sum = %f\n", sum);
}
```

Output

```
Enter value of n
99
Sum = 1.000001
Enter value of n
143
Sum = 0.999999
```

Fig. 4.6 *Round-off errors in floating point computations*

We know that the sum of n terms of 1/n is 1. However, due to errors in floating point representation, the result is not always 1.

4.14 TYPE CONVERSIONS IN EXPRESSIONS

Automatic Type Conversion

C permits mixing of constants and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversion. We know that the computer, considers one operator at a time, involving two operands.

If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of the higher type. A typical type conversion process is illustrated in Fig. 4.7.

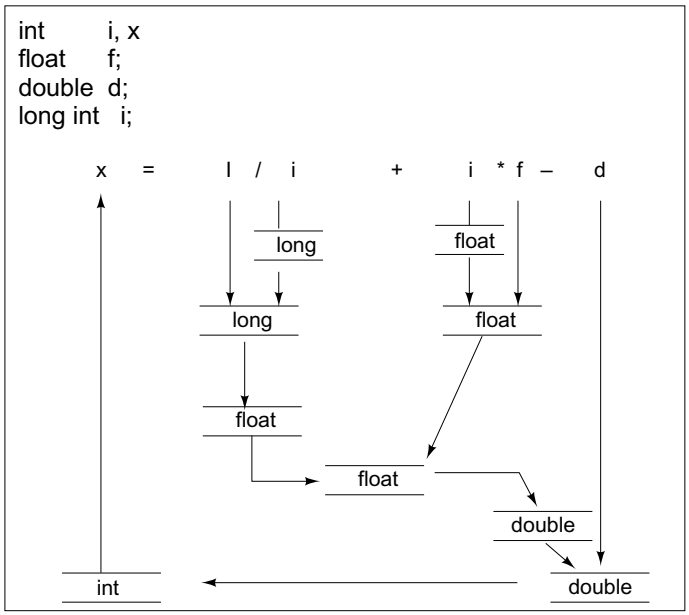


Fig. 4.7 *Process of automatical type conversion*

Given below is the sequence of rules that are applied while evaluating expressions.

All **short** and **char** are automatically converted to **int;** then

1. if one of the operands is **long double**, the other will be converted to **long double** and the result will be **long double**;
2. else, if one of the operands is **double**, the other will be converted to **double** and the result will be **double**;
3. else, if one of the operands is **float**, the other will converted to **float** and the result will be **float**;
4. else, if one of the operands is **unsigned long int**, the other will be converted to **unsigned long int** and the result will be **unsigned long int**;
5. else, if one of the operands is **long int** and the other is **unsigned int**, then:
   (a) if **unsigned int** can be converted to **long int**, the **unsigned int** operand will be converted as such and the result will be **long int**;
   (b) else, both operands will be converted to **unsigned long int** and the result will be **unsigned long int**;
6. else, if one of the operands is **long int**, the other will be converted to **long int** and the result will be **long int**;
7. else, if one of the operands is **unsigned int**, the other will be converted to **unsigned int** and the result will be **unsigned int**.

Note that some versions of C automatically convert all floating-point operands to double precision.

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment.

1. **float** to **int** causes truncation of the fractional part.

2. **double** to **float** causes rounding of digits.
3. **long int** to **int** causes dropping of the excess higher order bits.

Casting a Value

We have just discussed how C performs type conversion automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of females to males in a town.

```
ratio = female_number/male_number
```

Since **female_number** and **male_number** are declared as integers in the program, the decimal part of the result of the division would be lost and **ratio** would represent a wrong figure. This problem can be solved by converting *locally* one of the variables to the floating point as shown below:

```
ratio = (float) female_number/male_number
```

The operator (**float**) converts the **female_number** to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of result.

Note that in no way does the operator (**float**) affect the value of the variable **female_number**. And also, the type of **female_number** remains as **int** in the other parts of the program.

The process of such a local conversion is known as *casting* a value. The general form of a cast is:

(type-name) expression

Where *type-name* is one of the standard C data types. The *expression* may be a constant, variable or an expression. Some examples of casts and their actions are shown in Table 4.7.

Table 4.7 Use of Casts

Example	Action
x = (**int**) 7.5	7.5 is converted to integer by truncation.
a = (**int**) 21.3/(**int**)4.5	Evaluated as 21/4 and the result would be 5.
b = (**double**)sum/n	Division is done in floating point mode.
y = (**int**) (a + b)	The result of a + b is converted to integer.
z = (**int**)a + b	a is converted to integer and then added to b.
p = cos((**double**)x)	Converts x to double before using it.

Program

```
/* ************************************************************************** */
/*                PROGRAM SHOWING THE USE OF A CAST                           */
/* ************************************************************************** */
main()
{
    float sum;
    int n;

    sum = 0;
```

```
    for( n = 1; n <= 10; ++n )
    {
        sum = sum + 1/(float)n;
        printf("% 2d %6.4f\n", n, sum);
    }
}
Output
    1   1.0000
    2   1.5000
    3   1.8333
    4   2.0833
    5   2.2833
    6   2.4500
    7   2.5929
    8   2.7179
    9   2.8290
   10   2.9290
```

Fig. 4.8 *Use of a cast*

Casting can be used to round-off a given value. Consider the following statement:

x = (int) (y + 0.5);

If **y** is 27.6 **y** + 0.5 is 28.1 and on casting, the result becomes 28, the value that is assigned to **x**. Of course, the expression, being cast is not changed.

Example 4.6
Figure 4.8 shows a program using a cast to evaluate the equation

$$\text{sum} = \sum_{i=1}^{n} (1/i)$$

When combining two different types of variables in an expression, never assume the rules of automatic conversion. It is always a good practice to explicitly force the conversion. It is more safer safer and more portable. For example, when **y** and **p** are **double** and **m** is **int**, the following two statements are equivalent.

```
y = p + m;
y = p + (double)m;
```

However, the second statement is preferable. It will work the same way on all machines and is more readable.

4.15 OPERATOR PRECEDENCE AND ASSOCIATIVITY

Each operator in C has a precedence associated with it. This 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 one of the levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from left to right or from right to left, depending on the level. This is known as the *associativity* property of an operator. Table 4.8 pro-

vides a complete list of operators, their precedence levels, and their rules of association. The groups are listed in the order of decreasing precedence (rank 1 indicates the highest precedence level and 15 the lowest). The list also includes those operators which we have not yet discussed.

It is very important to note carefully, the order of precedence and associativity of operators. Consider the following conditional statement:

```
if (x == 10 + 15 && y < 10)
```

The precedence rules say that the *addition* operator has a higher priority than the logical operator (&&) and the relational operators (== and <). Therefore, the addition of 10 and 15 is executed first. This is equivalent to:

```
if (x == 25 && y < 10)
```

The next step is to determine whether **x** is equal to 25 and **y** is less than 10. If we assume a value of 20 for x and 5 for y, then

```
x == 25 is FALSE (0)
y < 10 is TRUE (1)
```

Note that since the operator < enjoys a higher priority compared to ==, y < 10 is tested first and then **x** == 25 is tested.

Finally we get:

```
if (FALSE && TRUE)
```

Because one of the conditions is FALSE, the complex condition is FALSE.

In the case of **&&**, it is guaranteed that the second operand will not be evaluated if the first is zero and in the case of ||, the second operand will not be evaluated if the first is non-zero.

CASE STUDIES

1. Salesman's Salary

A computer manufacturing company has the following monthly compensation policy to their sales-persons:

Minimum base salary	:	1500.00
Bonus for every computer sold	:	200.00
Commission on the total monthly sales	:	2 per cent

Since the prices of computers are changing, the sales price of each computer is fixed at the beginning of every month. A program to compute a sales-person's gross salary is given in Fig. 4.9.

Given the base salary, bonus, and commission rate, the inputs necessary to calculate the gross salary are, the price of each computer and the number sold during the month.

The gross salary is given by the equation:

```
Gross salary = base salary + (quantity * bonus rate)
             + (quantity * Price) * commission
```

Program

```
/ ************************************************************************************** /
/*              PROGRAM TO CALCULATE A SALESMAN'S SALARY                              */
/ ************************************************************************************** /

#define     BASE_SALARY     1500.00
#define     BONUS_RATE       200.00
#define     COMMISSION         0.02
main()

{
    int quantity;
    float gross_salary, price;
    float bonus, commission;

    printf("Input number sold and price\n");
    scanf("%d %f", &quantity, &price);

    bonus          = BONUS_RATE * quantity;
    commission     = COMMISSION * quantity * price;
    gross_salary   = BASE_SALARY + bonus + commission;

    printf("\n");
    printf("Bonus          = %6.2f\n", bonus);
    printf("Commission     = %6.2f\n", commission);
    printf("Gross salary   = %6.2f\n", gross-salary);

}
```

Output

```
Input number sold and price
5       20450.00

Bonus               = 1000.00
Commission          = 2045.00
Gross salary        = 4545.00
```

Fig. 4.9 *Program of salesman's salary*

Review Questions and Exercises

4.1 Which of the following arithmetic expressions are valid? If valid, give the value of the expression; otherwise give reason.

(a) 25/3 % 2
(b) + 9/4 + 5
(c) 7.5 % 3
(d) 14 % 3 + 7 % 2
(e) –14 % 3
(f) 15.25 + – 5.0
(g) (5/3) * 3 + 5 % 3
(h) 21 % (int) 4.5

4.2 Write C assignment statements to evaluate the following equations:

(a) Area = $\pi r^2 + 2\pi rh$

(b) Torque = $\dfrac{2m_1m_2}{m_1 + m_2} \cdot g$

(c) Side = $\sqrt{a^2 + b^2 - 2ab\cos(x)}$

(d) Energy = mass $\left[\text{acceleration} \times \text{height} + \dfrac{(\text{velocity})^2}{2}\right]$

4.3 Identify unnecessary parentheses in the following arithmetic expressions.

(a) ((x–(y/5)+z)%8) + 25 (b) ((x–y) * p)+q

(c) (m*n) + (–x/y) (d) x/(3*y)

4.4 Find errors, if any, in the following assignment statements and rectify them.

(a) x = y = z = 0.5, 2.0. –5.75; (b) m = ++a * 5;

(c) y = sqrt(100); (d) p * = x/y;

(e) s = /5; (f) a = b++ –c * 2

4.5 Determine the value of each of the following logical expressions if a = 5, b = 10 and c = –6.

(a) a > b && a < c (b) a < b && a > c

(c) a == c || b > a (d) b > 15 && c < 0 || a > 0

(e) (a/2.0 == 0.0 && b/2.0 ! = 0.0) || c < 0.0

4.6 The straight-line method of computing the yearly depreciation of the value of an item is given by

$$\text{Depreciation} = \frac{\text{Purchase Price} - \text{Salvage Value}}{\text{Years of Service}}$$

Write a program to determine the salvage value of an item when the purchase price, years of service, and the annual depreciation are given.

4.7 Write a program that will read a real number from the keyboard and print the following output in one line:

Smallest integer not less than the number	The given number	Largest integer not greater than the number

4.8 The total distance travelled by a vehicle in t seconds is given by

$$\text{distance} = ut + (at^2)/2$$

Where u is the initial velocity (meters per second), a is the acceleration (meters per second^2). Write a program to evaluate the distance travelled at regular intervals of time, given the values of u and a. The program should provide the flexibility to the user to select his own time intervals and repeat the calculations for different values of u and a.

4.9 In inventory management, the Economic Order Quantity for a single item is given by

$$\text{EOQ} = \sqrt{\frac{2 \times \text{demand rate} \times \text{setup costs}}{\text{holding cost per item per unit time}}}$$

and the optimal Time Between Orders

$$\text{TBO} = \sqrt{\frac{\overline{2} \times \text{setup costs}}{\text{demand rate} \times \text{holding cost per item per unit time}}}$$

Write a program to compute EOQ and TBO, given demand rate (items per unit time), setup costs (per order), and the holding cost (per item per unit time).

4.10 For a certain electrical circuit with an inductance L and resistance R, the damped natural frequency is given by

$$\text{Frequency} = \sqrt{\frac{1}{LC} - \frac{R^2}{4C^2}}$$

It is desired to study the variation of this frequency with C (capacitance). Write a program to calculate the frequency for different values of C starting from 0.01 to 0.1 in steps of 0.01.

Chapter 5

Managing Input and Output Operations

5.1 INTRODUCTION

Reading, processing, and writing of data are the three essential functions of a computer program. Most programs take some data as input and display the processed data, often known as *information* or *results*, on a suitable medium. So far we have seen two methods of providing data to the program variables. One method is to assign values to variables through the assignment statements such as **x = 5**; **a = 0**; and so on. Another method is to use the input function **scanf** which can read data from a terminal. We have used both the methods in most of our earlier example programs. For outputting results we have used extensively the function **printf** which sends results out to a terminal.

5.1.1 Input/output Statements and Header Files

Unlike other high-level languages, C does not have any built-in input/output statements as part of its syntax. All input/output operations are carried out through function calls such as **printf** and **scanf**. There exist several functions that have more or less become standard for input and output operations in C. These functions are collectively known as the *standard I/O library*. In this chapter we shall discuss some common I/O functions that can be used on many machines without any change. However, one should consult the system reference manual for exact details of these functions and also to see what other functions are available.

It may be recalled that we have included a statement

```
#include <math.h>
```

in the Sample Program 5 in Chapter 2, where a math library function cos(x) has been used. This is to instruct the compiler to fetch the function cos(x) from the math library, and that it is not a part of C language. Similarly, each program that uses a standard input/output function must contain the statement

```
#include <stdio.h>
```

at the beginning. However, there might be exceptions. For example, this is not necessary for the functions **printf** and **scanf** which have been defined as a part of the C language.

The file name **stdio.h** is an abbreviation for *standard input-output header* file. The instruction **#include <stdio.h>** tells the compiler 'to search for a file named *stdio.h* and place its contents at this point in the program. The contents of the header file become part of the source code when it is compiled.

5.2 READING A CHARACTER

The simplest of all input/output operations is reading a character from the standard input unit (usually the keyboard) and writing it to the standard output unit (usually the screen). Reading a single character can be done by using the function **getchar**. (This can also be done with the help of the **scanf** function which is discussed in Section 5.4) The **getchar** takes the following form:

variable_name = **getchar();**

variable_name is a valid C name that has been declared as **char** type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to **getchar** function. Since **getchar** is used on the right-hand side of an assignment statement, the character value of **getchar** is in turn assigned to the variable name on the left.
For example

```
char name;
name = getchar ();
```

will assign the character 'H' to the variable **name** when we press the key H on the keyboard. Since **getchar** is a function, it requires a set of parentheses as shown.

Example 5.1
The program in Fig. 5.1 shows the use of **getchar** function in an interactive environment.

The program displays a question of YES/NO type to the user and reads the user's response in a single character (Y or N). If the response is Y, it outputs the message

My name is BUSY BEE

otherwise, outputs "you are good for nothing".

The **getchar** function may be called successively to read the characters contained in a line of text. For example, the following program segment read characters from keyboard one after another until the 'Return' key is pressed.

```
--------------
--------------
char character;
character = ' ';
while (character ! = '\n')
{
   character = getchar();
}
--------------
--------------
```

Warning: The **getchar()** function accepts any character keyed in. This includes RETURN and TAB. This means that when we enter single character input, the newline character is waiting in the input queue after **getchar()** returns. This could create problems when we use **getchar ()** interactively in a loop. A dummy **getchar()** may be used to "eat" the unwanted newline character.

```
Program
/ ************************************************************************** /
/*              READING A CHARACTER FROM TERMINAL                          */
/ ************************************************************************** /

#include      <stdio.h>

main()
{
    char answer;

    printf("Would you like to know my name?\n");

    printf("Type Y for YES and N for NO: ");

    answer = getchar();      /* ..... Reading a character ..... */

    if(answer == 'Y'  ||answer == 'y')
    printf("\n\n My  name is BUSY BEE\' n");
    else
    printf ("\n\n You are good for nothing \n");
}
Output
Would you like to know my name?
Type Y for YES and N for NO: Y

My name is BUSY BEE

Would you like to know my name?
Type Y for YES and N for NO; n

You are good for nothing
```

Fig. 5.1 *Use of* ***getchar*** *function*

Example 5.2

The program of Fig. 5.2 requests the user to enter a character and displays a message on the screen telling the user whether the character is an alphabet or digit, or any other special character.

This program receives a character typed from the keyboard and tests whether it is a letter or digit and prints out a message accordingly. These tests are done with the help of the following functions:

isalpha(character)
isdigit(character)

For example, **isalpha** assumes a value non-zero (TRUE) if the argument **character** contains an alphabet; otherwise it assumes 0 (FALSE). Similar is the case with the function **isdigit**.

C supports many other similar functions which are given in Table 5.1. These character functions are contained in the file **ctype.h** and therefore the statement

#include < ctype.h >

must be included in the program.

```
Program
/ ************************************************************************** /
/*                      TESTING CHARACTER TYPE                             */
/ ************************************************************************** /

#include      <stdio.h>
#include      <ctype.h>

main()
{
    char character;
    printf("Press any key \n");
```

```
    character = getchar();

    if (isalpha(character) > 0)
      printf("The character is a letter.");
    else
    if(isdigit (character) > 0)
      printf("The character is a digit.");
    else
      printf("The character is not alphanumeric.");
}
```

Output

```
Press any key
h
The character is a letter.
Press any key
5
The character is a digit.
Press any key
*
The character is not alphanumeric.
```

Fig. 5.2 *Program to test the character type*

Table 5.1 Character Test Functions

Function	Test
isalnum(c)	Is c an alphanumeric character?
isalpha(c)	Is c an alphabetic character?
isdigit(c)	Is c a digit?
islower(c)	Is c a lower case letter?
isprint(c)	Is c a printable character?
ispunct(c)	Is c a punctuation mark?
isspace(c)	Is c a *white* space character?
isupper(c)	Is c an upper case letter?

5.3 WRITING A CHARACTER

Like **getchar**, there is an analogous function **putchar** for writing characters one at a time to the terminal. It takes the form as shown below:

putchar(*variable_name*);

where *variable-name* is a type **char** variable containing a character. This statement displays the character contained in the **variable-name** at the terminal. For example, the statements

```
answer = 'Y';
putchar(answer);
```

will display the character Y on the screen. The statement

```
putchar('\n');
```

would cause the cursor on the screen to move to the beginning of the next line.

Example 5.3
A program that reads a character from keyboard and then prints it in reverse case is given in Fig. 5.3 That is, if the input is upper case, the output will be lower case and vice-versa.

The program uses three new functions; **islower**, **toupper**, and **tolower**. The function **islower** is a conditional function and takes the value TRUE if the argument is a lower case alphabet; otherwise takes the value FALSE. The function **toupper** converts the lower case argument into an upper case alphabet while the function **tolower** does the reverse.

Program

```
/ ********************************************************************************** /
/*                  WRITING A CHARACTER TO THE TERMINAL                           */
/ ********************************************************************************** /

#include <stdio.h>
#include <ctype.h>

main()

{
    char alphabet;

    printf("Enter an alphabet");
    putchar('\n');
    alphabet = getchar();

    if (islower(alphabet))
        putchar(toupper(alphabet));
    else
        putchar(tolower(alphabet));
}
```

Output

```
Enter an alphabet
a
A

Enter an alphabet
Q
q

Enter an alphabet
z
Z
```

Fig. 5.3 *Reading and writing of alphabets in reverse case*

5.4 FORMATTED INPUT

Formatted input refers to an input data that has been arranged in a particular format. For example, consider the following data:

```
15.75    123    John
```

This line contains three pieces of data, arranged in a particular form. Such data has to be read conforming to the format of its appearance. For example, the first part of the data should be read into a variable **float**, the second into **int**, and the third part into **char**. This is possible in C using the **scanf** function.

We have already used this input function in a number of examples. Here, we shall explore all of the options that are available for reading the formatted data with **scanf** function. The general form of **scanf** is

scanf("*control string*", *arg1, arg2, ... argn*);

The *control string* specifies the field format in which the data is to be entered and the arguments *arg1, arg2, ... argn* specify the address of locations where the data is stored. Control string and arguments are separated by commas.

Control string contains field specifications which direct the interpretation of input data. It may include:

- Field (or format) specifications, consisting of the conversion character %, a data type character (or type specifier), and an *optional* number, specifying the field width.
- Blanks, tabs, or newlines.

Blanks, tabs and newlines are ignored. The data type character indicates the type of data that is to be assigned to the variable associated with the corresponding argument. The field width specifier is optional. The discussions that follow will clarify these concepts.

Inputting Integer Numbers

The field specification for reading an integer number is:

% *w* d

The per cent sign (%) indicates that a conversion specification follows. *w* is an integer number that specifies the *field width* of the number to be read and d, known as data type character, indicates that the number to be read is in integer mode. Consider the following example:

```
scanf("%2d %5d", &num1, &num2);
```

Data line:

```
50    31426
```

The value 50 is assigned to **num1** and 31426 to **num2**. Suppose the input data is as follows:

```
31426  50
```

The variable **num1** will be assigned 31 (because of %2d) and **num2** will be assigned 426 (unreal part of 31426). The value 50 that is unread will be assigned to the first variable in the next **scanf** call. This kind of errors may be eliminated if we use the field specifications without the field with specifications. That is, the statement

```
scanf("%d %d", &num1, &num2);
```

will read the data

```
31426  50
```

correctly and assign 31426 to **num1** and 50 to **num2**.

Input data items must be separated by spaces, tabs or newlines. Punctuations marks do not count as separators. When the **scanf** function searches the input data line for a value to be read, it will always bypass any white space characters.

What happens if we enter a floating point number instead of an integer? The fractional part may be stripped away! Also, **scanf** may skip reading further input.

When the **scanf** reads a particular value, reading of the value will terminate as soon as the number of characters specified by the field width is reached (if specified) or until a character that is not valid for the value being read is encountered. In the case of integers, valid characters are an optionally signed sequence of digits.

An input field may be skipped by specifying * in the place of field width. For example, the statement

```
scanf("%d %*d %d", &a, &b)
```

will assign the data

```
123   456   789
```

as follows:

```
123   to a
456   skipped (because of *)
789   to b
```

The data type character **d** may be preceded by 'l' (letter ell) to read long integers.

Example 5.4

Various input formatting options for reading integers are experimented with in the program shown in Fig. 5.4.

The first **scanf** requests input data for three integer values **a**, **b**, and **c**, and accordingly three values 1, 2, and 3 are keyed in. Because of the specification %**d* the value 2 has been skipped and 3 is assigned to the variable **b**. Notice that since no data is available for c, it contains garbage.

The second **scanf** specifies the format %2d and %4d for the variables **x** and **y** respectively. Whenever we specify field width for reading integer numbers, the input numbers should not contain more digits than the specified size. Otherwise, the extra digits on the right-hand side will be truncated and assigned to the next variable in the list. Thus, the second **scanf** has truncated the four digit number 6789 and assigned 67 to **x** and 89 to **y**. The value 4321 has assigned to the first variable in the immediately following **scanf** statement.

Program

```
/ ************************************************************************ /
/*                    READING INTEGER NUMBERS                           */
/ ************************************************************************ /

smain()
{
    int a,b,c,x,y,z;
    int p,q,r;

    printf("Enter three integer numbers\n");
    scanf("%d %*d %d",&a,&b,&c);
    printf("%d %d\n\n",a,b,c);

    printf("Enter two 4-digit numbers\n");
    scanf("%2d %4d",&x,&y);
    printf("%d %d\n\n",x,y);

    printf("Enter a two digit integers\n");
```

```
    scanf("%d %d", &a,&x);
    print("%d %d", &a,&x);

    printf("Enter a nine digit number¯\n");
    scanf("%3d %4d %3d",&p,&q,&r);
    printf("%d %d %d \n\n",p,q,r);

    printf("Enter two three digit numbers\n");
    scanf("%d %d",&x,&y);
    printf("%d %d",x,y);
}
```

Output

```
Enter three integer numbers
1   2   3
1   3   –3577
Enter two 4-digit numbers
6789   4321
67  89
Enter two integers
44      66
4321   44
Enter a nine digit number
123456789
66  1234   567
Enter two three digit numbers
123     456
89  123
```

Fig. 5.4 *Reading integers using* ***scanf***

Inputting Real Numbers

Unlike integer numbers, the field width of real numbers is not to be specified and therefore **scanf** reads real numbers using the simple specification **%f** for both the notations, namely, decimal point notation and exponential notation. For example, the statement

```
scanf("%f %f %f", &x, &y, &z);
```

with the input data

```
475.89  43.21E-1        678
```

will assign the value 475.89 to **x**, 4.321 to **y**, and 678.0 to **z**. The input field specifications may be separated by any arbitrary blank spaces.

If the number to be read is of **double** type, then the specification should be **%lf** instead of simple **%f**. A number may be skipped using **%*f** specification.

Example 5.5

Reading of real numbers (in both decimal point and exponential notation) is illustrated in Fig. 5.5.

Program

```
/ ************************************************************************** /
/ *                      READING OF REAL NUMBERS                           */
/ ************************************************************************** /

main()
{
    float x,y;
    double p,q;

    printf("Values of x and y:");
    scanf("%f %e", &x, &y);
    printf("\n");
    printf("x = %f\ny = %f\n\n", x, y);

    printf("Values of p and q:");
    scanf("%lf" %lf", &p, &q);
    printf("\np = %lf\nq = %e",p,q);
    printf("\n\np = %. 12 lf\np = %. 123", p,q);
}
```

Output

```
Values of x and y: 12.3456  17.5e-2

x = 12.345600
y = 0.175000

Values of p and q: 4.142857142857 18.5678901234567890

p = 4.4142857142857
q = 1.856789012346e+001
```

Fig. 5.5 *Reading of real numbers*

Input Character Strings

We have already seen how a single character can be read from the terminal using the **getchar** function. The same can be achieved using the **scanf** function also. In addition, a **scanf** function can input strings containing more than one character. Following are the specifications for reading character strings:

%ws or **%wc**

The corresponding argument should be a pointer to a character array. However, %c may be used to read a single character when the argument is a pointer to a **char** variable.

Example 5.6
Reading of strings using **%wc** and **%ws** is illustrated in Fig. 5.6.

The program in Fig. 5.6 illustrates the use of various field specifications for reading strings. When we use **%wc** for reading a string, the system will wait until the *w*th character is keyed in.

Note that the specification **%s** terminates reading at the encounter of a blank space. Therefore, **name2** has read only the first part of "New York" and the second part is automatically assigned to **name3**. However, during the second run, the string "New-York" is correctly assigned to **name2**.

Some versions of **scanf** support the following conversion specifications for strings:

%[characters] and %[characters]

The specification %**[characters]** means that only the characters specified within the brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character. The specification **%[characters]** does exactly the reverse. That is, the characters specified after the circumflex (^) are not permitted in the input string. The reading of the string will be terminated at the encounter of one of these characters.

Program

```
/ ********************************************************************** /
/ *                        READING STRINGS                             */
/ ********************************************************************** /

main()
{
    int no;
    char name1[15], name2[15], name3[15];

    printf("Enter serial number and name one\n");
    scanf("%d %15c", &no, name1);
    printf("%d %15s\n\n", no, name1);

    printf{"Enter serial number and name two\n");
    scanf("5d %s", &no, name2);
    printf("%d %15s\n\n", no, name2);

    printf("Enter serial number and name three\n");
    scanf("%d %15s", &no, name3);
    printf("%d %15s\n\n", no, name3);
}
```

Output

```
Enter serial number and name one
1   123456789012345
1   123456789012345r

Enter serial number and name two
2   New York
2                   New

Enter serial number and name three
2                   York
Enter serial number and name one
1   123456789012
1   123456789012   r

Enter serial number and name two
2    New York
2              New-York

Enter serial number and name three
3   London
3                 London
```

Fig. 5.6 *Reading of strings*

We have just seen that **%s** specifier cannot be used to read strings with blank spaces. But, this can be done with the help of %[ ] specification. Blank spaces may be included within the brackets, thus enabling the **scanf** to read strings with spaces. Remember that the lowercase and uppercase letters are distinct. Example 5.7 illustrates the use of %[ ] specification.

Example 5.7
The program in Fig. 5.7 illustrates the function of %[ ] specification.

Reading Mixed Data Types

It is possible to use one **scanf** statement to input a data line containing mixed mode data. In such cases, care should be exercised to ensure that the input data items match the control specifications *in order* and *type*. When an attempt is made to read an item that does not match the type expected, the **scanf** function does not read any further and immediately returns the values read. The statement

scanf("%d %c %f %s", &count, &code, &ratio, name);

will read the data

15 p 1.575 coffee

correctly and assign the values to the variables in the order in which they appear. Some systems accept integers in the place of real numbers and vice-versa, and the input data is converted to the type specified in the control string.

Detection of Errors in Input

When a **scanf** function completes its list, it returns the value of number of items that are successfully

Program-A

```
/ ********************************************************************************** /
/*                    ILLUSTRATION OF %[ ] SPECIFICATION                          */
/ ********************************************************************************** /
main()
{
    char address[80];
    printf("Enter address \n");
    scanf("%[a-z]", address);
    printf("%–80s \n\n",address);
}
```

Output

```
Enter address
new delhi 110 002
new delhi
```

Program-B

```
/ ********************************************************************************** /
/*                    ILLUSTRATION OF %[ ^ ] SPECIFICATION                        */
/ ********************************************************************************** /

main()
{
    char address[80];
    printf("Enter address\n");
    scanf("%[^\n]",address);
    printf("%-80s \n\n",address);
}
```

Output

```
Enter address
New Delhi 110 002
New Delhi 110 002
```

Fig. 5.7 *Illustration of conversion specification [. .] for strings*

read. This value can be used to test whether any errors occurred in reading the input. For example, the statement

```
scanf("%d %f %s", &a, &b, name);
```

will return the value 3 if the following data is typed in:

```
20      150.25  motor
```

and will return the value 1 if the following line is entered

```
20      motor   150.25
```

This is because the function would encounter a string when it was expecting a floating point value, and would therefore terminate its scan after reading the first value.

Example 5.8

The program presented in Fig. 5.8 illustrates the testing for correctness of reading of data by **scanf** function.

Program

```
/ ************************************************************************** /
/ *          TESTING FOR CORRECTNESS OF INPUT DATA                         */
/ ************************************************************************** /

main()
{
    int a;
    float b;
    char c;

    printf("Enter values of a, b and c\n");

    if (scanf("%d %f %c", &a, &b, &c) == 3)
      printf("a = %d b = %f c = %c\n", a, b, c);
    else
        printf("Error in input\n");
}
```

Output

```
Enter values of a, b and c
12   3.45   A
a = 12   b = 3.450000 c = A

Enter values of a, b and c
23  78  9
a = 23 b = 78.000000    c = 9
Enter values of a, b and c
8   A   5.25
Error in input.

Enter values of a, b and c
Y   12  67
Error in input.

Enter values of a, b and c
15.75  23  X
a = 15 b = 0.750000 c = 2
```

Fig. 5.8 *Detection of errors in* ***scanf*** *input*

The function **scanf** is expected to read three items of data and therefore, when the values for all the three variables are read correctly, the program prints out their values. During the third run, the second item

does not match with the type of variable and therefore the reading is terminated and the error message is printed. Same is the case with the fourth run.

In the last run, although data items do not match the variables, no error message has been printed. When we attempt to read a real number for an **int** variable, the integer part is assigned to the variable, and the truncated decimal part is assigned to the next variable. Note that the character '2' is assigned to the character variable **c**.

Commonly used **scanf** format codes are given in Table 5.2.

Table 5.2 scanf Format Codes

Code	Meaning
%c	read a single character
%d	read a decimal integer
%e	read a floating point value
%f	read a floating point value
%g	read a floating point value
%h	read a short integer
%i	read a decimal, hexadecimal, or octal integer
%o	read an octal integer
%s	read a string
%u	read an unsigned decimal integer
%x	read a hexadecimal integer
%[..]	read a string of word(s)

The following letters may be used as prefix for certain conversion characters.

h for short integers
l for long integers or double
L for long double

Points to Remember While Using scanf

If we do not plan carefully, some crazy things can happen with **scanf**. Since the I/O routines are not a part of C language, they are made available either as a separate module of the C library or as a part of the operating system (like UNIX). New features are added to these routines from time to time as new versions of systems are released. We should consult the system reference manual before using these routines. Given below are some of the general points to keep in mind while writing a **scanf** statement.

1. All function arguments, except the control string, *must* be pointers to variables.
2. Format specifications contained in the control string should match the arguments in order.
3. Input data items must be separated by spaces and must match the variables receiving the input in the same order.
4. The reading will be terminated, when **scanf** encounters an 'invalid mismatch' of data or a character that is not valid for the value being read.
5. When searching for a value, **scanf** ignores line boundaries and simply looks for the next appropriate character.

6. Any unread data items in a line will be considered as a part of the data input line to the next **scanf** call.
7. When the field width specifier *w* is used, it should be large enough to contain the input data size.

5.5 FORMATTED OUTPUT

We have seen the use of **printf** function for printing captions and numerical results. It is highly desirable that the outputs are produced in such a way that they are understandable and are in an easy-to-use form. It is therefore necessary for the programmer to give careful consideration to the appearance and clarity of the output produced by his program.

The **printf** statement provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the terminals. The general form of **printf** statement is

printf("*control string", arg1, arg2, ... argn*);

Control string consists of three types of items:

1. Characters that will be printed on the screen as they appear.
2. Format specifications that define the output format for display of each item.
3. *Escape sequence* characters such as \n, \t, and \b.

The control string indicates how many arguments follow and what their types are. The arguments *arg1, arg2, ... argn* are the variables whose values are formatted and printed according to the specifications of the control string. The arguments should match in number, order and type with the format specifications.

A simple format specification has the following form:

% w.p type-specifier

where *w* is an integer number that specifies the total number of columns for the output value and *p* is another integer number that specifies the number of digits to the right of the decimal point (of a real number) or the number of characters to be printed from a string. Both *w* and *p* are optional. Some examples of **printf** statement are:

```
printf("Programming in C");
printf(" ");
printf("\n");
printf("%d", x);
printf("a = %f\n b = %f", a, b);
printf("sum = %d", 1234);
printf("\n\n");
```

printf never supplies a newline automatically and therefore multiple **printf** statements may be used to build oneline of output. A newline can be introduced by the help of a *newline* character '\n' as shown in some of the examples above.

Output of Integer Numbers

The format specification for printing an integer number is

% *w* d

where *w* specifies the minimum field width for the output. However, if a number is greater than the specified field width, it will be printed in full, overriding the minimum specification, **d** specifies that the value to be printed is an integer. The number is written *right-justified* in the given field width. Leading blanks will appear as necessary. The following examples illustrate the output of the number 9876 under different formats:

Format	*Output*
printf("%d", 9876)	9 8 7 6
printf("%6d", 9876)	␣ ␣ 9 8 7 6
printf("%2d", 9876)	9 8 7 6
printf("%–6d", 9876)	9 8 7 6 ␣ ␣
printf("%06d", 9876)	0 0 9 8 7 6

It is possible to force the printing to be *left-justified* by placing a *minus* sign directly after the % character as shown in the fourth example above. It is also possible to pad with zeros the leading blanks by placing a 0 (zero) before the field width specifier as shown in the last item above.

Long integers may be printed by specifying **ld** in the place of **d** in the format specification.

Example 5.9

The program in Fig. 5.9 illustrates the output of integer numbers under various formats.

Program

```
/ ************************************************************************** /
/ *                  PRINTING OF INTEGER NUMBERS                           */
/ ************************************************************************** /

main()
{
    int m = 12345;
    long n = 987654;

    printf("%d\n",m);
    printf("%10d\n",m);
    printf("%010d\n",m);
    printf("%-10d\n",m);
    printf("%101d\n",n);
    printf("% 101d\n",_n);
}
```

Output

```
12345
     12345
0000012345
12345
    987654
   –987654
```

Fig. 5.9 *Formatted output of integers*

Output of Real Numbers

The output of a real number may be displayed in decimal notation using the following format specification:

% *w.p* f

The integer *w* indicates the minimum number of positions that are to be used for the display of the value and the integer *p* indicates the number of digits to be displayed after the decimal point (*precision*). The value, when displayed, is *rounded to p decimal places* and printed *right-justified* in the field of *w* columns. Leading blanks and trailing zeros will appear as necessary. The default precision is 6 decimal places. The negative numbers will be printed with the minus sign. The number will be displayed in the form [–] mmm . nnn.

We can also display a real number in exponential notation by using the specification

% *w.p* e

The display takes the form

[–] m.nnnne[±]xx

where the length of the string of n's is specified by the precision *p*. The default precision is 6. The field width *w* should satisfy the condition.

$$w \geq p+7$$

The value will be rounded off and printed right justified in the field of *w* columns.

Padding the leading blanks with zeros and printing with *left-justification* is also possible by introducing 0 or – before the field width specifier *w*.

The following examples illustrate the output of the number $y = 98.7654$ under different format specifications:

Format	*Output*
printf("%7.4f",y)	9 8 . 7 6 5 4
printf("%7.2f",y)	9 8 . 7 7
printf("%–7.2f",y)	9 8 . 7 7
printf("%f",y)	9 8 . 7 6 5 4
printf("%10.2e",y)	9 . 8 8 e + 0 1
printf("%11.4e",–y)	– 9 . 8 7 6 5 e + 0 1
printf("% – 10.2e",y)	9 8 8 e + 0 1
printf("% e",y)	9 8 7 6 5 4 0 e+ 0 1

Some systems also support a special field specification character that lets the user define the field size at run-time. This takes the following form:

printf("%*.*f", width, precision, number);

In this case, both the field width and the precision are given as arguments which will supply the values for *w* and *p*. For example,

```
printf("%*.*f",7,2,number);
```

is equivalent to

```
print("%7.2f",number);
```

The advantage of this format is that the values for *width* and *precision* may be supplied at run-time, thus making the format a *dynamic* one. For example, the above statement can be used as follows:

```
int width = 7;
int precision = 2;
.........
.........
printf("%*.*f", width, precision, number);
```

Example 5.10
All the options of printing a real number are illustrated in Fig. 5.10.

Program

```
/ ********************************************************************************** /
/*                          PRINTING OF REAL NUMBERS                              */
/ ********************************************************************************** /

main()
{
    float y = 98.7654;

    printf("%7.4f\n", y);
    printf("%f\n", y);
    printf("%7.2f\n", y);
    print("% -7.2f\n", y);
    printf("%0.72f\n", y);
    printf("%*.*f",7, 2, y);
    printf("\n");
    printf("% 10.2e\n", y);
    printf("12.4e\n", –y);
    printf("%–10.2e\n", y);
    printf("%e\n", y);
}
```

Output

```
98.7654
98.765404
   98.77
98.77
0098.77
98.77

9.88e+001
–9.8765e+001
9.88e+001
9.876540e+001
```

Fig. 5.10 *Formatted output of real numbers*

Note: Microsoft C supports three digits in exponent part.

Printing of a Single Character

A single character can be displayed in a desired position using the format

% *wc*

The character will be displayed *right-justified* in the field of *w* columns. We can make the display *left-justified* by placing a minus sign before the integer *w*. The default value for *w* is 1.

Printing of Strings

The format specification for outputting strings is similar to that of real numbers. It is of the form

% *w.ps*

where *w* specifies the field width for display and *p* instructs that only the first *p* characters of the string are to be displayed. The display is *right-justified*.

The following examples show the effect of a variety of specifications in printing a string "NEW DELHI 110001", containing 16 characters (including blanks)

Specification *Output*

	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
%s	N	E	W		D	E	L	H	I		1	1	0	0	0	1				
%20s					N	E	W		D	E	L	H	I		1	1	0	0	0	1
%20.10s											N	E	W		D	E	L	H	I	
%.5s	N	E	W		D															
%–20.10s	N	E	W		D	E	L	H	I											
%5s	N	E	W		D	E	L	H	I		1	1	0	0	0	1				

Example 5.11
Printing of characters and strings is illustrated in Fig. 5.11.

Mixed Data Output

It is permitted to mix data types in one printf statement. For example, the statement of the type

```
printf("%d %f %s %c", a, b, c, d);
```

is valid. As pointed out earlier, **printf** uses its control string to decide how many variables to be printed and what their types are. Therefore, the format specifications should match the variables in number, order, and type. If there are not enough variables or if they are of the wrong type, incorrect results will be output.

Program

```
/ ************************************************************************** /
/*              PRINTING OF CHARACTERS AND STRINGS                          */
/ ************************************************************************** /

main()
{
    char x = 'A';
    static char name[20] = "ANIL KUMAR GUPTA";
    printf("OUTPUT OF CHARACTERS\n");
    printf("%c\n%3c\n%5c\n", x,x,x);
    printf("%3c\n%c\n", x,x);
    printf("\n");

    printf("OUTPUT OF STRINGS\n\n");
    printf("%s\n", name);
    printf("%20s\n", name);
    printf("%20.10s\n", name);
    printf("%.5s\n", name);
    printf("%-20.10s\n", name);
    printf("%5s\n", name);
}
```

Output

```
OUTPUT OF CHARACTERS

A
   A
      A
   A
A
OUTPUT OF STRINGS

ANIL KUMAR GUPTA
   ANIL KUMAR GUPTA
          ANIL KUMAR
ANIL
ANIL KUMAR
ANIL KUMAR GUPTA
```

Fig. 5.11 *Printing of characters and strings*

Commonly used **printf** format codes are given in Table 5.3 and format flags in Table 5.4.

Table 5.3 printf Format Codes

Code	Meaning
%c	print a single character
%d	print a decimal integer
%e	print a floating point value in exponent form
%f	print a floating point value without exponent
%g	print a floating point value either e-type or f-type depending on value
%i	print a signed decimal integer
%o	print an octal integer, without leading zero
%s	print a string
%u	print an unsigned decimal integer
%x	print a hexadecimal integer, without leading 0x

The following letters may be used as prefix for certain conversion characters.

h for short integers,
l for long integers or double,
L for long double

Table 5.4 Output Format Flags

Flag	Meaning
–	Output is left-justified within the field. Remaining field will be blank.
+	+ or – will precede the signed numeric item.
0	Causes leading zeros to appear.
# (with o or x)	Causes octal and hex items to be preceded by O and Ox, respectively.
# (with e, f or g)	Causes a decimal point to be present in all floating point numbers, even if it is whole number. Also prevents the truncation of trailing zeros in g-type conversion.

Enhancing the Readability of Output

Computer outputs are used as information for analysing certain relationships between variables and for making decisions. Therefore the correctness and clarity of outputs is of utmost importance. While the correctness depends on the solution procedure, the clarity depends on the way the output is presented. Following are some of the steps we can take to improve the clarity and hence the readability and understandability of outputs.

1. Provide enough blank space between two numbers.
2. Introduce appropriate headings and variable names in the output.
3. Print special messages whenever a peculiar condition occurs in the output.
4. Introduce blank lines between the important sections of the output.

The system usually provides two blank spaces between the numbers. However, this can be increased by selecting a suitable field width for the numbers or by introducing a 'tab' character between the specifications. For example, the statement

```
printf("a = %d\t b = %d", a, b);
```

will provide four blank spaces between the two fields. We can also print them on two separate lines by using the statement

```
printf("a = %d\n b = %d", a, b);
```

Messages and headings can be printed by using the character strings directly in the **printf** statement. Examples:

```
printf("\n OUTPUT RESULTS  \n");
printf("code\t Name\t Age\n");
printf("Error in input data\n");
printf(Enter your name\n");
```

CASE STUDIES

1. Inventory Report

Problem: The ABC Electric Company manufactures four consumer products. Their inventory position on a particular day is given below:

Code	Quantity	Rate (Rs)
F105	275	575.00
H220	107	99.95
I 109	321	215.50
M315	89	725.00

It is required to prepare the inventory report table in the following format:

INVENTORY REPORT

Code	Quantity	Rate	Value
—	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—
		Total Value:	

The value of each item is given by the product of quantity and rate.
Program: The program given in Fig. 5.12 reads the data from the terminal and generates the required output. The program uses subscripted variables which are discussed in Chapter 8.

2. Reliability Graph

Problem: The reliability of an electronic component is given by

$$\text{reliability (r)} = e^{-\lambda t}$$

Program

```
/ ********************************************************************** /
/*                        INVENTORY REPORT                              */
/ ********************************************************************** /

#define ITEMS 4

main()
{ /* BEGIN */

    int i, quantity[5];
    float rate[5], value, total-value;
    char code[5][5];

    /* READING VALUES */

    i = 1;
    while (i <= ITEMS)
```

```
    {
        printf("Enter code, quantity, and rate:");
        scanf("%s %d %f", code[i], &quantity[i], &rate[i], i++;
    }
/* ..... Printing of Table and Column Headings ..... */
    printf("\n\n");
    printf("                    INVENTORY REPORT              \n");
    printf(" ------------------------------------------------------------ \n");
    printf("   Code      Quantity      Rate      Value          \n");
    printf(" ------------------------------------------------------------ \n");
/* ..... Preparation of Inventory Position ..... */
    total-value = 0;
    i = 1;
    while (i <= ITEMS);
    {
        value = quantity[i] * rate[i];
        printf("%5s % 10d % 10.2f %e\n" code[i], quantity[i], rate[i], value);
        total-value +=value;
        i++;
    }
/* ..... Printing of End of Table ..... */
    printf(" ------------------------------------------------------------ \n");
    printf("                          Total Value = %e\n", total-value);
    printf(" ------------------------------------------------------------ \n*);
}   /* END */
```

Output

```
Enter code, quantity, and rate:F105   275   575.00
Enter code, quantity, and rate:H220   107   99.95
Enter code, quantity, and rate:I019321      215.50
Enter code, quantity, and rate:M315         725.00

                    INVENTORY REPORT

------------------------------------------------------------------------
Code          Quantity          Rate                Value
------------------------------------------------------------------------
F105           275              575.00          1.581250e+005
H220           107               99.95          1.069465e+004
I019           321              215.50          6.917550e+004
M315            89              725.00          6.452500e+004
------------------------------------------------------------------------
                                    Total Value = 3.025202e+005
------------------------------------------------------------------------
```

Fig. 5.12 *Program for inventory report*

where λ is the component failure rate per hour and t is the time of operation in hours. A graph is required to determine the reliability at various times, from 0 to 300 hours. The failure rate λ (lamda) is 0.001.

Program

```
/ ************************************************************************** /
/*                          RELIABILITY GRAPH                             */
/ ************************************************************************** /

#include   <math.h>
```

```
#define    LAMDA    0.001
main()
{
    double t;
    float r;
    int i, R;
    for (i = 1, i<=27; ++i)
    {
        printf("—");
    }
    print("\n");
    for (t=0; t<=3000; t +=150)
    {
        r = exp(–LAMDA*t);
        R = (int)(50*r+0.5);
        printf("||");
        for (i=1; 1<=R; ++i)
        {
            print("*");
        }
        printf("#\n");
    }
    for (i = 1; i<3; ++i)
    {
        printf(" |\n");
    }
}
```

Output

```
--------------------------------------------------------------------------------
    **********************************************************#
    *****************************************************#
    ***************************************************#
    *****************************************#
    ***********************************#
    *********************************#
    *******************************#
    ****************************#
    ************************#
    **********************#
    ******************#
    ****************#
    **************#
    ************#
    **********#
    *********#
    *******#
    ****#
    ***#
    **#
```

Fig. 5.13 *Program to draw reliability graph*

Program: The program given in Fig. 5.13 produces a shaded graph. The values of t are self-generated by the for statement

for (t=0; t<=3000; t = t+150)

in steps of 150. The integer 50 in the statement

```
R = (int)(50*r+0.5)
```

is a scale factor which converts r to a large value where an integer is used for plotting the curve. Remember r is always less than 1.

Review Questions and Exercises

For questions 5.1 to 5.5 assume that the following declarations have been made in the program:

```
int     year, count;
float   amount, price;
char    code, city[10];
double  root;
```

5.1 State errors, if any, in the following input statements.
- (a) scanf("%c%f%d", city, &price, &year);
- (b) scanf("%s%d", city, amount);
- (c) scanf("%f, %d, &amount, &year);
- (d) scanf(\n"%f", root);
- (e) scanf("%c %d %ld", *code, &count, Root);

5.2 What will be the values stored in the variables **year** and **code** when the data

```
1988, x
```

is keyed in as a response to the following statements:
- (a) scanf("%d %c", &year, &code);
- (b) scanf("%c %d", &year, &code);
- (c) scanf("%d %c", &code, &year);
- (d) scanf("%s %c", &year, &code);

5.3 The variables **count**, **price**, and **city** have the following values:

```
count  ←  1275
price  ←  –235.74
city   ←  Cambridge
```

Show the exact output that the following output statements will produce:
- (a) printf("%d %f", count, price);
- (b) printf("%2d\n%f", count, price);
- (c) printf("%d %f", price, count);
- (d) printf("%10dxxxx%5.2f", count, price);
- (e) printf("%s", city);
- (f) printf("%–10d%–15s", count, city);

5.4 State what (if anything) is wrong with each of the following output statements:
- (a) printf(%d 7.2%f, year, amount);
- (b) printf("%–s, %c"\n, city, code);
- (c) printf("%f, %d, %s, price, count, city);
- (d) printf("%c%d%f\n", amount, code, year);

5.5 In response to the input statement

```
scanf("%4d% *%d", &year, &code, &count);
```

the following data is keyed in:

```
19883745
```

What values does the computer assign to the variables **year**, **code**, and **count**?

5.6 Given the string "WORDPROCESSING", write a program to read the string from the terminal and display the same in the following formats:

(a) WORD PROCESSING
(b) WORD
PROCESSING
(c) W.P.

5.7 Write a program to read the values of x and y and print the results of the following expressions in one line:

(a) $\frac{x+y}{x-y}$ (b) $\frac{x+y}{2}$ (c) $(x+y)(x-y)$

5.8 Write a program to read the following numbers, round them off to the nearest integers and print out the results in integer form:

35.7 50.21 –23.73 –46.45

Chapter 6

Decision Making and Branching

6.1 INTRODUCTION

We have seen that a C program is a set of statements which are normally executed sequentially in the order in which they appear. This happens when no options or no repetitions of certain calculations are necessary. However, in practice, we have a number of situations where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.

C language possesses such decision making capabilities and supports the following statements known as *control* or *decision* making statements.

1. **if** statement
2. **switch** statement
3. Conditional operator statement
4. **goto** statement

We have already used some of these statements in the earlier examples. Here we shall discuss their features, capabilities and applications in more detail.

6.2 DECISION MAKING WITH IF STATEMENT

The **if** statement is a powerful decision making statement and is used to control the flow of execution of statements. It is basically a *two-way* decision statement and is used in conjunction with an expression. It takes the following form:

if (*test expression*)

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression (relation or condition) is 'true' (non-zero) or 'false' (zero), it transfers the control to a particular statement. This point of program has two *paths* to follow, one for the true condition and the other for the false condition as shown in Fig. 6.1.

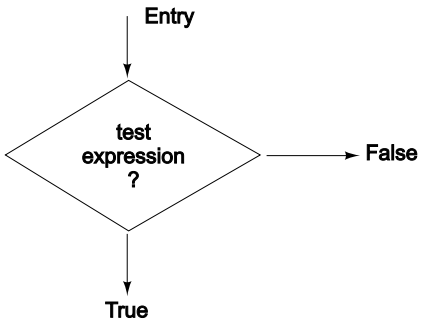


Fig. 6.1 *Two-way branching*

Some examples of decision making, using **if** statement are:

1. **if** (bank balance is zero)
   borrow money
2. **if** (room is dark)
   put on lights
3. **if** (code is 1)
   person is male
4. **if** (age is more than 55)
   person is retired

The **if** statement may be implemented in different forms depending on the complexity of conditions to be tested.

1. Simple **if** statement
2. **if** .... **else** statement
3. Nested **if** .... **else** statement
4. **else if** ladder.

6.3 SIMPLE IF STATEMENT

The general form of a simple **if** statement is

```
if(test expression)
  {
    statement-block;
  }
statement-x;
```

The 'statement-block' may be a single statement or a group of statements. If the *test expression* is true, the *statement-block* will be executed; otherwise the statement-block will be skipped and the execution will jump to the *statement-x*. Remember, when the condition is true both the statement-block and the statement-x are executed in sequence. This is illustrated in Fig. 6.2.

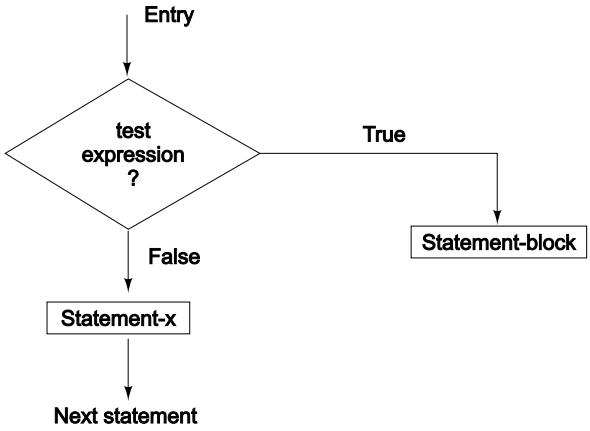


Fig. 6.2 *Flowchart of simple **if** control*

Consider the following segment of a program that is written for processing of marks obtained in an entrance examination.

```
.....
.....
if (category == SPORTS)
  {
    mark = marks + bonus_marks;
  }
printf("%f", marks);
.....
.....
```

The program tests the type of category of the student. If the student belongs to the SPORTS category, then additional bonus_marks are added to his marks before they are printed. For others, bonus_marks are not added.

Example 6.1

The program in Fig. 6.3 reads four values a, b, c, and d from the terminal and evaluates the ratio of (a + b) to (c – d) and prints the result, if c – d is not equal to zero.

The program given in Fig. 6.3 has been run for two sets of data to see that the paths function properly. The result of the first run is printed as

Ratio = –3.181818

The second run has neither produced any results nor any message. During the second run, the value of (c – d) is equal to zero and therefore the statements contained in the statement-block are skipped. Since no other statement follows the statement-block, program stops without producing any output.

Note the use of **float** conversion in the statement evaluating the **ratio**. This is necessary to avoid truncation due to integer division.

Program

```
/ ********************************************************************************** /
/*                          ILLUSTRATION OF if STATEMENT                          */
/ ********************************************************************************** /

main( )
{
      int a, b, c, d;
      float ratio;

      printf("Enter four integer values\n");
      scanf("%d %d %d", &a, &b, &c, &d);

      if (c–d  != 0)
        {
        ratio = (float) (a+b) / (float) (c–d);
        printf("Ratio = %f\n", ratio);
        }
}
```

Output

```
Enter four integer values
12    23    34     45
Ratio = –3.181818

Enter four integer values
12    23    34     34
```

Fig. 6.3 *Illustration of simple* ***if*** *statement*

The simple **if** is often used for counting purposes. The example 6.2 illustrates this.

Example 6.2

The program in Fig. 6.4 counts the number of boys whose weight is less than 50 kgs and height is greater than 170 cm.

The program has to test two conditions, one for weight and another for height. This is done using the compound relation

```
if (weight < 50 && height > 170)
```

This would have been equivalently done using two **if** statements as follows:

```
if (weight < 50)
   if (height > 170)
      count = count +1;
```

If the value on **weight** is less than **50**, then the following statement is executed, which is turn is another **if** statement. This **if** statement tests **height** and if the **height** is greater than **170**, then the **count** is incremented by 1.

Program

```
/ ************************************************************************** /
/*                          COUNTING WITH if                                */
/ ************************************************************************** /

main( )
{
        int count, i;
        float weight, height;

        count = 0;
        printf("Enter weight and height for 10 boys/n");
        for (i =1; i <= 10; i++)
        {
           scanf("%f %f", &weight, &height);
           if(weight < 50 && height > 170)
             count = count + 1;
        }
        printf("Number of boys with weight < 50 kgs\n");
        printf("and height > 170 cm = %d\n", count);
}
```

Output

```
Enter weight and height for 10 boys
45      176.5
55      174.2
47      168.0
49      170.7
54      169.0
53      170.5
49      167.0
48      175.0
47      167
51      170
Number of boys with weight < 50 kgs
and height > 170 cm = 3
```

Fig. 6.4 *Use of **if** for counting*

6.4 THE IF … ELSE STATEMENT

The **if** … **else** statement is an extension of the simple **if** statement. The general form is

```
if (test expression)
   {
      True-block statement(s)
   }
else
   {
      False-block statement(s)
   }
statement-x
```

If the *test expression* is true, then the *true-block statement(s)*, immediately following the **if** statement are executed; otherwise; the *false-block statement(s)* are executed. In either case, either *true-block* or

false-block will be executed, not both. This is illustrated in Fig. 6.5. In both the cases, the control is transferred subsequently to statement-x.

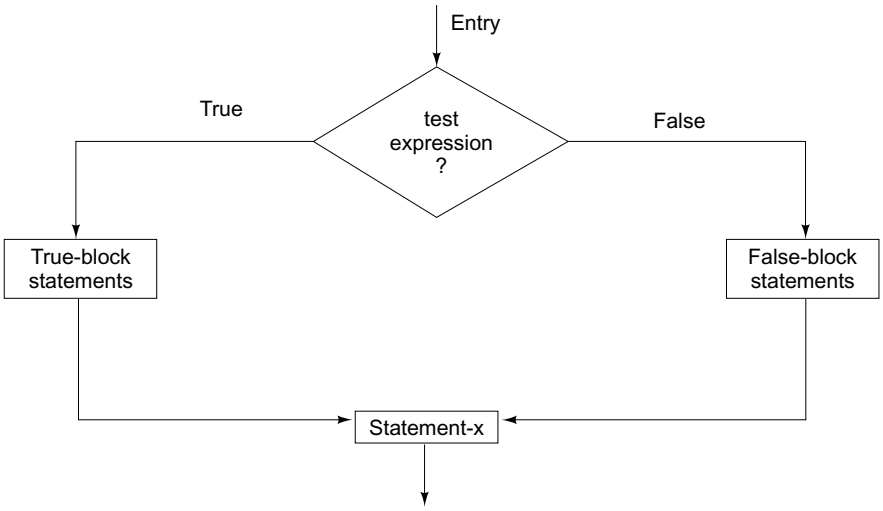


Fig. 6.5 *Flowchart of* ***if*** *…… **else** control*

Let us consider an example of counting the number of boys and girls in a class. We use code 1 for a boy and 2 for a girl. The program statement to do this may be written as follows:

```
.....
.....
if (code == 1)
    boy = boy + 1;
if (code == 2)
    girl = girl + 1;
.....
.....
```

The first test determines whether or not the student is a boy. If yes, the number of boys is increased by 1 and the program continues to the second test. The second test again determines whether the students is a girl. This is unnecessary. Once a student is identified as a boy, there is no need to test again for a girl. A student can be either a boy or a girl, not both. The above program segment can be modified using the **else** clause as follows:

```
.....
.....
if (code == 1)
    boy = boy + 1;
else
    girl = girl + 1;
xxxxxxxxxx
.....
```

Here, if the code is equal to 1, the statement **boy = boy + 1**; is executed and the control is transferred to statement **xxxxxx**, after skipping the **else** part. If the code is not equal to 1, the statement **boy = boy + 1**; is skipped and the statement in the **else** part **girl = girl + 1**; is executed before the control reaches the statement **xxxxxxxx**.

Consider the program given in Fig. 6.3. When the value (c – d) is zero, the ratio is not calculated and the program stops without any message. In such cases we may not know whether the program stopped due to a zero value or some other error. This program can be improved by adding the **else** clause as follows:

```
.....
.....
if (c – d != 0)
   {
   ratio = (float)(a+b)/(float)(c–d);
   printf("Ratio = %f\n", ratio);
   }
else
   printf("c–d is zero\n");
.....
.....
```

Example 6.3

A program to evaluate the power series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!},\ 0 < x < 1$$

is given in Fig. 6.6. It uses **if** ..... **else** to test the accuracy.

The power series contains the recurrence relationship of the type

$$T_n = T_{n-1}\left(\frac{x}{n}\right) \text{ for } n > 1$$

$$T_1 = x \qquad \text{for } n = 1$$

$$T_0 = 1$$

If T_{n-1} (usually known as *previous term*) is known, then T_n (known as *present term*) can be easily found by multiplying the *previous term* by x/n. Then

$$e^x = T_0 + T_1 + T_2 + \ldots\ldots + T_n = \text{sum}$$

The program uses **count** to count the number of terms added. The program stops when the value of the term is less than 0.0001 (ACCURACY). Note that when a term is less than ACCURACY, the value of **n** is set equal to 999 (a number higher than 100) and therefore the **while** loop terminates. The results are printed outside the **while** loop.

6.5 NESTING OF IF ... ELSE STATEMENTS

When a series of decisions are involved, we may have to use more than one **if ... else** statement in *nested* form as follows:

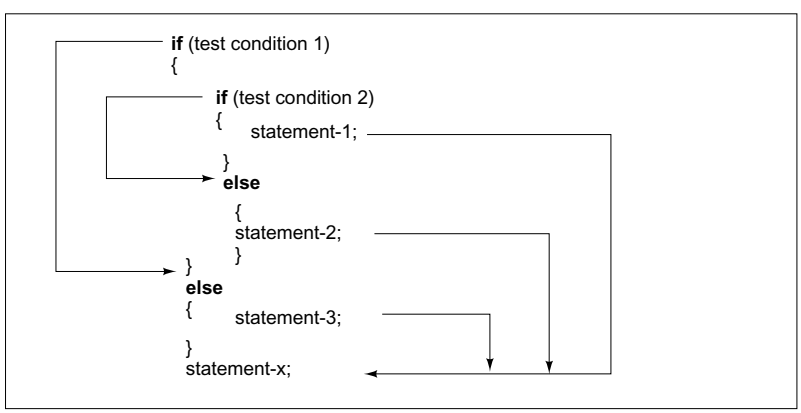
Program

```
/*********************************************************************************/
/*                    EXPERIMENT WITH if ... else STATEMENT                      */
/*********************************************************************************/

#define ACCURACY 0.0001

main()
{
      int n, count;
      float x, term, sum;

      printf("Enter value of x:");
      scanf("%f", &x);

      n = term = sum = count = 1;

      while (n <= 100)
      {
        term = term * x/n;
        sum = sum + term;
        count = count + 1;
        if (term <ACCURACY)
          n = 999;
        else
          n = n + 1;
      }

      printf("Terms = %d Sum = %f\n", count, sum);

}
```

Output

```
Enter value of x:0
Terms = 2 Sum = 1.000000

Enter value of x:0.1
Terms = 5 Sum = 1.105171

Enter value of x:0.5
Terms = 7 Sum = 1.648720

Enter value of x:0.75
Terms = 8 Sum = 2.116997

Enter value of x:0.99
Terms = 9 Sum = 2.691232

Enter value of x:1
Terms = 9 Sum = 2.718279
```

Fig. 6.6 *Illustration of* ***if ... else*** *statement*



The logic of execution is illustrated in Fig. 6.7. If the *condition-1* is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the *condition-2* is true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

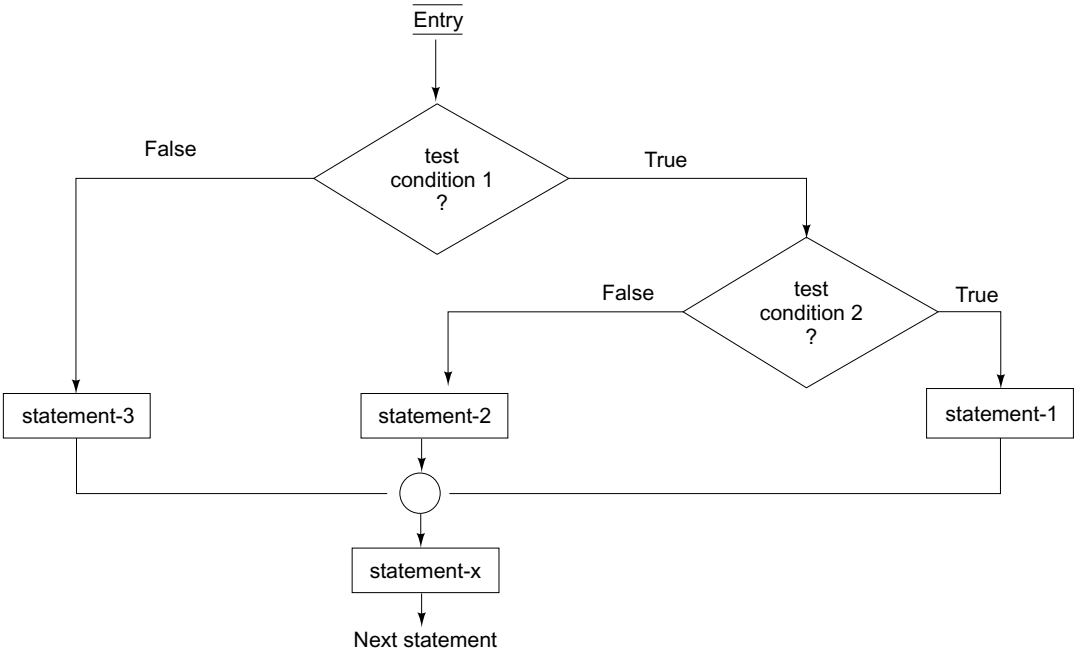


Fig. 6.7 *Flowchart of nested* ***if ... else*** *statements*

A commercial bank has introduced an incentive policy of giving bonus to all its deposit holders. The policy is as follows: A bonus of 2 per cent of the *balance* held on 31st December is given to every one, irrespective of their balances, and 5 per cent is given to female account holders if their balance is more than Rs. 5000. This logic can be coded as follows:

```
.....
if (sex is female)
   {
     if (balance > 5000)
        bonus = 0.05 * balance
     else
        bonus = 0.02 * balance;
   }
else
   {
     bonus 0.02 * balance;
   }
balance = balance + bonus;
.....
.....
```

When nesting, care should be exercised to match every **if** with an **else**. Consider the following alternative to the above program (which looks right at the first sight):

```
if (sex is female)
  if (balance > 5000)
    bonus = 0.05 * balance;
else
  bonus  =  0.02 * balance;
balance  =  balance + bonus;
```

There is an ambiguity as to over which **if** the **else** belongs to. In C, an **else** is linked to the closest non-terminated **if**. Therefore, the **else** is associated with the inner if and there is no else option for the outer **if**. This means that the computer is trying to execute the statement

```
balance  =  balance +  bonus;
```

without really calculating the bonus for the male account holders.

Consider another alternative which also looks correct:

```
if (sex is female)
  {
    if (balance > 5000)
      bonus = 0.05 * balance;
  }
else
  bonus = 0.02 * balance;
balance = balance + bonus;
```

In this case, **else** is associated with the outer **if** and therefore bonus is calculated for the male account holders. However, bonus for the female account holders, whose balance is equal to or less than 5000 is not calculated because of the missing **else** option for the inner **if**.

Example 6.4

The program in Fig. 6.8 selects and prints the largest of the three numbers using nested **if** ⋯ **else** statements.

Program

```
/ ********************************************************************************** /
/*          SELECTING THE LARGEST OF THREE VALUES                  */
/ ********************************************************************************** /

main( )
{
      float A, B, C;

      printf("Enter three values\n");
      scanf("%f %f %f", &A, &B, &C);

      printf("\n Largest value is");

      if (A > B)
        {
          if (A > C)
            printf("%f\n", A);
```

```
            else
              printf("%f\n", C);
          }
        else
          {
          if (C > B)
            printf("%f\n", C);
          else
            printf("%f\n", B);
          }
}
```

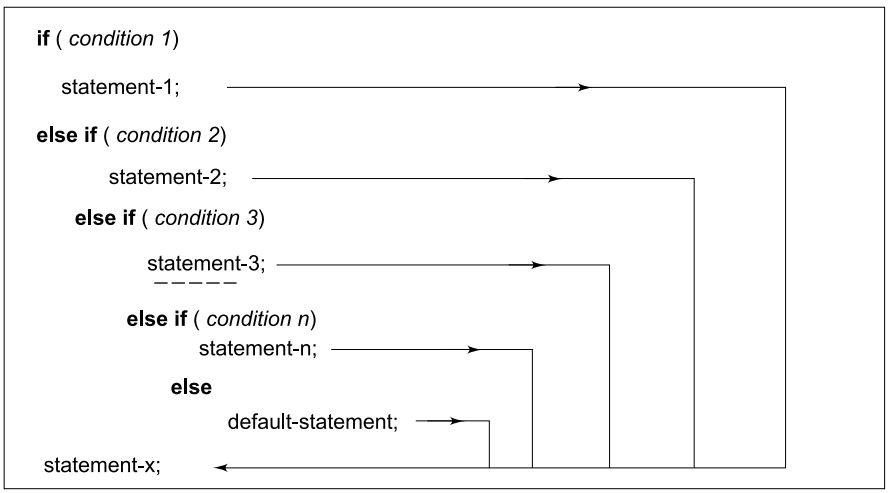
Output

```
Enter three values
23445 67379     88843

Largest value is 88843.000000
```

Fig. 6.8 *Selecting the largest of three numbers*

6.6 THE ELSE IF LADDER

There is another way of putting **if**s together when multipath decisions are involved. A multipath decision is a chain of **if**s in which the statement associated with each **else** is an **if**. It takes the following general form:



This construct is known as the **else if** ladder. The conditions are evaluated from the top (of the ladder), downwards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the *statement-x* (skipping the rest of the ladder). When all the n conditions become false, then the final **else** containing the *default-statement* will be executed. Figure 6.9 shows the logic of execution of **else if** ladder statements.

Let us consider an example of grading the students in an academic institution. The grading is done according to the following rules:

Average marks	*Grade*
80 to 100	Honours
60 to 79	First Division
50 to 59	Second Division
40 to 49	Third Division
0 to 39	Fail

This grading can be done using the **else if** ladder as follows (although syntactically, it is wrong!)

```
if (marks > 79)
  grade = "Honours";
else if (marks > 59)
        grade = "First Division";
     else if (marks > 49)
             grade = "Second Division";
          else if (marks > 39)
                  grade = "Third Division";
               else
                  grade = "Fail";
printf("%s\n", grade);
```

Consider another example given below:

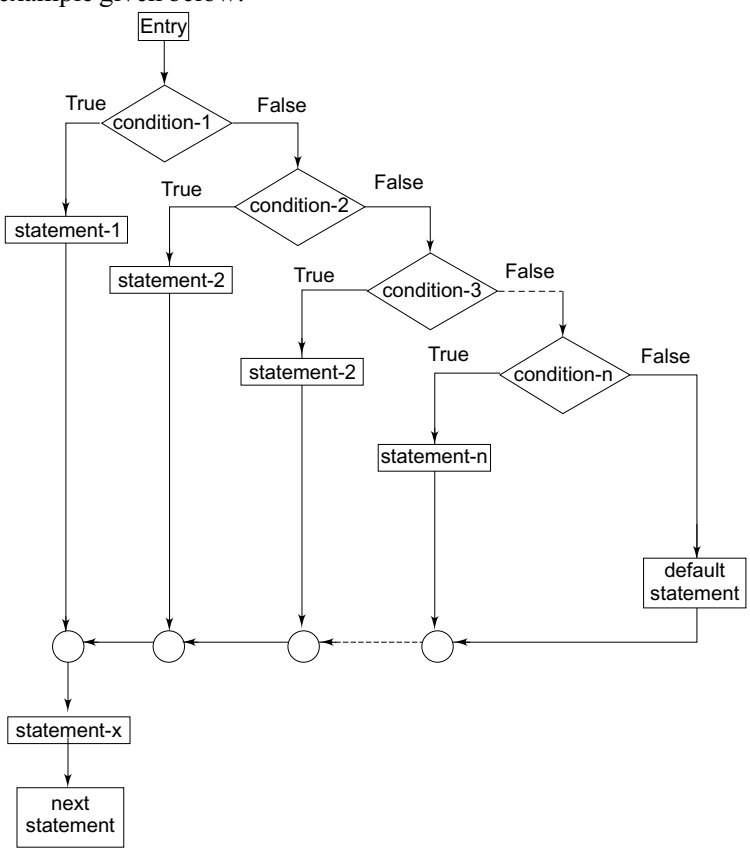


Fig. 6.9 *Flowchart of* ***else ... if*** *ladder*

```
.....
.....
if (code == 1)
  colour = "RED";
else if (code == 2)
        colour = "GREEN";
      else if (code == 3)
              colour = "WHITE";
            else
              colour = "YELLOW";
.....
.....
```

Code numbers other than 1, 2 or 3 are considered to represent YELLOW colour. The same results can be obtained by using nested **if** ⋯**else** statements.

```
if (code != 1)
   if (code != 2)
      if (code != 3)
         colour = "YELLOW";
     else
       colour = "WHITE";
  else
    colour = "GREEN";
else
  colour = "RED";
```

In such situations, the choice of the method is left to the programmer. However, in order to choose an **if** structure that is both effective and efficient, it is important that the programmer is fully aware of the various forms of an **if** statement and the rules governing their nesting.

Example 6.5

An electric power distribution company charges its domestic consumers as follows:

Consumption Units	*Rate of Charge*
0 - 20	Rs. 0.50 per unit
201 - 400	Rs. 100 plus Rs. 0.65 per unit excess of 200
401 - 600	Rs. 230 plus Rs. 0.80 per unit excess of 400
601 and above	Rs. 390 plus Rs. 1.00 per unit excess of 600

The program in Fig. 6.10 reads the customer number and power consumed and prints the amount to be paid by the customer.

6.7 THE SWITCH AND BREAK STATEMENTS

We have seen that when one of the many alternatives is to be selected, we can design a program using **if** statements to control the selection. However, the complexity of such a program increases dramatically

when the number of alternatives increases. The program becomes difficult to read and follow. At times, it may confuse even the person who designed it. Fortunately, C has a built-in multiway decision statement known as a **switch**. The **switch** statement tests the value of a given variable (or expression) against a list of **case** values and when a match is found, a block of statements associated with that **case** is executed. The general form of the **switch** statement is as shown in the next page.

Program

```
/ ********************************************************************************** /
/*                          USE OF else if LADDER                                 */
/ ********************************************************************************** /

main()
{
        int units, custnum;
        float charges;

        printf("Enter CUSTOMER NO. and UNITS consumed\n");
        scanf("%d %d", &custnum, &units);

        if (units <= 200)
            charges = 0.5* units;
        else if (units <= 400)
                charges = 100 + 0.65* (units – 200);
            else if (units <= 600)
                    charges = 230 + 0.8 * (units – 400);
                else
                    charges = 390 + (units – 600);
        printf("\n\nCustomer No: %d: Charges = %.2f\n",
                custnum, charges);
}
```

Output

```
Enter CUSTOMER NO. and UNITS consumed 101  150
Customer No: 101 Charges = 75.00

Enter CUSTOMER NO. and UNITS consumed 202  225

Customer No:202 Charges = 116.25

Enter CUSTOMER NO. and UNITS consumed 303  375

Customer No:303 Charges = 213.75

Enter CUSTOMER NO. and UNITS consumed 404  520

Customer No: 404 Charges = 326.00

Enter CUSTOMER NO. and UNITS consumed 505  625

Customer No:505 Charges = 415.00
```

Fig. 6.10 *Illustration of* ***else*** *⋯* ***if*** *ladder*

The *expression* is an integer expression or characters. *Value-1*, *value-2* …… are constants or constant expressions (evaluable to an integral constant) and are known as *case labels*. Each of these values should be unique within a **switch** statement **block-1**, **block-2** ⋯ are statement lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that **case** labels end with a colon (:).

When the **switch** is executed, the value of the expression is successively compared against the values *value-1*, *value-2*,..... If a case is found whose value matches with the value of the *expression*, then the block of statements that follows the case are executed.

```
switch (expression)
{
        case value-1:
                        block-1
                        break;
        case value-2:
                        block-2
                        break;
.......
.......
default:
                        default-block
                        break;
}
statement-x;
```

The **break** statement at the end of each block signals the end of a particular case and causes an exit from the **switch** statement, transferring the control to the **statement-x** following the **switch**.

The **default** is an optional case. When present, it will be executed if the value of the *expression* does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the **statement-x**.

The selection process of **switch** statement is illustrated in the flowchart shown in Fig. 6.11.

The **switch** statement can be used to grade the students as discussed in the last section. This is illustrated below:

```
.........
.........
index = marks/10;
switch (index)
{
   case 10:
   case 9:
   case 8:
         grade = "Honours";
         break;
   case 7:
   case 6:
         grade = "First Division";
         break;
   case 5:
         grade = "Second Division";
         break;
   case 4:
         grade = "Third Division";
         break;
   default:
         grade = "Fail";
         break;
```

```
}
printf("%s\n", grade);
.....
.....
```

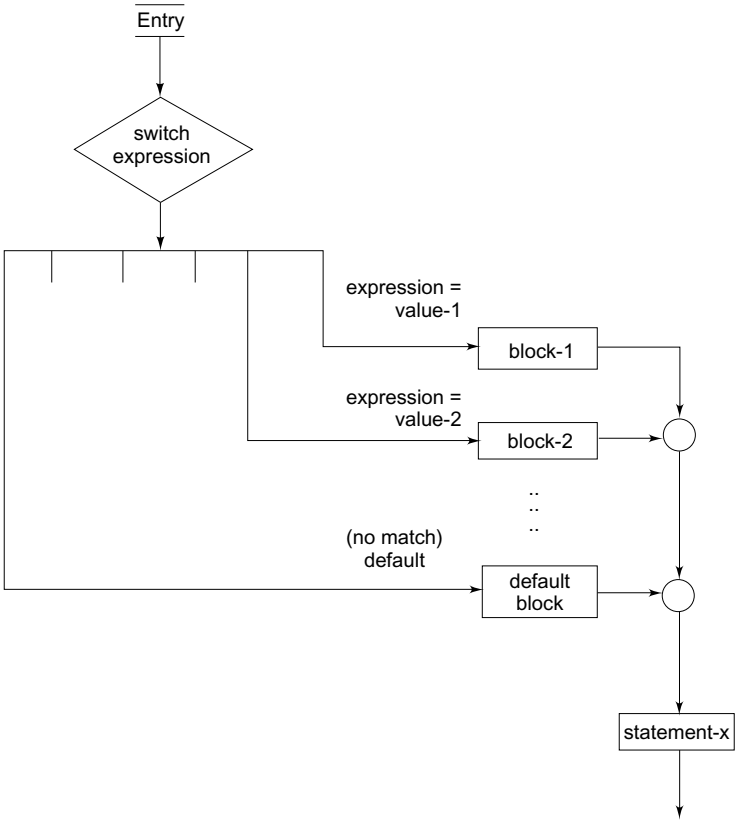


Fig. 6.11 *Selection process of the **switch** statement*

Note that we have used a conversion statement

index = marks / 10;

where, **index** is defined as an integer. The variable index takes the following integer values.

Marks	*Index*
100	10
90 – 99	9
80 – 89	8
70 – 79	7
60 – 69	6
50 – 59	5
40 – 49	4
:	:
0	0

This segment of the program illustrates two important features. First, it uses empty cases. The first three cases will execute the same statements

```
grade = "Honours";
break;
```

Same is the case with case 7 and case 6. Second, **default** condition is used for all other cases where marks is less than 40.

The **switch** statement is often used for menu selection. For example:

```
.....
.....
printf(" TRAVEL GUIDE \n\n");
printf("A Air Timings\n" );
printf(" T Train Timings\n");
printf(" B Bus Service\n" );
printf(" X to skip\n");
printf("\n Enter your choice\n");
character = getchar( );
switch (character)
{
   case 'A' :
            air-display( );
            break;
   case 'B' :
            bus-display( );
            break;
   case 'T' :
            train-display( );
            break;
   default  :
            printf("No choice\n");
}
.....
.....
```

It is possible to nest the **switch** statements, That is, a **switch** may be part of a **case** block. ANSI C allows 15 levels of nesting and as many as 257 *case labels*.

6.8 THE ? : OPERATOR

The C language has an unusual operator, useful for making two-way decisions. This operator is a combination of ? and :, and takes three operands. This operator is popularly known as the *conditional operator*. The general form of use of the conditional operator is as follows:

conditional expression **?** *expression1 : expression2*

The *conditional expression* is evaluated first. If the result is nonzero, *expression1* is evaluated and is returned as the value of the conditional expression. Otherwise, *expression2* is evaluated and its value is returned. For example, the segment

```
if (x < 0)
   flag = 0;
else
   flag = 1;
```

can be written as

```
flag = (x < 0) ? 0 : 1;
```

Consider the evaluation of the following function:

$y = 1.5x + 3$ for $x \leq 2$

$y = 2x + 5$ for $x > 2$

This can be evaluated using the conditional operator as follows:

```
y = (x > 2) ? (2 * x + 5) : (1.5 * x + 3);
```

The conditional operator may be nested for evaluating more complex assignment decisions. For example, consider the weekly salary of a salesgirl who is selling some domestic products. If x is the number of products sold in a week, her weekly salary is given by

$$\text{salary} = \begin{cases} 4x + 100 & \text{for } x < 40 \\ 300 & \text{for } x = 40 \\ 4.5x + 150 & \text{for } x > 40 \end{cases}$$

This complex equation can be written as

```
salary  = (x != 40) ? ((x < 40) ? (4*x + 100) : (4.5*x + 150)) : 300;
```

The same can be evaluated using **if** ⋯ **else** statements as follows:

```
if (x <= 40)
  if (x < 40)
    salary = 4 * x+100;
  else
    salary = 300;
else
    salary = 4.5 * x + 150;
```

When the conditional operator is used, the code becomes more concise and perhaps, more efficient. However, the readability is poor. It is better to use **if** statements when more than a single nesting of conditional operator is required.

Example 6.6

An employee can apply for a loan at the beginning of every six months, but he will be sanctioned the amount according to the following company rules:

Rule 1: An employee cannot enjoy more than two loans at any point of time.
Rule 2: Maximum permissible total loan is limited and depends upon the category of the employee.

A program to process loan applications and to sanction loans is given in Fig. 6.12.

The program uses the following variables:

loan3 - present loan amount requested
loan2 - previous loan amount pending
loan1 - previous to previous loan pending

maxloan-maximum permissible loan
sancloan-loan sanctioned

The rules for sanctioning new loan are:

1. loan1 should be zero.
2. loan2 + loan3 should not be more than maxloan.

Note the use of **long int** type to declare variables.

Program

```
/ ********************************************************************************** /
/*                          CONDITIONAL OPERATOR                                    */
/ ********************************************************************************** /

#define        MAXLOAN    50000

main()
{
    long int loan1, loan2, loan3, sancloan, sum23;

    printf("Enter the values of previous two loans:\n");
    scanf("%ld %ld", &loan1, &loan2);

    printf("\nEnter the value of new loan:\n");
    scanf(" %ld", &loan3);

    sum23 = loan2 + loan3;
    sancloan= (loan 1>0) ? 0 : ((sum23>MAXLOAN) ?
              MAXLOAN – loan2 : loan3);

    printf("\n\n");
    printf("Previous loans pending: \n%ld %ld\n", loan1, loan2);
    printf("Loan requested = %ld\n", loan3);
    printf("Loan sanctioned = %ld\n", sancloan);
}
```

Output

```
Enter the values of previous two loans:
0   20000

Enter the value of new loan:
45000


Previous loans pending:
0   20000
Loan requested   = 45000
Loan sanctioned = 30000

Enter the values of previous two loans:
1000   15000

Enter the value of new loan:
25000

Previous loans pending
1000  15000
Loan requested = 25000
Loan sanctioned = 0
```

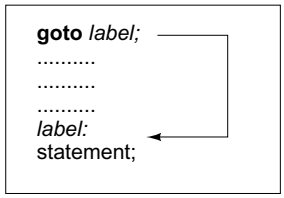
Fig. 6.12 *Illustration of the conditional operator*

6.9 THE GOTO STATEMENT

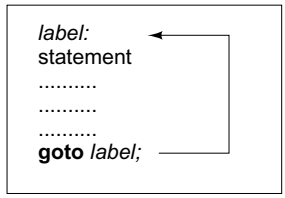
So far we have discussed ways of controlling the flow of execution based on certain specified conditions. Like many other languages, C supports the **goto** statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the **goto** statement in a highly structured language like C, there may be occasions when the use of **goto** might be desirable.

6.9.1 Label

The **goto** requires a *label* in order to identify the place where the branch is to be made. A **label** is any valid variable name, and must be followed by a colon. The *label* is placed immediately before the statement where the control is to be transferred. The general forms of **goto** and *label* statements are shown below:



Forward jump



Backward jump

The *label:* can be anywhere in the program either before or after the **goto** label**;** statement.

During running of a program when a statement like

goto begin;

is met, the flow of control will jump to the statement immediately following the label **begin:**. This happens unconditionally.

Note that a **goto** breaks the normal sequential execution of the program. If the *label*: is before the statement **goto** *label*; a *loop* will be formed and some statements will be executed repeatedly. Such a jump is known as a *backward jump*. On the other hand, if the *lable*: is placed after the **goto** *label*; some statements will be skipped and the jump is known as a *forward jump*.

A **goto** is often used at the end of a program to direct the control to go to the input statement, to read further data. Consider the following example:

```
main()
{
        double x, y;
        read:
        scanf("%f", &x);
        if (x < 0) goto read;
        y = sqrt(x);
        printf("%f %f\n", x, y);
        goto read;
}
```

This program is written to evaluate the square root of a series of numbers read from the terminal. The program uses two **goto** statements, one at the end, after printing the results to transfer the control back to the input statement and the other to skip any further computation when the number is negative.

Due to the unconditional **goto** statement at the end, the control is always transferred back to the input statement. In fact, this program puts the computer in a permanent loop known as an *infinite loop*. The

computer goes round and round until we take some special steps to terminate the loop. Such infinite loops should be avoided. Example 6.7 illustrates how such infinite loops can be eliminated.

Example 6.7

Program presented in Fig. 6.13 illustrates the use of the **goto** statement.

The program evaluates one square root for five numbers. The variable **count** keeps the count of numbers read. When **count** is less than or equal to 5, **goto read**; directs the control to the label **read**; otherwise, the program prints a message and stops.

Program

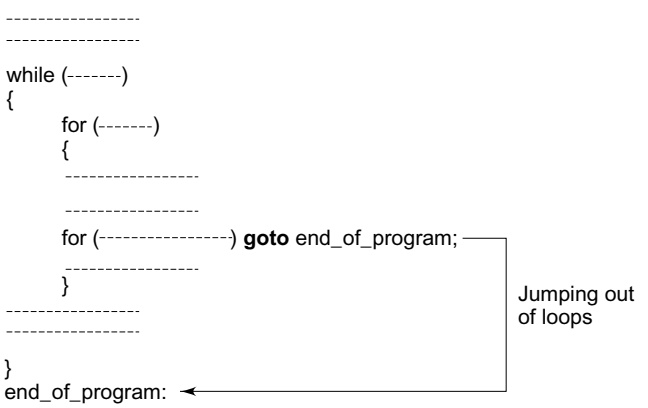
```
/ ********************************************************************************** /
/*                              USE OF goto STATEMENT                               */
/ ********************************************************************************** /
#include <math.h>

main()
{
    double x, y;
    int count;

    count = 1;

    printf("Enter FIVE real values in a LINE \n");
read:
    scanf("%lf", &x);
    printf("\n");
    if (x < 0)
        printf("Item—%d is negative \n",count);
    else
    {
        y = sqrt(x);
        printf("%lf\t %lf\n", x, y);
    }

    count = count + 1;

    if (count <= 5)
goto read;
    printf("\nEnd of computation");
}
```

Output

```
Enter FIVE real values in a LINE
50.70     40   –36     75     11.25
50.750000                 7.123903
40.000000                 6.324555

Item–3 is negative

75.000000                 8.660254
11.250000                 3.354102
End of Computation
```

Fig. 6.13 *Use of the **goto** statement*

Another use of the **goto** statement is to transfer the control out of a loop (or nested loops) when certain peculiar conditions are encountered. Example:



We should try to avoid using **goto** as far as possible. But there is nothing wrong, if we use it to enhance the readability of the program or to improve the execution speed.

CASE STUDIES

1. Range of Numbers

Problem: A survey of the computer market shows that personal computers are sold at varying costs by the vendors. The following is the list of costs (in hundreds) quoted by some vendors:

35.00	40.50	25.00	31.25	68.15
47.00	26.65	29.00	53.45	62.50

Determine the average cost and the range of values.
Problem analysis: Range is one of the measures of dispersion used in statistical analysis of a series of values. The range of any series is the difference between the highest and the lowest values in the series. That is

Range = highest value – lowest value

It is therefore necessary to find the highest and the lowest values in the series.
Program: A program to determine the range of values and the average cost of a personal computer in the market is given in Fig. 6.14.

When the value is read the first time, it is assigned to two *buckets*, **high** and **low**, through the statement

high = low = value;

For subsequent values, the value read is compared with **high**; if it is larger, the value is assigned to **high**. Otherwise, the value is compared with **low**; if it is smaller, the value is assigned to **low**. Note that at a given point, the buckets **high** and **low** hold the highest and the lowest values read so far.

Program

```
/ ************************************************************************************ /
/*                                  RANGE OF NUMBERS                                  */
/ ************************************************************************************ /

main()
{
        int count;
        float value, high, low, sum, average, range;

        sum = 0;
        count = 0;

        printf("Enter numbers in a line :
                    input a NEGATIVE number to end/n");
input:
        scanf("%f", &value);

        if (value < 0) goto output;
            count = count + 1;
        if (count == 1)
            high = low = value;
        else if (value > high)
                    high = value;
                else if (value < low)
                        low = value;

        sum = sum + value;
        goto input;
output:
        average = sum/count;
        range = high – low;
        printf("\n\n");
        printf("Total values    : %d\n", count);
        printf("Highest-value : %f\nLowest-value: %f\n", high, low);
        printf("Range           : %fAverage       :%f\n",
                    range, average);
}
```

Output

```
Enter numbers in a line : input a NEGATIVE number to end
35 40.50      25      31.25      68.15      47      26.65      29      53.45      62.50  – 1
Total values   :    10
Highest-value:    68.150002
Lowest-value :    25.000000
Range          :    43.150002
Average        :    41.849998
```

Fig. 6.14 *Calculation of range of values*

The values are read in an input loop created by the **goto input**; statement. The control is transferred out of the loop by inputting a negative number. This is caused by the statement

if (value < 0) goto output;

2. Pay-Bill Calculations

Problem: A manufacturing company has classified its executives into four levels for the benefit of certain perks. The levels and corresponding perks are shown as follows:

Level	Perks	
	Conveyance allowance	Entertainment allowance
1	1000	500
2	750	200
3	500	100
4	250	

An executive's gross salary includes basic pay, house rent allowance at 25% of basic pay and other perks. Income tax is withheld from the salary on a percentage basis as follows:

Gross salary	Tax rate
Gross <= 2000	No tax deduction
2000 < Gross <= 4000	3%
4000 < Gross <= 5000	5%
Gross > 5000	8%

Write a program that will read an executive's job number, level number, and basic pay and then compute the net salary after withholding income tax.

Problem analysis:
Gross salary = basic pay + house rent allowance + perks
Net salary = gross salary – income tax

The computation of perks depends on the level, while the income tax depends on the gross salary. The major steps are:

1. Read data.
2. Decide level number and calculate perks.
3. Calculate gross salary.
4. Calculate income tax.
5. Compute net salary.
6. Print the results.

Program: A program and the results of the test data are given in Fig. 6.15. Note that the last statement should be an executable statement. That is, the lable **stop**; cannot be the last line.

Program

```
/ ************************************************************************************ /
/*                          PAY-BILL CALCULATIONS                                     */
/ ************************************************************************************ /

#define   CA1   1000
#define   CA2   750
#define   CA3   500
#define   CA4   250
#define   EA1   500
#define   EA2   200
#define   EA3   100
#define   EA4   0
main()
```

```
{
        int     level, jobnumber;
        float   gross,
                basic,
                house_rent,
                perks,
                net,
                incometax;
        input:
        printf("\nEnter level, job number, and basic pay\n");
        printf(Enter 0 (zero) for level to END\n\n");
        scanf("%d", &level);
        if (level == 0) goto stop;
        scanf("%d %f", &jobnumber, &basic);
        switch (level)
        {
                case 1:
                        perks = CA1 + EA1;
                        break;
                case 2:
                        perks = CA2 + EA2;
                        break;
                case 3:
                        perks = CA3 + EA3;
                        break;
                case 4:
                        perks = CA4 + EA4;
                        break;
                default:
                        printf("Error in level code\n");
                        goto stop;
        }
        house_rent = 0.25 * basic;
        gross = basic + house_rent + perks;
        if (gross < = 2000)
            incometax = 0;
        else if (gross < = 4000)
                incometax = 0.03 * gross;
        else if (gross < = 5000)
                incometax = 0.05 * gross;
                else
                incometax = 0.08 * gross;
    net = gross – incometax;
    printf("%d %d %.2f\n", level, jobnumber, net);
    goto input;
    stop: printf("\n\nEND OF THE PROGRAM");
}
```

Output

```
Enter level, job number, and basic pay
Enter 0 (zero) for level to END

1  1111  4000
1  1111  5980.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
```

```
2 2222 3000
2 2222 4465.00

Enter level, job number, and basic pay
Enter 0 (zero) for level to END

3 3333 2000
3 3333 3007.00

Enter level, job number, and basic pay
Enter 0 (zero) for level to END

4 4444 1000
4 4444 1500.00

Enter level, job number, and basic pay
Enter 0 (zero) for level to END

0

END OF THE PROGRAM
```

Fig. 6.15 *Pay-bill calculations*

Review Questions and Exercises

6.1 Determine whether the following are true or false:
- (a) When **if** statements are nested, the last **else** gets associated with the nearest **if** without an **else.**
- (b) One **if** can have more than one **else** clause.
- (c) A **switch** statement can always be replaced by a series of **if**···**else** statements.
- (d) A **switch** expression can be of any type.
- (e) A program stops its execution when a **break** statement is encountered.

6.2 In what ways does a **switch** statement differ from an **if** statement?

6.3 Find errors, if any, in each of the following segments:

(a)
```
if (x + y = z && y > 0)
    printf("  ");
```
(b)
```
if (code > 1);
    a = b + c
else
    a = 0
```
(c)
```
if (p < 0) || (q < 0)
printf("sign is negative");
```

6.4 The following is a segment of a program:

```
x = 1;
y = 1;
if(n > 0)
    x = x + 1;
    y = y – 1;
printf("%d %d", x, y);
```

What will be the values of x and y if n assumes a value of (a) 1 and (b) 0.

6.5 Rewrite each of the following without using compound relations:

```
(a) if (grade <= 59 && grade >= 50)
        second = second + 1;
(b) if (number > 100 || number < 0)
        printf("Out of range");
        else
        sum = sum + number;
(c) if (M1 > 60 && M2 > 60) || T > 200)
        printf("Admitted\n");
    else
        printf("Not admitted\n");
```

6.6 Read all the programs discussed in this chapter. Identify any changes that might be necessary to improve either readability or efficiency of the programs.

6.7 Write a program to determine whether a number is 'odd' or 'even' and print the message

NUMBER IS EVEN
or
NUMBER IS ODD

(a) without using **else** option, and (b) with **else** option.

6.8 Write a program to find the number of and sum of all integers greater than 100 and less than 200 that are divisible by 7.

6.9 A set of two linear equations with two unknown x_1 and x_2 is given below:

$$ax_1 + bx_2 = m$$

$$cx_1 + dx_2 = n$$

The set has a unique solution

$$x_1 = \frac{md - bn}{ad - cb}$$

$$x_2 = \frac{na - mc}{ad - cb}$$

provided the denominator $ad - cb$ is not equal to zero.

Write a program that will read the values of constants a, b, c, d, m, and n and compute the values of x_1 and x_2. An appropriate message should be printed if $ad - cb = 0$.

6.10 Given a list of marks ranging from 0 to 100, write a program to compute and print the number of students:

(a) who have obtained more than 80 marks,
(b) who have obtained more than 60 marks,
(c) who have obtained more than 40 marks,
(d) who have obtained 40 or less marks,
(e) in the range 81 to 100,

(f) in the range 61 to 80,
(g) in the range 41 to 60, and
(h) in the range 0 to 40.

The program should use a minimum number of **if** statements.

6.11 Admission to a professional course is subject to the following conditions:

(a) Marks in mathematics > = 60
(b) Marks in physics > = 50
(c) Marks in chemistry > = 40
(d) Total in all three subjects > = 200

or

Total in mathematics and physics > = 150

Given the marks in the three subjects, write a program to process the applications to list the eligible candidates.

6.12 Write a program to print a two-dimensional Square Root Table as shown below, to provide the square root of any number from 0 to 9.9. For example, the value *x* will give the square root of 3.2 and *y* the square root of 3.9.

Square Root Table

Number	0.0	0.1	0.2	. . . . .	0.9
0.0 1.0 2.0					
3.0			x		y
9.0					

6.13 Shown below is a Floyd's triangle.

```
1
2  3
4  5  6
7  8  9  10
11 . .  . . . . 15
.
.
.
79   . . . . . . . . . . . . 91
```

(a) Write a program to print this triangle.
(b) Modify the program to produce the following form of Floyd's triangle.

```
1
0  1
1  0  1
0  1  0  1
1  0  1  0  1
```

6.14 A cloth showroom has announced the following seasonal discounts on purchase of items:

Purchase amount	Discount	
	Mill cloth	Handloom items
0 – 100	–	5%
101 – 200	5%	7.5%
201 – 300	7.5%	10.0%
Above 300	10.0%	15.0%

Write a program using **switch** and **if** statements to compute the net amount to be paid by a customer.

6.15 Write a program that will read the value of x and evaluate the following function

$$y = \begin{cases} 1 & \text{for } x > 0 \\ 0 & \text{for } x = 0 \\ -1 & \text{for } x < 0 \end{cases}$$

Using

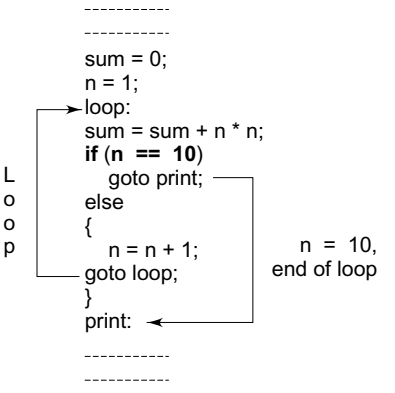
(a) nested **if** statements,
(b) **else if** statements, and
(c) conditional operator ? :

Chapter 7

Decision Making and Looping

7.1 INTRODUCTION

We have seen in the previous chapter that it is possible to execute a segment of a program repeatedly by introducing a counter and later testing it using the **if** statement. While this method is quite satisfactory for all practical purposes, we need to initialize and increment a counter and test its value at an appropriate place in the program for the completion of the loop. For example, suppose we want to calculate the sum of squares of all integers between 1 and 10. We can write a program using the **if** statement as follows:



This program does the following things:

1. Initializes the variable **n**.
2. Computes the square of **n** and adds it to **sum**.
3. Tests the value of **n** to see whether it is equal to 10 or not. If it is equal to 10, then the program prints the results.
4. If **n** is less than 10, then it is incremented by one and the control **goes back** to compute the **sum** again.

The program evaluates the statement

sum = sum + n*n;

10 times. That is, the loop is executed 10 times. This number can be decreased or increased easily by modifying the relational expression appropriately in the statement **if** (n == 10). On such occasions where the exact number of repetitions are known, there are more convenient methods of looping in C. These looping capabilities enable us to develop concise programs containing repetitive processes without the use of **goto** statements.

In looping, a sequence of statements are executed until some conditions for termination of the loop are satisfied. A *program loop* therefore consists of two segments, one known as the *body of the loop* and the other known as the *control statement*. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, a control structure may be classified either as the *entry-controlled loop* or as the *exit-controlled loop*. The flowcharts in Fig. 7.1 illustrate these structures. In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.

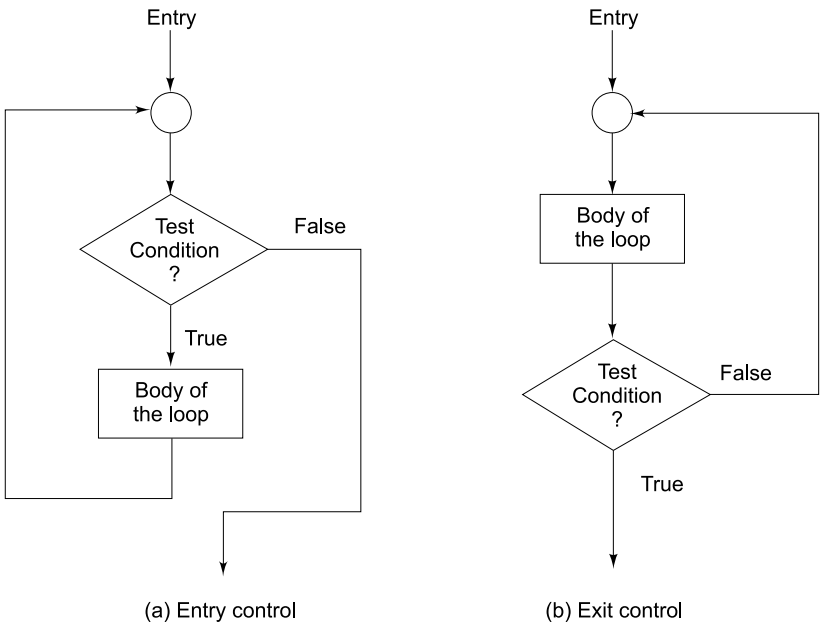


Fig. 7.1 *Loop control structures*

The test conditions should be carefully stated in order to perform the desired number of loop executions. It is assumed that the test condition will eventually transfer the control out of the loop. In case, due to some reason it does not do so, the control sets up an *infinite loop* and the body is executed over and over again.

A looping process, in general, would include the following four steps:

1. Setting and initialization of a counter.
2. Execution of the statements in the loop.

3. Test for a specified condition for execution of the loop.
4. Incrementing the counter.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.

The C language provides for three loop constructs for performing loop operations. They are:

1. The **while** statement.
2. The **do** statement.
3. The **for** statement.

We shall discuss the features and applications of each of these statements in this chapter.

7.2 THE WHILE STATEMENT

The simple888st of all the looping structures in C is the **while** statement. We have used **while** in many of our earlier programs. The basic format of the **while** statement is

```
while (test condition)
{
   body of the loop
}
```

The **while** is an *entry-controlled* loop statement. The *test-condition* is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes *false* and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.

We can rewrite the program loop discussed in Sec. 7.1 as follows:

```
.....
.....
sum = 0;
n = 1;
while(n <= 10)
{
   sum = sum + n * n;
   n = n+1;
}
printf("sum = %d\n", sum);
.....
```

The body of the loop is executed 10 times for n = 1, 2, ..., 10 each time adding the square of the value of n, which is incremented inside the loop. The test condition may also be written as **n < 11**; the result would be the same.

Another example of **while** statement which uses the keyboard input is shown below:

```
.....
.....
character = ' ';
while (character != 'Y')
        character = getchar();
xxxxxxx;
.....
.....
```

First the **character** is initialized to ′′. The **while** statement then begins by testing whether **character** is not equal to Y. Since the **character** was initialized to ′′, the test is true and the loop statement

character = getchar();

is executed. Each time a letter is keyed in, the test is carried out and the loop statement is executed until the letter Y is pressed. When Y is pressed, the condition becomes false because **character** equals Y, and the loop terminates, thus transferring the control to the statement **xxxxxxx**;.

Example 7.1

A program to evaluate the equation

$$y = x^n$$

when n is a non-negative integer, is given in Fig. 7.2.

The variable **y** is initialized to 1 and then multiplied by **x**, n times using the **while** loop. The loop control variable, count is initialized outside the loop and incremented inside the loop. When the value of **count** becomes greater than **n**, the control exits the loop.

7.3 THE DO STATEMENT

The **while** loop construct that we have discussed in the previous section makes a test of condition *before* the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the **do** statement. This takes the form:

```
do
{
body of the loop
}
while (test-condition);
```

On reaching the **do** statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the *test-condition* in the **while** statement is evaluated. If the condition is true, the program continues to evaluate the body of the *loop* once again. This process continues as long as the *condition* is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the **while** statement.

Program

```
/ ************************************************************************************ /
/*                          EXAMPLE OF while STATEMENT                                */
/ ************************************************************************************ /

main()
{
    int count, n;
    float x, y;
    printf("Enter the values of x and n:");
    scanf("%f %d", &x, &n);
    y = 1.0;
    count = 1;
    /* LOOP BEGINS */
    while (count <= n )
    {
            y = y*x;
            count++;
    }
    /* END OF LOOP */
    printf(" x = %f; n = %d; x to power n = %f n",x,n,y);
}
```

Output

```
Enter the values of x and n : 2.5  4
x = 2.500000; n = 4; x to power n = 39.062500

Enter the values of x and n : 0.5  4
x = 0.500000; n = 4; x to power n = 0.062500
```

Fig. 7.2 *Program to compute x to the power n using* ***while*** *loop*

Since the *test-condition* is evaluated at the bottom of the loop, the **do**···**while** construct provides an *exit-controlled* loop and therefore the body of the loop is *always executed at least once*.

A simple example of a **do**···**while** loop is:

```
.....
do
{
    printf("Input a number\n");
    number = getnum( );
}
while (number > 0);
.....
```

This segment of a program reads a number from the keyboard until a zero or a negative number is keyed in.

The test conditions may have compound relations as well. For instance, the statement

while (number > 0 && number < 100)

in the above example would cause the loop to be executed as long as the number keyed is lies between 0 and 100.

Consider another example:

```
.....
I = 1;
sum = 0;
do
{
   sum = sum + 1;
   I = I+2;
}
while(sum < 40 || < 10);
printf("%d %d\n", I, sum);
.....
```

The loop will be executed as long as one of the two relations is true.

Example 7.2

A program to print the multiplication table from 1×1 to 12×10 as shown below is given in Fig. 7.3.

```
1     2     3     4     .....     10
2     4     6     8     .....     20
3     6     9     12    .....     30
4                       .....     40
.                                 .
.                                 .
.                                 .
12                      .....     120
```

This program contains two **do**···**while** loops in nested form. The outer loop is controlled by the variable **row** and executed 12 times. The inner loop is controlled by the variable column and is executed 10 times, each time the outer loop is executed. That is, the inner loop is executed a total of 120 times, each time printing a value in the table.

Notice that the **printf** of the inner loop does not contain any new line character (\n). This allows the printing of all row values in one line. The empty **printf** in the outer loop initiates a new line to print the next row.

7.4 THE FOR STATEMENT

Simple 'for' Loops

The **for** loop is another *entry-controlled* loop that provides a more concise loop control structure. The general form of the **for** loop is

```
for (initialization ; test-condition ; increment)
{
   body of the loop
}
```

Program

```
/ ********************************************************************************** /
/*                    PRINTING OF MULTIPLICATION TABLE                              */
/ ********************************************************************************** /

#define COLMAX 10
#define ROWMAX 12

main()
{
    int row,column, y;

    row = 1;
    printf("                    MULTIPLICATION TABLE                    \n");
    printf("    ____________________________________________________ \n");

    /* ..... OUTER LOOP BEGINS ..... */
    do
    {
        column = 1;

        /* * INNER LOOP BEGINS ..... */
        do
        {
            y = row * column;
            printf("%4d", y);
            column = column + 1;
        }
            while (column <= COLMAX);
            /* * INNER LOOP ENDS ..... */
            printf("\n");
            row = row + 1;
    }

    while (row <= ROWMAX);
    /* .....  OUTER LOOP ENDS   ..... */
    printf (" ____________________________________________ \n");
}
```

Output

MULTIPLICATION TABLE

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100
11	22	33	44	55	66	77	88	99	110
12	24	36	48	60	72	84	96	108	120

Fig. 7.3 *Printing of a multiplication table using* ***do*** *⋯* ***while*** *loop*

The execution of the **for** statement is as follows:

1. *Initialization* of the *control variables* is done first, using assignment statements such as i = 1 and count = 0. The variables **i** and **count** are known as loop-control variables.

2. The value of the control variable is tested using the *test-condition*. The test-condition is a relational expression, such as i < 10 that determines when the loop will exit. If the condition is *true*, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now, the control variable is *incremented* using an assignment statement such as i = i+1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.
   Consider the following segment of a program;

```
for (x = 0; x <= 9; x = x+1)
{
   printf("%d", x);
}
printf("\n");
```

This **for** loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons. Note that there is no semicolon at the end of the *increment* section, x = x+1.

The **for** statement allows for *negative increments*. For example, the loop discussed above can be written as follows:

```
for (x = 9; x >= 0; x = x–1)
   printf("%d", x);
printf("\n");
```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Note that braces are optional when the body of the loop contains only one statement.

Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. For example,

```
for (x = 9; x < 9; x = x–1)
printf("%d", x);
```

will never be executed because the test condition fails at the very beginning itself.

Let us again consider the problem of sum of squares of integers discussed in Sec. 7.1. This problem can be coded using the for statement as follows:

```
.....
.....
sum = 0;
for (n = 1; n <= 10; n = n+1)
{
   sum = sum+ n*n;
}
printf("sum = %d\n", sum);
.....
.....
```

The body of the loop

```
sum = sum + n*n;
```

is executed 10 times for n = 1, 2, ·····, 10 each time incrementing the **sum** by the square of the value of **n**.

One of the important points about the **for** loop is that all the three actions, namely *initialization*, *testing*, and *incrementing*, are placed in the **for** statement itself, thus making them visible to the programmers and users, in one place. The **for** statement and its equivalent of **while** and do statements are shown in Table 7.1.

Table 7.1 Comparison of the Three Loops

for	while	do
for (n=1; n<=10; ++n) { .... }	n = 1; **while** (n<=10) { .... n = n+1; }	n = 1; **do** { .... n = n+1; } **while** (n<=10);

Example 7.3

The program in Fig. 7.4 uses a **for** loop to print the "Powers of 2" table for the power 0 to 20, both positive and negative.

The program evaluates the value

$$p = 2^n$$

successively by multiplying 2 by itself n times.

$$q = 2^{-n} = \frac{1}{p}$$

Note that we have declared **p** as a **long int** and **q** as a **double**.

Additional Features of for Loop

The **for** loop in C has several capabilities that are not found in other constructs. For example, more than one variable can be initialized at a time in the **for** statement. The statements

```
p = 1;
for (n=0; n<17; ++n)
```

can be rewritten as

Program

```
/ ************************************************************************************ /
/ *                                USE OF for LOOP                                   */
/ ************************************************************************************ /

main()
{
    long int    p;
    int         n;
    double      q;

    printf(" ------------------------------------------------------------------------------------------ \n");
    printf("  2 to power n                    n                    2 to power – n\n");
    printf("------------------------------------------------------------------------------------------ \n");
    p = 1;

    for (n = 0; n < 21 ; ++n) /* LOOP BEGINS */
    {
        if (n == 0)
          p = 1;
        else
          p = p * 2;
        q = 1.0/(double)p ;
        printf("% 10ld % 10d % 20.12lf\n", p, n, q);
}                                                          /* LOOP ENDS   */
printf(" ------------------------------------------------------------------------------------------ \n");
}
```

Output

```
------------------------------------------------------------------------------
 2 to power n                 n                2 to power -n
------------------------------------------------------------------------------
            1                 0          1.000000000000
            2                 1          0.500000000000
            4                 2          0.250000000000
            8                 3          0.125000000000
           16                 4          0.062500000000
           32                 5          0.031250000000
           64                 6          0.015625000000
          128                 7          0.007812500000
          256                 8          0.003906250000
          512                 9          0.001953125000
         1024                10          0.000976562500
         2048                11          0.000488281250
         4096                12          0.000244140625
         8192                13          0.000122070313
        16384                14          0.000061035156
        32768                15          0.000030517578
        65536                16          0.000015258789
       131072                17          0.000007629395
       262144                18          0.000003814697
       524288                19          0.000001907349
      1048576                20          0.000000953674
------------------------------------------------------------------------------
```

Fig. 7.4 *Program to print 'Power of 2' table using* ***for*** *loop*

Notice that the initialization section has two parts p = 1 and n = 1 separated by a *comma*.

Like the initialization section, the increment section may also have more than one part. For example, the loop

```
for (n=1, m=50; n<=m; n=n+1, m=m–1)
{
    p = m/n;
    printf("%d %d %d\n", n, m, p);
}
```

is perfectly valid. The multiple arguments in the increment section are separated by *commas*.

The third feature is that the test-condition may have any compound relation and the testing need not be limited only to the loop control variable. Consider the example below:

```
sum = 0;
for (i = 1; i < 20 && sum < 100; ++i)
{
    sum = sum+i;
    printf("%d %d\n", sum);
}
```

The loop uses a compound test condition with the control variable **i** and external variable **sum**. The loop is executed as long as both the conditions i < 20 and sum < 100 are true. The **sum** is evaluated inside the loop.

It is also permissible to use expressions in the assignment statements of initialization and increment sections. For example, a statement of the type

```
for (x = (m+n)/2; x > 0; x = x/2)
```

is perfectly valid.

Another unique aspect of **for** loop is that one or more sections can be omitted, if necessary. Consider the following statements:

```
.....
.....
m = 5;
for ( ; m != 100 ;)
{
    printf("%d\n", m);
    m = m+5;
}
.....
.....
```

Both the initialization and increment sections are omitted in the **for** statement. The initialization has been done before the for statement and the control variable is incremented inside the loop. In such cases, the sections are left blank. However, the semicolons separating the sections must remain. If the test-condition is not present, the **for** statement sets up and infinite loop. Such loops can be broken using **break** or **goto** statements in the loop.

We can set up *time delay loops* using the *null* statement as follows:

```
for (j = 1000; j > 0; j = j – 1)
;
```

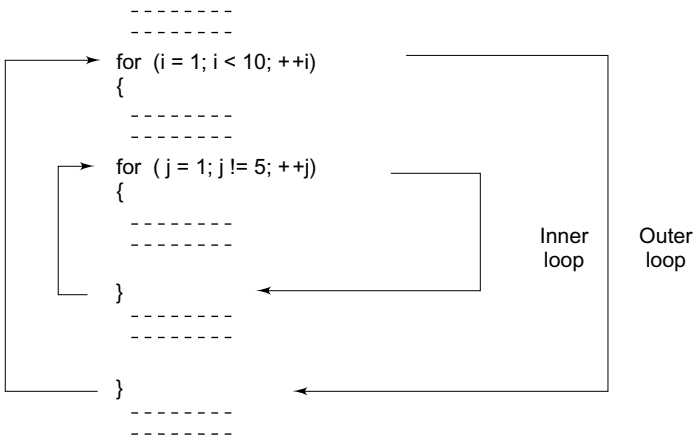
This loop is executed 1000 times without producing any output; it simple causes a time delay. Notice that the body of the loop contains only a semicolon, known as a *null* statement. This can also be written as

```
for (j=1000; j > 0; j = j–1);
```

This implies that the C compiler will not give an error message if we place a semicolon by mistake at the end of a **for** statement. The semicolon will be considered as a *null statement* and the program may produce some nonsense.

Nesting of for Loops

Nesting of loops, that is, one **for** statement within another **for** statement, is allowed in C. For example, two loops can be nested as follows:



The nesting may continue upto 15 levels in ANSI C; many compilers allow more. The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each **for** statement.

The program to print the multiplication table discussed in Example 7.2 can be written more concisely using nested for statements as follows:

```
.....
.....
for (row = 1; row <= ROWMAX; ++row)
{
   for (column = 1; column <= COLMAX; ++column)
   {
      y = row * column;
      printf("%4d", y);
   }
   print("\n");
}
.....
.....
```

The outer loop controls the rows while the inner loop controls the columns.

Example 7.4

A class of n students take an annual examination in m subjects. A program to read the marks obtained by each student in various subjects and to compute and print the total marks obtained by each of them is given in Fig. 7.5.

Program

```
/ ********************************************************************************** /
/ *                    ILLUSTRATION ON NESTED LOOPS                                */
/ ********************************************************************************** /

#define FIRST       360
#define SECOND    240

main()
{
    int n, m, i, j,
        roll–number, marks, total;
    printf("Enter number of students and subjects\n");
    scanf("%d %d", &n, &m);
    printf("\n");
    for (i = 1; i <= n; ++i)
    {
        printf("Enter roll-number: ");
        scanf("%d", &roll-number);
        total = 0 ;
        printf("\nEnter marks of %d subjects for ROLL NO %d
                \n",m,roll-number);
        for (j = 1; j <= m; j+ +)
        {
            scanf("%d", &marks);
            total = total + marks;
        }
        printf("TOTAL MARKS = %d", total);
        if (total >= FIRST)
            printf("(First Division) \n\n");
        else if (total >= SECOND)
            printf("(Second Division) \n\n");
        else
            printf("( *** F A I L ***) \n=n");
    }
}
```

Output

```
Enter number of students and subjects
3   6

Enter roll-number : 8701
Enter marks of 6 subjects for ROLL NO 8701
81  75  83  45  61  59

TOTAL MARKS =   404 ( First Division )
Enter roll-number : 8702

Enter marks of 6 subjects for ROLL NO 8702
51  49  55  47  65  41
TOTAL MARKS =   308 ( Second Division )

Enter roll-number : 8704

Enter marks of 6 subjects for ROLL NO 8704
40  19  31  47  39  25
TOTAL MARKS = 201 ( *** F A I L ***)
```

Fig. 7.5 *Illustration of nested* ***for*** *loops*

The program uses two for loops, one for controlling the number of students and the other for controlling the number of subjects. Since both the number of students and the number of subjects are requested by the program, the program may be used for a class of any size and any number of subjects.

The outer loop includes three parts: (1) reading of roll-numbers of students, one after another, (2) inner loop, where the marks are read and totalled for each student, and (3) printing of total marks and declaration of grades.

7.5 JUMPS IN LOOPS

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test-condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs. For example, consider the case of searching for a particular name in a list containing, say, 100 names. A program loop written for reading and testing the names a 100 times must be terminated as soon as the desired name is found. C permits a jump from one statement to another within a loop as well as a jump out of a loop.

Jumping Out of a Loop

An early exit from a loop can be accomplished by using the **break** statement or the **goto** statement. We have already seen the use of the **break** in the **switch** statement and the **goto** in the **if** ⋯ **else** construct. These statements can also be used within **while**, **do**, or **for** loops. They are illustrated in Fig. 7.6 and Fig. 7.7.

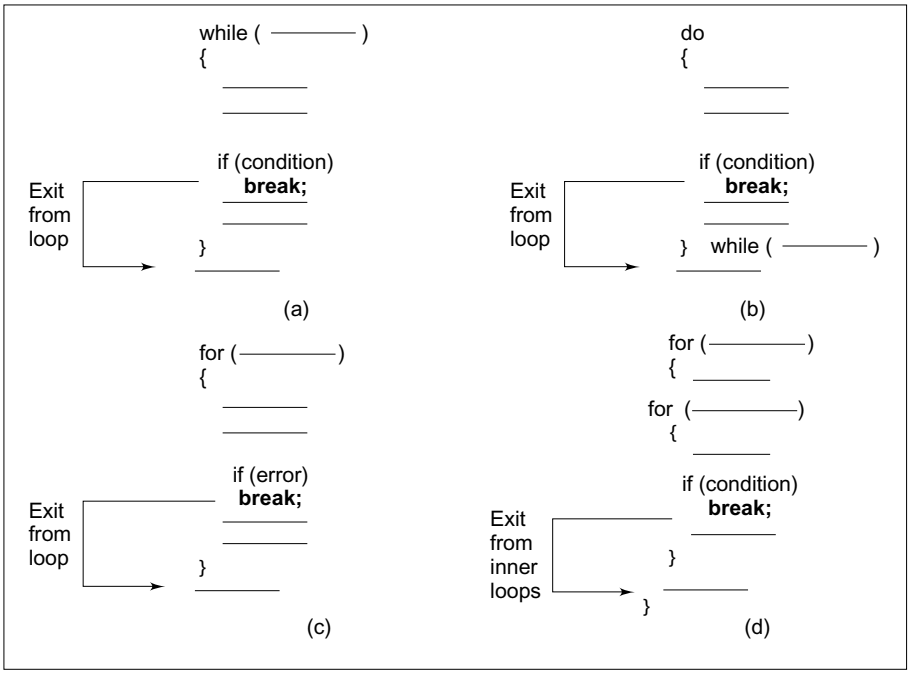


Fig. 7.6 *Exiting a loop with* ***break*** *statement*

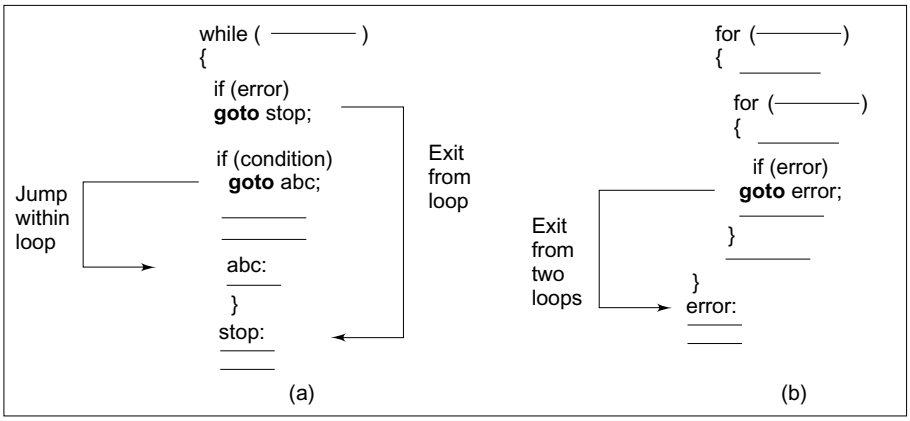


Fig. 7.7 *Jumping within and exiting from the loops with* ***goto*** *statement*

When the **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the **break** would only exit from the loop containing it. That is, the **break** will exit only a single loop.

Since a **goto** statement can transfer the control to any place in a program, it is useful to provide branching within a loop. Another important use of **goto** is to exit from deeply nested loops when an error occurs. A simple **break** statement would not work here.

Example 7.5

The program in Fig. 7.8 illustrates the use of the **break** statement in a C program.

The program reads a list of positive values and calculates their average. The **for** loop is written to read 1000 values. However, if we want the program to calculate the average of any set of values less than 1000, then we must enter a negative number after the last value in the list, to mark the end of input.

Each value, when it is read, is tested to see whether it is a positive number or not. If it is positive, the value is added to the **sum**; otherwise, the loop terminates. On exit, the average of the values read is calculated and the results are printed out.

Program

```
/ ************************************************************************************ /
/*                          USE OF break IN A PROGRAM                                  */
/ ************************************************************************************ /
main()
{
    int m;
    float x, sum, average;

    printf("This program computes the average of a
                    set of numbers\n");
    printf("Enter values one after another\n");
    printf("Enter a NEGATIVE number at the end.\n\n");
    sum = 0;
    for (m = 1 ; m <= 1000; ++m)
    {
        scanf("%f", &x);
        if (x < 0)
```

```
            break;
        sum += x ;
        }
    average = sum/(float)(m–1);
    printf("\n");
    printf("Number of values = %d\n", m–1);
    printf("Sum              = %f\n", sum);
    printf("Average          = %f\n", average);
}
```

Output

```
This program computes the average of a set of numbers
Enter values one after another
Enter a NEGATIVE number at the end.
21    23    24    22    26    22    –1
Number of values   = 6
Sum                = 138.000000
Average            = 23.000000
```

Fig. 7.8 *Use of **break** in a program*

Example 7.6

A program to evaluate the series

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + ... + x^n$$

for $-1 < x < 1$ to evaluate to 0.01 per cent accuracy is given in Fig. 7.9. The **goto** statement is used to exit the loop on achieving the desired accuracy.

We have used the **for** statement to perform the repeated addition of each of the terms in the series. Since it is an infinite series, the evaluation of the function is terminated when the term $\mathbf{x}^n$ reaches the desired accuracy. The value of n that decides the number of loops operations is not known and therefore we have decided arbitrarily a value of 100, which may or may not result in the desired level of accuracy.

Program

```
/ ********************************************************************************** /
/*                    EXAMPLE OF exit WITH goto STATEMENT                          */
/ **********************************************************************************/

#define     LOOP          100
#define     ACCURACY      0.0001
main()
{
    int n;
    float x, term, sum;

    printf("Input value of x : ");
    scanf("%f", &x);

    sum = 0;
    for (term = 1, n = 1; n <= LOOP ; ++n)
    {
        sum += term ;
        if (term <= ACCURACY)
            goto output; /* EXIT FROM THE LOOP */
        term *= x ;
    }
```

```
        printf("\nFINAL VALUE OF N IS NOT SUFFICIENT\n");
        printf("TO ACHIEVE DESIRED ACCURACY\n");
        goto end;
        output:
        printf("\nEXIT FROM LOOP\n");
        printf("Sum = %f; No. of terms = %d\n", sum, n);
        end:
        ;
}
```

Output

```
Input value of x : .21
EXIT FROM LOOP
Sum = 1.265800; No. of terms = 7

Input value of x : .75
EXIT FROM LOOP
Sum = 3.999774; No. of terms = 34

Input value of x : .99
FINAL VALUE OF N IS NOT SUFFICIENT
TO ACHIEVE DESIRED ACCURACY
```

Fig. 7.9 *Use of **goto** to exit from a loop*

The test of accuracy is made using an **if** statement and the **goto** statement exits the loop as soon as the accuracy condition is satisfied. If the number of loop repetitions is not large enough to produce the desired accuracy, the program prints an appropriate message.

Note that the **break** statement is not very convenient to use here. Both the normal exit and the **break** exit will transfer the control to the same statement that appears next to the loop. But, in the present problem, the *normal exit* prints the message

"FINAL VALUE OF N IS NOT SUFFICIENT
TO ACHIEVE DESIRED ACCURACY"

and the *forced exit* prints the results of evaluation. Notice the use of a *null* statement at the end. This is necessary because a program should not end with a label.

Skipping a Part of a Loop—The Continue Statement

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. For example, in processing of applications for some job, we might like to exclude the processing of data of applicants belonging to a certain category. On reading the category code of an applicant, a test is made to see whether his application should be considered or not. If it is not to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operation.

Like the **break** statement, C supports another similar statement called the **continue** statement. However, unlike the **break** which causes the loop to be terminated, the **continue**, as the name implies, causes the loop to be *continued* with the *next iteration* after *skipping* any statements in between. The **continue** statement tells the compiler, "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the **continue** statement is simply

```
continue;
```

The use of the **continue** statement in loops is illustrated in Fig. 7.10. In **while** and **do** loops, **continue** causes the control to go directly to the *test-condition* and then to continue the iteration process. In the case of **for** loop, the *increment* section of the loop is executed before the *test-condition* is evaluated.

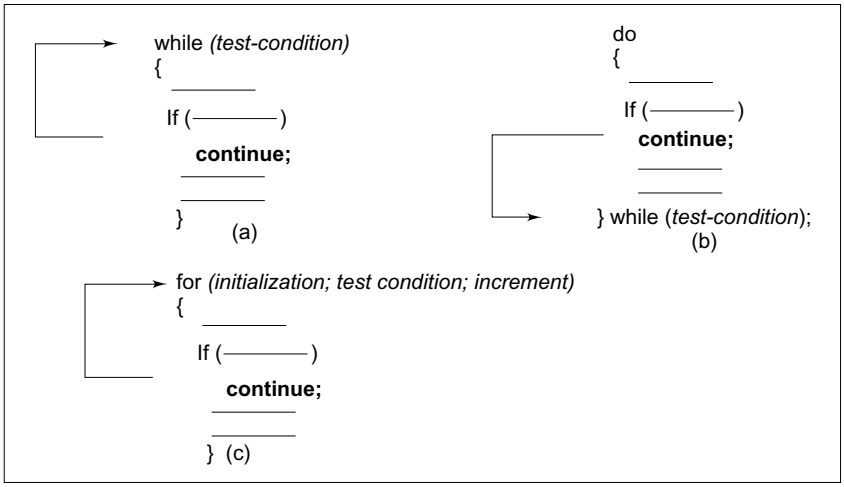


Fig. 7.10 *Bypassing and continuing in loops*

Example 7.7

The program in Fig. 7.11 illustrates the use of **continue** statement.

The program evaluates the square root of a series of numbers and prints the results. The process stops when the number 9999 is types in.

In case, the series contains any negative numbers, the process of evaluation of square root should be bypassed for such numbers because the square root of a negative number is not defined. The **continue** statement is used to achieve this. The program also prints a message saying that the number is negative and keeps an account of negative numbers.

The final output includes the number of positive values evaluated and the number of negative items encountered.

Program

```
/ ********************************************************************************** /
/*                          USE OF continue STATEMENT                              */
/ ********************************************************************************** /

#include <math.h>

main()
{
    int count, negative;
    double number, sqroot;
    printf("Enter 9999 to STOP\n");
    count = 0;
    negative = 0;
    while (count <= 100)
```

```
    {
        printf("Enter a number : ");
        scanf("%lf", &number);
        if   (number    == 9999)
            break;          /* EXIT FROM THE LOOP */
        if (number     <0)
        {
            printf("number is negative\n\n");
            negative++;
            continue;       /*   SKIP REST OF THE LOOP */
        }
        sqroot = sqrt(number);
        printf("Number    =   %lf\nSquare root = %lf\n=n",
                                number, sqroot);
        count++;
    }
    printf("Number of items done    =    %d\n", count);
    printf("\n\nNegative items                  = %d\n", negative);
    printf("END OF DATA\n");
}
```

Output

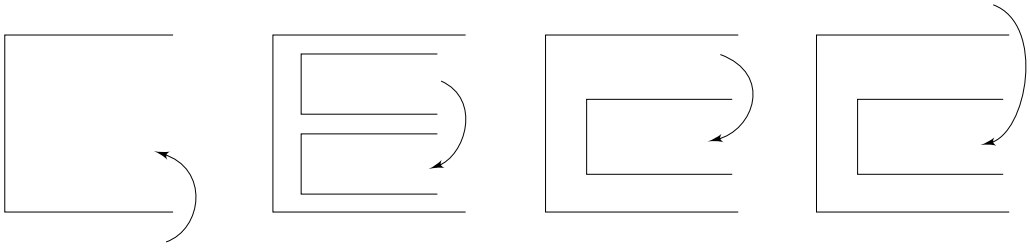
```
Enter 9999 to STOP
Enter a number : 25.0

Number        = 25.000000
Square root   = 5.000000

Enter a number   : 40.5
Number        = 40.500000
Square root   = 6.363961

Enter a number : –9
Number is negative

enter a number : 16
Number        = 16.000000
Square root   = 4.000000

Enter a number : –14.75
Number is negative

Enter a number : 80
Number        = 80.000000

Square root   = 8.944272

Enter a number : 9999

Number of items done   = 4
Negative items         = 2
END OF DATA
```

Fig. 7.11 *Use of **continue** statement*

Avoiding goto

As mentioned earlier, it is a good practice to avoid using **gotos**. There are many reasons for this. When **goto** is used, many compilers generate a less efficient code. In addition, using many of them makes a

program logic complicated and renders the program unreadable. It is possible to avoid using **goto** by careful program design. In case any **goto** is absolutely necessary, it should be documented. The following **goto** jumps would cause problems and therefore must be avoided:



Concise Test Expressions

We often use test expressions in the if, for, while and do statements that are evaluated and compared with zero for making branching decisions. Since every integer expression has a true/false value, we need not make explicit comparisons with zero. For instance, the expression **x** is true whenever **x** is not zero, and false when **x** is zero. Applying **!** operator, we can write concise test expressions without using any rational operators.

```
if (expression == 0)
```

is equivalent to

```
if (! expression)
```

Similarly,

```
if (expression ! = 0)
```

is equivalent to

```
if (expression)
```

Example
If (m%5 == 0 && n%5 == 0) is same as **if (! (m%5) &&! (n%5))**

7.6 STRUCTURED PROGRAMMING

Structured programming is an approach to the design and development of programs. It is a discipline of making a program's logic easy to understand by using only the basic three control structures:

- Sequence (straight line) structure
- Selection (branching) structure
- Repetition (looping) structure

While sequence and loop structures are sufficient to meet all the requirements of programming, the selection structure proves to be more convenient in some situations.

The use of structured programming techniques helps ensure well-designed programs that are easier to write, read, debug and maintain compared to those that are unstructured.

Structured programming discourages the implementation of unconditional branching using jump statements such as **goto**, **break** and **continue**. In its purest form, structured programming is synonymous with *"goto less programming"*.

Do not go to **goto** statement!

CASE STUDIES

1. Table of Binomial Coefficients

Problem: Binomial coefficients are used in the study of binomial distributions and reliability of multicomponent redundant systems. It is given by

$$B(m, x) = \left(\frac{m}{x}\right) = \frac{m!}{x!\,(m-x)!},\ m >= x$$

A table of binomial coefficients is required to determine the binomial coefficient for any set of m and x.

Problem Analysis: The binomial coefficient can be recursively calculated as follows:

$$B(m, o) = 1$$

$$B(m, x) = B(m, x-1)\left[\frac{m-x+1}{x}\right],\ x = 1, 2, 3, \cdots, m$$

Further,

$$B(o, o) = 1$$

That is, the binomial coefficient is one when either x is zero or m is zero. The program in Fig. 7.12 prints the table of binomial coefficients for $m = 10$. The program employs one **do** loop and one **while** loop.

Program

```
/ ********************************************************************************/
/*              EVALUATION OF BINOMIAL COEFFICIENTS                             */
/ ********************************************************************************/

#define      MAX      10
main()
{
    int  m,  x,  binom;
    printf("m x");
    for (m = 0; m <= 10; ++m)
    printf("%4d", m);
    printf("\n  ------------------------------------------------------------------------------\
                ----------\n");
    m = 0;
    do
    {
        printf("%2d ",m);
        x = 0; binom = 1;
```

```
            while (x <= m)
            {
                if(m == 0 || x == 0)
                    printf("%4d", binom);
                else
                    {
                        binom = binom * (m – x + 1)/x;
                        printf("%4d", binom);
                    }
                x = x + 1;
            }
            print("\n");
            m = m + 1;
        }
        while (m <= MAX);
        printf("----------------------------------------------------------------------------------
                ---------- \n");
}
```

Output

```
mx    0    1    2    3    4    5    6    7    8    9    10
-------------------------------------------------------------------------------------------------
0     1
1     1    1
  2     1    2    1
  3     1    3    3    1
  4     1    4    6    4    1
  5     1    5   10   10    5    1
  6     1    6   15   20   15    6    1
  7     1    7   21   35   35   21    7    1
  8     1    8   28   56   70   56   28    8    1
  9     1    9   36   84  126  126   84   36    9    1
 10     1   10   45  120  210  252  210  120   45   10    1
```

Fig. 7.12 *Program to print binomial coefficient table*

2. Histogram

Problem: In an organization, the employees are grouped according to their basic pay for the purpose of certain perks. The pay-range and the number of employees in each group are as follows:

Group	*Pay-Range*	*Number of Employees*
1	750 – 1500	12
2	1501 – 3000	23
3	3001 – 4500	35
4	4501 – 6000	20
5	above 6000	11

Draw a histogram to highlight the group sizes.

Problem analysis: Given the size of groups, it is required to draw bars representing the sizes of various groups. For each bar, its group number and size are to be written.

Program in Fig. 7.13 reads the number of employees belonging to each group and draws a histogram. The program uses four **for** loops and two **if** ⋯ **else** statements.

Program

```
/ ************************************************************************** /
/*                        PROGRAM TO DRAW HISTOGRAM                         */
/ ************************************************************************** /
#define        N          5
main( )
{
    int value[N];
    int i, j, n, x;
    for (n=0, n < N; ++n)
    {
        printf("Enter employees in Group – %d :",n+ 1);
        scanf("%d", &x);
        value[n] = x;
        printf("%d\n", value[n]);
    }
    printf("\n");
    printf("          |\n");
    for (n = 0; n < N: ++n)
    {
        for(i = 1; i <= 3; i++)
        {
            if (i == 2)
                printf("Group-% 1d   |",n+1);
            else
                printf("  |");
            for (j = 1 ; j <= value[n]; ++j)
                printf("*");
            if (i == 2)
                printf("(%d)\n", value[n]);
            else
                printf("\n");
            }
            printf("  |\n");
    }
}
```

Output

```
Enter employees in Group-1 : 12
12
Enter employees in Group-2 : 23
23
Enter employees in Group-3 : 35
35
Enter employees in Group-4 : 20
20
Enter Employees in Group-5 : 11
11
            |  ************
Group 1     |  ************(12)
            |  ************
            |
            |  ************************
Group 2     |  ************************(23)
            |  ************************
```

```
Group 3 | ***********************************
        | ***********************************(35)
        | ***********************************
        |
        | ********************
Group 4 | ********************(20)
        | ********************
        |
        | ***********
Group 5 | ***********(11)
        | ***********
```

Fig. 7.13 *Program to draw a histogram*

3. Minimum Cost

Problem: The cost of operation of a unit consists of two components C_1 and C_2 which can be expressed as functions of a parameter p as follows:

$$C_1 = 30 - 8p$$
$$C_2 = 10 + p^2$$

The parameter p ranges from 0 to 10. Determine the value of p with an accuracy of $\pm$ 0.1 where the cost of operation would be minimum.

Problem Analysis:

$$\text{Total cost} = C_1 + C_2 = 40 - 8p + p^2$$

The cost is 40 when $p = 0$, and 33 when $p = 1$ and 60 when $p = 10$. The cost, therefore, decreases first and then increases. The program in Fig. 7.14 evaluates the cost at successive intervals of p (in steps of 0.1) and stops when the cost begins to increase. The program employs **break** and **continue** statements to exit the loop.

Program

```
/ ********************************************************************************** /
/*                          PROGRAM OF MINIMUM COST                                 */
/ ********************************************************************************** /
main()
{
    float p, cost, p1, cost1;
    for (p = 0; p <= 10; p = p + 0.1)
    {
        cost = 40 – 8 * p + p * p;
        if(p == 0)
        {
          cost1 = cost;
          continue;
        }
        if(cost >= cost1)
          break;
        cost1 = cost;
        p1 = p;
```

```
    }
    p = (p + p1)/2.0;
    cost = 40 – 8 * p + p * p;
    printf("\nMINIMUM,COST = %.2f AT p = %. 1f\n", cost, p);
}
Output
MINIMUM COST = 24.00   AT p = 4.0
```

Fig. 7.14 *Program of minimum cost problem*

4. Plotting of Two Functions

Problem: We have two functions of the type

$$y_1 = \exp(-ax)$$
$$y_2 = \exp(-ax^2/2)$$

Plot the graphs of these functions for x varying from 0 to 5.0.

Problem analysis: Initially when $x = 0$, $y_1 = y_2 = 1$ and the graphs start from the same point. The curves cross when they are again equal at $x = 2.0$. The program should have appropriate branch statements to print the graph points at the following three conditions:

1. $y1 > y2$
2. $y1 < y2$
3. $y1 = y2$

The functions $y1$ and $y2$ are normalized and converted to integers as follows:

$$y1 = 50 \exp(-ax) + 0.5$$
$$y2 = 50 \exp(-ax^2/2) + 0.5$$

The program in Fig. 7.15 plots these two functions simultaneously. (0 for $y1$, * for $y2$, and # for the common point).

```
Program
/ ********************************************************************************** /
/*                        PLOTTING OF TWO FUNCTIONS                                 */
/ ********************************************************************************** /
#include   <math.h>

main()
{
    int  i;
    float a, x, y1, y2;

    a = 0.4;
    printf("                                          Y ——————————————> \n");
    printf("     0  ————————————————————————————————————————————————\
            ——————————————————— \n");

    for (x = 0; x < 5; x = x+0.25)
    {    /* BEGINNING OF FOR LOOP */
    /*.....................Evaluation of functions..... */
        y1 = (int) (50 * exp( –a * x ) + 0.5 );
        y2 = (int) (50 * exp( –a * x * x/2 ) + 0.5 );
```

```
        /* ..................Plotting when y1 = y2..................... */
            if (y1 == y2)
            {
            if (x == 2.5)
              printf("    X|");
            else
              printf("     |");
            for (i = 1; <= y1 – 1; + +i)
              printf(" ");
              printf("#\n");
            continue;
            }
        /* ..................Plotting when y1 > y2..................... */
            if (y1 > y2)
            {
            if (x == 2.5)
              printf("    X|");
            else
              printf("    |");
            for (i = 1; i <= y2 – 1; + +i)
              printf(" ");
            printf("*");
            for (i = 1; i <= (y1 – y2 – 1); + +i)
              printf("–");
            printf("0\n");
            continue;
            }
        /* ..................Plotting when y2 > y1..................... */
            if (x == 2.5)
              printf("    X|");
            else
              printf("     |");
            for (i = 1; i <= (y1 – 1); ++i)
              printf(" ");
            printf("0");
            for (i = 1; i <= (y2 – y1 – 1); + +i)
              printf("–");
            printf("*\n");
        } /* ..................END OF FOR LOOP..................... */
            printf(" |\n");
}
```

Output

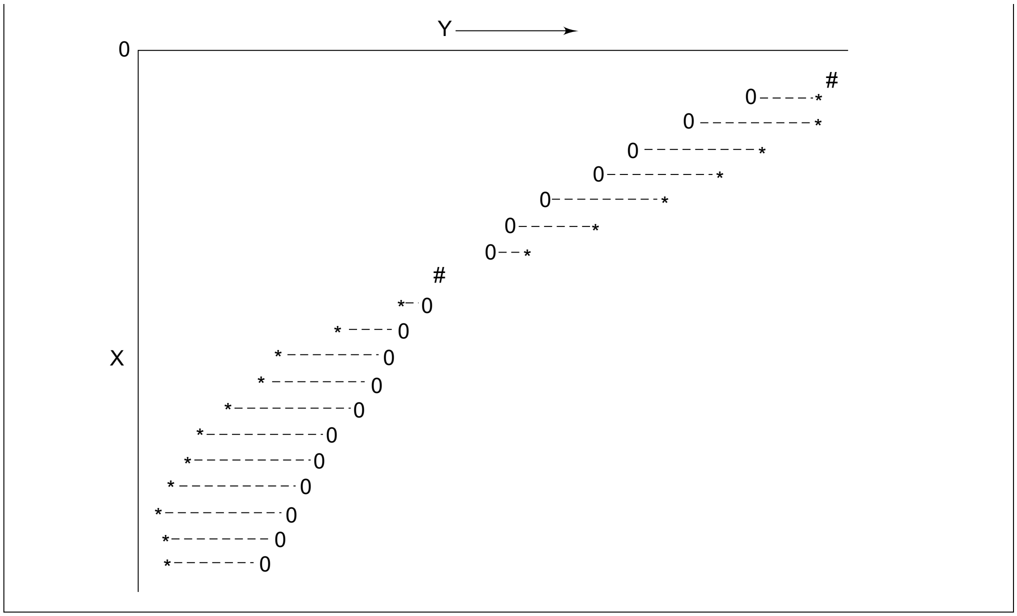


Fig. 7.15 *Plotting of two functions*

Review Questions and Exercises

7.1 Compare in terms of their functions, the following pairs of statements:
   (a) while and do ⋯ while.
   (b) while and for.
   (c) break and goto.
   (d) break and continue.
   (e) continue and goto.

7.2 Analyze each of the program segments that follows and determine how many times the body of each loop will be executed.

(a)
```
x = 5;
y = 50;
while (x <= y)
{
    x = y/x;
    ______
    ______
}
```

(b)
```
m = 1;
do
{
    ______
    ______
```

```
        m =  m+2;
    }
    while (m < 10);
(c) int i;
    for (i = 0; i <= 5; i = i+2/3)
    {
        _______
        _______
        _______

    }
(d) int m =  10;
    int n  =  7;
    while (m % n  > = 0)
    {
    _______

    m =  m + 1;
    n  =  n + 2;

    _______
}
```

7.3 Find errors, if any, in each of the following looping segments. Assume that all the variables have been declared and assigned values.

```
(a) while (count !=10);
    {
        count = 1;
        sum = sum + x;
        count = count + 1;
    }
(b) name = 0;
    do { name = name + 1;
    printf("My name is John\n");}
    while (name = 1)
(c) do:
    total = total + value;
    scanf("%f", &value);
    while(value != 999);
(d) for (x = 1, x > 10; x = x + 1)
    {
        _________
        _________
        _________
    }
(e) m = 1
    n = 0
    for (m + n < 19; ++n);
    printf("Hello\n");
    m = m+10
```

(f)
```
for (p = 10; p > 0;)
    p = p – 1;
    printf("%f", p);
```

7.4 What is a null statement? Explain its usefulness.

7.5 Given a number, write a program using **while** loomp to reverse the digits of the number. For example, the number

12345

should be written as

54321

(*Hint:* Use modulus operator to extract the last digit and the integer division by 10 to get the *n* - 1 digit number from the *n* digit number.)

7.6 The factorial of an integer *m* is the product of consecutive integers from 1 to *m*. That is,

$$\text{factorial } m = m! = m \times (m-1) \times \cdots\cdots \times 1.$$

Write a program that computes and prints a table of factorials for any given *m*.

7.7 Write a program to compute the sum of the digits of a given integer number.

7.8 The numbers in the sequence

1 1 2 3 5 8 13 21 ……

are called Fibonacci numbers. Write a program using a **do**···**while** loop to calculate and print the first *m* Fibonacci numbers.

(*Hint*: After the first two numbers in the series, each number is the sum of the two preceding numbers.)

7.9 Rewrite the program of the Example 7.1 using the **for** statement.

7.10 Write a program to evaluate the following investment; equation

$$V = P\,(1 + r)^n$$

and print the tables which would give the value of *V* for various combination of the following values of *P*, *r*, and *n*.

P : 1000, 2000, 3000, ..., 10,000
r : 0.10, 0.11, 0.12, ..., 0.20
n : 1, 2, 3, ..., 10

(*Hint*: *P* is the principal amount and *V* is the value of money at the end of *n* years. The equation can be recursively written as

$$V = P(1 + r)$$

$$P = V$$

That is, the value of money at the end of first year becomes the principal amount for the next year and so on.)

7.11 Write programs to print the following outputs using for loops.

(a)
```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

(b)
```
* * * * *
  * * * *
    * * *
      * *
        *
```

(c)
```
        1
      2 2
    3 3 3
  4 4 4 4
5 5 5 5 5
```

7.12 Write a program to read the age of 100 persons and count the number of persons in the age group 50 to 60. Use **for** and **continue** statements.

7.13 Rewrite the program of case study 7.4 (plotting of two curves) using **else** ⋯ **if** constructs instead of **continue** statements.

7.14 Write a program to print a table of values of the function

$$y = \exp(-x)$$

for x varying from 0.0 to 10.0 in steps of 0.10. The table should appear as follows:

Table for $Y = EXP(-X)$

x	0.1	0.2	0.3	.....	0.9
0.0					
1.0					
2.0					
3.0					
.					
.					
.					
9.0					

7.15 Write a program that will read a positive integer and determine and print its binary equivalent. (*Hint:* The bits of the binary representation of an integer can be generated by repeatedly dividing the number and the successive quotients by 2 and saving the remainder, which is either 0 or 1, after each division.)

UNIT II

Chapter 8

Arrays

8.1 INTRODUCTION

An *array* is a group of related data items that share a common name. For instance, we can define an array name **salary** to represent a *set of salaries* of a group of employees. A particular value is indicated by writing a number called *index* number of *subscript* in brackets after the array name. For example,

salary[10]

represents the salary of the 10th employee. While the complete set of values is referred to as an array, the individual values are called *elements*. Arrays can be of any variable type.

The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs. For example, a loop with the subscript as the control variable can be used to read the entire array, perform calculations and, print out the results. In this chapter, we shall discuss how arrays can be defined and used in C.

8.2 ONE-DIMENSIONAL ARRAYS

A list of items can be given one variable name using only one subscript and such a variable is called a *single-subscripted variable* or a *one-dimensional array*. In mathematics, we often deal with variables that are single-subscripted. For instance, we use the equation.

$$A = \frac{\sum_{i=1}^{n} x_i}{n}$$

to calculate the average of n values of x.

Accessing and Storing Elements

The subscripted variable x_i refers to the ith element of x. In C, single-subscripted variable x_i can be expressed as

***x*[1], *x*[2], *x*[3] .....*x*[*n*]**

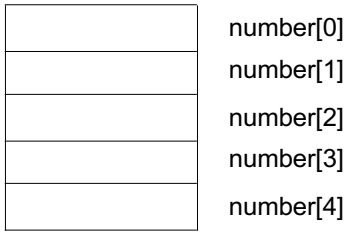
The subscript can begin with number 0. That is

x[0]

is allowed. For example, if we want to represent a set of five numbers, say (35, 40, 20, 57, 19), by an array variable **number**, then we may declare the variable **number** as follows

```
int number[5];
```

and the computer reserves five storage locations as shown below:



The values to the array elements can be assigned as follows:

```
number[0] = 35;
number[1] = 40;
number[2] = 20;
number[3] = 57;
number[4] = 19;
```

This would cause the array **number** to store the values as shown below:

number [0]	35
number [1]	40
number [2]	20
number [3]	57
number [4]	19

These elements may be used in programs just like any other C variable. For example, the following are valid statements:

```
a = number[0] + 10;
number[4] = number[0] + number [2];
number[2] = x[5] + y[10];
value[6] = number[i] * 3;
```

The subscript of an array can be integer constants, integer variables like i, or expressions that yield integers. *C performs no bounds checking and, therefore, care should be exercised to ensure that the array indices are within the declared limits.*

Declaration of Arrays

Like any other variable, arrays must be declared before they are used. The general form of array declaration is

type variable-name[size];

The *type* specifies the type of element that will be contained in the array, such as **int, float,** or **char** and the *size* indicates the maximum number of elements that can be stored inside the array. For example,

```
float height[50];
```

declares the **height** to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. Similarly,

```
int group[10];
```

declares the **group** as an array to contain a maximum of 10 integer constants. Remember, any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.

The C language treats character strings simply as arrays of characters. The *size* in a character string represents the maximum number of characters that the string can hold. For instance,

```
char name[10];
```

declares the **name** as a character array (string) variable that can hold a maximum of 10 characters. Suppose we read the following string constant into the string variable **name**.

"WELL DONE"

Each character of the string is treated as an element of the array **name** and is stored in the memory as follows:

'W'
'E'
'L'
'L'
' '
'D'
'O'
'N'
'E'
'\O'

When the compiler sees a character string, it terminates it with an additional null character. Thus, the element **name[9]** holds the null character '\0' at the end. When declaring character arrays, we must always allow one extra element space for the null terminator.

Example 8.1

Write a program using a single-subscripted variable to evaluate the following expressions:

$$\text{Total} = \sum_{i=1}^{10} x_i^2$$

The values of x_1, x_2, ... are read from the terminal.

Program in Fig. 8.1 uses a one-dimensional array **x** to read the values and compute the sum of their squares.

Initialization of Arrays

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

static *type array-name*[size] = { list of values };

The values in the list are separated by commas. For example, the statement

```
static int number[3] = { 0,0,0 };
```

will declare the variable **number** as an array of size 3 and will assign zero to each element. If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically. For instance,

```
static float total[5] = {0.0,15.75, – 10};
```

will initialize the first three elements to 0.0, 15.75, and –10.0 and the remaining two elements to zero.

Note that we have used the word 'static' before *type* declaration. This declares the variable as a *static variable*. More about static and other class of variables are discussed in Chapter 10.

Note: Under the old version (the K & R standard), automatic arrays cannot be initialized. Only external and static arrays may be initialized. However, the ANSI standard permits arrays with **auto** storage class to be initialized. That is, the keyword **static** could be omitted, when an ANSI compiler is used. Since ANSI compilers accept both the versions, we use the storage class **static** in our programs so that they can run under both the K & R and ANSI compilers.

The *size* may be omitted. In such cases, the compiler allocates enough space for all initialized elements. For example, the statement

```
static int counter[ ] = {1,1,1,1};
```

will declare the **counter** array to contain four elements with initial values 1. This approach works fine as long as we initialize every element in the array.

Character arrays may be initialized in a similar manner. Thus, the statement

```
static char name[ ] = {'J', 'o', 'h', 'n'};
```

declares the **name** to be an array of four characters, initialized with the string "John".

Initialization of arrays in C suffers two drawbacks.

1. There is no convenient way to initialize only selected elements.
2. There is no shortcut method for initializing a large number of array elements like the one available in FORTRAN.

Program

```
/ ********************************************************************************** /
/*                PROGRAM SHOWING ONE-DIMENSIONAL ARRAY                 */
/ ********************************************************************************** /

main( )
{
    int   i;
    float x[10], value, total;
/* ..... READING VALUES INTO ARRAY ..... */
    printf("ENTER 10 REAL NUMBERS\n");
    for(i = 0; i < 10; i++)
    {
        scanf("%f", &value) ;
        x[i] = value ;
    }
/* .....COMPUTATION OF TOTAL .....*/
    total = 0.0 ;
    for( i = 0 ; i < 10 ; i++ )
        total = total + x[i] * x[i];
/* .....PRINTING OF x[i] VALUES AND TOTAL ..... */
    printf("\n");
    for( i = 0 ; i < 10 ; i++ )
        printf("x[%2d] = %5.2f\n", i+1, x[i]) ;
    printf("\ntotal = %.2f\n", total) ;
}
```

Output

```
ENTER 10 REAL NUMBERS
1.1   2.2   3.3   4.4   5.5   6.6  7.7   8.8   9.9   10.10
x[ 1]   =   1.10
x[ 2]   =   2.20
x[ 3]   =   3.30
x[ 4]   =   4.40
x[ 5]   =   5.50
x[ 6]   =   6.60
x[ 7]   =   7.70
x[ 8]   =   8.80
x[ 9]   =   9.90
x[10]   =   10.10

Total   =   446.86
```

Fig. 8.1 *Program to illustrate one-dimensional array*

Consider an array of size, say 100. All the 100 elements have to be explicitly initialized. There is no way to specify a repeat count. In such situations, it would be better to use a **for** loop to initialize the elements. Consider the following segment of a C program:

```
----------
----------
for (i = 0; i < 100; i = i+1)
{
if (i < 50)
   sum[i] = 0.0;
else
```

```
        sum[i] = 1.0;
    }
    ----------
    ----------
```

The first 50 elements of the array **sum** are initialized to zero while the remaining 50 elements are initialized to 1.0.

Example 8.2

Given below is the list of marks obtained by a class of 50 students in an annual examination.

43	65	51	27	79	11	56	61	82	09
25	36	07	49	55	63	74	81	49	37
40	49	16	75	87	91	33	24	58	78
65	56	76	67	45	54	36	63	12	21
73	49	51	19	39	49	68	93	85	59

Write a program to count the number of students belonging to each of the following groups of marks: 0-9, 10-19, 20-29, ..., 100.

The program coded in Fig. 8.2 uses the array **group** containing 11 elements, one for each range of marks. Each element counts those values falling within the range of values it represents.

For any value, we can determine the correct group element by dividing the value by 10. For example, consider the value 59. The integer division of 59 by 10 yields 5. This is the element into which 59 is counted.

Note that the numbers with fractional part are rounded off to the nearest integer before the integer division occurs.

8.3 TWO-DIMENSIONAL ARRAYS

So far we have discussed the array variables that can store a list of values. There will be situations where a table of values will have to be stored. Consider the following data table, which shows the value of the sales of three items by four salesgirls:

	Item1	Item2	Item3
Salesgirl #1	310	275	365
Salesgirl #2	210	190	325
Salesgirl #3	405	235	240
Salesgirl #4	260	300	380

The table contains a total of 12 values, three in each line. We can think of this table as a matrix consisting of four *rows* and three *columns*. Each row represents the values of sales by a particular salesgirl and each column represents the values of sales of a particular item.

Program

```
/ ********************************************************************************** /
/*                    PROGRAM FOR FREQUENCY COUNTING                              */
/ ********************************************************************************** /

#define   MAXVAL    50
#define   COUNTER   11

main()
```

```
{
    float       value[MAXVAL];
    int         i, low, high;
    static int  group[COUNTER]  =  { 0,0,0,0,0,0,0,0,0,0};

/* ..... READING AND COUNTING ..... */
    for(i = 0; i < MAXVAL; i++)
    {
        /* ... READING OF VALUES ... */
        scanf("%f", &value[i]);
        /* ... COUNTING FREQUENCY OF GROUPS ... */
        ++ group[ (int) (value[i] + 0.5) / 10];
    }
/* ..... PRINTING OF FREQUENCY TABLE ..... */
    printf("\n");
    printf("GROUP   RANGE   FREQUENCY\n\n") ;
    for( i = 0 ; i < COUNTER ; i++ )
    {
        low = i * 10 ;
        if(i == 10)
          high = 100;
        else
          high = low + 9 ;
        printf("     %2d        %3d to %3d        %d\n",
                i+1, low, high, group[i] ) ;
    }
}
```

Output

```
43  65  51  27  79  11  56  61  82  09
25  36  07  49  55  63  74  81  49  37
40  49  16  75  87  91  33  24  58  78          (input data)
65  56  76  67  45  54  36  63  12  21
73  49  51  19  39  49  68  93  85  59
  GROUP     RANGE       FREQUENCY
     1        0 to   9          2
     2       10 to  19          4
     3       20 to  29          4
     4       30 to  39          5
     5       40 to  49          8
     6       50 to  59          8
     7       60 to  69          7
     8       70 to  79          6
     9       80 to  89          4
    10       90 to  99          2
    11      100 to 100          0
```

Fig. 8.2 *Program for frequency counting*

In mathematics, we represent a particular value in a matrix by using two subscripts such as v_{ij}. Here v denotes the entire matrix and v_{ij} refers to the value in the ith row and jth column. For example, in the above table v_{23} refers to the value 325.

C allows us to define such tables of items by using *two-dimensional* arrays. The table discussed above can be defined in C as

v[4][3]

Two-dimensional arrays are declared as follows:

type array_name [row_size][column_size];

Note that unlike most other languages, which use one pair of parentheses with commas to separate array sizes, C places each size in its own set of brackets.

Two dimensional arrays are stored in memory as shown in Fig. 8.3. As with the single-dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

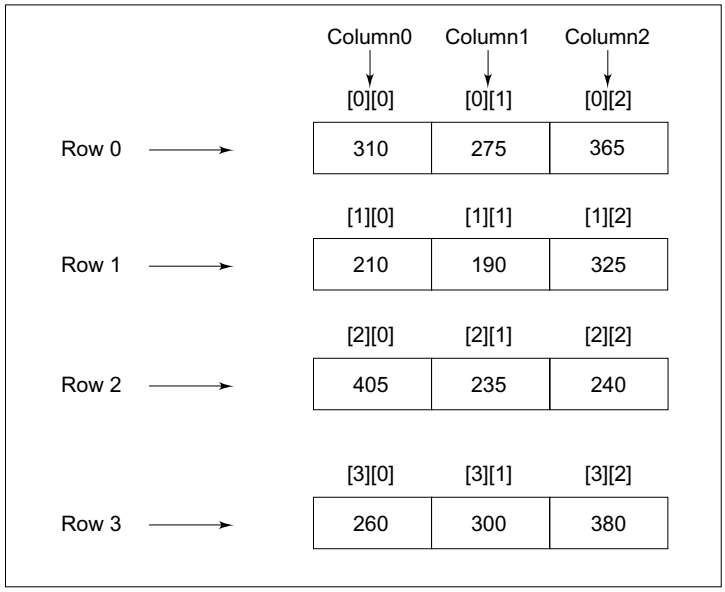


Fig. 8.3 *Representation of a two-dimensional array in memory*

Example 8.3

Write a program using a two-dimensional array to compute and print the following information from the table of data discussed above:

(a) Total value of sales by each girl.
(b) Total value of each item sold.
(c) Grand total of sales of all items by all girls.

The program and its output are shown in Fig. 8.4. The program uses the variable **value** in two-dimensions with the index **i** representing girls and **j** representing items. The following equations are used in computing the results:

$$\text{(a) Total sales by mth girl = (girl\_total[m])} = \sum_{j=0}^{2} \text{value [m][j]}$$

$$\text{(b) Total value of nth item = (item\_total[n])} = \sum_{i=0}^{3} \text{value[i][n]}$$

(c) Grand total $= \sum_{i=0}^{3} \sum_{j=0}^{2} \text{value}[i][j]$

$= \sum_{i=0}^{3} \text{girl\_total}[i]$

$= \sum_{j=0}^{2} \text{item\_total}[j]$

Program

```
/ ********************************************************************************** /
/*              PROGRAM SHOWING TWO-DIMENSIONAL ARRAYS                              */
/ ********************************************************************************** /
#define     MAXGIRLS     4
#define     MAXITEMS     3

main()
{
    int     value[MAXGIRLS][MAXITEMS];

    int     girl_total[MAXGIRLS], item_total [MAXITEMS];
    int     i, j, grand_total;

/* ..............READING OF VALUES AND COMPUTING girl_total ..............  */

    printf("Input data\n");
    printf("Enter values, one at a time, row-wise\n\n");

    for( i = 0 ; i < MAXGIRLS ; i++)
    {
        girl_total[i] = 0;
        for( j = 0 ; j < MAXITEMS ; j++)
        {
            scanf("%d", &value[i][j]);
            girl_total[i] = girl_total[i] + value[i][j];
        }
    }

/* . . . . . . . . . . . . . . . . . . . . . . . COMPUTING item_total . . . . . . . . . . . . . . . . . . . . . . .  */

    for( j = 0; j < MAXITEMS ; j++)
    {
        item_total[j] = 0;
        for( i = 0 ; i < MAXGIRLS ; i++ )
            item_total[j] = item_total[j] + value[i][j];
    }

/* . . . . . . . . . . . . . . . . . . . . . . . COMPUTING grand_total . . . . . . . . . . . . . . . . . . . . . . .  */

    grand_total = 0;
    for( i = 0 ; i < MAXGIRLS ; i++ )
        grand_total = grand_total + girl_total[i];

/* . . . . . . . . . . . . . . . . . . . . . . . PRINTING OF RESULTS . . . . . . . . . . . . . . . . . . . . . . .  */

    printf("\n GIRLS TOTALS\n\n");
    for( i = 0 ; i < MAXGIRLS ; i++ )
        printf{"Salesgirl[%d] = %d\n", i+1, girl_total[i] );
    printf("\n ITEM TOTALS\n\n");
    for(j = 0 ; j < MAXITEMS ; j++ )
        printf("Item[%d] = %d\n", j+1, item_total[j] );
```

```
    printf("\nGrand Total = %d\n", grand_total);
}
```

Output

```
Input data
Enter values, one at a time, row_wise

310    257    365
210    190    325
405    235    240
260    300    380

GIRLS TOTALS

Salesgirl[1] = 950
Salesgirl[2] = 725
Salesgirl[3] = 880
Salesgirl[4] = 940

ITEM TOTALS

Item[1] = 1185
Item[2] = 1000
Item[3] = 1310

Grand total = 3495
```

Fig. 8.4 *Illustration of two-dimensional arrays*

Example 8.4

Write a program to compute and print a multiplication table for numbers 1 to 5 as shown below:

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	·	·	·
4	4	8	·	·	·
5	5	10	·	·	25

The program shown in Fig. 8.5 uses a two-dimensional array to store the table values. Each value is calculated using the control variables of the nested **for** loops as follows:

product[i][j] = row * column

where i denotes rows and j denotes columns of the product table. Since the indices **i** and **j** range from 0 to 4, we have introduced the following transformation:

```
row = i+1
column = j+1
```

8.4 INITIALIZING TWO-DIMENSIONAL ARRAYS

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example,

```
static int table[2][3] = {0,0,0,1,1,1};
```

initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement can be equivalently written as

```
static int table[2][3] = {{0,0,0}, {1,1,1}};
```

by surrounding the elements of each row by braces.

We can also initialize a two-dimensional array in the form of a matrix as shown below:

```
static int table[2][3] = {
                              {0,0,0,},
                              {1,1,1}
                          };
```

Note the syntax of the above statements. Commas are required after each brace that closes off a row, except in the case of the last row.

If the values are missing in initializer, they are automatically set to zero. For instance, the statement

Program

```
/ ********************************************************************************** /
/*              PROGRAM TO PRINT MULTIPLICATION TABLE                              */
/ ********************************************************************************** /

#define   ROWS      5
#define   COLUMNS   5

main()
{
    int     row, column, product[ROWS][COLUMNS] ;
    int     i, j ;

    printf("     MULTIPLICATION TABLE\n\n") ;
    printf("     ") ;

    for( j = 1 ; j <= COLUMNS ; j++ )
        printf("%4d", j) ;
    printf("\n") ;
    printf("————————————————\n");

    for( i = 0 ; j < ROWS ; i++ )
    {
        row = i + 1 ;
```

```
            printf("%2d|", row) ;
            for(j = 1 ; j <= COLUMNS ; j++ )
            {
                column = j ;
                product[i][j] = row * column ;
                printf("%4d", product[i][j] );
            }
            printf("\n") ;
        }
    }
```

Output

```
MULTIPLICATION TABLE
          1   2   3   4   5
-------------------------------
1         1   2   3   4   5
2         2   4   6   8  10
3         3   6   9  12  15
4         4   8  12  16  20
5         5  10  15  20  25
```

Fig. 8.5 *Program to print multiplication table, using two-dimensional array*

```
static int table[2][3] = {
                            {1,1},
                            {2}
                         };
```

will initialize the first two elements of the first row to one, the first element of the second row to two, and all other elements to zero.

When all the elements are to be initialized to zero, the following short-cut method may be used.

```
static int m[3][5] = { {0}, {0} {0} };
```

The first element of each row is explicitly initialized to zero while other elements are automatically initialized to zero.

Example 8.5

A survey to know the popularity of four cars (Ambassador, Fiat, Dolphin and Maruti) was conducted in four cities (Bombay, Calcutta, Delhi and Madras). Each person surveyed was asked to give his city and the type of car he was using. The results, in coden form, are tabulated as follows:

```
M 1  C 2  B 1  D 3  M 2  B 4
C 1  D 3  M 4  B 2  D 1  C 3
D 4  D 4  M 1  M 1  B 3  B 3
C 1  C 1  C 2  M 4  M 4  C 2
D 1  C 2  B 3  M 1  B 1  C 2
D 3  M 4  C 1  D 2  M 3  B 4
```

Codes represent the following information:

M — Madras 1 — Ambassador

D — Delhi 2 — Fiat
C — Calcutta 3 — Dolphin
B — Bombay 4 — Maruti

Write a program to produce a table showing popularity of various cars in four cities.

A two-dimensional array **frequency** is used as an accumulator to store the number of cars used, under various categories in each city. For example, the element **frequency** [i][j] denotes the number of cars of type j used in city i. The **frequency** is declared as an array of size 5×5 and all the elements are initialized to zero.

The program shown in Fig. 8.6 reads the city code and the car code, one set after another, from the terminal. Tabulation ends when the letter X is read in place of a city code.

8.5 MULTIDIMENSIONAL ARRAYS

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multidimensional array is

type **array_name**[s1][s2][s3].....[sm];

where ***si*** is the size of the *i*th dimension. Some example are:

```
int survey[3][5][12];
float table[5][4][5][3];
```

survey is a three-dimensional array declared to contain 180 integer type elements. Similarly **table** is a four-dimensional array containing 300 elements of floating-point type.

The array **survey** may represent a survey data of rainfall during the last three years from January to December in five cities.

Program

```
/ ********************************************************************************** /
/ *                    PROGRAM TO TABULATE SURVEY DATA                             */
/ ********************************************************************************** /

main()
{
    int          i, j, car;
    static int   frequency[5][5] = { {0}, {0}, {0}, {0}, {0} };
    char         city;

    printf("For each person, enter the city code \n");
    printf("followed by the car code. \n");
    printf("Enter the letter X to indicate end.\n");

/* ..........TABULATION BEGINS ..........  */

    for(i = 1 ; i < 100 ; i++ )
    {
        scanf("%c", &city );
        if( city == 'X' )
          break;
        scanf("%d", &car );
        switch(city)
        {
```

```
                case 'B' :  frequency[1][car]++;
                            break;
                case 'C' :  frequency[2][car]++;
                            break;
                case 'D' :  frequency[3][car]++;
                            break;
                case 'M' :  frequency[4][car]++;
                            break;
            }
        }
/* ..... TABULATION COMPLETED AND PRINTING BEGINS ... */
        printf("\n\n");
        printf("              POPULARITY TABLE\n\n");
        printf("————————————————————————————————————————————————\n");
        printf("    City    Ambassador      Fiat    Dolphin    Maruti\n                ");
        printf("————————————————————————————————————————————————\n");
        for(i = 1 ; i <= 4; i++)
        {
            switch(i)
            {
                case 1:   printf("Bombay    ");
                          break;
                case 2:   printf("Calcutta  ");
                          break;
                case 3:   printf("Delhi     ");
                          break;
                case 4:   printf("Madras    ");
                          break;
            }
            for( j = 1 ; j <= 4; j++ )
                printf("%7d", frequency[i][j] ) ;
                printf("\n") ;
        }
        printf ("————————————————————————————————————————————————\n");
/* ……PRINTING ENDS …… */
}
```

Output

```
For each person, enter the city code
followed by the car code.
Enter the letter X to indicate end.
M 1" C 2 B 1 D 3 M 2 B 4
C 1 D 3 M 4 B 2 D 1 C 3
D 4 D 4 M 1 M 1 B 3 B 3
C 1 C 1 C 2 M 4 M 4 C 2
D 1 C 2 B 3 M 1 B 1 C 2
D 3 M 4 C 1 D 2 M 3 B 4 X
```

POPULARITY TABLE

City	Ambassador	Fiat	Dolphin	Maruti
Bombay	2	1	3	2
Calcutta	4	5	1	0
Delhi	2	1	3	2
Madras	4	1	1	4

Fig. 8.6 *Program to tabulate a survey data*

If the first index denotes year, the second city and the third month, then the element **survey[1][2][9]** denotes the rainfall in the month of October during the second year in city-3.

Remember that a three-dimensional array can be represented as a series of two-dimensional arrays as shown below:

Year 1

month / city	1	2	. . . . .	12
1				
.				
.				
5				

Year 2

month / city	1	2	. . . . .	12
1				
.				
5				

CASE STUDIES

1. Median of a List of Numbers

When all the items in a list are arranged in order, the middle value which divides the items into two parts with equal number of items on either side is called the *median*. Odd number of items have just one middle value while even number of items have two middle values. The median for even number of items is therefore designated as the average of the two middle values.

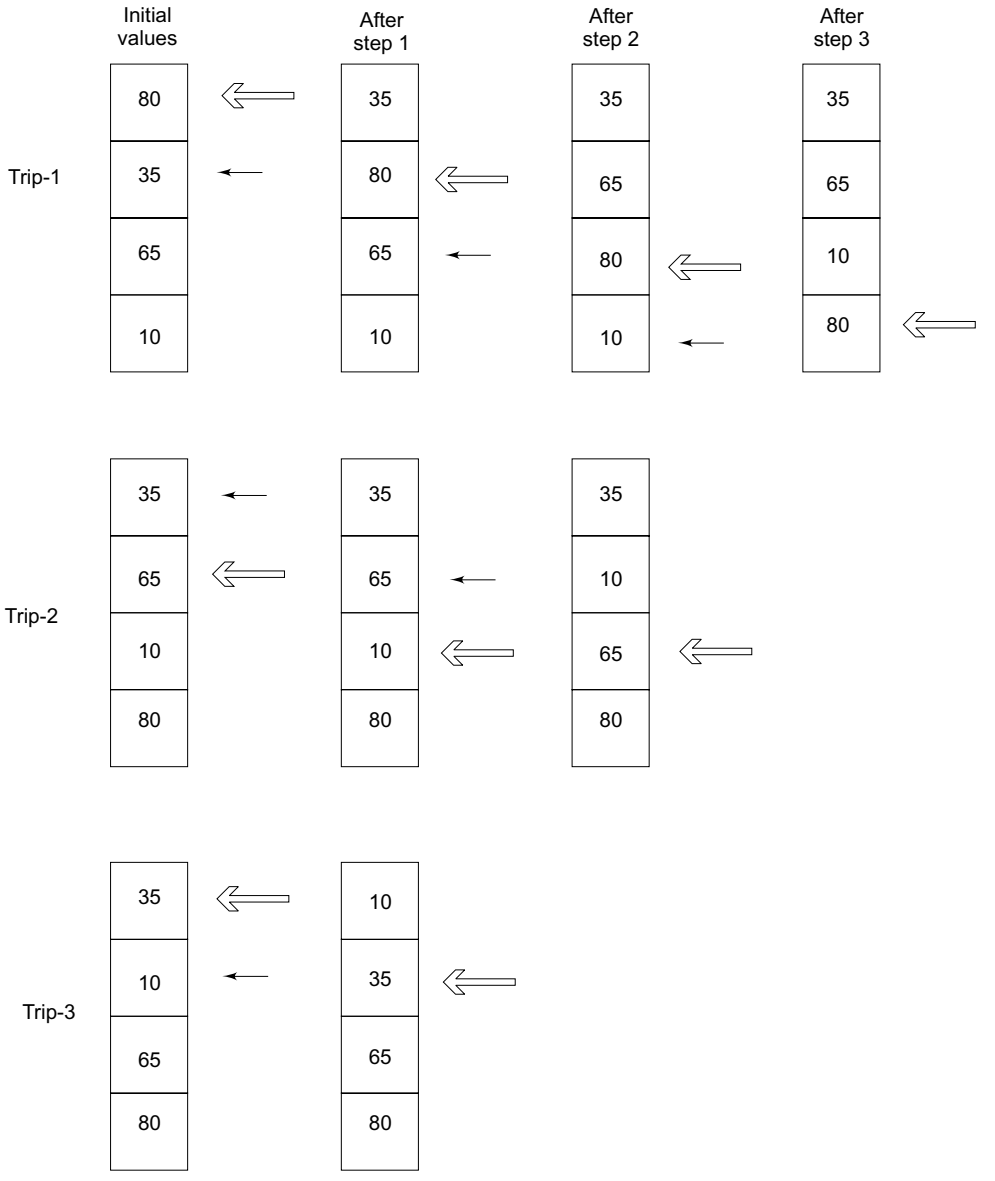
The major steps for finding the median are as follows:

1. Read the items into an array while keeping a count of the items.
2. Sort the items in increasing order.
3. Compute median.

The program and sample output are shown in Fig. 8.7. The sorting algorithm used is as follows:

1. Compare the first two elements in the list, say a[1], and a[2]. If a[2] is smaller than a[1], then interchange their values.
2. Compare a[2] and a[3]; interchange them if a[3] is smaller than a[2].
3. Continue this process till the last two elements are compared and interchanged.
4. Repeat the above steps n–1 times.

In repeated trips through the array, the smallest elements 'bubble up' to the top. Because of this bubbling up effect, this algorithms is called *bubble sorting*. The bubbling effect is illustrated below for four items.



During the first trip, three pairs of items are compared and interchanged whenever needed. It should be noted that the number 80, the largest among the items, has been moved to the bottom at the end of the first trip. This means that the element 80 (the last item in the new list) need not be considered any further. Therefore, trip-2 requires only two pairs to be compared. This time, the number 65 (the second largest value) has been moved down the list. Notice that each trip brings the smallest value 10 up by one level.

The number of steps required in a trip is reduced by one for each trip made. The entire process will be over when a trip contains only one step. If the list contains n elements, then the number of comparisons involved would be n(n–1)/2.

Program

```
/* ************************************************************************************ */
/*                    PROGRAM TO SORT A LIST AND FIND ITS MEDIAN                        */
/ ************************************************************************************  /

#define    N   10

main( )
{
    int i,j,n;
    float median,a[N], t;

    printf("Enter the number of items\n");
    scanf("%d", &n);

/*  Reading items into array a   */

    printf("Input %d values \n",n);
    for (i = 1; i <= n; i++)
        scanf("%f", &a[i]);

/*  Sorting begins    */

    for (i = 1 ; i <= n-1 ; i++)
    {   /* Trip-i begins.   */

        for (j = 1 ; j <=n – i ; j++)
        {
            if (a[j] <= a[j+1])
            { /* Interchanging values */
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
            else
            continue;
        }
    }   /* sorting ends */

/*  calculation of median */

        if (n % 2 == 0)
          median = (a[n/2] + a[n/2 + 1])/2.0 ;
        else
          median = a[n/2 + 1];

/*  Printing    */
        for (i = 1 ; i <= n ; i++)
            printf("%f ", a[i]);
        printf("\n\nMedian is %f\n", median);
}
```

Output

```
Enter the number of items
5
Input 5 values
1.111  2.222  3.333  4.444  5.555
5.555000  4.444000  3.333000  2.222000  1.111000

Median is 3.333000

Enter the number of items
6
Input 6 values
3   5   8   9   4   6
9.000000   8.000000   6.000000   5.000000   4.000000   3.000000

Median is 5.500000
```

Fig. 8.7 *Program to sort a list of numbers and to determine median*

2. Calculation of Standard Deviation

In statistics, standard deviation is used to measure deviation of data from its mean. The formula for calculating standard deviation of n items is

$$s = \sqrt{\text{variance}}$$

where $$\text{variance} = \frac{1}{n}\sum_{i=1}^{n}(x_i - m)^2$$

and $$m = \text{mean} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The algorithm for calculating the standard deviation is as follows:

1. Read n items.
2. Calculate sum and mean of the items.
3. Calculate variance.
4. Calculate standard deviation.

Complete program with sample output is shown in Fig. 8.8.

Program

```
/*  ************************************************************************************ */
/*              PROGRAM TO CALCULATE STANDARD DEVIATION                                  */
/   ************************************************************************************ /

#include   <math.h>
#define    MAXSIZE 100
main( )
{
    int i,n;
    float value [MAXSIZE}, deviation,
        sum,sumsqr,mean,variance,stddeviation;

    sum = sumsqr = n = 0;

    printf("Input values: input – 1 to end \n");
    for (i = 1; i<MAXSIZE; i++)
    {
        scanf("%f", &value[i]);
        if (value[i] == – 1)
        break;
        sum += value[i];
        n += 1;
    }
    mean = sum/(float)n;

    for (i = 1 ; i <= n; i++)
    {
        deviation = value[i] – mean;
        sumsqr +=deviation * deviation;
    }
```

```
    variance = sumsqr/(float)n ;
    stddeviation = sqrt(variance) ;
    printf("\nNumber of items : %d\n",n);
    printf("Mean : %f\n", mean);
    printf("Standard deviation :  %f=n", stddeviation);
}
```

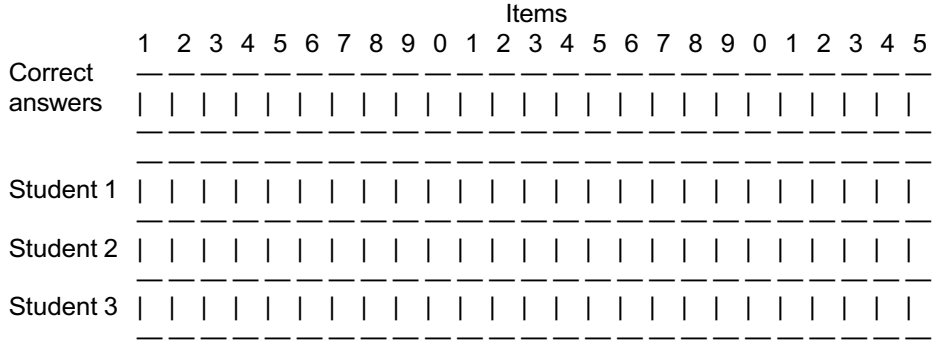
Output

```
Input values: input–1 to end
65 9   27 78 12 20 33 49 –1

Number of items : 8
Mean : 36.625000
Standard deviation : 23.510303
```

Fig. 8.8 *Program to calculate standard deviation*

3. Evaluating a Test

A test consisting of 25 multiple-choice items is administered to a batch of 3 students. Correct answers and student responses are tabulated as shown below:



Algorithm for evaluating the answers of students is as follows:

1. Read correct answers into an array.
2. Read the responses of a student and count the correct ones.
3. Repeat step-2 for each student.
4. Print the results.

A program to implement this algorithm is given in Fig. 8.9. The program uses the following arrays:

key[i] — To store correct answers of items
response[i] — To store responses of students
correct[i] — To identify items that are answered correctly.

Program

```
/ ********************************************************************************** /
/*              PROGRAM TO EVALUATE A MULTIPLE-CHOICE TEST          */
/ ********************************************************************************** /

#define    STUDENT    3
```

```
#define    ITEMS          25
main( )
{
    char    key[ITEMS+1], response[ITEMS+1];
    int     count, i, student,n,
            correct[ITEMS+1];
/*  Reading of Correct answers */
    printf("Input key to the items\n");
    for(i=0; i < ITEMS; i++)
        scanf("%c",&key[i]);
    scanf("%c",&key[i]);
    key[i] = '\0';
/*  Evaluation begins  */
    for(student = 1; student <= STUDENTS; student++)
    {
    /*Reading student responses and counting correct ones*/
        count = 0;
        printf("\n");
        printf("Input responses of student—%d\n",student);
        for(i=0; i < ITEMS ; i++)
            scanf("%c",&response[i]);
        scanf("%c",&response[i]);
        response[i] = '\0';
        for (i = 0; i < ITEMS; i++)
            correct[i] = 0;
        for(i=0; < ITEMS ; i++)
            if(response[i] == key[i])
            {
                count = count + 1;
                correct[i] = 1;
            }
        /*  printing of results */
        printf("\n");
        printf("Student—%d\n", student);
        printf("Score is %d out of %d\n",count, ITEMS);
        printf("Response to the items below are wrong\n");
        n = 0;
        for(i=0; i < ITEMS ; i++)
            if(correct[i] == 0)
            {
                printf("%d",i+1);
                n = n+1;
            }
        if(n == 0)
          printf("NIL\n");
        printf(("\n");
    } /* Go to next student */
/* Evaluation and print ends */
}
```

Output

```
Input key to the items
abcdabcdabcdabcdabcdabcda
```

```
Input responses of student-1
abcdabcdabcdabcdabcdabcda

Student-1
Score is 25 out of 25
Response to the following items are wrong
NIL

Input responses of student-2
abcddcbaabcdabcddddddddddd

Student-2
Score is 14 out of 25
Response to the following items are wrong
5  6  7  8  17 18 19 21 22 23 25

Input responses of student-3
aaaaaaaaaaaaaaaaaaaaaaaaa

Student-3
Score is 7 out of 25
Response to the following items are wrong
2 3 4 6 7 8 10 11 12 14 15 16 18 19 20 22 23 24
```

Fig. 8.9 *Program to evaluate responses to a multiple-choice test*

4. Production and Sales Analysis

A company manufactures five categories of products and the number of items manufactured and sold are recorded productwise every week in a month. The company reviews its production schedule at every month-end. The review may require one or more of the following information:

(a) Value of weekly production and sales.
(b) Total value of all the products manufactured.
(c) Total value of all the products sold.
(d) Total value of each product, manufactured and sold.

Let us represent the products manufactured and sold by two two-dimensional arrays M and S respectively. Then,

M =

M11	M12	M13	M14	M15
M21	M22	M23	M24	M25
M31	M32	M33	M34	M35
M41	M42	M43	M44	M45

S =

S11	S12	S13	S14	S15
S21	S22	S23	S24	S25
S31	S32	S33	S34	S35
S41	S42	S43	S44	S45

where Mij represents the number of jth type product manufactured in ith week and Sij the number of jth product sold in ith week. We may also represent the cost of each product by a single dimensional array C as follows:

C =

C1	C2	C3	C4	C5

where Cj is the cost of jth type product.

We shall represent the value of products manufactured and sold by two value arrays, namely, **Mvalue** and **Svalue**. Then,

Mvalue[i][j] = Mij x Cj
Svalue[i][j] = Sij x Cj

A program to generate the required outputs for the review meeting is shown in Fig. 8.10. The following additional variables are used:

Mweek[i] = Value of all the products manufactured in week i.

$$= \sum_{j=1}^{5} \text{Mvalue}[i][j]$$

Sweek[i] = Value of all the products sold in week i

$$= \sum_{j=1}^{5} \text{Svalue}[i][j]$$

Mproduct[j] = Value of jth type product manufactured during the month

$$= \sum_{j=1}^{4} \text{Mvalue}[i][j]$$

Sproduct[j] = Value of jth type product sold during the month

$$= \sum_{j=1}^{4} \text{Svalue}[i][j]$$

Mtotal = Total value of all the products manufactured during the month

$$= \sum_{j=1}^{4} \text{Mweek}[i] \quad = \sum_{j=1}^{5} \text{Mproduct}[j]$$

Stotal = Total value of all the products sold during the month

$$= \sum_{j=1}^{4} \text{Sweek}[i] \quad = \sum_{j=1}^{5} \text{Sproduct}[j]$$

Program

```
/* ************************************************************************** */
/*                    PRODUCTION AND SALES ANALYSIS                           */
/* ************************************************************************** */

main( )
{
    int M[5][6], S[5][6],C[6]
        Mvalue[5][6], Svalue[5][6],
```

```
        Mweek[5], Sweek[5],
        Mproduct[6],  Sproduct[6],
        Mtotal, Stotal, i,j,number;
/*      Input data      */
    printf("Enter products manufactured week_wise \n");
    printf(" M11, M12 ——, M21, M22, * etc\n");
    for(i=1; i<=4;  i++)
        for(j=1; j<=5;  j++)
            scanf("%d",&M[i][j]);
    printf("Enter products sold week_wise\n");
    printf(" S11, S12, ——, S21, S22, —— etc\n ");
    for(i=1; I<=4; i++)
        for(j=1; j<=5; j++)
            scanf("%d", &S[i][j]);
    printf("Enter cost of each product\n");
    for(j=1; j<=5; j++)
        scanf("%d",&C[j]);
/*  Value matrices of production and sales    */
    for(i=1; i<=4; i++)
        for(j=1; j<=5; j++)
        {
            Mvalue[i][j] = M[i][j] * C[j];
            Svalue[i][j] = S[i][j] * C[j];
        }
/*  Total value of weekly production and sales    */
    for(i=1; i<=4; i++)
    {
        Mweek[i] = 0 ;
        Sweek[j] = 0 ;
        for(j=1; j<=5; j++)
        {
            Mweek[i] + = Mvalue[i][j];
            Sweek[i] + = Svalue[i][j];
        }
    }
/*  Monthly value of product_wise production and sales */
    for(j=1; j<=5; j++)
    {
        Mproduct[j] = 0 ;
        Sproduct[j] = 0 ;
        for(i=1; i<=4; i++)
        {
            Mproduct[j] + = Mvalue[i][j];
            Sproduct[j] +  = Svalue[i][j];
        }
    }
/*  Grand total of production and sales values    */
    Mtotal = Stotal = 0;
    for(i = 1; i<=4; i++)

    {
        Mtotal + = Mweek[i];
```

```
        Stotal + = Sweek[i];
    }
    /*****************************************************************************
                    Selection and printing of information required
    *****************************************************************************/
    printf("\n\n");
    printf(" Following is the list of things you can\n");
    printf(" request for Enter appropriate item number\n");
    printf(" and press RETURN Key\n\n");
    printf(" 1. Value matrices of production & sales\n");
    printf(" 2. Total value of weekly production & sales\n");
    printf(" 3. Product_wise monthly value of production &");
    printf(" sales\n");
    printf(" 4. Grand total value of production & sales\n");
    printf(" 5. Exit\n");
    number = 0;
    while(1)
    {   /* Beginning of while loop */
        printf("\n\n ENTER YOUR CHOICE:");
        scanf("%d",&number);
        printf("\n");
        if(number == 5)
        {
            printf("G O O D B Y E\n\n");
            break;
        }
        switch(number)
        {   /* Beginning of switch */
/*  V A L U E  M A T R I C E S */
            case 1:
                printf("VALUE MATRIX OF PRODUCTION\n\n");
                for(i=1; i <=4; i++)
                {
                    printf(" Week(%d)\t",i);
                    for(j=1; j <=5; j++)
                        printf("%7d", Mvalue[i][j]);
                    printf("\n");
                }
                printf({"\n VALUE MATRIX OF SALES\n\n");
                for (i=1; i < =4; i++)
                {
                    printf(" Week(%d)\t",i);
                    for(j=1; j <=5; j++)
                        printf("%7d", Svalue[i][j]);
                    printf("\n");
                }
                break;
    W E E K L Y  A N A L Y S I S */
    case 2:
            printf(" TOTAL WEEKLY PRODUCTION & SALES\n=n");
            printf("                    PRODUCTION SALES\n");
```

```
            printf("                    ————————  ——— \n");
            for(i=1; i <=4; i++)
            {
                printf("Week(%d)\t", i);
                printf("%7d\t%7d\n", Mweek[i], Sweek[i]);
            }
            break;
/*      P R O D U C T W I S E   A N A L Y S I S       */
        case 3:
            printf(" PRODUCT_WISE TOTAL PRODUCTION &");
            printf(" SALES\n\n");
            printf("                    PRODUCTION SALES\n");
            printf("                    ————————  ———*\n");
            for(j=1; j <=5; j++)
            {
                printf(" Product(%d)\t", j);
                printf("%7d\t%7d\n",Mproduct[j],Sproduct[j]);
            }
            break;
/*      G R A N D   T O T A L S       */
        case 4:
            printf(" GRAND TOTAL OF PRODUCTION & SALES\n");
            printf("\n Total production = %d\n", Mtotal);
            printf(" Total sales    = %d\n", Stotal);
            break;
/*      D E F A U L T   */
        default :
            printf(" Wrong choice, select again\n\n");
            break;
        } /* End of switch */
    } /* End of while loop */
    printf(" Exit from the program\n\n");
}   /* End of main */
```

Output

```
Enter products manufactured week_wise
M11,M12, ———, M21,M22, ——— etc
11  15  12  14  13
13  13  14  15  12
12  16  10  15  14
14  11  15  13  12

Enter products sold week_wise
S11,S12,———,S21,S22,——— etc
10  13   9  12  11
12  10  12  14  10
11  14  10  14  12
12  10  13  11  10

Enter cost of each product
10  20  30  15  25

Following is the list of things you can
request for. Enter appropriate item number
and press RETURN key
1.  Value matrices of production & sales
```

```
2. Total value of weekly production & sales
3. Product_wise monthly value of production & sales
4. Grand total value of production & sales
5. Exit
ENTER YOUR CHOICE:1
VALUE MATRIX OF PRODUCTION
Week(1)    110    300    360    210    325
Week(2)    130    260    420    225    300
Week(3)    120    320    300    225    350
Week(4)    140    220    450    185    300
VALUE MATRIX OF SALES
Week(1)    100    260    270    180    275
Week(2)    120    200    360    210    250
Week(3)    110    280    300    210    300
Week(4)    120    200    390    165    250
ENTER YOUR CHOICE 2
TOTAL WEEKLY PRODUCTION & SALES
           PRODUCTION    SALES
Week(1)      1305        1085
Week(2)      1335        1140
Week(3)      1315        1200
Week(4)      1305        1125
ENTER YOUR CHOICE:3
PRODUCT_WISE TOTAL PRODUCTION SALES
           PRODUCTION    SALES
Product(1)    500         450
Product(2)   1100         940
Product(3)   1530        1320
Product(4)    855         765
Product(5)   1275        1075
ENTER YOUR CHOICE:4
GRAND TOTAL OF PRODUCTION & SALES
Total production  = 5260
Total sales       = 4550
ENTER YOUR CHOICE:5
G O O D    B Y E
Exit from the program
```

Fig. 8.10 *Program for production and sales analysis*

Review Questions and Exercises

8.1 Explain the need for array variables.

8.2 Identify errors, if any, in each of the following array declaration statements.

(a) int score (100);
(b) float values[10,15];
(c) float average[ROW],[COLUMN];
(d) char name[15];
(e) int sum[];

8.3 Identify errors, if any, in each of the following initialization statements.

(a) static int number[ ] = {0,0,0,0,0,};
(b) static float item[3][2] = {0,1,2,3,4,5};
(c) static char word[ ] = {'A','R','R','A','Y'};
(d) static int m[2,4] = {(0,0,0,0)(1,1,1,1)};
(e) static float result[10] = 0;

8.4 Assume that the arrays A and B are declared as follows:

```
int     A[5][4];
float   B[4];
```

Find the errors (if any) in the following program segments.

(a)
```
for (i=1; i<=5; i++)
    for(j=1; j<=4; j++)
        A[i][j] = 0;
```

(b)
```
for (i=1; i<4; i++)
    scanf("%f", B[i]);
```

(c)
```
for (i=0; i<=4; i++)
    B[i] = B[i]+i;
```

(d)
```
for (i=4; j>=0; 4; j– –)
    for (j=0; j < 4; j ++)
        A[i][j] = B[j] + 1.0;
```

8.5 Write the segment of a program that initializes the array A as follows:

1	0	0	0	0	. . . .	0
0	1	0	0	0	. . . .	0
0	0	1	0	0	. . . .	0
. . . . .	. . . . .	. . . . .	. . . . .	. . . . .		. . . . .
0	0	0	0	0	. . . .	1

All diagonal elements are initialized to one and others to zero.

8.6 Write a program for fitting a straight line through a set of points $(x_i, y_i), i = 1, \cdots, n$. The straight line equation is

$$y = mx + c$$

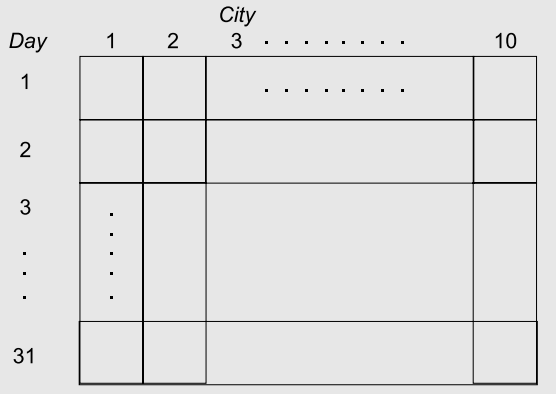
and the values of m and c are given by

$$m = \frac{n \Sigma (x_i\ y_i) - (\Sigma\ x_i)(\Sigma\ y_i)}{n \left(\Sigma\ x_i^2\right) - (\Sigma\ x_i)^2}$$

$$c = \frac{1}{n}\ (\Sigma y_i - m \Sigma x_i)$$

All summations are from 1 to n.

8.7 The daily maximum temperatures recorded in 10 cities during the month of January (for all 31 days) have been tabulated as follows:



Write a program to read the table elements into a two-dimensional array **temperature**, and to find the city and day corresponding to (a) the highest temperature and (b) the lowest temperature.

8.8 An election is contested by 5 candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and count the votes cast for each candidate using an array variable count. In case, a number read is outside the range 1 to 5, the ballot should be considered as a 'spoilt ballot' and the program should also count the number of spoilt ballots.

8.9 The following set of numbers is popularly known as Pascal's triangle.

```
1
1   1
1   2    1
1   3    3    1
1   4    6    4    1
1   5   10   10    5    1
.   .    .    .    .    .    .
.   .    .    .    .    .    .    .
```

If we denote rows by i and columns by j, then any element (except the boundary elements) in the triangle is given by

$$p_{ij} = p_{i-1,\ j-1} + p_{i-1,\ j}$$

Write a program to calculate the elements of the Pascal triangle for 10 rows and print the results.

8.10 The annual examination results of 100 students are tabulated as follows:

Roll No.	Subject 1	Subject 2	Subject 3

Write a program to read the data and determine the following:

(a) Total marks obtained by each student.
(b) The highest marks in each subject and the Roll No. of the student who secured it.
(c) The student who obtained the highest total marks.

8.11 Given are two one-dimensional arrays **A** and **B** which are sorted in ascending order. Write a program to *merge* them into a single sorted array **C** that contains every item from arrays **A** and **B**, in ascending order.

8.12 Two matrices that have the same number of rows and columns can be multiplied to produce a third matrix. Consider the following two matrices.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \ldots\ldots & a_{1n} \\ a_{21} & a_{22} \ldots\ldots & a_{2n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \ldots\ldots\ldots & a_{nn} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \ldots\ldots & b_{1n} \\ b_{21} & b_{22} \ldots\ldots & b_{2n} \\ . & & . \\ . & & . \\ . & & . \\ b_{n1} & \ldots\ldots\ldots & b_{nn} \end{bmatrix}$$

The product of **A** and **B** is a third matrix **C** of size *nxn* where each element of **C** is given by the following equation.

$$\mathbf{C}_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{ki}$$

Write a program that will read the values of elements of **A** and **B** and produce the product matrix **C**

Chapter 9

Handling of Character Strings

9.1 INTRODUCTION

A string is an array of characters. We have used strings in a number of examples in the past. Any group of characters (except double quote sign) defined between double quotation marks is a constant string. Example:

"Man is obviously made to think."

If we want to include a double quote in the string, then we may use it with a back slash as shown below.

"\" Man is obviously made to think, \" said Pascal."

For example,

```
printf ("/"well Done !/" ");
```

will output the string

"Well Done !"

while the statement

```
printf("Well Done !");
```

will output the string

Well Done !

Character strings are often used to build meaningful and readable programs. The common operations performed on character strings are:

- Reading and Writing strings.
- Combining strings together.
- Copying one string to another.
- Comparing strings for equality.
- Extracting a portion of a string.

In this chapter we shall discuss these operations in detail and develop programs that involve these operations.

9.2 DECLARING AND INITIALIZING STRING VARIABLES

A string variable is any valid C variable name and is always declared as an *array*. The general form of declaration of a string variable is

char *string_name*[size];

The *size* determines the number of characters in the *string-name*. Some examples are:

```
char city[10];
char name[30];
```

When the compiler assigns a character string to a character array, it automatically supplies a *null* character ('\0') at the end of the string. Therefore, the size should be equal to the maximum number of characters in the string *plus* one.

Character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms:

```
static char city[9] = "NEW YORK";
static char city[9] = {'N','E','W', ' ','Y','O','R','K','/0'};
```

The reason that city had to be 9 elements long is that the string NEW YORK contains 8 characters and one element space is provided for the null terminator. Note that when we initialize a character array by listing its elements, we must supply explicitly the null terminator.

C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized. For example, the statement

```
static char string [ ] = {'G', 'O', 'O', 'D', '\0'};
```

defines the array **string** as a five element array.

Note that the word **static** may be omitted when using ANSI compilers.

9.3 READING STRINGS FROM TERMINAL

Reading Words

The familiar input function **scanf** can be used with **%s** format specification to read in a string of characters. Example:

```
char address[15];
scanf("%s", address);
```

The problem with the **scanf** function is that it terminates its input on the first white space it finds. (A white space includes blanks, tabs, carriage returns, form feeds, and new lines.) Therefore, if the following line of text is typed in at the terminal,

NEW YORK

then only the string "NEW" will be read into the array **address**, since the blank space after the word 'NEW' will terminate the string.

Note that unlike previous **scanf** calls, in the case of character arrays, the ampersand (&) is not required before the variable name. The **scanf** function automatically terminates the string that is read with a *null* character and therefore the character array should be large enough to hold the input string plus the null character.

If we want to read the entire line "NEW YORK", then we may use two character arrays of appropriate sizes. That is,

```
scanf("%s %s", adr1, adr2);
```

with the line of text

```
NEW YORK
```

will assign the string "NEW" to **adr1** and "YORK" to **adr2**.

Example 9.1

Write a program to read a series of words from a terminal using **scanf** function.

The program shown in Fig. 9.1 reads four words and displays them on the screen. Note that the string 'Oxford Road' is treated as two words while the string 'Oxford-Road' as *one word*.

Program

```
/*********************************************************************************** /
/*       READING A SERIES OF WORDS USING scanf FUNCTION              */
/*********************************************************************************** /

main*( )
{
    char word1 [40], word2[40], word3[40], word4[40];
    printf("Enter text : \n");
    scanf("%s %s", word1, word2);
    scanf("%s", word3);
    scanf("%s", word4);

    printf("=n");
    printf("word1 = %s\nword2 = %s\n", word1, word2);
    printf("word3 = %s\nword4 = %s\n", word3, word4);
}
```

Output

```
Enter text:
Oxford Road, London M17ED

word1 = Oxford
word2 = Road,
word3 = London
word4 = M17ED

Enter text:
Oxford-Road, London-M17ED United Kingdom

word1 = Oxford-Road,
word2 = London-M17ED
word3 = United
word4 = Kingdom
```

Fig. 9.1 *Reading a series of words using* ***scanf***

Reading a Line of Text

In many text processing applications, we need to read in an entire line of text from the terminal. It is not possible to use **scanf** function to read a line containing more than one word. This is because the **scanf** terminates reading as soon as a space is encountered in input.

We have discussed in Chapter 5 as to how to read a single character from the terminal, using the function **getchar**. We can use this function repeatedly to read successive single characters from the input and place them into a character array. Thus, an entire line of text can be read and stored in an array. The reading is terminated when the newline character ('\n') is entered and the *null* character is then inserted at the end of the string.

Example 9.2

Write a program to read a line of text containing a series of words from the terminal.

The program shown in Fig. 9.2 can read a line of text (upto a maximum of 80 characters) into the string **line**. Every time a character is read, it is assigned to its location in the string **line** and then tested for *newline* character. When the *newline* character is read (signalling the end of line), the reading loop is terminated and the *newline* character is replaced by the *null* character to indicate the end of character string.

Program

```
/*******************************************************************************/
/*          PROGRAM TO READ A LINE OF TEXT FROM TERMINAL        */
/*******************************************************************************/

#include      <stdio.h>

main( )
{
    char   line [81], character;
    int    c;

    c = 0;

    print("Enter text. Press <Return> at end\n");

    do
    {
        character  = getchar( );
        line[c]    = character;
        c++;
    }
    while(character ! = '\n');

    c = c – 1;
    line[c] = '\0';
    printf("\n%s\n", line);
}
```

Output

```
Enter text. Press <Return> at end
Programming in C is interesting.
Programming in C is interesting.

Enter text. Press <Return> at end
National Computing Centre, United Kingdom.
National Computing Centre, United Kingdom.
```

Fig. 9.2 *Program to read a line of text from terminal*

When the loop is exited, the value of the index **c** is one number higher than the last character position in the string (since it has been incremented after assigning the new character to the string). Therefore the index value **c–1** gives the position where the *null* character is to be stored.

C does not provide operators that work on strings directly. For instance we cannot assign one string to another directly. For example, the assignment statements

```
string = "ABC";
string1 = string2;
```

are not valid. If we really want to copy the characters in **string2** into **string1**, we may do so on a character-by-character basis.

Example 9.3

Write a program to copy one string to another and count the number of characters copied.

The program is shown in Fig. 9.3. We use a **for** loop to copy the characters contained inside **string2** into the **string1**. The loop is terminated when the *null* character is reached. Note that we are again assigning a null character to the **string1**.

9.4 WRITING STRINGS TO SCREEN

We have used extensively the **printf** function with **%s** format to print strings to the screen. The format **%s** can be used to display an array of characters that is terminated by the *null* character. For example, the statement

```
printf("%s", name);
```

can be used to display the entire contents of the array **name**.

We can also specify the precision with which the array is displayed. For instance, the specification

```
% 10.4
```

indicates that the *first four* characters are to be printed in a field width of 10 columns.

However, if we include the minus sign in the specification (e.g., %–10.4s), the string will be printed left justified. Example 9.4 illustrates the effect of various %s specifications.

Example 9.4

Write a program to store the string "United Kingdom" in the array **country** and display the string under various format specifications.

The program and its output are shown in Fig. 9.4. The output illustrates the following features of the **%s** specifications.

1. When the field width is less than the length of the string, the entire string is printed.

Program

```
/*********************************************************************************** /
/*              C+OPYING ONE STRING INTO ANOTHER                                    */
/***********************************************************************************/

main( )
{
    char   string1[80], string2[80];
    int    i;
```

```
    printf("Enter a string \n");
    printf("?");
    scanf("%s", string2);
    for(i=0; string2[i] ! = '\0'; i+ +)
    string1[i] = string2[i];
    string1[i] = '/0';
    printf("\n");
    printf("%s\n", string1);
    printf("Number of characters = %d\n", i);
}
```

Output

```
Enter a string
?Manchester

Manchester
Number of characters = 10


Enter a string
?Westminster

Westminster
Number of characters = 11
```

Fig. 9.3 *Copying one string into another*

2. The integer value on the right side of the decimal point specifies the number of characters to be printed.
3. When the number of characters to be printed is specified as zero, nothing is printed.
4. The minus sign in the specification causes the string to be printed left-justified.

The **printf** on UNIX supports another nice feature that allows for variable field width or precision. For instance

printf("%*.*s\n", w,d, string);

prints the first *d* characters of the **string** in the field width of *w*.

This feature comes in handy for printing a sequence of characters. Example 9.5 illustrates this.

Program

```
/*********************************************************************************/
/*                    WRITING STRINGS USING %s FORMAT                            */
/*********************************************************************************/

Main()
{
    static char country[15] = "United Kingdom";
    printf("\n\n");
    printf("————————————— \n");
    printf(" |% 15s |\n", country);
    printf(" |%5s |\n", country);
    printf(" |% 15.6s |\n", country);
```

```
    printf(" |% 15.6s |\n", country);
    printf(" |% 15.0s |\n", country);
    printf(" |%.3s |\n", country);
    printf(" |%s |\n", country);
    printf("————————————— \n");
}
```

Output

```
┌──────────────┐
| United Kingdom
|United Kingdom |
|        United
|United
|
|Uni |
|United Kingdom
└──────────────┘
```

Fig. 9.4 *Writing strings using* ***%s*** *format*

Example 9.5

Write a program using **for** loop to print the following output.

```
C
CP
CPr
CPro
. . . . .
. . . . .
CProgramming
```

```
CProgramming
. . . . .
. . . . .
CPro
CPr
CP
C
```

Program

```
/*********************************************************************************/
/*                PRINTING SEQUENCES OF CHARACTERS                               */
/*********************************************************************************/

main( )
{
    int  c,   d;
    static char     string[ ] = "CProgramming";
    printf("\n\n");
    printf("————————————— \n");
    for(c = 0; c <=11 ; c++)
    {
        d = c + 1;
        printf(" |%–12.*s |\n", d, string);
```

```
    }
    printf("|————————————|\n");
    for(c = 11 ; c >=0; c– –)
    {
        d = c + 1;
        printf("|%–12.*s|\n", d, string);
    }
    printf("————————————\n");
}
```

Output

```
C
CP
CPr
CPro
CProg
CProgr
CProgra
CProgram
CProgramm
CProgrammi
CProgrammin
CProgramming

CProgramming
CProgrammin
CProgrammi
CProgramm
CProgram
CProgra
CProgr
CProg
CPro
CPr
CP
C
```

Fig. 9.5 *Illustration of variable field specifications*

The outputs of the program in Fig. 9.5, for variable specifications **%12.*s**, **%. *s**, and **%*.1s** are shown in Fig. 9.6 which further illustrates the variable field width and the precision specifications.

9.5 ARITHMETIC OPERATIONS ON CHARACTERS

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into an integer value by the system. The integer value depends on the local character set of the system.

To write a character in its integer representation, we may write it as an integer. For example, if the machine uses the ASCII representation, then,

```
x ='a';
printf("%d\n",x);
```

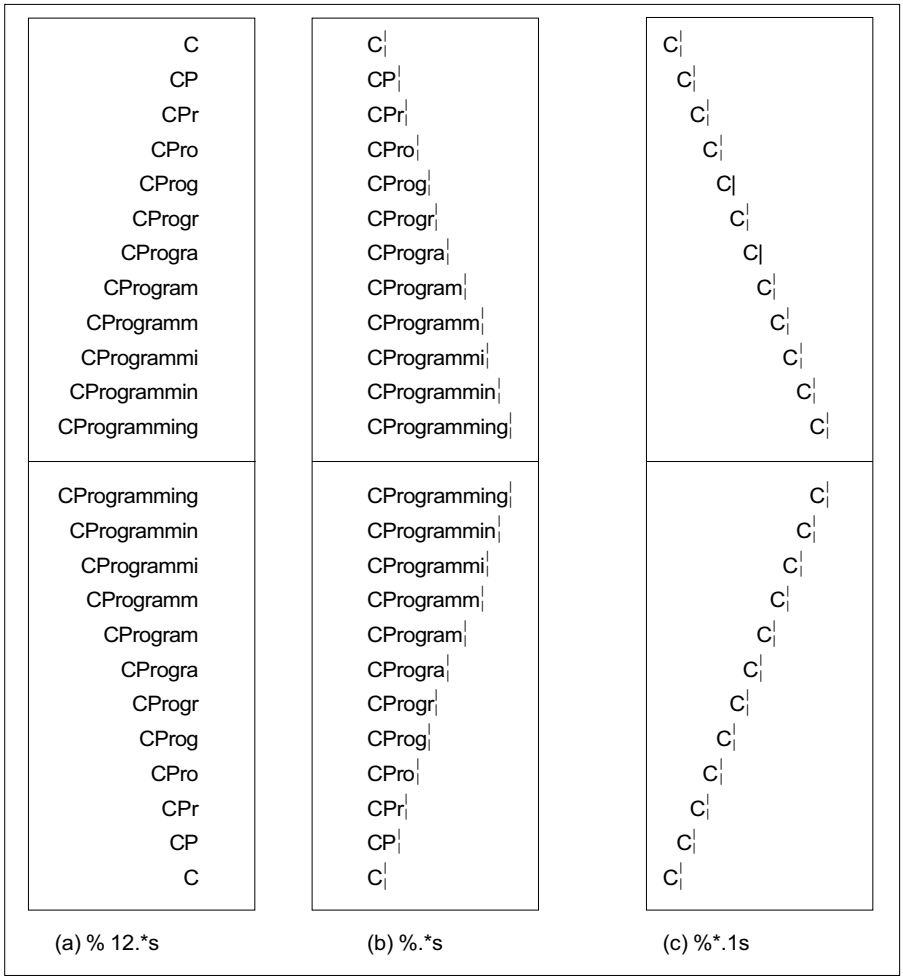


Fig. 9.6 *Further illustrations of variable specifications*

will display the number 97 on the screen.

It is also possible to perform arithmetic operations on the character constants and variables. For example,

```
x = 'z'–1;
```

is a valid statement. In ASCII, the value of '**z**' is 122 and therefore, the statement will assign the value 121 to the variable **x**.

We may also use character constants in relational expressions. For example, the expression

```
ch >= 'A' && ch <= 'Z'
```

would test whether the character contained in the variable **ch** is an upper-case letter.

We can convert a character digit to its equivalent integer value using the following relationship:

```
x = character - '0';
```

where **x** is defined as an integer variable and **character** contains the character digit. For example, let us assume that the **character** contains the digit '7', Then,

```
x  = ASCII value of '7' – ASCII value of '0'
   = 55 – 48
   = 7
```

The C library supports a function that converts a string of digits into their integer values. The function takes the form

```
x = atoi(string)
```

x is an integer variable and **string** is a character array containing a string of digits. Consider the following segment of a program:

```
---------
number = "1988";
year = atoi(number);
---------
```

number is a string variable which is assigned the string constant "1988". The function **atoi** converts the string "1988" (contained in **number**) to its numeric equivalent 1988 and assigns it to the integer variable **year**.

Example 9.6
Write a program which would print the alphabet set a to z and A to Z in decimal and character form.

The program is shown in Fig. 9.7. In ASCII character set, the decimal numbers 65 to 90 represent upper-case alphabets and 97 to 122 represent lower-case alphabets. The values from 91 to 96 are excluded using an **if** statement in the **for** loop.

9.6 PUTTING STRINGS TOGETHER

Just as we cannot assign one string to another directly, we cannot join two strings together by the simple arithmetic addition. That is, the statements such as

```
string3 = string1 + string2;
string2 = string1 + "hello";
```

are not valid. The characters from **string1** and **string2** should be copied into the **string3** one

Program

```
/*********************************************************************************/
/*   PRINTING ALPHABET SET IN DECIMAL AND CHARACTER FORM    */
/*********************************************************************************/

main( )
{
    char    c ;

    printf("\n\n");
    for(c = 65; c <= 122; c = c + 1)
    {
        if(c > 90 && c < 97)
            continue;
        printf("%5d – %c ", c, c);
    }

    printf("\n");
}
```

Output

```
  65   -   A   66   -   B   67   -   C   68   -   D   69   -   E   70   -   F
  71   -   G   72   -   H   73   -   I   74   -   J   75   -   K   76   -   L
```

```
 77 - M  78 - N  79 - O  80 - P  81 - Q  82 - R
 83 - S  84 - T  85 - U  86 - V  87 - W  88 - X
 89 - Y  90 - Z  97 - a  98 - b  99 - c 100 - d
101 - e 102 - f 103 - g 104 - h 105 - i 106 - j
107 - k 108 - l 109 - m 110 - n 111 - o 112 - p
113 - q 114 - r 115 - s 116 - t 117 - u 118 - v
119 - w 120 - x 121 - y 122 - z
```

Fig. 9.7 *Printing of the alphabet set in decimal and character form*

after the other. The *size* of the array **string3** should be large enough to hold the total characters.

The process of combining two strings together is called *concatenation*. Example 9.7 illustrates the concatenation of three strings.

Example 9.7

The names of employees of an organization are stored in three arrays, namely **first_name, second_name**, and **last_name**. Write a program to concatenate the three parts into one string to be called **name**.

The program is given in Fig. 9.8. Three **for** loops are used to copy the three strings. In the first loop, the characters contained in the **first_name** are copied into the variable **name** until the *null* character is reached. The *null* character is not copied; instead it is replaced by a *space* by the assignment statement

```
name[i] = ' ';
```

Similarly, the *second–name* is copied into **name**, starting from the column just after the space created by the above statement. This is achieved by the assignment statement

```
name[i+j+1] = second_name{j];
```

If **first_name** contains 4 characters, then the value of **i** at this point will be 4 and therefore the first character from **second_name** will be placed in the *fifth cell* of **name**. Note that we have stored a space in the *fourth cell*.

In the same way, the statement

```
name[i+j+k+2] = last_name[k];
```

is used to copy the characters from **last_name** into the proper locations of **name**.

At the end, we place a **null** character to terminate the concatenated string **name**. In this example, it is important to note the use of the expressions **i**+**j**+**1** and **i**+**j**+**k**+**2**.

```
Program
/*************************************************************************************/
/*                          CONCATENATION OF STRINGS                                 */
/*************************************************************************************/

main( )
{
    int i,   j,   k;
    static char     first_name[10]   =  {"VISWANATH"};
    static char second_name[10]   =  {"PRATAP"};
    static char     last_name[10]   =  {"SINGH"};
    char    name[30];
```

```
    /* Copy first_name into name */
    for(i = 0; first_name[i] != '\0'; i++)
        name[i] = first_name[i];
    /* End first_name with a space */
        name[i] = ' ' ;
    /* Copy second_name into name */
    for (j = 0; second_name[j] ! = `\0'; j++)
        name [i + j + 1] = second_name[j];
    /* End second_name with a space */
    name[i + j + 1]= ' ' ;
    /* Copy last_name into name */
    for (k = 0; last_name[k] ! = `\0'; k++)
        name[i + j + k + 2] = last_name[k];
    /* End name with a null character */
    name[i+j+k+2] = '\0';
    printf("\n\n");
    printf("%s\n", name);
}
Output
VISWANATH PRATAP SINGH
```

Fig. 9.8 *Concatenation of strings*

9.7 COMPARISON OF TWO STRINGS

Once again, C does not permit the comparison of two strings directly. That is, the statements such as

```
if(name 1  == name2)
if(name    == "ABC")
```

are not permitted. It is therefore necessary to compare the two strings to be tested, character by character. The comparison is done until there is a mismatch or one of the strings terminates into a *null* character, whichever occurs first. The following segment of a program illustrates this.

```
----------
i=0;
while(str1[i] == str2[i] && str1[i] !='\0'
          && str2[i] !='\0')
    i = i+1;
if (str1[i] == '=0' && str2[i] == '\0')
    printf("strings are equal\n");
else
    printf("strings are not equal\n");
----------
```

9.8 STRING-HANDLING FUNCTIONS

Fortunately, the C library supports a large number of string-handling functions that can be used to carry out many of the string manipulations discussed so far. Following are the most commonly used string-handling functions.

Function	Action
strcat()	concatenates two strings
strcmp()	compares two strings
strcpy()	copies one string over another
strlen()	finds the length of a string

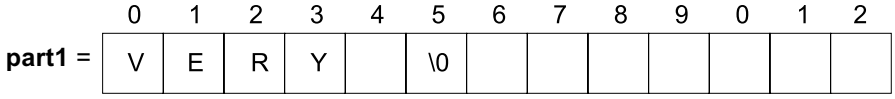
We shall discuss briefly how each of these functions can be used in the processing of strings.

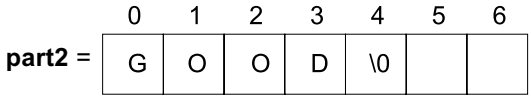
strcat( ) Function

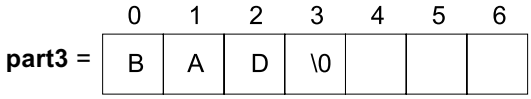
The **strcat** function joins two strings together. It takes the following form:

```
strcat(string 1, string2);
```

string1 and **string2** are character arrays. When the function **strcat** is executed, **string2** is appended to **string1**. It does so by removing the *null* character at the end of **string1** and placing **string2** from there. The string at **string2** remains unchanged. For example, consider the following three strings:



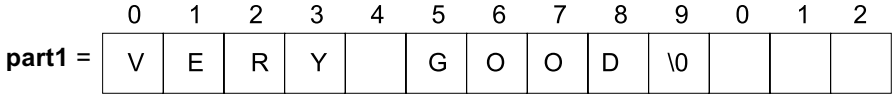


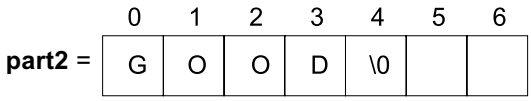


Execution of the statement

```
strcat(part1, part2);
```

will result in:

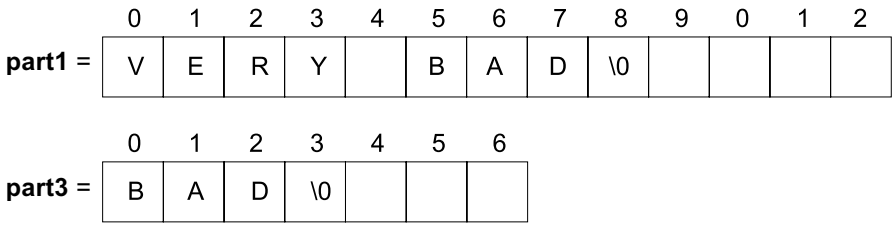




while the statement

```
strcat(part1, part3);
```

will result in:



We must make sure that the size of **string1** (to which **string2** is appended) is large enough to accommodate the final string.

strcat function may also append a string constant to a string variable. The following is valid.

```
strcat(part1, "GOOD");
```

C permits nesting of **strcat** functions. For example, the statement

```
strcat(strcat(string1,string2), string3);
```

is allowed and concatenates all the three strings together. The resultant string is stored in **string1**.

strcmp( ) Function

The **strcmp** function compares two strings identified by the arguments and has a value 0 if they are equal. If they are not, it has the numeric difference between the first nonmatching characters in the strings. It takes the form:

```
strcmp(string1, string2);
```

string1 and **string2** may be string variables or string constants. Examples are:

```
strcmp(name1, name2);
strcmp(name1, "John");
strcmp("Rom", "Ram");
```

Our major concern is to determine whether the strings are equal; if not, which is alphabetically above. The value of the mismatch is rarely important. For example, the statement

```
strcmp("their", "there");
```

will return a value of –9 which is the numeric difference between ASCII "i" and ASCII "r". That is, "i" minus "r" in ASCII code is –9. If the value is negative, **string1** is alphabetically above **string2**.

strcpy( ) Function

The **strcpy** function works almost like a string-assignment operator. It takes the form

```
strcpy(string1, string2);
```

and assigns the contents of **string2** to **string1**. **string2** may be a character array variable or a string constant. For example, the statement

```
strcpy(city, "DELHI");
```

will assign the string "DELHI" to the string variable **city**. Similarly, the statement

```
strcpy(city1, city2);
```

will assign the contents of the string variable **city2** to the string variable **city1.** The size of the array **city1** should be large enough to receive the contents of **city2.**

strlen() Function

This function counts and returns the number of characters in a string.

```
n = strlen(string);
```

where **n** is an integer variable which receives the value of the length of the **string**. The argument may be a string constant. The counting ends at the first *null* character.

Example 9.8

s1, **s2**, and **s3** are three string variables. Write a program to read two string constants into **s1** and **s2** and compare whether they are equal or not. If they are not, join them together. Then copy the contents of **s1** to the variable **s3**. At the end, the program should print the contents of all the three variables and their lengths.

The program is shown in Fig. 9.9. During the first run, the input strings are "New" and "York". These strings are compared by the statement

```
x = strcmp(s1, s2);
```

Since they are not equal, they are joined together and copied into **s3** using the statement

```
strcpy(s3, s1);
```

The program outputs all the three strings with their lengths.

During the second run, the two strings **s1** and **s2** are equal, and therefore, they are not joined together. In this case all the three strings contain the same string constant "London".

9.9 TABLE OF STRINGS

We often use lists of character strings, such as a list of names of students in a class, list of names of employees in an organization, list of places, etc. A list of names can be treated as a table of strings and a two-dimensional character array can be used to store the entire list. For example, a character array **student[30][15]** may be used to store a list of 30 names, each of length not more than 15 characters. Shown below is a table of five cities:

C	h	a	n	d	i	g	a	r	h
M	a	d	r	a	s				
A	h	m	e	d	a	b	a	d	
H	y	d	e	r	a	b	a	d	
B	o	m	b	a	y				

This table can be conveniently stored in a character array **city** by using the following declaration:

```
static char city[ ] [ ]
                    {
                        "Chandigarh",
                        "Madras",
                        "Ahmedabad",
                        "Hyderabad",
                        "Bombay"
                        };
```

To access the name of the *i*th city in the list, we write

```
city[i–1]
```

Program

```
/*****************************************************************************/
/*          ILLUSTRATIONS OF STRING-HANDLING FUNCTIONS                       */
/*****************************************************************************/

#include      <string.h>

main( )
{
    char   s1[20],     s2[20],     s3[20];
    int    x,  |1,     |2,  |3;

    printf("\n\nEnter two string constants \n");
    printf("?");
    scanf("%s %s", s1, s2);

    /* comparing s1 and s2 */

    x = strcmp(s1, s2);
    if(x != 0)
    {
        printf("\n\nStrings are not equal \n");
        strcat(s1, s2);    /* joining s1 and s2 */
    }
    else
        printf("\n\nStrings are equal \n");

    /* copying s1 to s3 */

    strcpy(s3, s1);

    /* Finding length of strings */
```

```
    |1 = strlen(s1);
    |2 = strlen(s2);
    |3 = strlen(s3);
    /* output */
    printf("\ns1 = % s\t length = %d characters\n", s1, |1)
    printf("s2 = %s\t length = %d characters \n", s2, |2);
    printf("s3 = %s\t length = %d characters\n", s3, |3);
}
Output
Enter two string constants
? New York
Strings are not equal

s1 = New York        length = 7 characters
s2 = York            length = 4 characters
s3 = New York        length = 7 characters

Enter two string constants
? London      London

Strings are equal

s1 = London          length = 6 characters
s2 = London          length = 6 characters
s3 = London          length = 6 characters
```

Fig. 9.9 *Illustration of string handling functions*

and therefore **city[0]** denotes "Chandigarh", **city[1]** denotes "Madras" and so on. This shows that once an array is declared as two-dimensional, it can be used like a one-dimensional array in further manipulations. That is, the table can be treated as a column of strings.

Example 9.9

Write a program that would sort a list of names in alphabetical order.

A program to sort the list of strings in alphabetical order is given in Fig. 9.10. It employs the method of *bubble* sorting described in Case Study 1 in the previous chapter.

```
Program
/*************************************************************************************/
/*            SORTING STRINGS IN ALPHABETICAL ORDER                                  */
/*************************************************************************************/

#define ITEMS       5
#define MAXCHAR  20

main( )
{
    char    string[ITEMS][MAXCHAR], dummy[MAXCHAR];
    int     i = 0,   j = 0;
    /* Reading the list */

    printf ("Enter names of %d items \n ",ITEMS);
    while (i < ITEMS)
        scanf ("%s", string[i+ +]);
```

```
    /* Sorting begins */

    for (i=1; i < ITEMS; i++) /* Outer loop begins */
    {
        for (j=1; j <=ITEMS–i; j++) /*Inner loop begins*/
        {
            if (strcmp (string[j–1], string[j]) > 0)
            {
                /* Exchange of contents */
                strcpy (dummy, string[j–1]);
                strcpy (string[j–1], string[j]);
                strcpy {string[j], dummy);

            }

        } /* Inner loop ends */

    } /* Outer loop ends */

    /* Sorting completed */

    printf ("\nAlphabetical list \n\n");
    for (i=0; i < ITEMS; i++)
        printf ("%s", string[i]);
}
```

Output

```
Enter names of 5 items
London Manchester Delhi Paris Moscow

Alphabetical list

Delhi
London
Manchester
Moscow
Paris
```

Fig. 9.10 *Sorting of strings*

Note that a two-dimensional array is used to store the list of strings. Each string is read using a **scanf** function with **%s** format. Remember, if any string contains a white space, then the part of the string after the white space will be treated as another item in the list by the **scanf**. In such cases, we should read the entire line as a string using a suitable algorithm.

CASE STUDIES

1. Counting Words in a Text

One of the practical applications of string manipulations is counting the words in a text. We assume that a word is a sequence of any characters, except escape characters and blanks, and that two words are separated by one blank character. The algorithm for counting words is as follows:

1. Read a line of text.
2. Beginning from the first character in the line, look for a blank. If a blank is found, increment **words** by 1.
3. Continue *steps* 1 and 2 until the last line is completed.

The implementation of this algorithm is shown in Fig. 9.11. The first **while** loop will be executed once for each line of text. The end of text is indicated by pressing the 'Return' key an *extra time* after the entire text has been entered. The extra 'Return' key causes a *newline* character as input to the last line and as a result, the last line contains only the *null* character.

The program checks for this special line using the statement

```
if (line[0] = = '\0')
```

and if the first (and only the first) character in the line is a *null* character, then counting is terminated. Note the difference between a *null* character and a blank character.

The program also counts the number of lines read and the total number of characters in the text. Remember, the last line containing the *null* string is not counted.

After the first **while** loop is exited, the program prints the results of counting.

Program

```
/*****************************************************************************/
/*       COUNTING CHARACTERS, WORDS AND LINES IN A TEXT          */
/*****************************************************************************/

#include      <stdio.h>

main( )
{
    char   line[81], ctr;
    int    i,c,
           end = 0,
           characters = 0,
           words = 0,
           lines = 0;

    printf("KEY IN THE TEXT. \n");
    printf("GIVE ONE SPACE AFTER EACH WORD.\n");
    printf("WHEN COMPLETED, PRESS 'RETURN'.\n\n");

    while(end = = 0)
    {
          /* Reading a line of text */

          c = 0;
          while((ctr=getchar( )) != '\n')
              line[c++] = ctr;
          line[c] = '\0';

          /* counting the words in a line */

          if(line[0] =='\0')
              break;
          else
          {
              words ++;
              for(i=0; line[i] != '\0';i++)
                  if(line[i] == ' ' ||line[i] == '\t')
```

```
                    words + +;
                }
                /* counting lines and characters */
                lines = lines + 1;
                characters = characters + strlen(line);
        }
        printf ("\n");
        printf("Number of lines = %d\n", lines);
        printf("Number of words = %d/n", words);
        printf("Number of characters = %d\n", characters);
}
```

Output

```
KEY IN THE TEXT.
GIVE ONE SPACE AFTER EACH WORD.
WHEN COMPLETED, PRESS 'RETURN'.

Admiration is a very short-lived passion.
Admiration involves a glorious obliquity of vision.
Always we like those who admire us but we do not
like those whom we admire.
Fools admire, but men of sense approve.

Number of lines = 5
Number of words = 36
Number of characters = 205
```

Fig. 9.11 *Counting of characters, words and lines in a text*

2. Processing of a Customer List

Telephone numbers of important customers are recorded as follows:

Full name	*Telephone number*
Joseph Louis Lagrange	869245
Jean Robert Argand	900823
Carl Freidrich Gauss	806788
---	---
---	---

It is desired to prepare a revised alphabetical list with surname (last name) first, followed by a comma and the initials of the first and middle names. For example,

Argand,J.R:

We create a table of strings, each row representing the details of one person, such as first_name, middle_name, last_name, and telephone_number. The columns are interchanged as required and the list is sorted on the last_name. Figure 9.12 shows a program to achieve this.

Program

```
/*****************************************************************************/
/*                        PROCESSING OF CUSTOMER LIST                        */
/*****************************************************************************/

#define    CUSTOMERS    10
main( )
{
    char    first_name[20][10], second_name[20][10],
            surname[20][10], name[20][20],
            telephone[20][10], dummy[20];

    int     i,j;

    printf("Input names and telephone numbers \n");
    printf("?");
    for(i=0; i < CUSTOMERS; i++)
    {
        scanf("%s %s %s %s", first_name[i],
            second_name[i], surname[i], telephone[i]);
        /* converting full name to surname with initials */

        strcpy(name[i], surname[i]);
        strcat(name[i], ",");
        dummy[0] = first_name[i][0];
        dummy[1] = '\0';
        strcat(name[i], dummy);
        strcat(name[i], ".");
        dummy[0] = second_name[i][0];
        dummy[1] = '\0';
        strcat(name[i], dummy);
    }
    /* Alphabetical ordering of surnames */

        for(i=1; i<= CUSTOMERS–1; i++)
            for[j=1; j<=CUSTOMERS–i; j++)
                if(strcmp (name[j–1], name[j]) > 0)
                {
                /*  Swaping names */
                    strcpy(dummy, name[j–1]);
                    strcpy(name[j–1], name[j]);
                    strcpy(name[j], dummy);

                /*  Swaping telephone numbers */
                    strcpy(dummy, telephone[j–1]);
                    strcpy(telephone[j–1], telephone[j]);
                    strcpy(telephone[j], dummy);

                }

    /* printing alphabetical list */
    printf("=nCUSTOMERS LIST IN ALPHABETICAL ORDER \n\n");
    for(i=0; i < CUSTOMERS; i++)
    printf(" %–20s\t %–10s/n"), name[i], telephone[i];
}
```

Output

```
Input names and telephone numbers
?Gottfried Wilhelm Leibniz 711518
Joseph Louis Lagrange 869245
```

```
Jean Robert Argand 900823
Carl Freidrich Gauss 806788
Simon Denis Poisson 853240
Friedrich Wilhalm Bessel 719731
Charles Francois Sturm 222031
George Gabriel Stokes 545454
Mohandas Karamchand Gandhi 362718
Josian Willard Gibbs 123145

CUSTOMERS LIST IN ALPHABETICAL ORDER

Argand,J.R    900823
Bessel,F.W    719731
Gandhi,M.K    362718
Gauss,C.F     806788
Gibbs,J.W     123145
Lagrange,J.L  869245
Leibniz,G.W   711518
Poisson,S.D   853240
Stokes,G.G    545454
Sturm,C.F     222031
```

Fig. 9.12 *Program to alphabetize a customer list*

Review Questions and Exercises

9.1 Describe the limitations of using **getchar** and **scanf** functions for reading srings.

9.2 Character strings in C are automatically terminated by the *null* character. Explain how this feature helps in string manipulations.

9.3 String variables can be assigned values in three ways:

(a) During type declaration

```
static char string[ ] = {".....”};
```

(b) Using **strcpy** function

```
strcpy(string, ".....");
```

(c) Reading-in a string using **scanf** function

```
scanf("%s", string);
```

Compare them critically and describe situations where one is superior to the others.

9.4 Assuming the variable **string** contains the value "The sky is the limit" determine what output of the following program segments will be.

```
(a) printf("%s", string);
(b) printf("%25.10s", string);
(c) printf("%s", string[0]);
(d) for (i=0; string[i] != ".”; i+ +)
        printf("%c", string[i]);
(e) for (i=0; string[i] != '\0'; i+ +;)
        printf("%d\n", string[i]);
```

(f)
```
for (i=0; i <=strlen[string]; ;)
{
    string[i++] = i;
    printf("%s\n", string[i]);
}
```

(g)
```
printf("%c\n", string[10] + 5);
```

(h)
```
printf("%c\n", string[10] + '5');
```

9.5 Write a program which reads your name from the keyboard and outputs a list of ASCII codes which represent your name.

9.6 Write a program to do the following:
- (a) To output the question "Who is the inventor of C?"
- (b) To accept an answer.
- (c) To print out "Good" and then stop, if the answer is correct.
- (d) To output the message "try again", if the answer is wrong.
- (e) To display the correct answer when the answer is wrong even at the third attempt and stop.

9.7 Write a program to extract a portion of a character string and print the extracted string. Assume that *m* characters are extracted, starting with the *n*th character.

9.8 Write a program which will read a text and count all occurrences of a particular word.

9.9 Write a program which will read a string and rewrite it in the alphabetical order. For example, the word STRING should be written as GINRST.

9.10 Write a program to replace a particular word by another word in a given string. For example, the word "PASCAL" should be replaced by "C" in the text "It is good to program in PASCAL language."

9.11 A Maruti car dealer maintains a record of sales of various vehicles in the following form:

Vehicle type	*Month of sales*	*Price*
MARUTI-800	02/87	75000
MARUTI-DX	07/87	95000
GYPSY	04/88	110000
MARUTI-VAN	08/88	85000

Write a program to read this data into a table of strings and output the details of a particular vehicle sold during a specified period. The program should request the user to input the vehicle type and the period (starting month, ending month).

Chapter 10

User-Defined Functions

10.1 INTRODUCTION

We have mentioned earlier that one of the strengths of C language is that C functions are easy to define and use. We have used functions in every program that we have discussed so far. However, they have been primarily limited to the three functions, namely, **main**, **printf**, and **scanf**. In this chapter, we shall consider in detail how a function is designed, how two or more functions are put together and how they communicate with one another.

10.1.1 Standard Library Functions

C functions can be classified into two categories, namely, *library* functions and *user-defined* functions. **main** is an example of user-defined functions. **printf** and **scanf** belong to the category of library functions. We have also used other library functions such as **sqrt**, **cos**, **strcat**, etc. The main distinction between these two categories is that library functions are not required to be written by us whereas a user-defined function has to be developed by the user at the time of writing a program. However, a user-defined function can later become a part of the C program library.

ANSI C Standard Library Functions

The C language is accompanied by a number of library functions that perform various tasks. The ANSI committee has standardized header files which contain these functions. What follows is a list of commonly used functions and the header files where they are defined. For a more complete list, the reader should refer to the manual of the version of C that is being used.
The header files that are included in this Appendix are:

<ctype.h>	Character testing and conversion functions
<math.h>	Mathematical functions
<stdio.h>	Standard I/O library functions
<stdlib.h>	Utility functions such as string conversion routines, memory allocation routines, random number generator, etc.

<string.h>	String manipulation functions
<time.h>	Time manipulation functions

Note: The following function parameters are used:

c - character type argument
d - double precision argument
f - file argument
i - integer argument
l - long integer argument
p - pointer argument
s - string argument
u - unsigned integer argument
An asterisk (*) denotes a pointer

Function	Data type returned	Task
<ctype.h>		
isalnum(c)	int	Determine if argument is alphanumberic. Return nonzero value if true; 0 otherwise.
isalpha(c)	int	Determine if argument is alphabetic. Return nonzero value if true; 0 otherwise.
isascii(c)	int	Determine if argument is an ASCII character. Return nonzero value if true; 0 otherwise.
iscntrl(c)	int	Determine if argument is an ASCII control character. Return nonzero value if true; 0 otherwise.
isdigit(c)	int	Determine if argument is a decimal digit. Return nonzero value if true; 0 otherwise.
isgraph(c)	int	Determine if argument is a graphic printing ASCII character. Return nonzero value if true; 0 otherwise.
islower(c)	int	Determine if argument is lowercase. Return nonzero value if true; 0 otherwise.
isodigit(c)	int	Determine if argument is an octal digit. Return nonzero value if true; 0 otherwise.
isprint(c)	int	Determine if argument is a printing ASCII character. Return nonzero value if true; 0 otherwise.
ispunct(c)	int	Determine if argument is a punctuation character. Return nonzero value if true; 0 otherwise.
isspace(c)	int	Determine if argument is a whitespace character. Return nonzero value if true; 0 otherwise.
isupper(c)	int	Determine if argument is uppercase. Return nonzero value if true; 0 otherwise.
isxdigit(c)	int	Determine if argument is a hexadecimal digit. Return nonzero value if true; 0 otherwise.
toascii(c)	int	Convert value of argument to ASCII.
tolower(c)	int	Convert letter to lowercase.
toupper(c)	int	Convert letter to uppercase.

(*Contd.*)

Function	Data type returned	Task
<math.h>		
acos(d)	double	Return the arc cosine of d.
asin(d)	double	Return the arc sine of d.
atan(d)	double	Return the arc tangent of d.
atan2(d1,d2)	double	Return the arc tangent of d1/d2.
ceil(d)	double	Return a value rounded up to the next higher integer.
cos(d)	double	Return the cosine of d.
cosh(d)	double	Return the hyperbolic cosine of d.
exp(d)	double	Raise e to the power d.
fabs(d)	double	Return the absolute value of d.
floor(d)	double	Return a value rounded down to the next lower integer.
fmod(d1,d2)	double	Return the remainder of d1/d2 (with same sign as d1).
labs(l)	long int	Return the absolute value of l.
log(d)	double	Return the natural logarithm of d.
log10(d)	double	Return the logarithm (base 10) of d.
pow(d1,d2)	double	Return d1 raised to the d2 power.
sin(d)	double	Return the sine of d.
sinh(d)	double	Return the hyperbolic sine of d.
sqrt(d)	double	Return the square root of d.
tan(d)	double	Return the tangent of d.
tanh(d)	double	Return the hyperbolic tangent of d.
<stdio.h>		
fclose(f)	int	Close file f. Return 0 if file is successfully closed.
feof(f)	int	Determine if an end-of-file condition has been reached. If so, return a non zero value; otherwise, return 0.
fgetc(f)	int	enter a single character from file f.
fgets(s, i, f)	char*	Enter string s, containing i characters, from file f.
fopen(s1,s2)	file*	Open a file named s1 of type s2. Return a pointer to the file.
fprintf(f,...)	int	Send data items to file f.
fputc(c,f)	int	Send a single character to file f.
fputs(s,f)	int	Send string s to file f.
fread(s,i1,i2,f)	int	Enter i2 data items, each of size i1 bytes, from file f to string s.
fscant(f,...)	int	Enter data items from file f.
fseek(f,1,i)	int	Move the pointer for file f a distance 1 bytes from location i.
ftell(f)	long int	Return the current pointer position within file f.
fwrite(s,i1,i2,f)	int	Send i2 data items, each of size i1 bytes from string s to file f.
getc(f)	int	Enter a single character from file f.
getchar(void)	int	Enter a single character from the standard input device.
gets(s)	char*	Enter string s from the standard input device.
printf(...)	int	Send data items to the standard output device.
putc(c,f)	int	Send a single character to file f.
putchar(c)	int	Send a single character to the standard output device.
puts(s)	int	Send string s to the standard output device.
rewind(f)	void	Move the pointer to the beginning of file f.

(*Contd.*)

Function	Data type returned	Task
scanf(...)	int	Enter data items from the standard input device.
<stdlib.h>		
abs(i)	int	Return the absolute value of i.
atof(s)	double	Convert string s to a adouble-precision quantity.
atoi(s)	int	Convert string s to an integer.
atol(s)	long	Convert string s to a long integer.
calloc(u1,u2)	void*	Allocate memory for an array having u1 elements, each of length u2 bytes. Return a pointer to the beginning of the allocated space.
exit(u)	void	Close all files and buffers, and terminate the program. (Value of u is assigned by the function, to indicate termination status.)
free(p)	void	Free a block of allocated memory whose beginning is indicated by p.
malloc(u)	void*	Allocate u bytes of memory. Return a pointer to the beginning of the allocated space.
rand(void)	int	Return a random positive integer.
realloc(p, u)	void*	Allocate u bytes of new memory to the pointer variable p. Return a pointer to the beginning of the new memory space.
srand(u)	void	Initialize the random number generator.
system(s)	int	Pass command string s to the operating system. Return 0 if the command is successfully executed; otherwise, return a nonzero value typically—1.
<string.h>		
strcmp(s1,s2)	int	Compare two strings lexicographically. Return a negative value if s1 < s2; 0 if s1 and s2 are identical; and a positive value if s1 > s2.
strcmpi(s1, s2)	int	Compare two strings lexicographically, without regard to case. Return a negative value if s1 < s2; 0 if s1 and s2 are identical; and a value if s1 > s2.
strcpy(s1, s2)	char*	Copy string s2 to string s1.
strlen(s)	int	Return the number of characters is string s.
strset(s, c)	char*	Set all characters within s to c (excluding the terminating null character\0).
<time.h>		
difftime(11,12)	double	Return the time difference 11 ~ 12. Where 11 and 12 represent elapsed times beyond a designated base time (see the time function).
time(p)	long int	Return the number of seconds elapsed beyond a designated base time.

10.2 NEED FOR USER-DEFINED FUNCTIONS

As pointed out earlier, **main** is a specially recognized function in C. Every program must have a **main** function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only **main** function, it leads to a number of problems. The program may become too large and complex and as a result the task of debugging, testing, and maintaining becomes difficult. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. These subprograms called 'functions' are much easier to understand, debug, and test.

There are times when some type of operation or calculation is repeated at many points throughout a program. For instance, we might use the factorial of a number at several points in the program. In such

situations, we may repeat the program statements wherever they are needed. Another approach is to design a function that can be called and used whenever required. This saves both time and space.

This sub-sectioning approach clearly results in a number of advantages.

1. It facilitates top-down modular programming as shown in Fig. 10.1. In this programming style, the high level logic of the overall problem is solved first while the details of each lower-level function are addressed later.
2. The length of a source program can be reduced by using functions at appropriate places. This factor is particularly critical with microcomputers where memory space is limited.
3. As mentioned earlier, it is easy to locate and isolate a faulty function for further investigations.
4. A function may be used by many other programs. This means that a C programmer can build on what others have already done, instead of starting over, from scratch.

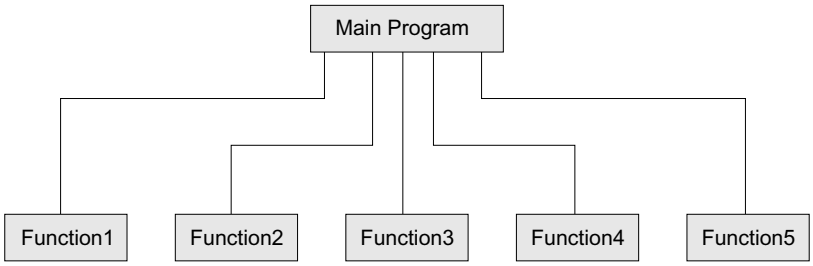


Fig. 10.1 *Top-down modular programming, using functions*

10.3 A MULTI-FUNCTION PROGRAM

A function is a self-contained block of code that performs a particular task. Once a function has been designed and packed, it can be treated as a 'black box' that takes some data from the main program and returns a value. The inner details of operation are invisible to the rest of the program. All that the program knows about a function is: What goes in and what comes out. Every C program can be designed using a collection of these black boxes.

Consider a set of statements as shown below:

```
printline( )
{
    int i;
    for (i=1; i<40; i++)
        printf("–");
    printf("\n");
}
```

The above set of statements defines a function called **printline** which could print a line of 39-character length. This function can be used in a program as follows:

```
main( )
{
    printline( );
    printf("This illustrates the use of C functions \n");
    printline( );
}
printline( )
{
```

```
    int i;
    for(i=1; i<40; i++)
        printf("–");
        printf("\n");
}
```

This program will print the following output:

```
This illustrates the use of C functions
```

The above program contains two user-defined functions:

main()
printline()

As we know, the program execution always begins with the **main** function. During execution of the **main**, the first statement encountered is

printline();

which indicates that the function **printline** is to be executed. At this point, the program control is transferred to the function **printline**. After executing the **printline** function which outputs a line of 39 character length, the control is transferred back to the **main**. Now, the execution continues at the point where the function call was executed. After executing the **printf** statement, the control is again transferred to the printline function for printing the line once more.

The **main** function calls the user-defined **printline** function two times and the library function **printf** once. We may notice that the **printline** function itself calls the library function **printf** repeatedly.

Any function can call any other function. In fact, it can call itself. A 'called function' can also call another function. A function can be called more than once. In fact, this is one of the main features of using functions. Figure 10.2 illustrates the flow of control in a multi-function program.

Except the starting point, there are no other predetermined relationships, rules of precedence, or hierarchies among the function that make up a complete program. The functions can be placed in any order. A called function can be placed either before or after the calling function. However, it is the usual practice to put all the called functions at the end.

Important Note

During the last few years of applications of C, the way the functions have been defined and declared has undergone changes. The major change is the way the function arguments are declared. There are now two established methods of declaring the parameters. The older method (known as "classic" method) declares function arguments separately after the definition of function name (known as *function header*). The newer method (known as "modern" or ANSI method) combines the function definition and arguments declaration in one line. Luckily the modern compilers support both the methods. That is, we can write a function using either of them and execute it successfully.

Since a large number of existing programs and functions have been written using the older method, we should be familiar with this approach. The sections that follow (Secs 10.4 through 10.15) will discuss and use the classic form of function definitions and declarations. The modern ANSI method is discussed in Sec. 10.16.

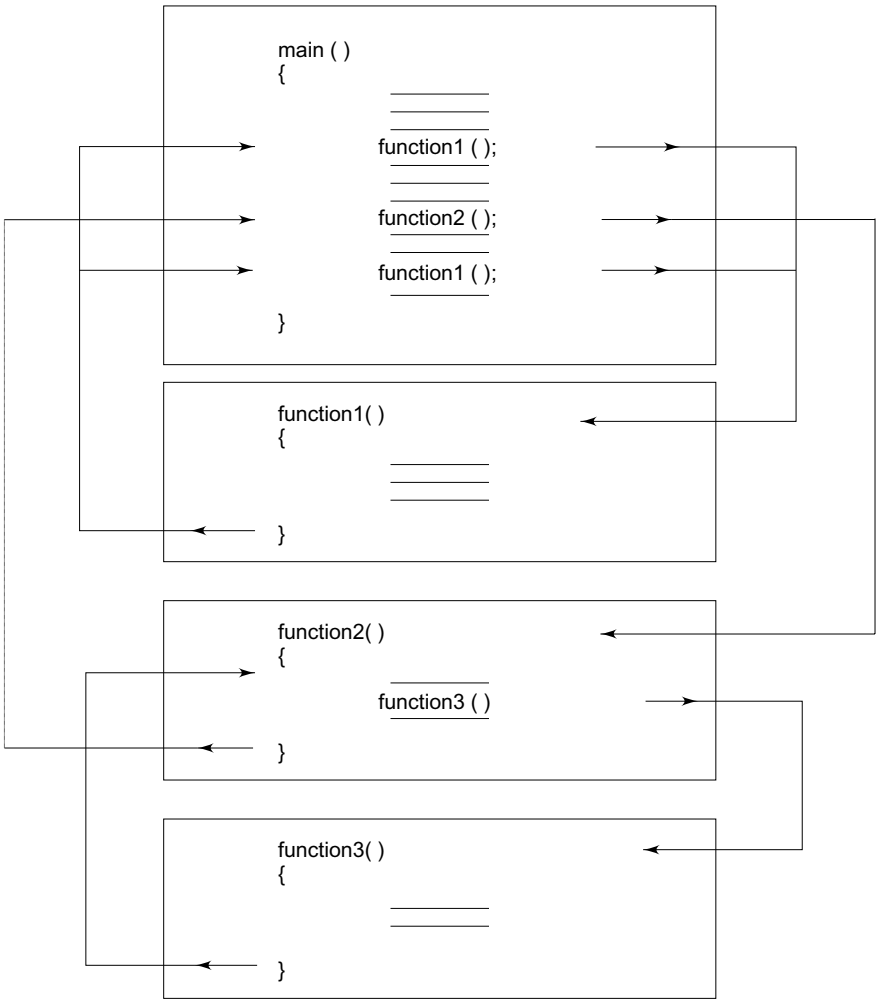


Fig. 10.2 *Flow of control in a multi-function program*

10.4 THE FORM OF C FUNCTIONS

All functions have the form:

```
function-name (argument list)
argument declaration;
{
local variable declarations:
executable statement1;
executable statement2;
---------
---------
return (expression);
}
```

All parts are not essential. Some may be absent. For example, the argument list or parameter list and its associated argument declaration parts are optional. We may recall that the **main** functions discussed thus far have not included any arguments. Same is the case with the **printline** function discussed earlier.

Similarly, the declaration of local variables is required only when any local variables are used in the function. In the example discussed in the previous section, the **main** function does not use any local variables, while the **printline** function uses one local variable **i**.

A function can have any number of executable statements. A function that does nothing, may not include any executable statements at all. For example:

```
do_nothing( ) { }
```

The **return** statement is the mechanism for returning a value to the calling function. This is also an optional statement. Its absence indicates that no value is being returned to the calling function.

Function Name

A function must follow the same rules of formation as other variable names in C. Additional care must be taken to avoid duplicating library routine names or operating system commands.

Parameter List

The parameter list contains valid variable names separated by commas. The list must be surrounded by parentheses. Note that no semicolon follows the closing parenthesis. The argument variables receive values from the calling function, thus providing a means for data communication from the calling function to the called function. Some examples of functions with parameters are:

```
quadratic(a,b,c)
power(x,n)
mul(a,b)
square(y)
copy(name1, name2)
```

All parameter variables must be declared for their types after the function header and before the opening brace of the function body. Example:

```
power(x,n)
float x;
int   n;
{
   ---------
   ---------
   ---------
}
```

10.5 RETURN VALUES AND THEIR TYPES

Does a function send back any information to the calling function? If so, how does it achieve this?

A function may or may not send back any value to the calling function. If it does, it is done through the **return** statement. While it is possible to pass to the called function any number of values the called function can only return *one value* per call, at the most.

The **return** statement can take one of the following forms

```
return;
or
return(expression);
```

The first, the 'plain' **return** does not return any value; it acts much as the closing brace of the function. When a **return** is encountered, the control is immediately passed back to the calling function. An example of the use of a simple **return** is as follows:

```
if(error)
   return;
```

The second form of **return** with an expression returns the value of the expression. For example, the function

```
mul(x,y)
int    x,y;
{
   int p;
   p = x*y;
   return(p);
}
```

returns the value of **p** which is the product of the values of **x** and **y**. The last two statements can be combined into one statement as follows:

```
return (x*y);
```

A function may have more than one **return** statements. This situation arises when the value returned is based on certain conditions. For example:

```
if(x < = 0)
   return(0);
else
   return(1);
```

What type of data does a function return? All functions by default return **int** type data. But what happens if a function must return some other type? We can force a function to return a particular type of data by using a *type specifier* in the function header. Example:

```
double product(x,y)
float sqr_root(p)
```

When a value is returned, it is automatically cast to the function's type. In functions that do computations using **double**, yet return **int**, the returned value will be truncated to an integer.

10.6 CALLING A FUNCTION

A function can be called by simply using the function name in a statement. Example:

```
main( )
{
    int p;
    p = mul(10,5);
    printf("%d\n", p);
}
```

When the compiler encounters a function call, the control is transferred to the function **mul(x,y)**. This function is then executed line by line as described and a value is returned when a **return** statement is encountered. This value is assigned to **p**.

A function which returns a value can be used in expressions like any other variable. Each of the following statements is valid:

```
printf("%d\n", mul(p,q));
y = mul(p,q) / (p+q);
if (mul(m,n)>total) printf("large");
```

However, a function cannot be used on the right side of an assignment statement. For instance,

```
mul(a,b) = 15;
```

is invalid.

A function that does not return any value may not be used in expressions; but can be called to perform certain tasks specified in the function. The function **printline( )** discussed in Sec. 10.3 belongs to this category. Such functions may be called in by simply stating their names as independent statements. Example:

```
main( )
{
    printline( );
}
```

Note the presence of a semicolon at the end.

10.7 CATEGORY OF FUNCTIONS

A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories:

Category 1: Functions with no arguments and no return values
Category 2: Functions with arguments and no return values
Category 3: Functions with arguments and return values.

In the sections to follow, we shall discuss these categories with examples.

10.8 NO ARGUMENTS AND NO RETURN VALUES

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function. In effect, there is no data transfer between the calling function and the called function. This is depicted in Fig. 10.3. The dotted lines indicate that there is only a transfer of control but not data.

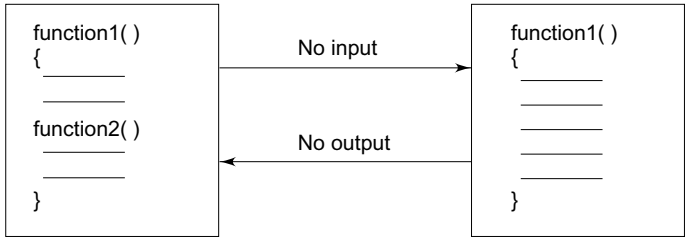


Fig. 10.3 *No data communication between functions*

As pointed out earlier, a function that does not return any value cannot be used in an expression. It can only be used as an independent statement.

Example 10.1
Write a program with multiple functions that do not communicate any data between them.

A program with three user-defined functions is given in Fig. 10.4. **main** is the calling function that calls **printline** and **value** functions. Since both the called functions contain no arguments, there are no argument declarations. The **printline** function, when encountered, prints a line with a length of 35 characters as prescribed in the function. The **value** function calculates the value of principal amount after a certain period of years and prints the results. The following equation is evaluated repeatedly:

value = principal(1 + interest-rate)

```
Program
/*************************************************************************************/
/*                                                                                   */
        FUNCTIONS WITH NO ARGUMENTS, NO RETURN VALUES
/*************************************************************************************/

/*
                                % % % % % % % % % %
                                %          main function          %
                                % % % % % % % % % %
main()                                                                          */
{
    printline( );
    value( );
    printline( );
}
/*      Function1: printline( )             */
printline()     /* contains no arguments */
{
    int i;
    for(i=1; i < = 35; i++)
        printf("%c", '–');
        printf("\n");
}
/*      Function2: value( )                 */
value ( )   /* contains no arguments    */
{
    int    year, period;
    float  inrate, sum, principal;
```

```
    printf("Principal amount?");
    scanf("%f", &principal);
    printf("Interest rate?      ");
    scanf("%f", &inrate);
    printf("Period?             ");
    scanf("%d", &period);
    sum = principal;
    year = 1;
    while(year <= period)
    {
        sum = sum *(1 + inrate);
        year = year + 1;
    }
    printf("\n%8.2f %5.2f %5d % 12.2f\n"
        principal, inrate, period, sum);
}
```

Output

```
Principal amount? 5000
Interest rate?    0.12
Period?           5
5000.00   0.12        5      8811.71
```

Fig. 10.4 *Functions with no arguments and no return values*

It is important to note that the function **value** receives its data directly from the terminal. The input data includes principal amount, interest rate and the period for which the final value is to be calculated. The **while** loop calculates the final value and the results are printed by the library function **printf**. When the closing brace of **value**() is reached, the control is transferred back to the calling function **main**. Since everything is done by the **value** itself there is in fact nothing left to be sent back to the called function.

Note that no **return** statement is employed. When there is nothing to be returned, the **return** statement is optional. The closing brace of the function signals the end of execution of the function, thus returning the control, back to the calling function.

10.9 ARGUMENTS BUT NO RETURN VALUES

In Fig. 10.4 the **main** function has no control over the way the functions receive input data. For example, the function **printline** will print the same line each time it is called. Same is the case with the function, **value**. We could make the calling function to read data from the terminal and pass it on to the called function. This approach seems to be wiser because the calling function can check for the validity of data, if necessary, before it is handed over to the called function.

The nature of data communication between the calling function and the called function with arguments but no return value is shown in Fig. 10.5.

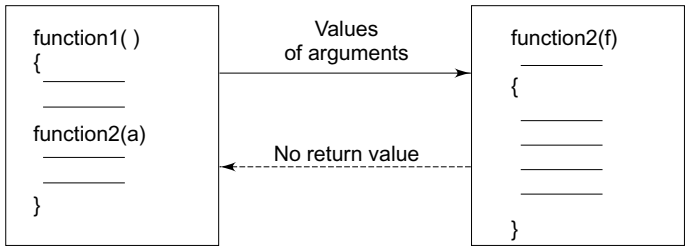


Fig. 10.5 *One-way data communication*

We shall modify declaration of both the called functions to include arguments as follows:

```
printline(ch)
value(p,r,n)
```

The arguments **ch**, **p**, **r**, and **n** are called the *formal arguments*. The calling function can now send values to these arguments using function calls containing appropriate arguments. For example, the function call

```
value(500,0.12,5)
```

would send the values 500,0.12 and 5 to the function

```
value(p,r,n)
```

and assign 500 to **p**, 0.12 to **r** and 5 to **n**. The values 500,0.12, and 5 are the *actual arguments* which become the values of the *formal arguments* inside the called function.

The *actual* and *formal* arguments should match in number, type, and order. The values of actual arguments are assigned to the formal arguments on a *one to one* basis, starting with the first argument as shown in Fig. 10.6.

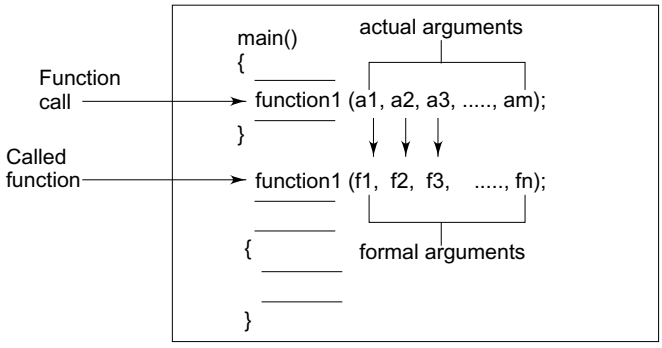


Fig. 10.6 *Arguments matching between the function call and the called function*

We should ensure that the function call has matching arguments. In case, the actual arguments are more than the formal arguments (m > n), the extra actual arguments are discarded. On the other hand, if the actual arguments are less than the formal arguments, the unmatched formal arguments are initialized to some garbage values. Any mismatch in data type may also result in passing of garbage values. Remember, no error message will be generated.

While the formal arguments must be valid variable names, the actual arguments may be variable names, expressions, or constants. The variables used in actual arguments must be assigned values before the function call is made.

Remember that, when a function call is made, *only a copy of the values of actual arguments is passed into the called function*. What occurs inside the function will have no effect on the variables used in the actual argument list.

Example 10.2

Modify the program of Example 10.1 to include the arguments in the function calls.

The modified program with function arguments is presented in Fig. 10.7. Most of the program is identical to the program in Fig. 10.4. The input prompt and **scanf** assignment statement have been moved from **value** function to **main**. The variables **principal, inrate,** and **period** are declared in **main** because they are used in **main** to receive data. The function call

```
value(principal, inrate, period);
```

passes information it contains to the function **value.**

The function header of **value** has three formal arguments **p**,**r**, and **n** which correspond to the actual arguments in the function call, namely, **principal**, **inrate**, and **period**. The formal arguments are declared immediately after the function header. On execution of the function call, the values of the actual arguments are assigned to the corresponding formal arguments. In fact, the following assignments are accomplished across the function boundaries:

```
p = principal;
r = inrate;
n = period;
```

The variables declared inside a function are known as *local* variables and therefore their values are local to the function and cannot be accessed by any other function. We shall discuss more about this later in the chapter.

The function **value** calculates the final amount for a given period and prints the results as before. Control is transferred back on reaching the closing brace of the function. Note that the function does not return any value.

The function **printline** is called twice. The first call passes the character 'Z', while the second passes the character 'C' to the function. These are assigned to the formal argument **ch** for printing lines (see the output).

10.10 ARGUMENTS WITH RETURN VALUES

The function **value** in Fig. 10.7 receives data from the calling function through arguments, but does not send back any value. Rather, it displays the results of calculations at the terminal.

Program

```
/*************************************************************************************/
/*       FUNCTIONS WITH ARGUMENTS BUT NO RETURN VALUES       */
/*************************************************************************************/

main()
{
```

```
    float principal, inrate;
    int period;

    printf("Enter principal amount, interest");
    printf("rate, and period \n");
    scanf("%f %f %d",&principal, &inrate, &period);
    printline('Z');
    value(principal,inrate,period);
    printline('C');
}
/*-----------------------------------------------------------------------*/
printline(ch)
char ch;
{
    int i;
    for(i=1; i <= 52; i++)
        printf("%c",ch);
    printf("\n");
}
/*-----------------------------------------------------------------------*/
value(p,r,n)
int n;
float p,r;
{
    int year;
    float sum;
    sum = p;
    year = 1;
    while(year <= n)
    {
        sum =sum * (1+r);
        year = year + 1;
    }
    printf("%f\t%f\%d\t%f\n", p,r,n,sum);
}
/*-----------------------------------------------------------------------*/
Output
Enter principal amount, interest rate, and period
5000   0.12   5
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
5000.000000          0.120000          5          8811.708984
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Fig. 10.7 *Functions with arguments but no return values*

However, we may not always wish to have the result of a function displayed. We may use it in the calling function for further processing. Moreover, to assure a high degree of portability between programs, a function should generally be coded without involving any I/O operations. For example, different programs may require different output formats for displaying results. These shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program.

A self-contained and independent function should behave like a 'black box' that receives a predefined form of input and outputs a desired value. Such functions will have two-way data communication as shown in Fig. 10.8.

We shall modify the program in Fig. 10.7 to illustrate the use of two-way data communication between the calling and the called functions.

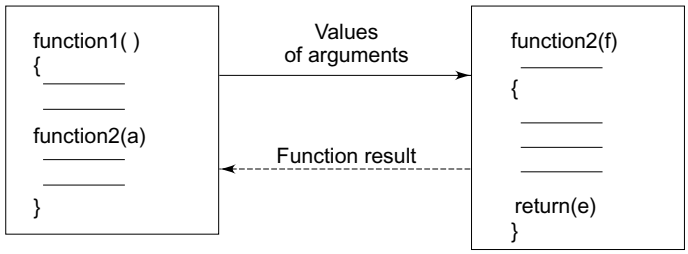


Fig. 10.8 *Two-way data communication between functions*

Example 10.3

In the program presented in Fig. 10.7, modify the function **value**, to return the final amount calculated to the **main**, which will display the required output at the terminal. Also extend the versatility of the function **printline** by having it to take the length of the line as an argument.

The modified program with the proposed changes is presented in Fig. 10.9. One major change is the movement of the **printf** statement from **value** to **main**.

The calculated value is passed on to **main** through the statement:

return(sum);

The integer value of **sum** at this point is returned to **main** and assigned to the variable **amount** by the function call

amount = value(principal,inrate,period);

The following events occur, in order, when the above function call is executed:

1. The function call transfers the control along with copies of the values of the actual arguments to the function **value** where the formal arguments **p**, **r**, and **n** are assigned the values of **principal**, **inrate**, and **period** respectively.
2. The called function **value** is executed line by line in a normal fashion until the **return(sum);** statement is encountered. At this point, the value of **sum** is passed back to the function-call in the **main** and the following indirect assignment occurs:

value(principal,inrate,period) = sum;

```
Program
/*************************************************************************************/
/*            FUNCTIONS WITH ARGUMENTS AND RETURN VALUES                             */
/*************************************************************************************/

main( )
{
    float principal, inrate, amount;
    int period;
```

```
    printf("Enter principal amount, interest");
    printf("rate, and period\n");
    scanf("%f %f %d", &principal, &inrate, &period);
    printline ('*', 52);
    amount = value (principal, inrate, period);
    printf("\n%f\t%f\t%d\t%f\n\n", principal,
                inrate,period,amount);
    printline('=',52);
}
/*-------------------------------------------------------------------------------------------*/
printline(ch,len)
int len;
char ch;
{
    int i;
    for(i=1;i<=len;i++) printf("%c",ch);
    printf("\n");
}
/*-------------------------------------------------------------------------------------------*/
value(p,r,n)
int n;
float p,r;
{
    int year;
    float sum;
    sum = p; year = 1;
    while(year <=n)
    {
        sum = sum * (1 + r);
        year = year + 1;
    }
    return(sum);
}
/*-------------------------------------------------------------------------------------------*/
```

Output

```
Enter principal amount, interest rate, and period
5000        0.12        5
*******************************************************************************************
5000.000000          0.120000                5                      8811.000000
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
```

Fig. 10.9 *Functions with arguments and return values*

3. The calling statement is executed normally and the returned value is thus assigned to **amount**.

Note that the value returned by the function is only the integer part of **sum**. Another important change is the inclusion of a second argument to **printline** function to receive the value of length of the line from the calling function. Thus, the function call

printline('*', 52);

will transfer the control to the function **printline** and assign the following values to the formal arguments **ch**, and **len**:

```
ch      = '*';
len     = 52;
```

10.11 HANDLING OF NON-INTEGER FUNCTIONS

We mentioned earlier that a C function returns a value of the type **int** as the default case when no other type is specified explicitly. For example, the function **value** of Example 10.3 does all calculations using **floats** but the return statement

```
return(sum);
```

returns only the integer part of **sum**. This is due to the absence of the *type-specifier* in the function header. In this case, we can accept the integer value of **sum** because the truncated decimal part is insignificant compared to the integer part. However, there will be times when we may find it necessary to receive the **float** or **double** type of data. For example, a function that calculates the mean or standard deviation of a set of values should return the function value in either **float** or **double**.

We must do two things to enable a calling function to receive a non-integer value from a called function:

1. The explicit type-specifier, corresponding to the data type required must be mentioned in the function header. The general form of the function definition is

```
type-specifier function-name (argument list)
argument declaration;
{
   function statements;
}
```

   The type-specifier tells the compiler, the type of data the function is to return.
2. The called function *must be declared* at the start of the body in the calling function, like any other variable. This is to tell the calling function the type of data that the function is actually returning.

The program given below illustrates the transfer of a floating-point value between functions is done in a multifunction program.

```
main( )
{
   float a,b,mul();
   double div();
   a = 12.345;
   b = 9.82;
   printf("%f\n", mul(a,b));
   printf("%if\n", div(a,b));
}
float mul(x,y)
float x,y;
{
   return(x*y);
```

```
}
double div(p,q)
double p,q;
{
    return(p/q);
}
```

The declaration part of **main** function declares not only the variables but the functions **mul** and **div** as well. This only tells the compiler that **mul** will return a float-type value and **div** a double-type value. Parentheses that follow **mul** and **div** specify that they are functions instead of variables.

It we have a mismatch between the type of data that the called function returns and the type of data that the calling function expects, we will have unpredictable results. We must, therefore, be very careful to make sure that both types are compatible.

Example 10.4

Write a function **power** that computes x raised to the power y for integers x and y and returns double-type value.

Figure 10.10 shows a **power** function that returns a **double**. The declaration

```
double power( );
```

appears in **main**, before **power** is called.

Program

```
/*************************************************************************************/
/*                              POWER FUNCTIONS                                      */
/*************************************************************************************/
main( )
{
    int x,y;
    double power( );
    printf("Enter x,y;");
    scanf("%d %d", &x,&y);
    printf("%d to power %d is %f\n", x,y,power (x,y));
}
/*-----------------------------------------------------------------------------------*/
double power(x,y) /* computes x to power y */
int x,y;
{
    double p;
    p = 1.0;  /* x to power zero */
    if(y >= 0)
     while(y--) /* computes positive powers */
        p *= x;
    else
        while(y++) /* computes negative powers */
          p /= x;
    return(p); /* returns double type */
}
/*-----------------------------------------------------------------------------------*/
```

Output

```
Enter x,y:16 2
```

```
16 to power 2 is 256.000000
Enter x,y:16 –2
16 to power –2 is 0.003906
```

Fig. 10.10 *Illustration of return of float values*

Another way to guarantee that **power's** type is declared before it is called in **main** is to define the **power** function before we define **main**. **Power's** type is then known from its definition, so we no longer need its type declaration in **main**.

Functions Returning Nothing

We have seen earlier that in many occasions, functions do not return any values. They perform only some printing or house keeping operations. Consider the program in Fig. 10.4.

```
main ( )
{
   printline();
   value( );
   printline();
}
printline()
{
   . . . .
   . . . .
}
value()
{
   . . . .
   . . . .
}
```

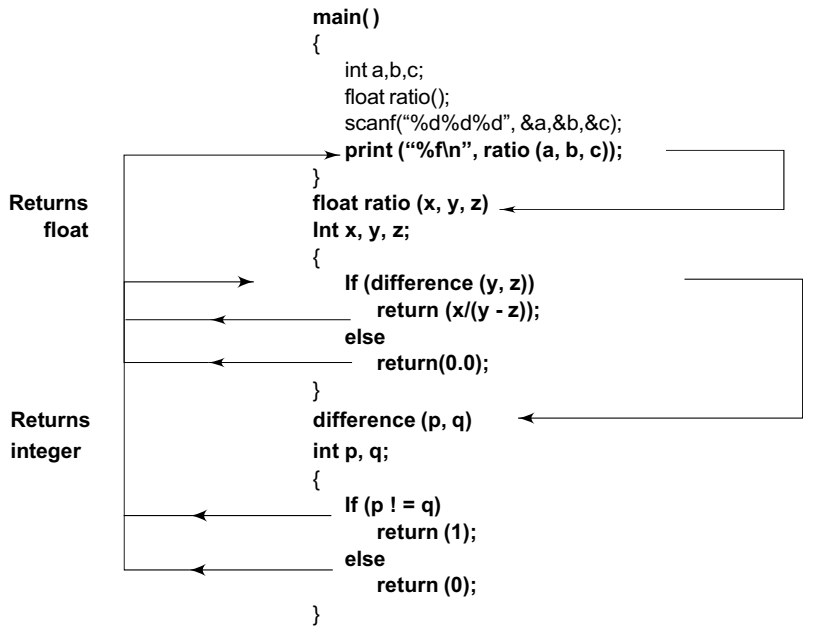
The functions **printline()** and **value()** do not return any values and therefore they were not declared in the **main**. Although the program works nicely, we can declare them in the **main** with the qualifier **void**. This states explicitly that the functions do not return values. This prevents any accidental use of these functions in expressions. The program in Fig. 10.4 may be modified as follows:

```
main()
{
   void printline();      /* declaration */
   void value();          /* declaration */
   . . . .
   . . . .
}
void printline()
{
   . . . .
   . . . .
}
void value()
```

```
{
    . . . .
    . . . .
}
```

10.12 NESTING OF FUNCTIONS

C permits nesting of functions freely. **main** can call **function1**, which calls **function2**, which calls **function3,** ....and so on. There is in principle no limit as to how deeply functions can be nested. Consider the following program:



The above program calculates the ratio

$$\frac{a}{b-c}$$

and prints the result. We have the following three functions:

main()
ratio()
difference()

main reads the values of a, b and c and calls the function **ratio** to calculate the value a/(b-c). This ratio cannot be evaluated if (b – c) = 0. Therefore, **ratio** calls another function **difference** to test whether the difference (b – c) is zero or not. **difference** returns 1, if b is not equal to c; otherwise returns zero to the function **ratio**. In turn, **ratio** calculates the value a/(b–c) if it receives 1 and returns the result in **float**. In case, **ratio** receives zero from **difference**, it sends back 0.0 to **main** indicating that (b–c) = 0.

Nesting of function calls is also possible. For example, a statement like

```
p = mul(mul(5,2),6);
```

is valid. This represents two sequential function calls. The inner function call is evaluated first and the returned value is again used as an actual argument in the outer function call. If **mul** returns the product of its arguments, then the value of **p** would be 60 (= 5 × 2 × 6).

10.13 RECURSION

When a called function in turn calls another function a process of 'chaining' occurs. *Recursion* is a special case of this process, where a function calls itself. A very simple example of recursion is presented below:

```
main()
{
   printf("This is an example of recursion\n");
   main();
}
```

When executed, this program will produce an output something like this:

```
This is an example of recursion
This an example of recursion
This is an example of recursion
This is an ex
```

Execution is terminated abruptly; otherwise the execution will continue indefinitely.

Another useful example of recursion is the evaluation of factorials of a given number. The factorial of a number n is expressed as a series of repetitive multiplications as shown below:

factorial of n = n(n–1)(n–2)....1.

For example,

factorial of 4 = 4 × 3 × 2 × 1 = 24

A function to evaluate factorial of n is as follows:

```
factorial(n)
int n;
{
   int fact;
   if (n==1)
      return(1);
   else
      fact =  n*factorial(n–1);
   return(fact);
}
```

Let us see how the recursion works. Assume n = 3. Since the value of n is not 1, the statement

```
fact = n * factorial(n–1);
```

will be executed with n = 3. That is,

```
fact = 3 * factorial(2);
```

will be evaluated. The expression on the right-hand side includes a call to **factorial** with n = 2. This call will return the following value:

```
2 * factorial(1)
```

Once again, **factorial** is called with n = 1. This time, the function returns 1. The sequence of operations can be summarized as follows:

```
fact  = 3 * factorial(2)
      = 3 * 2 * factorial(1)
      = 3 * 2 * 1
      = 6
```

Recursive functions can be effectively used to solve problems where the solution is expressed in terms of successively applying the same solution to subsets of the problem. When we write recursive functions, we must have an if statement somewhere to force the function to return without the recursive call being executed. Otherwise, the function will never return.

10.14 FUNCTIONS WITH ARRAYS

Like the values of simple variables, it is also possible to pass the values of an array to a function. To pass an array to a called function, it is sufficient to list the name of the array, *without any subscripts*, and the size of the array as arguments. For example, the call

```
largest(a,n);
```

will pass all the elements contained in the array **a** of size **n**. The called function expecting this call must be appropriately defined. The **largest** function header might look like:

```
float largest(array, size)
float array[ ];
int size;
```

The function **largest** is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array. The declaration of the formal argument **array** is made as follows:

```
float array[ ];
```

The pair of brackets informs the compiler that the argument **array** is an array of numbers. It is not necessary to specify the size of the **array** here.

Let us consider a problem of finding the largest value in an array of elements. The program is as follows:

```
main( )
{
   float largest( );
   static float value[4] = {2.5,–4.75, 1.2,3.67};
```

```
    printf("%f\n", largest(value,4));
}
float largest(a,n)
float a[ ];
int n;
{
    int i;
    float max;
    max = a[0];
    for(i = 1; i < n; i++)
        if(max < a[i])
            max = a[i];
    return(max);
}
```

When the function call **largest** (value, 4) is made, the values of all elements of the array **value** are passed to the corresponding elements of array **a** in the called function. The **largest** function finds the largest value in the array and returns the result to the **main**.

Example 10.5

Write a program to calculate the standard deviation of an array of values. The array elements are read from the terminal. Use functions to calculate standard deviation and mean.

Standard deviation of a set of n values is given by

$$\text{S.D.} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\bar{x} - x_i)^2}$$

where $\bar{x}$ is the mean of the values.

A multifunction program consisting of **main, std-dev,** and **mean** functions is shown in Fig. 10.11. **main** reads the elements of the array **value** from the terminal and calls the function **std-dev** to print the standard deviation of the array elements. **std-dev**, in turn, calls another function **mean** to supply the average value of the array elements.

Both **std-dev** and **mean** are defined as **floats** and therefore they are declared as **floats** in the declaration part of their calling functions.

When dealing with array arguments, we should remember one major distinction. If a function changes the values of an array elements, then these changes will be made to the original array that passed to the function. When an entire array is passed as an argument, the contents of the array are not copied into the formal parameter array; instead, information about the addresses of array elements are passed on to the function. Therefore, any changes introduced to the array elements are truly reflected in the original array in the calling function.

Program

```
/*****************************************************************************/
/*                          FUNCTIONS WITH ARRAYS                            */
/*****************************************************************************/
#include        <math.h>
#define SIZE      5
main()
```

```
{
    float value[SIZE], std_dev();
    int i;
    printf("Enter %d float values\n", SIZE);
    for (i=0; i < SIZE; i++)
        scanf("%f", &value[i];
    printf("Std.deviation is %f\n", std_dev(value, SIZE)) ;
}
float std_dev(a,n)
float a[];
int  n;
{
    int i;
    float mean(), x, sum = 0.0;
    x = mean (a,n);
    for(i=0; i <n; i++)
        sum += (x–a[i]*(x–a[i]);
    return(sqrt(sum/(float)n));
}
float mean(a,n)
float a[];
int n;
{
    int i;
    float sum = 0.0;
    for(i=0; i < n; i++)
        sum = sum + a[i];
    return(sum/(float)n);
}
Output
Enter 5 float values
35.0 67.0 79.5 14.20 55.75
Std.deviation is 23.231582
```

Fig. 10.11 *Passing of arrays to a function*

However, this does not apply when an individual element is passed on as argument. Example 10.6 highlights these concepts.

Example 10.6

Write a program that uses a function to sort an array of integers.

A program to sort an array of integers using the function **sort**( ) is given in Fig. 10.12. Its output clearly shows that a function can change the values in an array passed as an argument.

```
Program
/*********************************************************************************/
/*                          SORTING OF ARRAY ELEMENTS                            */
/*********************************************************************************/
main()
{
    int  i;
    static int marks[5] = {40, 90, 73, 81, 35};
```

```
    printf("Marks before sorting\n");
    for(i = 0; i < 5; i++)
        printf("%d", marks[i]);
    printf("\n\n");
    sort (5, marks);
    printf("Marks after sorting\n");
    for(i = 0; i < 5; i++)
        printf("%4d", marks[i]);
    printf("\n");
}
sort(m,x)
int m, x[ ];
{
    int i, j, t;
    for(i = 1; i < = m–1; i++)
        for(j = 1; j <=m–i; j++)
            if(x[j-1]>=x[j])
            {
                t = x[j–1];
                x[j–1] = x[j];
                x[j] = t;
            }
}
Output
Marks before sorting
40 90 73 81 35
Marks after sorting
   35  40  73  81  90
```

Fig. 10.12 *Sorting of array elements using a function*

10.15 THE SCOPE AND LIFETIME OF VARIABLES IN FUNCTIONS—STORAGE CLASSES

Variables in C differ in behaviour from those in most other languages. For example, in a BASIC program, a variable retains its value throughout the program. It is not always the case in C. It all depends on the 'storage' class a variable may assume.

As mentioned earlier, a variable in C can have any one of the four storage classes:

1. Automatic variables
2. External variables
3. Static variables
4. Register variables

We shall briefly discuss the *scope* and *longevity* of each of the above class of variables. The scope of variable determines over what part(s) of the program a variable is actually available for use (active). Longevity refers to the period during which a variable retains a given value during execution of a program (alive). So longevity has a direct effect on the utility of a given variable.

The variables may also be broadly categorized, depending on the place of their declaration, as *internal* (local) or *external* (global). Internal variables are those which are declared within a particular function, while external variables are declared outside of any function.

It is very important to understand the concept of storage classes and their utility in order to develop efficient multifunction programs.

Automatic Variables

Automatic variables are declared inside a function in which they are to be utilized. They are *created* when the function is called and *destroyed* automatically when the function is exited, hence the name automatic. Automatic variables are therefore private (or local) to the function in which they are declared. Because of this property, automatic variables are also referred to as *local* or *internal* variables.

A variable declared inside a function without storage class specification is, by default, an automatic variable. For instance, the storage class of the variable **number** in the example below is automatic.

```
main( )
{
    int number;
    ----------
    ----------
}
```

We may also use the keyword **auto** to declare automatic variables explicitly.

```
main( )
{
    auto int number;
    ----------
    ----------
}
```

One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other function in the program. This assures that we may declare and use the same variable name in different functions in the same program without causing any confusion to the compiler.

Example 10.7
Write a multifunction to illustrate how automatic variables work.

A program with two subprograms **function1** and **function2** is shown in Fig. 10.13. **m** is an automatic variable and it is declared at the beginning of each function. **m** is initialized to 10, 100, and 1000 in **function1**, **function2**, and **main** respectively.

When executed, **main** calls **function2** which in turn calls **function**1. When **main** is active, m = 1000; but when **function2** is called, the **main's** **m** is temporarily put on the shelf and the new local **m** = 100 becomes active. Similarly, when **function1** is called, both the previous values of **m** are put on the shelf and the latest value of **m** (=10) becomes active. As soon as **function1** (m = 10) is finished, **function2** (m = 100) takes over again. As soon it is done, **main** (m = 1000) takes over. The output clearly shows that the value assigned to **m** in one function does not affect its value in the other functions; and the local value of **m** is destroyed when it leaves a function.

There are two consequences of the scope and longevity of **auto** variables worth remembering. First, any variable local to **main** will normally *live* throughout the whole program, although it is *active* only in

Program

```
/*****************************************************************************/
/*          ILLUSTRATION OF WORKING OF auto VARIABLES                        */
/*****************************************************************************/
main()
{
    int m = 1000;
    function2();
    printf("%d\n",m);
}
function1()
{
    int m = 10;
    printf("%d\n",m);
}
function2()
{
    int m = 100;
    function1();
    printf("%d\n",m);
}
```

Output

```
10
100
1000
```

Fig. 10.13 *Working of automatic variables*

main. Secondly, during recursion, the nested variables are unique **auto** variables, a situation similar to function-nested **auto** variables with identical names.

Automatic variables can also be defined within a set of braces known as "blocks". They are meaningful only inside the blocks where they are defined. Consider the example below:

```
main()
{
    int n, a, b;
    . . . .
    . . . .
    if(n <= 100)
    {
        int n, sum;
        . . . .
        . . . .
    }
    . . . . /* sum not valid here */
    }. . . .
```

Scope level 1

Scope level 2

The variables **n**, **a** and **b** defined in **main** have scope from the beginning to the end of **main**. However, the variable **n** defined in the **main** cannot enter into the block of scope level 2 because the scope level 2

contains another variable named **n**. The second **n** (which takes precedence over the first **n**) is available only inside the scope level 2 and no longer available the moment control leaves the **if** block. Of course, if no variable named **n** is defined inside the **if** block, then **n** defined in the **main** would be available inside scope level 2 as well. The variable **sum** defined in the **if** block is not available outside that block.

External Variables

Variables that are both *alive* and *active* throughout the entire program are known as *external* variables. They are also known as *global* variables. Unlike local variables, global variables can be accessed by *any* function in the program. External variables are declared outside a function. For example, the external declaration of integer **number** and float **length** might appear as:

```
int number;
float length = 7.5;
main()
{
   ----------
   ----------
}
function1()
{
   ----------
   ----------
}
function2()
{
   ----------
   ----------
}
```

The variables **number** and **length** are available for use in all the three functions. In case a local variable and a global variable have the same name, the local variable will have precedence over the global one in the function where it is declared. Consider the following example:

```
int count;
main()
{
   count = 10;
   ----------
   ----------
}
function()
{
   int count = 0;
   ----------
   ----------
   count = count+1;
}
```

When the **function** references the variable **count**, it will be referencing only its local variable, not the global one. The value of **count** in **main** will not be affected.

Example 10.8

Write a multifunction program to illustrate the properties of global variables.

A program to illustrate the properties of global variables is presented in Fig. 10.14. Note that variable **x** is used in all functions, but none except **fun2**, has a definition for **x**. Because **x** has been declared 'above' all the functions, it is available to each function without having to pass **x** as a function argument.

Program

```
/*****************************************************************************/
/*       ILLUSTRATION OF PROPERTIES OF GLOBAL VARIABLES          */
/*****************************************************************************/

int x;
main()
{
    x = 10;
    printf("x = %d\n", x);
    printf("x = %d\n", fun1());
    printf("x = %d\n", fun2());
    printf("x = %d\n", fun3());
}
fun1()
{
    x = x + 10;
    return(x);
}
fun2()
{
    int x;
    x = 1;
    return(x);
}
fun3()
{
    x = x + 10;
    return(x);
}
```

Output

```
x = 10
x = 20
x = 1
x = 30
```

Fig. 10.14 *Illustration of global variables*

Once a variable has been declared as global, any function can use it and change its value. Then, subsequent functions can reference only that new value. Because of this property, we should try to use global variables only for tables or for variables shared between functions when it is inconvenient to pass them as parameters.

One other aspect of a global variable is that it is visible only from the point of declaration to the end of the program. Consider a program segment as shown below:

```
main( )
{
   y = 5;
   ----------
   ----------
}
int y;
func1( )
{
   y = y+1;
}
```

We have a problem here. As far as **main** is concerned, **y** is not defined. So, the compiler will issue an error message. Unlike local variables, global variables are initialized to zero by default. The statement

```
y = y +1;
```

in **fun1** will, therefore, assign 1 to **y**.

External Declaration

In the program segment above, the **main** cannot access the variable **y** as it has been declared after the **main** function. This problem can be solved by declaring the variable with the storage class **extern**. For example:

```
main()
{
   extern int y; /* external declaration */
   . . . .
   . . . .
}
func1()
{
   extern int y; /* external declaration */
   . . . .
   . . . .
}
int y; /* definition */
```

Although the variable **y** has been defined after both the functions, the *external declaration* of **y** inside the functions informs the compiler that **y** is an integer type defined somewhere else in the program. Note that the **extern** declaration does not allocate storage space for variables. In case of arrays, the definition should include their size as well.

Example:

```
main()
{
   int i;
```

```
    void print_out();
    extern float height [];
    . . . .
    . . . .
    print_out();
}
void print_out()
{
    extern float height [];
    int i;
    . . . .
    . . . .
}
float height[SIZE];
```

An **extern** within a function provides the type information to just that one function. We can provide type information to all functions within a file by placing external declarations before any of them, as shown below:

```
extern float height [];
main()
{
    int i;
    void print_out();
    . . . .
    . . . .
    print_out();
}
void print_out()
{
    int i;
    . . . .
    . . . .
]
float height[SIZE];
```

The distinction between definition and declaration also applies to functions. A function is defined when its parameters and function body are specified. This tells the compiler to allocate space for the function code and provides type information for the parameters. Since functions are external by default, we declare them (in the calling functions) without the qualifier **extern**. Therefore, the declaration

```
void print_out();
```

is equivalent to

```
extern void print_out();
```

Function declarations outside of any function behave the same way as variable declarations.

Multifile Programs

So far we have been assuming that all the functions (including the **main**) are defined in one file. However, in real-life programming environment we may use more than one source files which may be

compiled separately and linked later to form an executable object code. This approach is very useful because any change in one file does not affect other files thus eliminating the need for recompilation of the entire program.

Multiple source files can share a variable provided it is declared as an external variable appropriately. Variables that are shared by two or more files are global variables and therefore we must declare them accordingly in one file and then explicitly define them with **extern** in other files. Figure 10.15 illustrates the use of extern declarations in a multifile program.

The function **main** in **file1** can reference the variable **m** that is declared as global in **file2**. Remember, **function1** cannot access the variable **m**. If, however, the **extern int m;** statement is placed before **main**, then both the functions could refer to **m**. This can also be achieved by using **extern int m;** statement inside each function in **file1**.

The **extern** specifier tells the compiler that the following variable types and names have already been declared elsewhere and no need to create storage space for them. It is the responsibility of the *linker* to resolve the reference problem. It is important to note that a multifile global variable should be declared *without* **extern** in one (and only one) of the files. The **extern** declaration is done in places where secondary references are made. If we declare a variable as global in two different files used by a single program, then the linker will have a conflict as to which variable to use and, therefore, issues a warning.

file1.c

```
main()
{
    extern int m;
    int i;
    . . . .
    . . . .
}
function1()
{
    int j;
    . . . .
    . . . .
}
```

file2.c

```
int m /* global variable */
function2()
{
    int i;
    . . . .
    . . . .
}
function3()
{
    int count;
    . . . .
    . . . .
}
```

Fig. 10.15 *Use of* ***extern*** *in a multifile program*

The multifile program shown in Fig. 10.15 may be modified as shown in Fig. 10.16.

When a function is defined in one file and accessed in another, the later file must include a function *declaration*. The declaration identifies the function as an external function whose definition appears elsewhere. We usually place such declarations at the beginning of the file, before all functions. Although all functions are assumed to be external, it would be a good practice to explicitly declare such functions with the storage class **extern**.

Static Variables

As the name suggests, the value of static variables persists until the end of the program. A variable can be declared static using the keyword **static** like.

file1.c

```
int m; /* global variable */

main()
{
    int i;
    . . . .
    . . . .
}
function1()
{
    int j;
    . . . .
    . . . .
}
```

file2.c

```
extern int m;

function2()
{
    int i;
    . . . .
    . . . .
}
function3()
{
    int count;
    . . . .
    . . . .
}
```

Fig. 10.16 *Another version of a multifile program*

```
static int x;
static float y;
```

A static variable may be either an internal type or an external type, depending on the place of declaration.

Internal static variables are those which are declared inside a function. The scope of internal static variables extend upto the end of the function in which they are defined. Therefore, internal **static** variables are similar to **auto** variables, except that they remain in existence (alive) throughout the remainder of the program. Therefore, internal **static** variables can be used to retain values between function calls. For example, it can be used to count the number of calls made to a function.

Example 10.9

Write a program to illustrate the properties of a static variable.

The program in Fig. 10.17 explains the behaviour of a static variable.

Program

```
/*********************************************************************************** /
/*                          ILLUSTRATION OF STATIC VARIABLE                        */
/***********************************************************************************/
main ( )
{
    int i;
    for(i=1; i<=3; i++)
        stat( );
}
stat( )
{
    static int x = 0;
    x = x + 1;
    printf("x = %d\n", x);
```

```
{
Output
x = 1
x = 2
x = 3
```

Fig. 10.17 *Illustration of static variable*

A static variable is initialized only once, when the program is compiled. It is never initialized again. During the first call to **stat, x** is incremented to 1. Because **x** is static, this value persists and therefore, the next call adds another 1 to **x** giving it a value of 2. The value of **x** becomes three when the third call is made.

Had we declared **x** as an **auto** variable, the output would have been:

```
x = 1
x = 1
x = 1
```

This is because each time **stat** is called, the auto variable **x** is initialized to zero. When the function terminates, its value of 1 is lost.

An external **static** variable is declared outside of all functions and is available to all the functions in that program. The difference between a **static** external variable and a simple external variable is that the **static** external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

It is also possible to control the scope of a function. For example, we would like a particular function accessible only to the functions in the file in which it is defined, and not to any function in other files. This can be accomplished by defining 'that' function with the storage class **static**.

Register Variables

We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory (where normal variables are stored). Since a register access is much faster than a memory access, keeping the frequently accessed variables (e.g., loop control variables) in the register will lead to faster execution of programs. This is done as follows:

```
register int count;
```

Although, ANSI standard does not restrict its application to any particular data type, most compilers allow only **int** or **char** variables to be placed in the register.

Since only a few variables can be placed in the register, it is important to carefully select the variables for this purpose. However, C will automatically convert **register** variables into nonregister variables once the limit is reached.

Table 10.1 summarizes the information on the visibility and lifetime of variables in functions and files.

Table 10.1 Scope and Lifetime of Declarations

Storage class	Where declared	Visibility (Active)	Lifetime (Active)
None	Before all functions in a file (may be initialized)	Entire file plus other files where variable is declared with **extern**.	Entire program (Global)
extern	Before all functions in a file (cannot be initialized)	Entire file plus other files where variable is declared **extern** and the file where originally declared as global.	Global
static	Before all functions in a file	Only in that file	Global
None or **auto**	Inside a function (or a block)	Only in that function or block	Until end of function or block
register	Inside a function or block	Only in that function or block	Until end of function or block
static	Inside a function	Only in that function	Global

10.16 ANSI C FUNCTIONS

Function Definition

As mentioned in the beginning of this chapter, ANSI standard combines the function definition and arguments declaration in one line (the function header). The general form of ANSI C functions is:

```
data-type function-name (type1 a1, type2 a2,.. typeN aN)
{
      . . . .
      . . . . (body of the function)
      . . . .
}
```

Here *data-type* refers to the type of the value returned by the function **function-name** and *type1, type2,..., typeN* are types of arguments *a1*. *a2*,..., *aN*.

As usual, if no data type is specified for the function, then, by default, the function is assumed to return an integer value. The arguments declaration is done using the *comma* separator. Remember, each

argument must be independently declared with type specifier and name. That is, similar type arguments cannot be combined under one type name, like variable declarations. For example,

double funct(int a, int b, double c)

is a valid function definition, whereas

double funct(int a, b, double c)

is incorrect.

This new version of function definition, which combines the argument list and the arguments declaration, is called a *function prototype*. Function prototypes were not part of the original C language. It is one of the important recommendations of the ANSI C committee.

Function Declaration

We know that every function that is being referenced should be declared for its type in the calling function. For example:

```
main()
{
   float a, b, x;
   float mul(); /* function declaration (old type)*/
   . . . .
   . . . .
   x = mul(a,b); /* function call */
}
```

Note that the function declaration as stated above specifies the type of value returned by the function **mul** and does not talk anything about its arguments. The actual arguments are specified only when the function is called.

The introduction of "function prototypes" requires that the function declaration must include not only the type of return value but also the type and number of arguments to expect. Above program segment may be modified using the function prototype as follows:

```
main()
{
   float a, b, x;
   float mul(float a, float b);     /* function prototype */
  . . . . .                         /* ANSI declaration */
   . . . .
   x = mul(a, b);
}
```

The general form of function declaration using ANSI prototype is:

data-type **function-name** (*type1 a1, type2 a2, ...,typeN aN*);

One major reason for using function prototypes in function declarations is that they enable the compiler check for any mismatch of type and number of arguments between the function calls and the function definitions. They also help the compiler to perform automatic type conversions on function parameters. When a function is called, the actual arguments are automatically converted to the types in the function definition using the normal rules of assignment.

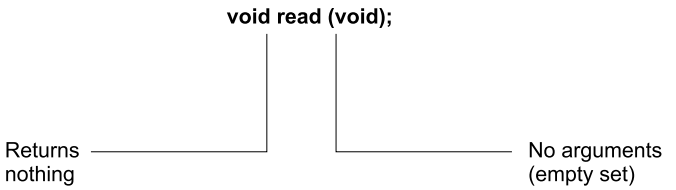
The argument names (*a1, a2, ..., aN*) used in the function declaration are for documentation purpose only and they do not define the variables in the function. This means, we may use any "dummy" argument names (which are recognised only within the declaration). For example, the following function declaration is valid.

```
main()
{
   float a, b, x;
   float mul(float length, float breadth); /* declaration */
   . . . .
   . . . .
   x = mul(a,b);
}
```

But the normal practice is to use the names of the "actual" arguments appearing in the function call. Since only the types (and not the names) of the arguments are important, it is possible to declare a function without the names of arguments. Example:

```
float mul(float, float); /* declaration */
```

In case, a function does not return anything, or it does not have any arguments to declare, it can be declared using **void** as follows:



The type **void** is often used for declaring functions returning pointers that can point to any type of data.

Example 10.10

Rewrite the power function program in Fig. 10.10 using function prototypes.

The rewritten program is shown in Fig. 10.18. The program uses function prototypes in both the function definition and function declaration. The program also illustrates the following aspects:

1. When the function is called, the names of the actual arguments need not be the same as the names given in the declaration. Nevertheless, their data types must match the types in declaration.
2. The names of the arguments in prototype declaration need not be the same names given in the prototype definition. Here again, the data types must match.
3. The variable name *p* has been defined in both the functions. Since they are defined as local **auto** variables, they do not conflict.

Program

```
/*****************************************************************************/
/*              POWER FUNCTION PROGRAM WITH PROTOTYPES                       */
/*****************************************************************************/

main()
{
    int x, y, m, n, p;
    double q;
    double power(int x, int y); /* prototype declaration*/

    x = 16;  y = 2;
    p = power(x, y);

    m = 16; n = – 2;
    q = power(m,n);       /*   arguments names not same */

    printf("\n p = %d\n q = %f\n", p,q);
}

double power(int a, int b) /* prototype definition */

{
    double p;
    p = 1.0;

    if(b > = 0)
        while(b––) p *= a;
    else
        while(b+ +) p /= a;
    return(p);
}
```

Output

```
p = 256
q = 0.003906
```

Fig. 10.18 *Use of function prototypes*

Variable Number of Arguments

Some functions have a variable number of arguments and data types which cannot be known at compile time. The **print** and **scanf** functions are typical examples. The ANSI standard proposes new symbol called the *ellipsis* to handle such functions. The ellipsis consists of three periods (...) and used as shown below:

double area(float d, ...)

Both the function declaration and definition should use ellipsis to indicate that the arguments are arbitrary both in number and type.

The Preprocessor

10.17 THE PREPROCESSOR

C is a unique language in many respects. We have already seen features such as structures and pointers. Yet another unique feature of the C language is the *preprocessor.* The C preprocessor provides several

tools that are unavailable in other high-level languages. The programmer can use these tools to make his program easy to read, easy to modify, portable, and more efficient.

The preprocessor, as its name implies, is a program that processes the source code before it passes through the compiler. It operates under the control of what is known as *preprocessor command lines or directives*. Preprocessor directives are placed in the source program before the main line. Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any, appropriate actions (as per the directives) are taken and then the source program is handed over to the compiler.

Preprocessor directives follow special syntax rules that are different from the normal C syntax. They all begin with the symbol # in column one and do not require a semicolon at the end. We have already used the directives **#define** and **#include** to a limited extent. A set of commonly used preprocessor directives and their functions is given in Table 10.2.

Table 10.2 Preprocessor Directives

Directive	Function
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies the files to be included
#ifdef	Test for a macro definition
#endif	Specifies the end of #if
#ifndef	Tests whether a macro is not defined
#if	Test a compile-time condition
#else	Specifies alternatives when #if test fails

These directives can be divided into three categories:

1. Macro substitution directives.
2. File inclusion directives.
3. Compiler control directives.

Macro Substitution

Macro substitution is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. The preprocessor accomplishes this task under the direction of **#define** statement. This statement, usually known as a *macro definition* (or simply a macro) takes the following general form:

#define *identifier string*

If this statement is included in the program at the beginning, then the preprocessor replaces every occurrence of the **identifier** in the source code by the string. The keyword **#define** is written just as shown (starting from the first column) followed by the *identifier* and a *string*, with at least one blank space between them. Note that the definition is not terminated by a semicolon. The *string* may be any text, while the *identifier* must be a valid C name.

There are different forms of macro substitution. The most common forms are:

1. Simple macro substitution.

2. Argumented macro substitution.
3. Nested macro substitution.

Defining Macro

Simple string replacement is commonly used to define constants. Examples of definition of constants are:

#define	COUNT	100
#define	FALSE	0
#define	SUBJECTS	6
#define	PI	3.1415926
#define	CAPITAL	"DELHI"

Notice that we have written all macros (identifiers) in capitals. It is a convention to write all macros in capitals to identify them as symbolic constants.

Undefining a Macro

A defined macro can be undefined, using the statement

#undef *identifier*

This is useful when we want to restrict the definition only to a particular part of the program.

File Inclusion

An external file containing functions or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions. This is achieved by the preprocessor directive

#include "filename"

where *filename* is the name of the file containing the required definitions or functions. At this point, the preprocessor inserts the entire contents of *filename* into the source code of the program. When the *filename* is included within the double quotation marks, the search for the file is made first in the current directory and then in the standard directories.

Alternatively this directive can take the form

#include <filename>

without double quotation marks. In this case, the file is searched only in the standard directories.

Nesting of included files is allowed. That is, an included file can include other files. However, a file cannot include itself.

If an included file is not found, an error is reported and compilation is terminated.

Let use assume that we have created the following three files:

SYNTAX.C	contains syntax definitions.
STAT.C	contains statistical functions.
TEST.C	contains test functions.

We can make use of a definition or function contained in any of these files by including them in the program as:

```
#include      <stdio.h>
#include      "SYNTAX.C"
#include      "STAT.C"
#include      "TEST.C"
#define       M          100
main ()
{
      ----------
      ----------
      ----------
}
```

Compiler Control Directives

While developing large programs, you may face one or more of the following situations:

1. You have included a file containing some macro definitions. It is not known whether a particular macro (say, TEST) has been defined in that header file. However, you want to be certain that Test is defined (or not defined).
2. Suppose a customer has two different types of computers and you are required to write a program that will run on both the systems. You want to use the same program, although certain lines of code must be different for each system.
3. You are developing a program (say, for sales analysis) for selling in the open market. Some customers may insist on having certain additional features. However, you would like to have a single program that would satisfy both types of customers.
4. Suppose you are in the process of testing your program, which is rather a large one. You would like to have print calls inserted in certain places to display intermediate results and messages in order to trace the flow of execution and errors, if any. Such statements are called 'debugging' statements. You want these statements to be a part of the program and to become 'active' only when you decide so.

One solution to these problems is to develop different programs to suit the needs of different situations. Another method is to develop a single, comprehensive program that includes all optional codes and then directs the compiler to skip over certain parts of source code when they are not required. Fortunately, the C preprocessor offers a feature known as *conditional compilation*, which can be used to 'switch' on or off a particular line or group of lines in a program.

10.18 ANSI ADDITIONS

ANSI committee has added some more preprocessor directives to the existing list given in Table 14.1. They are:

#elif	Provides alternative test facility
#pragma	Specifies certain instructions
#error	Stops compilation when an error occurs

The ANSI standard also includes two new preprocessor operations:

#	Stringizing operator
##	Token-pasting operator

elif Directive

The #elif enables us to establish an "if..else..if.." sequence for testing multiple conditions. The general form of use of **#elif** is:

```
#if    expression 1
       statement sequence 1
  #elif  expression 2
         statement sequence 2
         . . . . .
         . . . . .
            #elif    expression N
                 statement sequence N
#endif
```

For example:

```
#if    MACHINE == HCL
       #define FILE "hcl.h"

          #elif      MACHINE == WIPRO
                #define FILE "wipro.h"

                #elif      MACHINE == DCM
                      #define FILE "dcm.h"
#endif
#include FILE
```

#pragma Directive

The **#pragma** is an implementation oriented directive that allows us to specify various instructions to be given to the compiler. It takes the following form:

#pragma *name*

where, *name* is the name of the **pragma** we want. For example, under Microsoft C,

#pragma loop_opt(on)

causes loop optimization to be performed. It is ignored, if the compiler does not recognize it.

#error Directive

The **#error** directive is used to produce diagnostic messages during debugging. The general form is

#error *error message*

When the **#error** directive is encountered, it displays the error message and terminates processing. Example.

```
#if !defined(FILE_G)
#error NO GRAPHICS FACILITY
#endif
```

Note that we have used a special processor operator **defined** along with **#if**. **defined** is a new addition and takes a *name* surrounded by parentheses. If a compiler does not support this, we can replace it as follows:

```
#if  !defined     by     #ifndef
#if  defined      by     #ifdef
```

Stringizing Operator

ANSI C provides an operator # called *stringizing operator* to be used in the definition of macro functions. This operator allows a formal argument within a macro definition to be converted to a string. Consider the example below:

```
#define sum(xy) printf(#xy " = %f\n", xy)
main()
{
   … …
   … …
   sum(a+b);
   … …
}
```

The preprocessor will convert the line

```
sum(a+b);
```

into

```
printf("a+b" "=%f\n", a+b);
```

which is equivalent to

```
printf("a+b =%f\n", a+b);
```

Note that the ANSI standard also stipulates that adjacent strings will be concatenated.

Token Pasting Operator

The token pasting operator ## defined by ANSI standard enables us to combine two tokens within a macro definition to form a single token. For example:

```
#define combine(s1,s2) s1 ## s2
main()
{
        ... ..
        ... ..
```

```
        printf("%f", combine(total, sales));
        ... ..
        ... ..
}
```

The preprocessor transforms the statement

```
printf("%f", combine(total, sales));
```

into the statement

```
printf("%f", totalsales);
```

Consider another macro definition:

```
#define print(i) printf("a" #i "=%f", a##i)
```

This macro will convert the statement

```
print(5);
```

into the statement

```
printf("a5 = %f", a5)
```

CASE STUDY

Calculation of Area Under a Curve

One of the applications of computers in numerical analysis is computing the area under a curve. One simple method of calculating the area under a curve is to divide the area into a number of trapezoids of same width and summing up the areas of individual trapezoids. The area of a trapezoid is given by

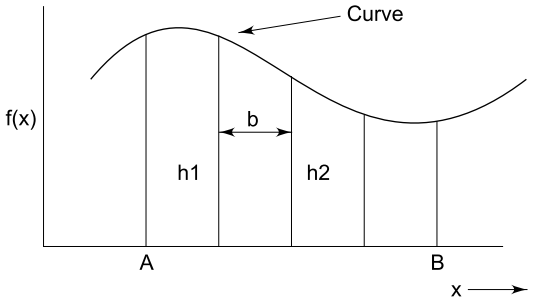
$$\text{Area} = 0.5 * (h1 + h2) * b$$

where h1 and h2 are the heights of two sides and b is the width as shown below:
The program in Fig. 10.19 calculates the area for a curve of the function

$$f(x) = x^2 + 1$$

between any two given limits, say, A and B.

Input
Lower limit (A)
Upper limit (B)
Number of trapezoids



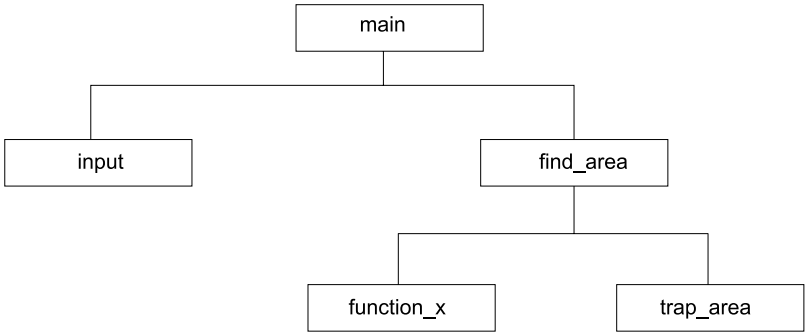
Output

Total area under the curve between the given limits.

Algorithm

1. Input the lower and upper limits and the number of trapezoids.
2. Calculate the width of trapezoids.
3. Initialize the total area.
4. Calculate the area of trapezoid and add to the total area.
5. Repeat step-4 until all the trapezoids are completed.
6. Print total area.

The algorithm is implemented in top-down modular form as shown below:



The evaluation of f(x) has been done using a separate function so that it can be easily modified to allow other functions to be evaluated.

The output for two runs shows that better accuracy is achieved with larger number of trapezoids. The actual area for the limits 0 and 3 is 12 units (by analytical method).

Program

```
/*****************************************************************************/
/*                          AREA UNDER A CURVE                               */
/*****************************************************************************/

#include <stdio.h>

float   start_point,        /* GLOBAL VARIABLES */
        end_point,
        total_area;
int     numtraps;


main()
{
    void    input(void);
    float   find_area(float a, floatb, int n);

    printf("AREA UNDER A CURVE");
    printf("/n");
```

```
    total_area = find_area (start_point, end_point, numtraps);
    printf("TOTAL AREA = %f", total_area)
}
void input(void)
{
    printf("\n Enter lower limit:");
    scanf("%f", &start_point);
    printf(" Enter upper limit:");
    scanf("%f", &end_point);
    printf("Enter number of trapezoids:");
    scanf("%d", &numtraps);
}

float find_area(float a, float b, int n)
{

    float   base, lower, h1, h2; /* LOCAL VARIABLES */
    float    function_x(float x);
    float   trap_area(float h1, float h2, float base);

    base   = (b–a)/n;
    lower  = a;

    for(lower = a; lower <=b – base; lower = lower + base)
    {
        h1 =   function_x(lower);
        h2 =   function_x(lower + base);
        total_area += trap_area(h1, h2, base);
    }
    return(total_area);
}

float trap_area(float height_1, float height_2, float base)
{
    float area;      /* LOCAL VARIABLES */
    area = 0.5 * (height_1 + height_2) * base;
    return(area);
}

float function_x(float x)
{

/* F(X) = X * X + 1 */

return(x * x + 1);
}
```

Output

```
AREA UNDER A CURVE
 Enter lower limit: 0
 Enter upper limit: 3
 Enter number of trapezoids: 30
TOTAL AREA = 12.005000
```

```
AREA UNDER A CURVE
 Enter lower limit: 0
 Enter upper limit: 3
 Enter number of trapezoids: 100
TOTAL AREA = 12.000438
```

Fig. 10.19 *Computing area under a curve*

POINTS TO REMEMBER

The strength of C language lies primarily in the effective use of functions. While we encourage the use of modular, multifunction programming approach, it is whortwhile to remember some important points about functions.

1. A function may or may not return a value. If it does, it can return only one value.
2. Actual and formal arguments must agree in data types. Otherwise, garbage will be passed, without generating any error message.
3. Functions return integer value by default.
4. When a function is supposed to return a non-integer value, its type must be explicitly specified in the function header. Such functions should be declared at the start of the calling functions.
5. A function without a **return** statement cannot return any value. However, a global variable used in the function will retain its value for future use.
6. A function that returns a value may be used in expressions (arithmetic or relational) like any other C variable.
7. When the value returned is assigned to a variable, the value will be converted to the type of the variable receiving it.
8. A function cannot be the target of an assignment.
9. A **return** statement can occur anywhere within the body of a function. A function can also have more than one **return** statement.
10. A function may be placed either after or before the **main** function.
11. Where more functions are used, they may appear in any order.
12. A function is treated as an external variable. Therefore, a function cannot be defined inside another function.
13. A variable declared inside a function is known only to that function. It is created when the function is called and destroyed when the function is finished.
14. A global (external) variable is visible only from the point of its declaration to the end of the program.
15. A variable that has been declared as **static** inside a function retains its value even after the function is *exited*. (Since static variables are initialized during compilation, they are initialized only once.)
16. A multifile global variable must be declared without **extern** in one (and only one) of the files. The **extern** declaration is done in places where secondary references are made.

Review Questions and Exercises

10.1 Explain what is likely to happen when the following situations are encountered in a program.
- (a) Actual arguments are less than the formal arguments in a function.
- (b) Data type of one of the actual arguments does not match with the data type of the corresponding formal argument.
- (c) The type of the *expression* in **return** (expression) does not match with the type of the function.
- (d) Same variable name is declared in two different functions.

10.2 State whether the following statements are *true* or *false*.
- (a) Functions should be arranged in the order in which they are called.
- (b) C functions can return only one value.
- (c) We can pass any number of arguments to a function.
- (d) A function in C should have at least one argument.
- (e) A function always returns an integer value.
- (f) A function can call itself.
- (g) A global variable can be used only in **main** function.
- (h) The values of formal arguments can be changed inside the function where they are declared.
- (i) Every function should have a **return** statement.

10.3 Distinguish between the following:
- (a) Actual and formal arguments.
- (b) Global and local variables.
- (c) Automatic and static variables.
- (d) Global and extern variables.

10.4 **Main** is a user-defined function. How does it differ from other user-defined functions?

10.5 Which of the following function headers are invalid? And why?
- (a) average(x,y,z);
- (b) power(a,n–1)
- (c) product(m,10)
- (d) double minimum(float a; float b;)

10.6 The following function returns the value of x/y.

```
divide(x,y)
float x,y;
{
    return(x/y);
}
```

What will be the value of the following function calls:
- (a) divide(10,2)
- (b) divide(9,2)
- (c) divide(4.5,1.5)

10.7 Determine the output of the following program:

```
main( )
{
```

```
    int x = 10;
    int y = 20;
    int p,q;
    p = prod(x,y);
    q = prod(p,prod(x,2));
    printf("%d %d\n", p,q);
}
prod(a,b)
int a,b;
{
    return (a*b);
}
```

10.8 Write a function **exchange** to interchange the values of two variables, say **x** and **y**. Illustrate the use of this function, in a calling function. (*Hint:* Assume that **x** and **y** are defined as global variables.)

10.9 Write a function **space**(**x**) that can be used to provide a space of x positions between two output numbers.

10.10 Use recursive calls to evaluate

$$f(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

10.11 An *n*_order polynomial can be evaluated as follows:

$$P = (....(((a_0 \times + a_1)x + a_2)x + a_3)x + ... + a_n)$$

Write a function to evaluate the polynomial, using an array variable **a**.

10.12 The Fibonacci numbers are defined recursively as follows:

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

Write a function that will generate and print the first *n* Fibonacci numbers.

10.13 Write a function that will round a floating-point number to an indicated decimal place. For example the number 17.457 would yield the value 17.46 when it is rounded off to two decimal places.

10.14 Write a function **prime** that returns 1 if its argument is a prime number and returns zero otherwise.

10.15 Write a function that will scan a character string passed as an argument and convert all lowercase characters into their upper-case equivalents.

10.16 Write a function to raise a number to a power when both are floating point numbers.

Note: It is a good practice to use function prototypes as recommended in ANSI C. Try them if your compiler supports ANSI standards.

10.17 Explain the role of the C preprocessor.

10.18 What is a macro and how is it different from a C variable name?

10.19 When does a programmer use **#include** directive?

10.20 Distinguish between **#ifdef** and **#if** directives.

10.21 In **#include** directives, some file names are enclosed in angle brackets while others are enclosed in double quotation marks. Why?

UNIT III

Chapter 11

Pointers

11.1 INTRODUCTION

Pointers are another important feature of C language. Although they may appear a little confusing for a beginner, they are a powerful tool and handy to use once they are mastered. There are a number of reasons for using pointers.

1. A pointer enables us to access a variable that is defined outside the function.
2. Pointers are more efficient in handling the data tables.
3. Pointers reduce the length and complexity of a program.
4. They increase the execution speed.
5. The use of a pointer array to character strings results in saving of data storage space in memory.

The real power of C lies in the proper use of pointers. In this chapter, we will examine the pointers in detail and learn how to use them in program development.

11.2 UNDERSTANDING POINTERS

As you know, computers use their memory for storing the instructions of a program, as well as the values of the variables that are associated with it. The computer's memory is a sequential collection of 'storage cells' as shown in Fig. 11.1. Each cell, commonly known as a *byte*, has a number called *address* associated with it. Typically, the addresses are numbered consecutively, starting from zero. The last address depends on the memory size. A computer system having 64 K memory will have its last address as 65,535.

Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number. Consider the following statement:

```
int    quantity = 179;
```

This statement instructs the system to find a location for the integer variable **quantity** and puts the value 179 in that location. Let us assume that the system has chosen the address location 5000 for **quantity**. We may represent this as shown in Fig. 11.2.

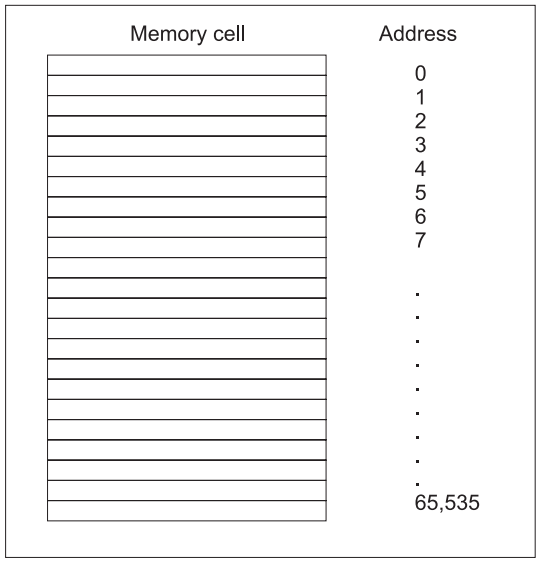


Fig. 11.1 *Memory organization*

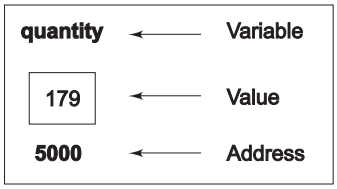


Fig. 11.2 *Representation of a variable*

During execution of the program, the system always associates the name **quantity** with the address 5000. (This is something similar to having a house number as well as a house name.) We may have access to the value 179 by using either the name **quantity** or the address 5000. Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory, like any other variable. Such variables that hold memory addresses are called *pointers*. A pointer is, therefore, nothing but a variable that contains an address which is a location of another variable in memory.

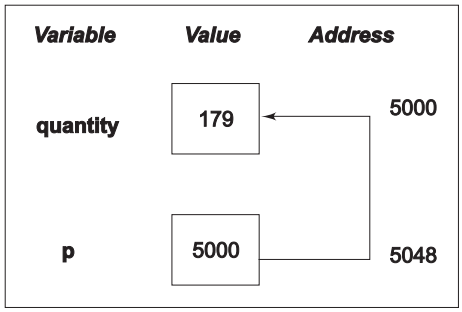


Fig. 11.3 *Pointer as a variable*

Remember, since a pointer is a variable, its value is also stored in the memory in another location. Suppose, we assign the address of **quantity** to a variable **p**. The link between the variables **p** and **quantity** can be visualized as shown in Fig. 11.3. The address of **p** is 5048.

Since the value of the variable **p** is the address of the variable **quantity**, we may access the value of **quantity** by using the value of **p** and therefore, we say that the variable **p** 'points' to the variable **quantity**. Thus, **p** gets the name 'pointer'.

11.3 ACCESSING THE ADDRESS OF A VARIABLE

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. How can we then determine the address of a variable? This can be done with the help of the operator **&** available in C. We have already seen the use of this *address operator* in the **scanf** function. The operator **&** immediately preceding a variable returns the address of the variable associated with it. For example, the statement

```
p = &quantity;
```

would assign the address 5000 (the location of **quantity**) to the variable **p**. The **&** operator can be remembered as 'address of'.

The **&** operator can be used only with a simple variable or an array element. The following are illegal use of address operator:

1. **&125** (pointing at constants).
2. int x[10];
   &x (pointing at array names)
3. **&(x+y)** (pointing at expressions).

If **x** is an array, then expressions such as

```
&x[0] and &x[i+3]
```

are valid and represent the addresses of 0th and (i+3)th elements of x.

Example 11.1
Write a program to print the address of a variable along with its value.

The program shown in Fig. 11.4, declares and initializes four variables and then prints out these values with their respective storage locations. Notice that we have used %u format for printing address values. Memory addresses are unsigned integers.

Program

```
/*************************************************************************************/
/*                    ACCESSING ADDRESSES OF VARIABLES                               */
/*************************************************************************************/
main( )
{
    char a;
    int  x;
    float p, q;
    a = 'A';
```

```
    x = 125;
    p = 10.25, q = 18.76;
    printf("%c is stored at addr %u.\n", a, &a);
    printf("%d is stored at addr %u.\n", x, &x);
    printf("%f is stored at addr %u.\n", p, &p);
    printf("%f is stored at addr %u.\n", q, &q);
}
```

Output

```
A is stored at addr 4436.
125 is stored at addr 4434.
10.250000 is stored at addr 4442
18.760000 is stored at addr 4438.
```

Fig. 11.4 *Accessing the address of a variable*

11.4 DECLARING AND INITIALIZING POINTERS

In C, every variable must be declared for its type. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:

```
data type *pt_name;
```

This tells the compiler three things about the variable **pt_name**.

1. The asterisk (*) tells that the variable **pt_name** is a pointer variable.
2. **pt_name** needs a memory location.
3. **pt_name** points to a variable of type *data type*.

For example,

```
int   *p;
```

declares the variable **p** as a pointer variable that points to an integer data type. Remember that the type **int** refers to the data type of the variable being pointed to by **p** and not the type of the value of the pointer. Similarly, the statement

```
float   *x;
```

declares x as a pointer to a floating point variable.

Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement such as

```
p = &quantity;
```

which causes **p** to point to **quantity**. That is, **p** now contains the address of **quantity**. This is known as pointer *initialization*. Before a pointer is initialized, it should not be used.

We must ensure that the pointer variables always point to the corresponding type of data. For example,

```
float  a, b;
int    x, *p;
```

```
p  = &a;
b  = *p;
```

will result in erroneous output because we are trying to assign the address of a **float** variable to an integer pointer. When we declare a pointer to be of **int** type, the system assumes that any address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors, care should be taken to avoid wrong pointer assignments.

And also assigning an absolute address to a pointer variable is prohibited. The following is wrong.

```
int       *ptr;
....
ptr       = 5368;
....
....
```

A pointer variable can be initialized in its declaration itself. For example,

```
int x, *p = &x;
```

is perfectly valid. It declares **x** as an integer variable and **p** as a pointer variable and then initializes **p** to the address of **x**. Note carefully that this is an initialization of **p**, not ***p**. And also remember that the target variable **x** is declared first. The statement

```
int   *p = &x, x;
```

is not valid.

11.5 ACCESSING A VARIABLE THROUGH ITS POINTER

Once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer. This is done by using another unary operator * (asterisk), usually known as the *indirection* operator. Consider the following statements:

```
int   quantity, *p, n;
quantity = 179;
p  = &quantity;
n  = *p;
```

The first line declares **quantity** and **n** as integer variables and **p** as a pointer variable pointing to an integer. The second line assigns the value 179 to **quantity** and the third line assigns the address of **quantity** to the pointer variable **p**. The fourth line contains the indirection operator *. When the operator * is placed before a pointer variable in an expression (on the right-hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address. In this case, ***p** returns the value of the variable **quantity**, because **p** is the address of **quantity**. The * can be remembered as 'value at address'. Thus the value of **n** would be 179. The two statements

```
p = &quantity;
n = *p;
```

are equivalent to

n = *&quantity;

which in turn is equivalent to

n = quantity;

In C, the assignment of pointers and addresses is always done symbolically, by means of symbolic names. You cannot access the value stored at the address 5368 by writing *5368. It will not work. Example 11.2 illustrates the distinction between pointer value and the value it points to.

Example 11.2
Write a program to illustrate the use of indirection operator '*' to access the value pointed to by a printer.

The program and output are shown in Fig. 11.5. The program clearly shows how we can access the value of a variable using a pointer. You may notice that the value of the pointer **ptr** is 4104 and the value it points to is 10. Further, you may also note the following equivalences:

x = *(&x) = *ptr = y
&x = &*ptr

Program

```
/*************************************************************************** /
/*                    ACCESSING VARIABLES USING POINTERS                    */
/*************************************************************************** /
main( )
{
    int  x, y;
    int  *ptr;

    x = 10;
    ptr = &x;
    y = *ptr;

    printf("Value of x is %d\n\n",x);
    printf("%d is stored at addr %u\n", x, &x);
    printf("%d is stored at addr %u\n", *&x, &x);
    printf("%d is stored at addr %u\n", *ptr, ptr);
    printf("%d is stored at addr %u\n", y, &*ptr);
    printf("%d is stored at addr %u\n", ptr, &ptr);
    printf("%d is stored at addr %u\n", y, &y);

    *ptr = 25;
    printf("\nNow x = %d\n",x);
}
```

Output

```
Value of x is 10

10 is stored at addr 4104
10 is stored at addr 4104
10 is stored at addr 4104
10 is stored at addr 4104
4104 is stored at addr 4106
10 is stored at addr 4108

Now x = 25
```

Fig. 11.5 *Accessing a variable through its pointer*

The actions performed by the program are illustrated in Fig. 11.6. The statement **ptr = &x** assigns the address of **x** to **ptr** and **y = *ptr** assigns the value pointed to by the pointer **ptr** to **y**.

Note the use of the assignment statement

```
*ptr = 25;
```

This statement puts the value of 25 at the memory location whose address is the value of **ptr**. We know that the value of **ptr** is the address of **x** and therefore the old value of **x** is replaced by 25. This, in effect, is equivalent to assigning 25 to **x**. This shows how we can change the value of a variable *indirectly* using a pointer and the *indirection operator*.

11.6 POINTER EXPRESSIONS

Like other variables, pointer variables can be used in expressions. For example, if **p1** and **p2** are properly declared and initialized pointers, then the following statements are valid.

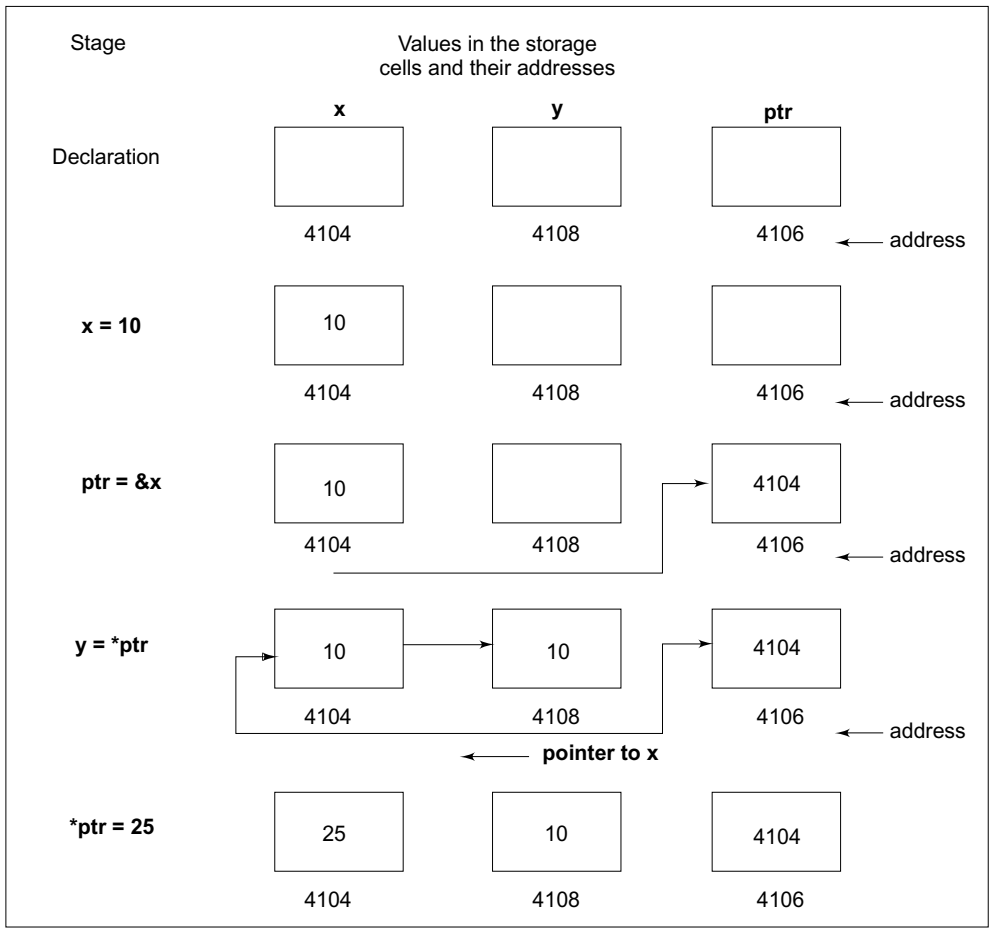


Fig. 11.6 *Illustration of pointer assignments*

```
y    = *p1 * *p2;          same as     (*p1) * (*p2)
sum  = sum + *p1;
z    = 5* – *p2/ *p1;      same as     (5 * (– (*p2)))/(*p1)
*p2  = *p2 + 10;
```

Note that there is a blank space between / and * in the item3 above. The following is wrong.

```
z = 5* – *p2 /*p1;
```

The symbol /* is considered as the beginning of a comment and therefore the statement fails.

C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another. p1 + 4, p2 – 2 and p1 – p2 are all allowed. If **p1** and **p2** are both pointers to the same array, then **p2** – **p1** gives the number of elements between **p1** and **p2**.

We may also use short-hand operators with the pointers.

```
p1++;
— p2;
sum += *p2;
```

In addition to arithmetic operations discussed above, pointers can also be compared using the relational operators. The expressions such as **p1** > **p2**. **p1** = – **p2**, and **p1** != **p2** are allowed. However, any comparison of pointers that refer to separate and unrelated variables makes no sense. Comparisons can be used meaningfully in handling arrays and strings.

We may not use pointers in division or multiplication. For example, expressions such as

p1 / **p2** or **p1** * **p2** or **p1 / 3**

are not allowed. Similarly, two pointers cannot be added. That is, **p1** + **p2** is illegal.

Example 11.3

Write a program to illustrate the use of pointers in arithmetic operations.

The program in Fig. 11.7 shows how the pointer variables can be directly used in expressions. It also illustrates the order of evaluation of expressions. For example, the expression

```
4* – *p2/*p1 + 10
```

is evaluated as follows:

```
((4 * (–(*p2))) / (*p1)) + 10
```

When *p1 = 12 and *p2 = 4, this expression evaluates to 9. Remember, since all the variables are of type **int**, the entire evaluation is carried out using the integer arithmetic.

Program

```
/*************************************************************************** /
/*                    ILLUSTRATION OF POINTER EXPRESSIONS                  */
/*************************************************************************** /
main( )
{
    int a, b, *p1, *p2, x, y, z;

    a  = 12;
    b  = 4;
    p1 = &a;
    p2 = &b;
```

```
    x   = *p1 * *p2 – 6;
    y   = 4* – *p2 / *p1 + 10;

    printf("Address of a = %u\n", p1);
    printf("Address of b = %u\n", p2);
    printf("\n");
    printf("a = %d, b = %d\n", a, b);
    printf("x = %d, y = %d\n", x, y);

    *p2     = *p2 + 3;
    *p1     = *p2 –  5;
    z       = *p1 * *p2 – 6;

    printf("\na = %d, b = %d,", a, b);
    printf("z = %d\n", z):
}
```

Output

```
Address of a = 4020
Address of b = 4016

a = 12, b = 4
x = 42, y = 9

a = 2, b = 7, z = 8
```

Fig. 11.7 *Evaluation of pointer expressions*

11.7 POINTER INCREMENTS AND SCALE FACTOR—ADDRESS ARITHMETIC

We have seen that the pointers can be incremented like

```
p1 = p2 + 2;
p1 = p1 + 1;
```

and so on. Remember, however, an expression like

```
p1++;
```

will cause the pointer **p1** to point to the next value of its type. For example, if **p1** is an integer pointer with an initial value, say 2800, then after the operation **p1** = **p1** + **1**, the value of **p1** will be 2802, and not 2801. That is, when we increment a pointer, its value is increased by the length of the data type that it points to. This length is called the *scale factor*.

For an IBM PC, the lengths of various data types are as follows:

characters	1 byte
integers	2 bytes
floats	4 bytes
long integers	4 bytes
doubles	8 bytes

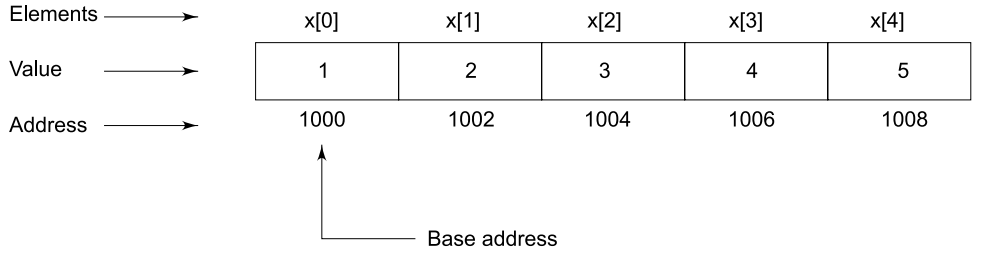
The number of bytes used to store various data types depends on the system and can be found by making use of the **sizeof** operator. For example, if **x** is a variable, then **sizeof(x)** returns the number of bytes needed for the variable.

11.8 POINTERS AND ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element. Suppose we declare an array **x** as follows:

static int x[5] = {1, 2, 3, 4, 5};

Suppose the base address of **x** is 1000 and assuming that each integer requires two bytes, the five elements will be stored as follows:



The name **x** is defined as a constant pointer pointing to the first element, **x[0]** and therefore the value of **x** is 1000, the location where **x[0]** is stored. That is,

x = &x[0] = 1000

If we declare **p** as an integer pointer, then we can make the pointer **p** to point to the array **x** by the following assignment:

p = x;

This is equivalent to

p = &x[0];

(ANSI standard permits a pointer to an array to be obtained by applying the **&** operator to the array name itself.)

Now, we can access every value of **x** using p++ to move from one element to another. The relationship between **p** and **x** is shown below:

p = &x[0] (= 1000)
p+1 = &x[1] (= 1002)
p+2 = &x[2] (= 1004)
p+3 = &x[3] (= 1006)
p+4 = &x[4] (= 1008)

You may notice that the address of an element is calculated using its index and the scale factor of the data type. For instance,

address of **x[3]** = base address + (3 × scale factor of **int**)
= 1000 + (3 × 2) = 1006

When handling arrays, instead of using array indexing, we can use pointers to access array elements. Note that ***(p+3)** gives the value of **x[3]**. The pointer accessing method is much faster than array indexing.

The example 11.4 illustrates the use of pointer accessing method.

Example 11.4

Write a program using pointers to compute the sum of all elements stored in an array.

The program shown in Fig. 11.8 illustrates how a pointer can be used to traverse an array. Since incrementing an array pointer causes it to point to the next element, we need only to add one to **p** each time we go through the loop.

Program

```
/*************************************************************************** /
/*              POINTERS IN ONE-DIMENSIONAL ARRAY                          */
/*************************************************************************** /

main( )
{
    int *p, sum, i;
    static int x[5] = {5, 9, 6, 3, 7};

    i   =  0;
    p   =  x;

    sum = 0;
    printf("Element   Value  Address\n\n");
    while(i < 5)
    {
        printf("x[%d]      %d      %u\n", i, *p, p);
        sum = sum + *p;
        i++, p++;
    }
    printf("\n  Sum   =  %d\n", sum);
    printf("\n &x[0]   =  %u\n", &x[0]);
    printf("\n p        =   %u\n", p);
}
```

Output

```
Element          Value          Address
   x[0]            5              166
   x[1]            9              168
   x[2]            6              170
   x[3]            3              172
   x[4]            7              174

Sum   = 55

&x[0]  = 165

p      = 176
```

Fig. 11.8 *Accessing array elements using the pointer*

It is possible to avoid the loop control variable **i** as shown below:

```
p = x;
while(p <= &x[4])
```

```
{
    sum += *p;
    p++;
}
.....
```

Here, we compare the pointer **p** with the address of the last element to determine when the array has been traversed.

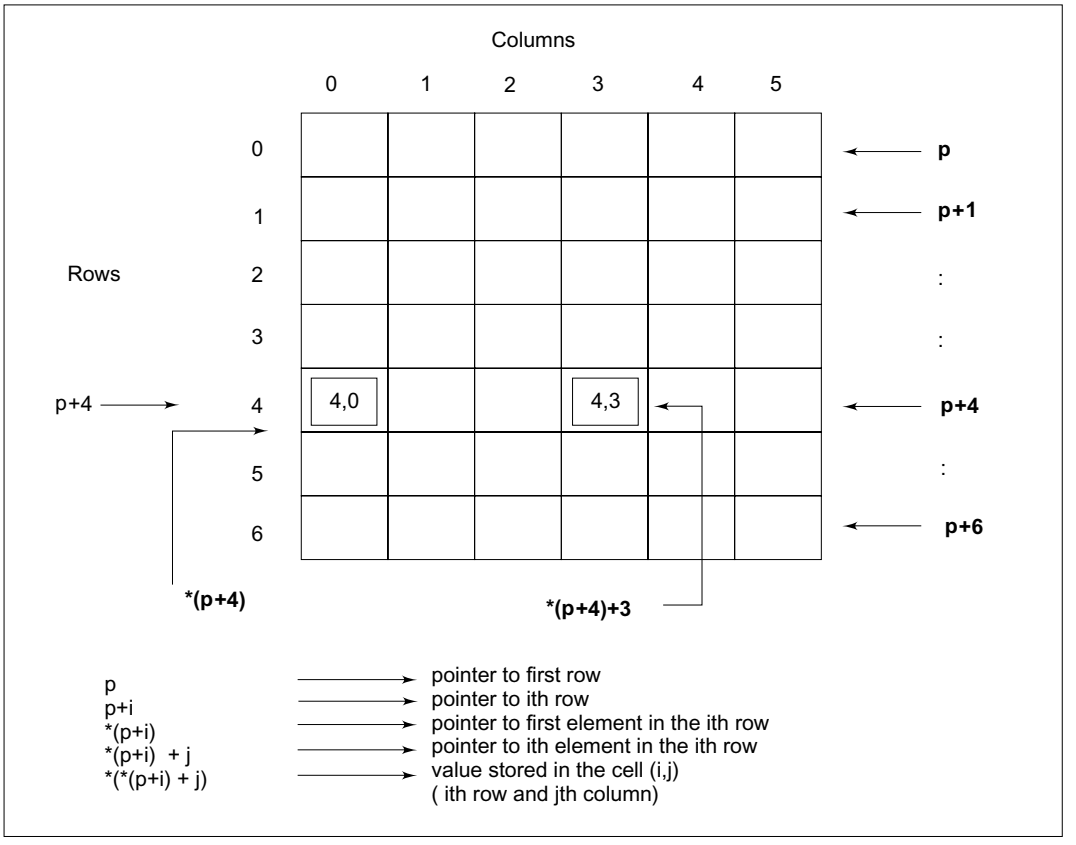


Fig. 11.9 *Pointers to two-dimensional arrays*

Pointers can be used to manipulate two-dimensional arrays as well. We know that in a one-dimensional array **x**, the expression

***(x + i)** or ***(p+i)**

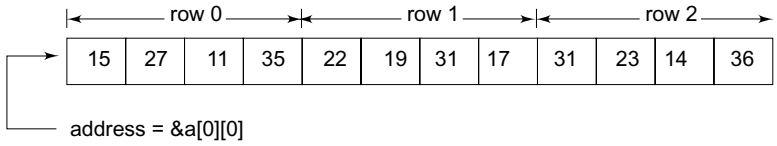
represents the element **x[i]**. Similarly, an element in a two-dimensional array can be represented by the pointer expression as follows:

***(*(a+i)+j)** or ***(*(p+i)+j)**

Figure 11.9 illustrates how this expression represents the element **a[i][j]**. The base address of the array **a** is &a[0][0] and starting at this address, the compiler allocates contiguous space for all the elements, *row-wise*. That is, the first element of the second row is placed immediately after the last element of the first row, and so on. Suppose we declare an array **a** as follows:

```
int a[3][4] = {{15,27,11,35}},
              {22,19,31,17},
              {31,23,14,36}
             };
```

The elements of **a** will be stored as shown below:



If we declare **p** as an **int** pointer with the initial address of &a[0][0], then

a[i][j] is equivalent to *(p+4 x i+j)

You may notice that, if we increment **i** by 1, the **p** is incremented by 4, the size of each row, making **p** element **a[2][3]** is given by ***(p** + **2** × **4** + **3**) = ***(p** + **11**).

This is the reason why, when a two-dimensional array is declared, we must specify the size of each row so that the compiler can determine the correct storage mapping.

11.9 POINTERS AND CHARACTER STRINGS

We know that a string is an array of characters, terminated with a null character. Like in one-dimensional arrays, we can use a pointer to access the individual characters in a string. This is illustrated by the example 11.5.

Example 11.5

Write a program using pointers to determine the length of a character string.

A program to count the length of a string is shown in Fig. 11.10. The statement

char *cptr = name;

Program

```
/*************************************************************************** /
/*                    POINTERS AND CHARACTER STRINGS                       */
/*************************************************************************** /

main( )
{
    char   *name;
    int    length;
    char *cptr = name;

    name = "DELHI";

    while(*cptr != '\0')
    {
        printf("%c is stored at address %u\n", *cptr, cptr);
        cptr++;
    }
    length = cptr – name;
    printf("\nLength of the string = %d\n", length);
```

```
}
Output
D   is stored at address 54
E   is stored at address 55
L   is stored at address 56
H   is stored at address 57
I   is stored at address 58
Length of the string = 5
```

Fig. 11.10 *String handling by pointers*

declares **cptr** as a pointer to a character and assigns the address of the first character of **name** as the initial value. Since a string is always terminated by the null character, the statement

```
while(*cptr != '\0')
```

is true until the end of the string is reached.

When the **while** loop is terminated, the pointer **cptr** holds the address of the null character. Therefore, the statement

```
length = cptr – name;
```

gives the length of the string **name**.

The output also shows the address location of each character. Note that each character occupies one memory cell (byte).

In C, a constant character string always represents a pointer to that string. And therefore the following statements are valid:

```
char    *name;
name    = "Delhi";
```

These statements will declare **name** as a pointer to character and assign to **name** the constant character string "Delhi". You might remember that this type of assignment does not apply to character arrays. The statements like

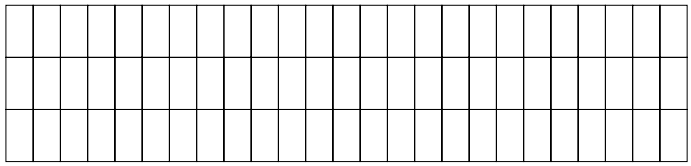
```
char    name[20];
name    = "Delhi"'
```

do not work.

One important used of pointers is in handling of a table of strings. Consider the following array of strings:

```
char name[3][25];
```

This says that the **name** is a table containing three names, each with a maximum length of 25 characters (including null character). The total storage requirements for the **name** table are 75 bytes.



We know that rarely the individual strings will be of equal lengths. Therefore, instead of making each row a fixed number of characters, we can make it a pointer to a string of varying length. For example,

```
static char *name[3] = {
                         "New Zealand",
                         "Australia",
                         "India"
                       };
```

declares **name** to be an *array of three pointers* to characters, each pointer pointing to a particular name as shown below:

name[0] → New Zealand
name[1] → Australia
name[2] → India

This declaration allocates only 28 bytes, sufficient to hold all the characters as shown below:

N	e	w		Z	e	a	l	a	n	d	\0
A	u	s	t	r	a	l	i	a	\0		
I	n	d	i	a	\0						

The following statement would print out all the three names:

```
for(i = 0; i <= 2; i++)
   printf("%s\n", name[i]);
```

To access the j^{th} character in the i^{th} name, we may write as

```
*(name[i]+j]
```

The character arrays with the rows of varying length are called ragged arrays and are better handled by pointers.

Remember the difference between the notations ***p[3]** and **(*p)[3]**. Since * has a lower precedence than [ ], *p[3] declares **p** as an array of 3 pointers while (*p)[3] declares **p** as a pointer to an array of three elements.

11.10 POINTERS AND FUNCTIONS

Pointers as Function Arguments

We have noted earlier that when an array is passed to a function as an argument, only the address of the first element of the array is passed, but not the actual values of the array elements. If **x** is an array, when we call **sort(x)**, the address of **x[0]** is passed to the function **sort**. The function uses this address for manipulating the array elements. Similarly, we can pass the address of a variable as an argument to a function in the normal fashion.

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of variable is known as *call by reference*. (You know, the process of passing the actual value of variables is known as *call by value*.) The function which is called by 'reference' can change the value of the variable used in the call. Consider the following code:

```
main()
{
    int x;
    x = 20;
    change(&x);
    printf("%d\n", x);
}
change(p)
int *p;
{
    *p = *p + 10;
}
```

When the function **change**( ) is called, the address of the variable **x**, not its value, is passed into the function **change**( ). Inside **change**( ), the variable **p** is declared as a pointer and therefore **p** is the address of the variable **x**. The statement

```
*p = *p + 10;
```

means 'add 10 to the value stored at the address **p**'. Since **p** represents the address of **x**, the value of **x** is changed from 20 to 30. Therefore, the output of the program will be 30, not 20.

Thus, call by reference provides a mechanism by which the function can change the stored values in the calling function.

Example 11.6

Write a function using pointers to exchange the values stored in two locations in the memory.

The program in Fig. 11.11 shows how the contents of two locations can be exchanged using their address locations. The function **exchange( )** receives the addresses of the variables **x** and **y** and exchanges their contents.

Program

```
/*****************************************************************************/
/*                    POINTERS AS FUNCTION PARAMETERS                        */
/*****************************************************************************/
main( )
{
    int     x, y;
    x  =  100;
    y  =  200;
    printf("Before exchange:    x = %d   y = %d\n\n", x, y);
    exchange(&x,&y);
    printf("After exchange  :    x = %d   y = %d\n\n", x, y);
}
exchange (a,b)
int *a, *b;
{
    int  t;
    t = *a; /* Assign the value at address a to t */
    *a = *b; /* put the value at b into a */
    *b = t; /* put t into b */
}
```

```
Program
Before exchange:    x = 100  y = 200
After exchange:     x = 200  y = 100
```

Fig. 11.11 *Passing of pointers as function parameters*

You may note the following points:

1. The function parameters are declared as pointers.
2. The dereferenced pointers are used in the function body.
3. When the function is called, the addresses are passed as actual arguments.

Pointer parameters are commonly employed in string functions. Consider the function copy which copies one string to another.

```
copy(s1, s2)
char *s1, *s2;
{
   while((*s1++ = *s2++) ! = '\0')
}
```

This copies the contents of **s2** into the string **s1**. Parameters **s1** and **s2** are the pointers to character strings, whose initial values are passed from the calling function. For example, the calling statement

```
copy(name1, name2);
```

will assign the address of the first element of **name1** to **s1** and the address of the first element of **name2** to **s2**.

Note that the value of ***s2**++ is the character that **s2** pointed to before **s2** was incremented. Due to the postfix ++, **s2** is incremented only after the current value has been fetched. Similarly, **s1** is incremented only after the assignment has been completed.

Each character, after it has been copied, is compared with '\0' and therefore copying is terminated as soon as the '\0' is copied.

Pointers to Pointers

A pointer to a pointer in C can be used to indirectly point to the memory address of a variable. A pointer is a variable that contains the memory address of a variable. As a result, a pointer to a pointer contains the memory address of a pointer, which holds the memory address of a variable.

In C, a pointer is a data type that points to the address where the value of a variable is stored in memory. Therefore, you can have a pointer to a pointer that points to the memory address of a pointer. In C, a pointer to a pointer can point to any type of pointer irrespective of the fact that the pointer is referencing an integer or character value. For example, you can create a pointer **ptr2** pointing to a pointer, **ptr1**, which points to the memory address in which the value of **'Name'** variable is stored as shown under:

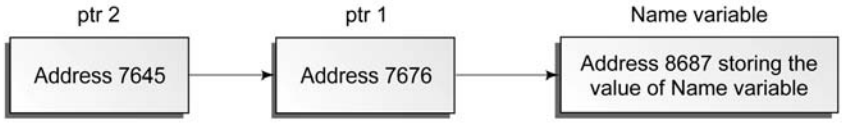


Fig. 11.12

Here, **ptr2,** stored at memory location 7645, points to memory address 7676 in which pointer **ptr1** is contained. The pointer **ptr1** points to the memory address 8687 in which the value of the **'Name'** variable is stored.

Pointers to pointers are particularly useful while handling arrays or while passing address values as parameters to a function. A pointer to a pointer can be declared in the following way:

data type ** pointername

In the above declaration, data type can be any type, such as **char** or **int** and **pointername** can be any valid name. The ** in the above declaration indicates that the pointer with the name **pointername** is a pointer to a pointer, which is containing the memory address of a variable of type, **data type.**

In case of arrays, you can use pointer to a pointer when passing two-dimensional arrays between functions.

As an example, consider the following declarations:

```
int x;
int *ptr1;
ptr1 = &x;
int **ptr2;
ptr2 = &ptr1;
```

In the above declarations, *x* is an integer type variable and **ptr1** is a pointer of integer type holding the address of *x*. ****ptr2** is an integer type pointer to the pointer **ptr1** and contains its address.

The following code is an example of how to use a pointer to pointer in C.

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
clrscr();
int num1, num2, mul, *ptr1,*ptr2,*ptr3,**ptr4;
printf("Enter the first number for multiplication:\n");
scanf("%d",&num1);
printf("Enter the second number for multiplication:\n");
scanf("%d",&num2);
ptr1 = &num1;
ptr2 = &num2;
mul = *ptr1 * *ptr2;
ptr3 = &mul;
ptr4 = &ptr3;
printf("The multiplication of %d and %d is %d", num1, num2,**ptr4);
getch();
return 0;
}
```

Fig. 11.13 *C program for pointer to pointer*

In the above code, **ptr4** is a pointer to **ptr3** pointer, which points to the memory address containing the value generated after the multiplication of two numbers. The **ptr4** pointer is then used in the **printf()** function to display the result of multiplying two integer values. The output of the above code is:

```
Enter the first number for multiplication:
45
Enter the second number for multipliction:
67
The multiplication of 45 and 67 is 3015
```

Pointers to Functions

A function, like a variable, has an address location in the memory. It is, therefore, possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows:

type **(fptr)();**

This tells the compiler that **fptr** is a pointer to a function which returns *type* value. The parentheses around ***fptr** are necessary. Remember that a statement like

type ***gptr();**

would declare **gptr** as a function returning a pointer to *type*.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer. For example, the statements

```
double (*p1)(), mul();
p1 = mul;
```

declare **p1** as a pointer to a function and **mul** as a function and then make **p1** to point to the function **mul**. To call the function **mul**, we may now use the pointer **p1** with the list of parameters. That is,

```
(*p1)(x,y)
```

is equivalent to

```
mul(x,y)
```

Note the parentheses around ***p1**.

Example 11.7

Write a program that uses a function pointer as a function argument.

A program to print the function values over a given range of values is shown in Fig. 11.14. The printing is done by the function **table** by evaluating the function passed to it by the **main**.

With **table**, we declare the parameter **f** as a pointer to a function as follows:

```
double (*f)();
```

The value returned by the function is of type **double**. When **table** is called in the statement

```
table (y, 0.0, 2, 0.5);
```

Program

```
/*************************************************************************** /
/*              ILLUSTRATION OF POINTERS TO FUNCTIONS                      */
/*************************************************************************** /
#include <math.h>
#define PI 3.1415926

main( )
{
    double y( ), cos( ), table( );

    printf("Table of y(x) = 2*x*x–x+1\n\n");
    table(y, 0.0, 2.0, 0.5);

    printf("\nTable of cost(x)\n\n");
    table(cos, 0.0, PI, 0.5);
}
double table(f, min, max, step)
double (*f)( ), min, max, step;
{
    double a, value;
    for(a = min; a <= max; a + = step)
    {
        value = (*f)(a);
        printf("%5.2f %10.4f\n", a, value);
    }
}
double y(x)
double x;
{
    return(2*x*x–x+1);
}
```

Output

```
Table of y(x) = 2*x*x – x+1
0.00            1.0000
0.50            1.0000
1.00            2.0000
1.50            4.0000
2.00            7.0000

Table of cos(x)

0.00             1.0000
0.50             0.8776
1.00             0.5403
1.50             0.0707
2.00            –0.4161
2.50            –0.8011
3.00            –0.9900
```

Fig. 11.14 *Use of pointers to functions*

we pass a pointer to the function **y** as the first parameter of **table**. Note that **y** is not followed by a parameter list.

During the execution of **table**, the statement

```
value = (*f)(a);
```

calls the function **y**, which is pointed to by **f**, passing it the parameter **a**. Thus the function **y** is evaluated over the range 0.0 to 2.0 at the intervals of 0.5.

Similarly, the call

```
table (cos, 0,0, PI, 0.5);
```

passes a pointer to **cos** as its first parameter and therefore, the function **table** evaluates the value of **cos** over the range 0.0 to PI at the intervals of 0.5.

11.11 POINTERS AND STRUCTURES

We know that the name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variables. Suppose **product** is an array variable of **struct** type. The name **product** represents the address of its zeroth element. Consider the following declaration:

```
struct inventory
{
    char  name[30];
    int   number;
    float price;
} product[2], *ptr;
```

This statement declares **product** as an array of two elements, each of the type **struct inventory** and **ptr** as a pointer to data objects of the type **struct inventory**.

The assignment

```
ptr = product;
```

would assign the address of the zeroth element of **product** to **ptr**. That is, the pointer **ptr** will now point to **product[0]**. Its members can be accessed using the following notation.

```
ptr → name
ptr → number
ptr → price
```

The symbol – > is called the *arrow operator* and is made up of a minus sign and a greater than sign. Note that **ptr** – > is simply another way of writing **product[0]**.

When the pointer **ptr** is incremented by one, it is made to point to the next record, i.e., product[1]. The following **for** statement will print the values of members of all the elements of product array.

```
for(ptr = product; ptr < product+2; ptr++)
   printf("%s %d %f\n",ptr→ name,ptr→ number,ptr→ price);
```

We could also use the notation

```
(*ptr).number
```

to access the member **number**. The parentheses around ***ptr** are necessary because the member operator "." has a higher precedence than the operator *.

Example 11.8

Write a program to illustrate the use of structure pointers.

A program to illustrate the use of a structure pointer to manipulate the elements of an array of structures is shown in Fig. 11.15. The program highlights all the features discussed above. Note that the pointer **ptr** (of type **struct invent**) is also used as the loop control index in **for** loops.

Program

```
/*******************************************************************************/
/*                    POINTERS TO STRUCTURE VARIABLES                          */
/*******************************************************************************/

struct invent
{
    char    *name[20];
    int     number;
    float   price;
};
main( )
{
    struct invent product[3], *ptr;
    printf("INPUT\n\n");
    for(*ptr = product; ptr < product+3; ptr++)
        scanf("%fs %d %f". ptr–>name, &ptr–>number, &ptr–>price);
    printf("\nOUTPUT\n\n");

    ptr = product;
    while(ptr < product + 3)
    {
        printf("%–20s %5d % 10.2f\n",
                ptr → name.
                ptr → number;
                ptr → price);
        ptr++;
    }
}
```

Output

```
INPUT

Washing_machine 5 7500
Electric_iron 12 350
Two_in_one 7 1250

OUTPUT

Washing_machine          5          7500.00
Electric_iron           12           350.00
Two_in_one               7          1250.00
```

Fig. 11.15 *Pointer to structure variables*

While using structure pointers, we should take care of the precedence of operators.

The operators '–>' and '.', and ( ) and [ ] enjoy the highest priority among the operators. They bind very tightly with their operands. For example, given the definition

```
struct
{
    int count;
    float *p;
}   *ptr;
```

then the statement

++ptr → count;

increments **count**, not **ptr**. However,

(++ptr) → count;

increments **ptr** first, and then links **count**. The statement

ptr++ → count;

is legal and increments **ptr** after accessing **count**.

The following statements also behave in the similar fashion.

*ptr → p Fetches whatever **p** points to.
*ptr → p++ Increments **p** after accessing whatever it points to.
(*ptr → p)++ Increments whatever **p** points to.
*ptr ++ → p Increments **ptr** after accessing whatever it points to.

In the previous chapter, we discussed about passing of a structure as an argument to a function. We also saw an example where a function receives a copy of an entire structure and returns it after working on it. As we mentioned earlier, this method is inefficient in terms of both, the execution speed and memory. We can overcome this drawback by passing a pointer to the structure and then using this pointer to work on the structure members. Consider the following function:

```
print_invent(item)
struct invent *item;
{
    printf("Name: %s\n", item –> name);
    printf("Price: %f\n", item –> price);
}
```

This function can be called by

```
print_invent(&product);
```

The formal argument **item** receives the address of the structure **product** and therefore it must be declared as a pointer of type **struct invent**, which represents the structure of **product**.

CASE STUDIES

1. Processing of Examination Marks

Marks obtained by a batch of students in the Annual Examination are tabulated as follows.

Student name	*Marks obtained*
S.Laxmi	45 67 38 55
V.S.Rao	77 89 56 69

It is required to compute the total marks obtained by each student and print the rank list based on the total marks.

The program in Fig. 11.16 stores the student names in the array **name** and the marks in the array **marks**. After computing the total marks obtained by all the students, the program prepares and prints the rank list. The declaration

```
int   marks[STUDENTS][SUBJECTS+1];
```

defines **marks** as a pointer to the array's first row. We use **rowptr** as the pointer to the row of **marks**. The **rowptr** is initialized as follows:

```
int (*rowptr)[SUBJECTS+1] = array;
```

Note that **array** is the formal argument whose values are replaced by the values of the actual argument **marks**. The parentheses around ***rowptr** makes the **rowptr** as a pointer to an array of SUBJECTS+1 integers. Remember, the statement

Program

```
/*************************************************************************************/
/*                    POINTERS AND TWO-DIMENSIONAL ARRAYS                            */
/************************************************************************************* /
#define   STUDENTS 5
#define   SUBJECTS 4
#include  <string.h>

main( )
{
    char name[STUDENTS] [20];
    static int marks[STUDENTS][SUBJECTS+1];

    printf(*input students names & their marks in four subjects\n");

    get_list(name, marks, STUDENTS, SUBJECTS);

    get_sum(marks, STUDENTS, SUBJECTS+1);

    printf("\n");
    print_list(name,marks,STUDENTS,SUBJECTS+1);

    get_rank_list(name, marks, STUDENTS, SUBJECTS+1);

    printf("\nRanked List\n\n");
    print_list(name,marks,STUDENTS,SUBJECTS+1);
}
/*  Input student name and marks                */
get_list(string,array,m,n)
int      m, n;
char    *string[ ];
int  array[][SUBJECTS+1];
{
    int  i, j, (*rowptr)[SUBJECTS+1] = array;

    for(i = 0; i < m; i++)
    {
```

```
        scanf("%s", string[i]);
        for{j = 0; j < SUBJECTS; j++)
            scanf("%d", &(*(rowptr + i))[j]);
    }
}
/*  Compute total marks obtained by each student          */
get_sum(array, m, n)
int  m,n;
int  array[ ][SUBJECTS+1];
{
    int  i, j, (*rowptr)[SUBJECTS+1] = array;
    for(i = 0; i < m; i++}
    {
        (*(rowptr + i))[n–1] = 0;
        for(j = 0; j < n–1; j++)
            (*(rowptr + i))[n–1] + = (*(rowptr + i))[j];
    }
}
/*  Prepare rank list based on total marks          */
get_rank_list(string,array,m,n)
int     m,n;
char    *string[ ];
int     array[ ][SUBJECTS+1];
{
    int i, j, k, (*rowptr)[SUBJECTS+1] = array;
    char *temp;
    for(i = 1; i < = m-1; i++)
        for(j = 1; j <= m-i; j++)
            if((*(rowptr + j–1))[n–1] <(*(rowptr + j))[n–1])
            {
                swap_string(string[j-1], string[j]);
                for(k = 0; k < n; k++)
                    swap_int(&(*(rowptr + j–1))[k],&(*(rowptr+j))[k]);
            }
}
/*                  Print out the ranked list                */
print_list(string,array, m, n)
int     m,n;
char    string[ ];
int     array[ ][SUBJECTS+1];
{
    int  i, j, (*rowptr)[SUBJECTS+1] = array;
    for(i = 0; i < m; i++)
    {
        printf("%-20s", string[i]);
        for(j = 0; j < n; j++)
            printf("%5d", (*(rowptr + i))[j]);
            printf("\n");
    }
}
/*          Exchange of integer values          */
swap_int(p,q)
int *p, *q;
{
```

```
    int  temp;

    temp   = *p;
    *p     = *q;
    *q     = temp;
}
/*          Exchange of strings              */

swap_string(s1, s2)
char        s1[], s2[];
{
    char swaparea[256];
    int  i;

    for(i = 0; i < 256; i++)
        swaparea[i] = '\0';
    i = 0;
    while(s1[i] != '\0' && i < 256)
    {
        swaparea[i] = s1[i];
        i++;
    }
    i = 0;
    while(s2[i] != '\0' && i < 256)
    {
    s1[i] = s2[];
    s1[++i} = '\0';
    }
    i = 0;
    while(swaparea[i] ! = '\0')
    {
        s2[i] = swaparea[i];
        s2[++i] = '\0';
    }
}
```

Output

```
Input students names & their marks in four subjects
S.Laxmi    45   67   38   55
V.S.Rao    77   89   56   69
A.Gupta    66   78   98   45
S.Mani     86   72    0   25
R.Daniel   44   55   66   77

S. Laxmi                    45   67   38   55  205
V.S.Rao                     77   89   56   69  291
A.Gupta                     66   78   98   45  287
S.Mani                      86   72    0   25  183
R.Daniel                    44   55   66   77  242

Ranked List

V.S.Rao                     77   89   56   69  291
A.Gupta                     66   78   98   45  287
R.Daniel                    44   55   66   77  242
S.Laxmi                     45   67   38   55  205
S.Mani                      86   72    0   25  183
```

Fig. 11.16 *Preparation of the rank list of a class of students*

```
int *rowptr[SUBJECTS+1];
```

would declare **rowptr** as an array of SUBJECTS+1 elements.

When we increment the **rowptr** (by **rowptr+1**), the incrementing is done in units of the size of each row of **array**, making **rowptr** point to the next row. Since **rowptr** points to a particular row, **(*rowptr)[x]** points to the xth element in the row.

2. Inventory Updating

The price and quantity of items stocked in a store changes every day. They may either increase or decrease. The program in Fig. 11.17 reads the incremental values of price and quantity and computes the total value of the items in stock.

Program

```
/*****************************************************************************/
/*                    STRUCTURES AS FUNCTION PARAMETERS                      */
/*                          Using structure pointers                         */
/*****************************************************************************/
struct stores
{
    char name[20];
    float    price;
    int      quantity;
};
main( )
{
    void update( );
    float        mul( ), p_increment, value;
    int          q-increment;

    static struct stores item = {
    "XYZ", 25.75, 12};
    struct stores *ptr = &item;

    printf("\nInput increment values:");
    printf("price increment and quantity increment/n");
    scanf("%f %d", &p_increment, &p_increment);
/*---------------------------------------------------------------------------*/
                    update(&item, p_increment, q_increment);
/*---------------------------------------------------------------------------*/
    printf("Updated values of item\n\n");
    printf("Name      : %s\n",ptr–>name);
    printf("Price     :  %f\n"ptr–>price);
    printf("Quantity  :  %d\n", ptr–>quantity);
/*---------------------------------------------------------------------------*/
    value = mul(&item);
/*---------------------------------------------------------------------------*/
    printf("\nValue of the item = %f\n", value);
}
void update(product, p, q)
struct stores *product;
float    p;
int      q;
{
```

```
    product–>price += p;
    product–>quantity += q;
}
float mul(stock)
struct stores *stock;
{
    return(stock–>price * stock–>quantity);
}
```

Output

```
Input increment values: price increment and quantity increment
10      12
Updated values of item
Name      : XYZ
Price     : 35.750000
Quantity  : 24
Value of the item = 858.000000
```

Fig. 11.17 *Use of structure pointers as function parameters*

The program illustrates the use of structure pointers as function parameters. **&item**, the address of the structure **item**, is passed to the functions **update(product,p,q)** and **mul(stock)**. The formal arguments **product** and **stock**, which receive the value of **&item**, are declared as pointers of type **struct stores**.

POINTS ON POINTERS

While pointers provide enormous power and flexibility to the programmers, they may cause nightmares if they are not properly handled. It is advisable to master all the intricacies of pointers before using them. We should make sure that we know where each pointer is pointing in a program. Here are some general observations and common errors that might be useful to remember.

1. A pointer contains garbage until it is initialized. Since compilers cannot detect uninitialized or wrongly initialized pointers, the errors may not be known until we execute the program. Remember that even if we are able to locate a wrong result, it may not provide any evidence for us to suspect problems in the pointers.
2. The abundance of C operators is another cause of confusion that leads to errors. For example, the expressions such as

   `*ptr++, *p[ ], (*p)[ ], (ptr).member,`

   etc., should be carefully used. A proper understanding of the precedence and associativity rules of C plays a critical role in pointer applications.
3. When an array is passed as an argument to a function, a pointer is actually passed. In the header function, we must declare such arrays with proper sizes, except the first, which is optional.
4. If we define an array in a function with **auto** class, we cannot pass the address of that array back to the **main** for subsequent work.
5. A very common error is to use (or not to use) the address operator**&** and the indirection operator * in certain places. The user-friendly compiler may not warn you of such mistakes.

Review Questions and Exercises

11.1 What is a pointer?

11.2 How is a pointer initialized?

11.3 Explain the effects of the following statements:

(a) `int a, *b = &a;`
(b) `int p, *p;`
(c) `char *s;`
(d) `a = (float *) &x;`
(e) `double (*f)( );`

11.4 If **m** and **n** have been declared as integers and **p1** and **p2** as pointers to integers, then state errors, if any, in the following statements:

(a) `p1 = &m;`
(b) `p2 = n;`
(c) `*p1 = &n`
(d) `p2 = &*&m;`
(e) `m = p2–p1;`
(f) `p1 = &p2;`
(g) `m = *p1 + *p2++.`

11.5 Distinguish between (*m)[5] and *m[5].

11.6 State whether each of the following statements is true or false. Give reasons.

(a) An integer can be added to a pointer.
(b) A pointer can never be subtracted from another pointer.
(c) When an array is passed as an argument to a function, a pointer is passed.
(d) Pointers cannot be used as formal parameters in headers to function definitions.
(e) Value of a local variable in a function can be changed by another function.

11.7 Explain the difference between 'call by reference' and call by value'.

11.8 Write a program using pointers to read in an array of integers and print its elements in reverse order.

11.9 We know that the roots of a quadratic equation of the form

$$ax^2 + bx + c = 0$$

are given by the following equations;

$$x_1 = \frac{-b + \text{square root } (b^2 - 4ac)}{2a}$$

$$x_2 = \frac{-b - \text{square root } (b^2 - 4ac)}{2a}$$

Write a function to calculate the roots. The function must use two pointer parameters, one to receive the coefficients a, b, and c, and the other to send the roots to the calling function.

11.10 Write a function that receives a sorted array of integers and an integer value, and inserts the value in its correct place.

11.11 Write a function using pointers to add two matrices and to return the resultant matrix to the calling function.

11.12 Using pointers, write a function that receives a character string and a character as argument and deletes all occurrences of this character in the string. The function should return the corrected string with no holes.

11.13 Write a function **day_name** that receives a number n and returns a pointer to a character string containing the name of the corresponding day. The day names should be kept in a **static** table of character strings local to the function.

11.14 Write a program to read in an array of names and to sort them in alphabetical order. Use **sort** function that receives pointers to the functions **stremp** and **swap. sort** in turn should call these functions via the pointers.

11.15 Given an array of sorted list of integer numbers, write a function to search for a particular item, using the method of *binary search*. And also show how this function may be used in a program. Use pointers and pointer arithmetic.

Hint: In binary search, the target value is compared with the array's middle element. Since the table is sorted, if the required value is smaller, we know that all values greater than the middle element can be ignored. That is, in one attempt, we eliminate one-half the list. This search can be applied recursively till the target value is found.

UNIT IV

Chapter 12

Structures and Unions

12.1 INTRODUCTION

We seen that arrays can be used to represent a group of data items that belong to the same type, such as **int** or **float**. However, if we want to represent a collection of data items of different types using a single name, then we cannot use an array. Fortunately, C supports a constructed data type known as *structure*, which is a method for packing data of different types. A structure is a convenient tool for handling a group of logically related data items. For example, it can be used to represent a set of attributes, such as student_name, roll_number and marks. The concept of a structure is analogous to that of a 'record' in many other languages.

Structures help to organize complex data in a more meaningful way. It is a powerful concept that we may often need to use in our program design. The chapter is devoted to the study of structures and their applications in program development. Another related concept known as *unions is* also discussed.

12.2 STRUCTURE DEFINITION—DECLARING STRUCTURES

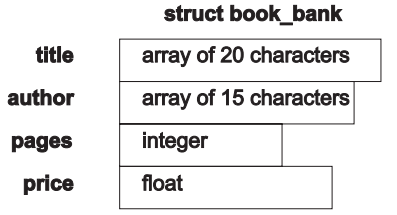
A structure definition creates a format that may be used to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book database consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

```
struct book_bank
{
    char  title[20];
    char  author[15];
    int   pages;
    float price;
};
```

The keyword **struct** declares a structure to hold the details of four fields, namely **title**, **author**, **pages**, and **price**. These fields are called *structure elements or members*. Each member may belong to a differ-

ent type of data. **book_bank** is the name of the structure and is called the *structure tag*. The tag name may be used subsequently to declare variables that have the tag's structure.

Note that the above declaration has not declared any variables. It simply describes a format called *template* to represent information as shown below.



The general format of a structure definition is as follows:

```
struct   tag_name
{
         data_type     member1;
         data_type     member2;
            . . .          . . .
            . . .          . . .
};
```

We can declare structure variables using the tag name anywhere in the program. For example, the statement

```
struct book_bank book1, book2, book3;
```

declares **book1**, **book2,** and **book3** as variables of type **struct book_bank**.

Each one of these variables has four members as specified by the template. The complete declaration might look like this:

```
struct book_bank
{
   char  title[20];
   char  author[15];
   int   pages;
   float price;
};
struct book_bank book1, book2, book3;
```

Remember that the members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as **book1**.

In defining a structure you may note the following syntax:

1. The template is terminated with a semicolon.
2. While the entire declaration is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name such as **book_bank** can be used to declare structure variables of its type, later in the program.

It is also allowed to combine both the template declaration and variables declaration in one statement. The declaration

```
struct book_bank
{
   char  title[20];
   char  author[15];
   int   pages;
   float price;
} book1, book2, book3;
```

is valid. The use of tag name is optional. For example,

```
struct
{
   ........
   ........
   ........
} book1, book2, book3;
```

declares **book1, book2**, and **book3** as structure variables representing three books, but does not include a tag name for later use in declarations.

Normally, structure definitions appear at the beginning of the program file, before any variables or functions are defined. They may also appear before the **main**, along with macro definitions, such as **#define**. In such cases, the definition is *global* and can be used by other functions as well.

12.3 GIVING VALUES TO MEMBERS

We can assign values to the members of a structure in a number of ways. As mentioned earlier, the members themselves are not variables. They should be linked to the structure variables in order to make them meaningful members. For example, the word **title**, has no meaning where as the phrase 'title of book3' has a meaning. The link between a member and a variable is established using the *member operator* '.' which is also known as 'dot operator' or 'period operator'. For example,

```
book1.price
```

is the variable representing the price of **book1** and can be treated like any other ordinary variable. Here is how we would assign values to the members of **book1**:

```
strcpy(book1.title, "BASIC");
strcpy(book1.author, "Balagurusamy");
book1.pages  = 250;
book1.price    = 28.50;
```

We can also use **scanf** to give the values through the keyboard.

```
scanf("%s\n", book1. title);
scanf("%d\n", &book1.pages);
```

are valid input statements.

Accessing Structures

Structure definition along with the program is shown in Fig. 12.1. The **scanf** and **printf** functions illustrate how the member operator '.' is used to link the structure members to the structure variables. The variable name with a period and the member name is used like an ordinary variable.

Example 12.1

Define a structure type, **struct personal**, that would contain person name, date of joining and salary. Using this structure, write a program to read this information for one person from the keyboard and print the same on the screen.

Program

```
/*******************************************************************************/
/*   DEFINING AND ASSIGNING VALUES TO STRUCTURE MEMBERS     */
/*******************************************************************************/

struct personal
{
    char    name[20];
    int     day;
    char    month[10];
    int     year;
    float   salary;
};

main()
{
    struct personal person;

    printf("Input Values\n");
    scanf("%s %d %s %d %f",
                  person.name,
                 &person.day,
                  person.month,
                 &person.year,
                 &person.salary);
    printf("%s %d %s d%, %.2f\n",
                 person.name,
                 person.day,
                 person.month,
                 person.year,
                 person.salary);
}
```

Output

```
Input Values
M.L. Goel 10 January 1945 4500
M.L. Goel 10 January 1945 4500.00
```

Fig. 12.1 *Defining and accessing structure members*

12.4 STRUCTURE INITIALIZATION

Like any other data type, a structure variable can be initialized. However, a structure must be declared as **static** if it is to be initialized inside a function (similar to arrays).

Note: This condition is not applicable to ANSI compilers. The ANSI standard permits initialization of structure variables with **auto** storage class. We could therefore delete the word **static** when using ANSI compilers. However, the use of the keyword **static** in the programs would enable us to run them under both the old and ANSI standard compilers.

```
main( )
{
   static struct
   {
      int weight;
      float height;
   }
   student = {60, 180.75};
   .....
   .....
}
```

This assigns the value 60 to **student.weight** and 180.75 to **student.height**. There is a one-to-one correspondence between the members and their initializing values.

A lot of variation is possible in initializing a structure. The following statements initialize two structure variables. Here, it is essential to use a tag name.

```
main( )
{
   struct st_record
   {
      int    weight;
      float  height;
   };
   static struct st_record student1 = {60, 180.75};
   static struct st_record student2 = {53, 170.60};
   .....
   .....
}
```

Another method is to initialize a structure variable outside the function as shown below:

```
struct st_record    /*  No static word    */
{
   int    weight,
   float  height;
}    student1 = {60, 180.75};
main()
{
   static struct st_record student2 = {53, 170.60};
   .....
   .....
}
```

C language does not permit the initialization of individual structure members within template. The initialization must be done only in the declaration of the actual variables.

12.5 COMPARISON OF STRUCTURE VARIABLES

Two variables of the same structure type can be compared the same way as ordinary variables. If **person1** and **person2** belong to the same structure, then the following operations are valid:

Operation	Meaning
person1 = **person2**	Assign **person2** to **person1**.
person1 == **person2**	Compare all members of **person1** and **person2** and return 1 if they are equal, 0 otherwise.
person1 != **person2**	Return 1 if all the members are not equal, 0 otherwise.

Note that not all compilers support these operations. For example, Microsoft C version does not permit any logical operations on structure variables. In such cases, individual members can be compared using logical operators.

Example 12.2

Write a program to illustrate the comparison of structure variables.

The program shown in Fig. 12.2 illustrates how a structure variable can be copied into another of the same type. It also performs memberwise comparison to decide whether two structure variables are identical.

12.6 ARRAYS OF STRUCTURES

We use structures to describe the format of a number of related variables. For example, in analysing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable. For example,

```
struct class student[100];
```

defines an array called **student**, that consists of 100 elements. Each element is defined to be of the type **struct class**. Consider the following declaration:

```
struct marks
{
    int  subject1;
    int  subject2;
    int  subject3;
};
main( )
{
    static struct marks student[3] =
        {{45,68,81}, {75,53,69}, {57,36,71}};
```

This declares the **student** as an array of three elements **student[0]**, **student[1]**, and **student[2]** and initializes their members as follows:

```
student[0].subject1 =  45;
student[0].subject2 =  68;
        .....
        .....
student[2].subject3 =  71;
```

Program

```
/*************************************************************************************/
/*                    COMPARISON OF STRUCTURE VARIABLES                              */
/*************************************************************************************/

struct class
{
    int     number;
    char    name[20];
    float   marks;
};

main( )
{
    int x;
    static struct class student1 = {111,"RAO",72.50};
    static struct class student2 = {222,"Reddy" , 67.00};
    struct class student3;

    student3 = student2;

    x = ((student3.number   == student2.number)&&
        (student3.marks     == student2.marks))? 1:0;
    if(x == 1)
    {
        printf("\nstudent2 and student3 are same\n\n");
        printf("%d %s %f\n",    student3.number,
                                student3.name,
                                student3.marks);
    }
else
    printf("\nstudent2 and student3 are different\n\n");
}
```

Output

```
student2 and student3 are same

222 Reddy 67.000000
```

Fig. 12.2 *Comparing and copying structure variables*

Note that the array is declared just as it would have been, with any other array. Since **student** is an array, we use the usual array-accessing methods to access individual elements and then the member operator to access members.

An array of structures is stored inside the memory in the same way as a multi-dimensional array. The array **student** actually looks as shown in Fig. 12.3.

Example 12.3
For the **student** array discussed above, write a program to calculate the subjectwise and studentwise totals and store them as a part of the structure.

student[0].subject1	45
. subject2	68
. subject3	81
student[1].subject1	75
. subject2	53
. subject3	69
student[2].subject1	57
. subject2	36
. subject3	71

Fig. 12.3 *The array **student** inside memory*

The program is shown in Fig. 12.4. We have declared a four-member structure, the fourth one for keeping the student-totals. We have also declared an array **total** to keep the subject-totals and the grand-total. The grand-total is given by **total.total**. Note that a member name can be any valid C name and can be the same as an existing structure variable name. The linked name **total.total** represents the **total** member of the structure variable total.

Program

```
/*************************************************************************************/
/*                              ARRAYS OF STRUCTURES                                 */
/*************************************************************************************/
struct marks
{
    int  sub1;
    int  sub2;
    int  sub3;
    int  total;
};
main()
{
    int  i;
    static struct marks student[3] =    {   {45,67,81,0},
                                            {75,53,69,0},
                                            {57,36,71,0}};
    static struct marks total;
    for(i = 0; i <= 2; i++)
    {
        student[i].total =  student[i].sub1 +
                            student[i].sub2 +
                            student[i].sub3;
        total.sub1 = total.sub1 + student[i].sub1;
        total.sub2 = total.sub2 + student[i].sub2;
```

```
        total.sub3 = total.sub3   + student[i].sub3;
        total.total = total.total     + student[i].total;
}
printf("STUDENT                   TOTAL\n\n\");
for(i = 0; i <= 2; i++)
    printf("Student[%d]       %d\n", i+1.student[i].total);
printf("\n SUBJECT          TOTAL\n\n");

    printf("%s     %d\n%s       %d\n%s       %d\n",
        "Subject   1        ",   total.sub1,
        "Subject   2        ",   total.sub2,
        "Subject   3        ",   total.sub3);
    printf("\nGrand Total = %d\n", total.total);
}
```

Output

```
STUDENT          TOTAL

Student[1]         193
Student[2]         197
Student[3]         164

SUBJECT          TOTAL

Subject 1          177
Subject 2          156
Subject 3          221

Grand Total = 554
```

Fig. 12.4 *Illustration of subscripted structure variables*

12.7 ARRAYS WITHIN STRUCTURES

C permits the use of arrays as structure members. We have already used arrays of characters inside a structure. Similarly, we can use single- or multi-dimensional arrays of type **int** or **float**. For example, the following structure declaration is valid:

```
struct marks
{
    int     number;
    float subject[3];
}       student[2];
```

Here, the member **subject** contains three elements, **subject[0]**, **subject[1]** and **subject[2]**. These elements can be accessed using appropriate subscripts. For example, the name

student[1].subject[2];

would refer to the marks obtained in the third subject by the second student.

Example 12.4

Rewrite the program of Example 12.3 using an array member to represent the three subjects.

The modified program is shown in Fig. 12.5. You may notice that the use of array name for subjects has simplified the code.

12.8 STRUCTURES WITHIN STRUCTURES—NESTED STRUCTURES

Structures within a structure means nesting of structures. Nesting of structures is permitted in C. Let us consider the following structure defined to store information about the salary of employees.

```
struct salary
{
    char  name[20];
    char  department[10];
    int   basic_pay;
    int   dearness_allowance;
    int   house_rent_allowance;
    int   city_allowance;
}
employee;
```

This structure defines name, department, basic pay and three kinds of allowances. We can group all the items related to allowance together and declare them under a substructure as shown below:

```
struct salary
{
    char  name[2];
    char  department[10];
    struct
    {
        int  dearness;
        int  house_rent;
        int  city;
    }
    allowance;
}
employee;
```

The salary structure contains a member named **allowance** which itself is a structure with three members. The members contained in the inner structure namely **dearness**, **house_rent**, and **city** can be referred to as

```
employee.allowance.dearness
employee.allowance.house_rent
employee.allowance.city
```

An inner-most member in a nested structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most) with the member using dot operator. The following being invalid:

Program

```
/*************************************************************************/
/*                    ARRAYS WITHIN A STRUCTURE                          */
/*************************************************************************/

main( )
{
    struct      marks
    {
        int  sub[3];
        int  total;
    };
    static struct marks student[3] =
    {45,67,81,0,75,53,69,0,57,36,71,0};
    static struct marks total;
    int i,j;

    for(i = 0; i <= 2; i++)
    {
        for (j = 0; j < = 2; j++)
        {
            student[i].total += student[i].sub[j];
            total sub[j] += student[i].sub[j];
        }
        total.total += student[i].total;
    }

    printf("STUDENT            TOTAL\n\n");
    for(i = 0; i <= 2; i++)
        printf("Student[%d]        %d\n", i+1.student[i].total);

    printf("\nSUBJECT            TOTAL\n\n");
    for(j = 0; j <= 2; j+ +)
        printf("Subject-%d         %d\n",j+1.total.sub[j]);

    printf("\nGrand Total    =   %d\n",total.total);
}
```

Output

```
STUDENT          TOTAL

Student[1]         193
Student[2]         197
Student[3]         164

SUBJECT          TOTAL

Subject-1          177
Subject-2          156
Subject-3          221
Grand Total    =   554
```

Fig. 12.5 *Use of subscripted members in structures*

employee.allowance	(actual member is missing)
employee.house_rent	(inner structure variable is missing)

An inner structure can have more than one variable. The following form of declaration is legal:

```
struct salary
{
    .....
    struct
    {
        int dearness;
        .....
    }
    allowance;
    arrears;
}
employee[100];
```

The inner structure has two variables, **allowance** and **arrears**. This implies that both of them have the same structure template. Note the comma after the name **allowance**. A base member can be accessed as follows:

```
employee[1].allowance.dearness
employee[1].arrears.dearness
```

We can also use tag names to define inner structures. Example:

```
struct pay
{
    int     dearness;
    int     house_rent;
    int     city;
};
struct salary
{
    char  name[20];
    char  department[10];
    struct pay allowance;
    struct pay arears;
};
struct salary employee[100];
```

pay template is defined outside the **salary** template and is used to define the structure of **allowance** and **arrears** inside the **salary** structure.
It is also permissible to nest more than one type of structures.

```
struct personal_record
{
    struct name_part name;
    struct addr_part address;
    struct date date_of_birth;
    . . . .
    . . . .
};
struct personal_record person1;
```

The first member of this structure is **name** which is of the type struct **name_part**. Similarly, other members have their structure types.

12.9 STRUCTURES AND FUNCTIONS

We know that the main philosophy of C language is the use of the functions. Therefore, it is natural that C supports the passing of structure values as arguments to functions. There are three methods by which the values of a structure can be transferred from one function to another.

The first method is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the most elementary method and becomes unmanageable and inefficient when the structure size is large.

The second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function. All compilers may not support this method of passing the entire structure as a parameter.

The third approach employs a concept called *pointers* to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This is similar to the way arrays are passed to functions. This method is more efficient as compared to the second one.

In this section, we discuss in detail the second method, while the third approach using pointers is discussed in the next chapter, where pointers are dealt in detail.

The general format of sending a copy of a structure to the called function is:

```
function name (structure variable name)
```

The called function takes the following form:

```
data_type function name(st_name)
struct_type st_name;
{
    ......
    ......
    return(expression);
}
```

The following points are important to note:

1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as **struct** with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same **struct** type.
3. The **return** statement is necessary only when the function is returning some data. The *expression* may be any simple variable or structure variable or an expression using simple variables.
4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called function must be declared in the calling function for its type, if it is placed after the calling function.

Example 12.5

Write a simple program to illustrate the method of sending an entire structure as a parameter to a function.

A program to update an item is shown in Fig. 12.6. The function **update** receives a copy of the structure variable **item** as one of its parameters. Note that both the function **update** and the formal parameter **product** are declared as type **struct stores**. It is done so because the function uses the parameter **product** to receive the structure variable **item** and also to return the updated values of **item**.

The function **mul** is of **type** float because it returns the product of **price** and **quantity**. However, the parameter **stock**, which receives the structure variable **item** is declared as type **struct stores**.

The entire structure returned by **update** can be copied into a structure of identical type. The statement

item = update(item,p-increment,q_increment);

replaces the old values of **item** by the new ones.

Program

```
/ ********************************************************************************** /
/*                    STRUCTURE AS FUNCTION PARAMETERS                              */
/*                       Passing a copy of the entire structure                     */
/ ********************************************************************************** /
struct stores
{
    chart   name[20];
    float   price;
    int     quantity,
};
main( )
{
    struct stores update( );
    float   mul(), p_increment, value;
    int     q_increment;

    static struct stores item = {"XYZ", 25.75, 12};

printf("\nInput increment values:");
printf(" price increment and quantity increment\n");
scanf("%f %d", &p_increment, &q_increment);
/*------------------------------------------------------------------------------------------------- */
    item = update(item, p_increment, q_increment);
/*------------------------------------------------------------------------------------------------- */
    printf("Update values of item\n\n");
    printf("Name       : %s\n",item.name);
    printf("Price      : %f\n",item.price);
    prinft("Quantity   : %d\n",item.quantity);

/*-------------------------------------------------------------------------------------------------*/
    value = mul(item);
/*-------------------------------------------------------------------------------------------------*/
    printf("\nValue of the item = %f\n", value);
}

struct stores update(product, p, q)
struct store product;
float   p;
int     q;
{
    product.price + = p;
    product.quantity += q;
    return(product);
}
```

```
float mul(stock)
struct stores stock;
{
    return(stock.price * stock.quantity);
}
```

Output

```
Input increment values:      price increment and quantity increment
    10      12
    Update values of item
    Name       : XYZ
    Price          : 35.750000
    Quantity       : 24
    Value of the item = 858.000000
```

Fig. 12.6 *Using structure as a function parameter*

You may notice that the template of **stores** is defined before **main()**. This has made the data type **struct stores** as *global* and has enabled the functions **update** and **mul** to make use of this definition.

12.10 UNIONS

Unions are a concept borrowed from structures and therefore follow the same syntax as structures. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword **union** as follows:

```
union item
{
    int    m;
    float  x;
    char   c;
} code;
```

This declares a variable **code** of type **union item**. The union contains three members, each with a different data type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

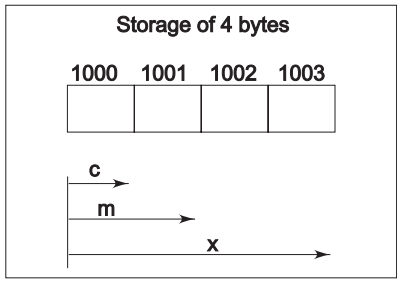


Fig. 12.7 *Sharing of a storage locating by union members*

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above the member **x** requires 4 bytes which is the largest among the members.

Figure 12.7 shows how all the three variables share the same address. This assumes that a **float** variable requires 4 bytes of storage.

To access a union member, we can use the same syntax that we use for structure members. That is,

code.m
code.x
code.c

are all valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored. For example, the statements such as

```
code.m = 379;
code.x = 7859.36;
printf("%d", code.m);
```

would produce erroneous output (which is machine dependent).

In effect, a union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supercedes the previous member's value.

Unions may be used in all places where a structure is allowed. The notation for accessing a union member which is nested inside a structure remains the same as for the nested structures.

typedef

C provides the **typedef** keyword that allows you to specify a new name for a data type already provided in the C programming language. In other words, you can use the **typedef** keyword to specify an identifier for an existing data type. The declaration for the **typedef** keyword is:

typedef data type newname

In the above example, *data type* represents the existing data type for which you want to specify an identifier and newname refers to that identifier.

As an example, consider the following declaration:

typedef int num;

Here, **int** is the data type available in C and a new identifier **num** has been specified for this data type. The following code shows an example for using the **typedef** keyword:

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
typedef int num;
num a, b, add;
a=10;
b=20;
add=0;
add = a+b;
clrscr();
printf("The sum of %d and %d is: %d",a,b,add);
getch();
return 0;
}
```

Fig. 12.8 *C program for typedef*

In the above code, the **typedef** keyword is used to specify a new name for the **int** data type. The output of the above code is:

```
The sum of 10 and 20 is: 30
```

12.11 SIZE OF STRUCTURES

We normally use structures, unions, and arrays to create variables of large sizes. The actual size of these variables in terms of bytes may change from machine to machine. We may use the unary operator **sizeof** to tell us the size of a structure (or any variable). The expression

sizeof (**struct** ***x***)

will evaluate the number of bytes required to hold all the members of the structure **x**. If **y** is a simple structure variable of type **struct x**, then the expression

sizeof(y)

would also give the same answer. However, if **y** is an array variable of type **struct x**, then

sizeof(y)

would give the total number of bytes the array **y** requires.

This kind of information would be useful to determine the number of records in a database. For example, the expression

size of(y)/**sizeof**(x)

would give the number of elements in the array **y**.

12.12 BIT FIELDS

So far, we have been using integer fields of size 16 bits to store data. There are occasions where data items require much less than 16 bits space. In such cases, we waste memory space. Fortunately, C permits us to use small *bit fields* to hold data items and thereby to pack several data items in a word of memory. Bit fields allow direct manipulation of string of a string of preselected bits as if it represented an integral quantity.

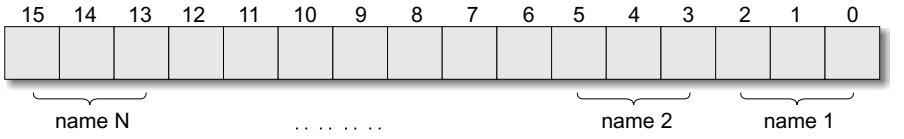
A *bit field* is a set of adjacent bits whose size can be from 1 to 16 bits in length. A word can therefore be divided into a number of bit fields. The name and size of bit fields are defined using a structure. The general form of bit field definition is:

```
struct tag-name
{
        data-type name1: bit–length;
        data-type name2: bit–length;
        . . . . . .
        . . . . . .
        data-type nameN: bit-length;
}
```

The *data-type* is either **int** or **unsigned int** or **signed int** and the *bit-length* is the number of bits used for the specified name. Remember that a **signed** bit field should have at least 2 bits (one bit for sign).

Note that the field name is followed by a colon. The *bit-length* is decided by the range of value to be stored. The largest value that can be stored is 2^{n-1}, where **n** is bit-length.

The internal representation of bit fields is machine dependent. That is, it depends on the size of **int** and the ordering of bits. Some machines store bits from left to right and others from right to left. The sketch below illustrates the layout of bit fields, assuming a 16-bit word that is ordered from right to left.



There are several specific points to observe:

1. The first field always starts with the first bit of the word.
2. A bit field cannot overlap integer boundaries. That is, the sum of lengths of all the fields in a structure should not be more than the size of a word. In case, it is more, the overlapping field is automatically forced to the beginning of the next word.
3. There can be unnamed fields declared with size. Example:

   Unsigned : *bit-length*

   Such fields provide padding within the word.
4. There can be unused bits in a word.
5. We cannot take the address of a bit field variable. This means we cannot use **scanf** to read values into bit fields. We can neither use pointer to access the bit fields.
6. Bit fields cannot be arrayed.
7. Bit fields should be assigned values that are within the range of their size. If we try to assign larger values, behaviour would be unpredicted.

Suppose, we want to store and use personal information of employees in compressed form, this can be done as follows:

```
struct personal
{
unsigned sex          :    1
unsigned age          :    7
unsigned m_status     :    1
unsigned children     :    3
unsigned              :    4
} emp;
```

This defines a variable name **emp** with four bit fields. The range of values each field could have is follows:

Bit field	*Bit length*	*Range of value*
sex	1	0 or 1
age	7	0 or 127 ($2^7 - 1$)
m_status	1	0 or 1
children	3	0 to 7 (2^3–1)

Once bit fields are defined, they can be referenced just as any other structure-type data item would be referenced. The following assignment statements are valid.

```
emp.sex = 1;
emp.age = 50;
```

Remember, we cannot use **scanf** to read values into a bit field. We may have to read into a temporary variable and then assign its value to the bit field. For example:

```
scanf(%d %d", &AGE,&CHILDREN);
emp.age = AGE;
emp.children = CHILDREN;
```

One restriction in accessing bit fields is that a pointer cannot be used. However, they can be used in normal expressions like any other variable. For example:

```
sum = sum + emp.age;
if(emp.m_status). . . . .;
printf("%d\n", emp.age);
```

are valid statements.

It is possible to combine normal structure elements with bit field elements. For example:

```
struct personal
{
        char          name[20];     /* normal variable */
        struct addr   address;      /* structure variable */
        unsigned      sex : 1;
        unsigned      age : 7;
        . . . . .
        . . . . .
}
emp[100];
```

This declares **emp** as a 100 element array of type **struct personal.** This combines normal variable name and structure type variable **address** with bit fields.

Bit fields are packed into words as they appear in the definition. Consider the following definition.

```
struct pack
{
        unsigned  a:2;
        int count;
        unsigned  b : 3;
};
```

Here, the bit field **a** will be in one word, the variable **count** will be in the second word and the bit field **b** will be in the third word. The fields **a** and **b** would not get packed into the same word.

CASE STUDY

Book Shop Inventory

A book shop uses a personal computer to maintain the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher, stock position, etc. Whenever a customer wants a book, the shopkeeper inputs the title and author of the book and the system replies whether it is

in the list or not. If it is not, an appropriate message is displayed. If book is in the list, then the system displays the book details and asks for number of copies. If the requested copies are available, the total cost of the books is displayed; otherwise the message "Required copies not in stock" is displayed.

A program to accomplish this is shown in Fig. 12.8. The program uses a template to define the structure of the book. Note that the date of publication, a member of **record** structure, is also defined as a structure.

When the title and author of a book are specified, the program searches for the book in the list using the function

look_up(table, s1, s2, m)

The parameter **table** which receives the structure variable **book** is declared as type **struct record**. The parameters **s1** and **s2** receive the string values of **title** and **author** while **m** receives the total number of books in the list. Total number of books is given by the expression

sizeof(book)/sizeof(struct record)

The search ends when the book is found in the list and the function returns the serial number of the book. The function returns –1 when the book is not found. Remember that the serial number of the first book in the list is zero. The program terminates when we respond "NO" to the question

Do you want any other book?

Note that we use the function

get(string)

to get title, author, etc. from the terminal. This enables us to input strings with spaces such as "C Language". We cannot use **scanf** to read this string since it contains two words.

Since we are reading the quantity as a string using the **get(string)** function, we have to convert it to an integer before using it in any expressions. This is done using the **atoi()** function.

Program

```
/ ********************************************************************************** /
/*                              BOOK SHOP INVENTORY                                 */
/ ********************************************************************************** /

#include   <stdio.h>
#include   <string.h>

struct record
{
    chart   author[20];
    char    title[30];
    float   price;
    struct
    {
    char    month[10];
    int     year;
    }
    date;
    char    publisher[10];
    int     quantity;
};
```

```
main( )
{
    chart title[30], author[20];
    int     index, no_of_records, look_up( );
    char    response[10], quantity[10];

    static struct record book [ ] = {

    {“Ritche”,”C Language”,45.00,”May”,1977,”PHI”,10},
    {“Kochan”,“Programming in C”,75.50,“July”, 1983,“Hayden”,5},
    {“Balagurusamy”,“BASIC”,30.00,“January”,1984,“TMH”,0},
    {“Balagurusamy”,“COBOL”,60.00,“December”,1998,“Macmillan”,25}

    no_of_records = sizeof(book)/sizeof(struct record);
    do
    {
        printf(“Enter title and author name as per the list\n”);
        printf(“\nTitle;          ”);
        get(title);

        printf(“Author:          ”);
        get(author);
        index = look_up(book, title, author, no_of_records);
        if(index != –1)     /*   Book found */
        {
            printf(“\n%s %s %.2f %s %d %s\n\n”,
                book[index].author,
                book[index].title,
                book[index].price,
                book[index].date.month,
                book][index].date.year,
                book[index].publisher);

            printf(“Enter number of copies:”);
            get(quantity);
            if(atoi(quantity) <book[index].quantity)

                printf(“Cost of %d copies = %.2f\n”, atoi(quantity),
                    book[index].price * atoi(quantity));
            else
                printf(“\nRequired copies not in stock\n\n”);
        }
        else
            printf(“\nBook not in list\n\n”);
    printf(“\nDo you want any other book? (YES/NO);”);
    get(response);
}
    while(response[0] == ‘Y’ || response[0] = = ‘y’);
    printf(“\n\nThank you. Good bye!\n”);
}
get(string)
char    string[ ];
{
    char    c;
    int     i = 0;
    do
    {
```

```
        c = getchar( );
        string[i++] = c;
    }
    while(c != '\n');
    string[i – 1] = '\0';
}
int look_up (table, s1, s2, m)
struct record table[ ];
char s1[ ], s2[ ];
int     m;
{
    int     i;
    for(i = 0; i < m; i++)
        if(strcmp(s1, table[i].title) == 0 &&
        strcmp(s2, table[i].author) == 0)
        return(i);              /* book found              */
    return (– 1);               /* book not found          */
}
```

Output

```
Enter title and author name as per the list

Title:      BASIC
Author:     Balagurusamy

Balagurusamy BASIC 30.00 January 1984 TMH

Enter number of copies:5

Required copies not in stock

Do you want any other book? (YES/NO):y
Enter title and author name as per the list

Title:      COBOL
Author:     Balagurusamy

Balagurusamy COBOL 60.00 December 1988 Macmillan

Enter number of copies:7
Cost of 7 copies = 420.00

Do you want any other book? (YES/NO):y
Enter title and author name as per the list

Title:      C Programming
Author:     Ritche

Book not in list
Do you want any other book? (YES/NO):n

Thank you. Good bye!
```

Fig. 12.9 *Program of book shop inventory*

Review Questions and Exercises

12.1 How does a structure differ from an array?
12.2 Explain the meaning and purpose of the following:

(a) Template.
(b) Tag.
(c) sizeof.
(d) struct.

12.3 Describe what is wrong in the following structure declaration:

```
struct
{
    int     number;
    float   price;
}
main()
{
    .....
    .....
}
```

12.4 The following code appears at the beginning of a function. Identify errors, if any, in the code.

```
static int count[ ] = {
        10, 15, 20, 30;
    float value;
```

12.5 What is meant by the following terms?
(a) Nested structure.
(b) Array of structures.

12.6 Define a structure that can describe a hotel. It should have members that include the name, address, grade, average room charge, and number of rooms.
Write functions to perform the following operations:
(a) To print out hotels of a given grade in order of charges.
(b) To print out hotels with room charges less than a given value.

12.7 Define a structure called **cricket** that will describe the following information:

```
player name
team name
batting average
```

using **cricket**, declare an array **player** with 50 elements and write a program to read the information about all the 50 players and print a team wise list containing names of players with their batting average.

12.8 Rewrite the programs in Fig. 12.6 and Fig. 12.8 using ANSI function prototypes.

UNIT V

Chapter 13

File Management in C

13.1 INTRODUCTION

Until now, we have been using the functions such as **scanf** and **printf** to read and write data. These are console oriented I/O functions which always use the terminal (keyboard and screen) as the target place. This works fine as long as the data is small. However, many real-life problems involve large volumes of data and in such situations, the console oriented I/O operations pose two major problems.

1. It becomes cumbersome and time consuming to handle large volumes of data through terminals.
2. The entire data is lost when either the program is terminated or the computer is turned off.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of *files* to store data. A file is a place on the disk where a group of related data is stored. Like most other languages, C supports a number of functions that have the ability to perform basic file operations, which include:

- naming a file,
- opening a file,
- reading data from a file,
- writing data to a file, and
- closing a file.

There are two distinct ways to perform file operations in C. The first one is known as the *low-level I/O* and uses UNIX system calls. The second method is referred to as the *high-level I/O* operation and uses functions in C's standard I/O library. We shall discuss in this chapter, the important file handling functions that are available in the C library. They are listed in Table 13.1.

There are many other functions. Not all of them are supported by all compilers. You should check your C library before using a particular I/O function.

Table 13.1 High level I/O functions

Function name	Operation
fopen()	• Creates a new file for use. • Opens an existing file for use.
fclose()	• Closes a file which has been opened for use.
getc()	• Reads a character from a file.
putc()	• Writes a character to a file.
fprintf()	• Writes a set of data values to a file.
fscanf()	• Reads a set of data values from a file.
getw()	• Reads an integer from a file.
putw()	• Writes an integer to a file.
fseek()	• Sets the position to a desired point in the file.
ftell()	• Gives the current position in the file (in terms of bytes from the start).
rewind()	• Sets the position to the beginning of the file.

Text Files and Binary Files

The text files are those files that contain letters, digits and symbols. A text file mainly contains characters coded from the ASCII character set. On the other hand, a binary file is a file that uses all eight bits of a byte for storing information. The text file is mainly in a form, which can be read and understood by human beings. The binary file, on the other hand, is generally in a form, which can be interpreted and understood by a computer system.

In contrast to a binary file, the text file uses only seven bits allowing the 8^{th} bit to be 0. In a computer system, all executable or *.exe* files, Dynamic Link Libraries (DLL) and many database files are stored as binary files. One of the differences between text and binary file is that the data contained in the text file can be read by a word processor while the binary file data cannot be read by a word processor.
In C when you access a file for reading and writing then you can choose to open the file as a binary or text file. C programming language provides the **fopen()** function to open a file in a specific mode.

When you specify the mode with the **fopen()** function, you can choose to open the file in a read mode for reading, write mode for writing to a file, or append mode for appending data to the end of a file. It is also determined at the time of opening a file using **fopen()** function that whether the file should be opened as a binary or text file. The C programming language provides different read, write and append modes for reading and writing to text and binary files. If you use the following modes with **fopen()** function then the file is opened as a text file.

- **r:** This mode allows you to open a file as a text file for reading data from it.
- **w:** This mode allows you to open a file as a text file for writing data to it.
- **a:** This mode allows you to open a file as a text file for appending data at the end of the file.
- **r+:** This mode allows you to open a file as a text file for reading as well as writing data to a file.
- **w+:** This mode allows you to open a file as a text file for writing as well as reading data from a file.
- **a+:** This mode allows you to open a file as a text file for both reading and writing data to a file.

On the other hand, you need to add **b** for binary file with the **r, w** and **a** modes to open a file as a binary file. In other words to open a file as a binary file, you can use the following modes:

- **rb:** This mode allows you to open a file as a binary file for reading data from it.
- **wb:** This mode allows you to open a file as a binary file for writing data to it.
- **ab:** This mode allows you to open a file as a binary file for appending data at the end of the file.
- **rb+:** This mode allows you to open a file as a binary file for reading as well as writing data to a file.
- **wb+:** This mode allows you to open a file as a binary file for writing as well as reading data from a file.
- **ab+:** This mode allows you to open a file as a binary file for both reading and writing data to a file.

In C, the process for writing of text and binary files is also different. In case of a binary file, data is contained as lines of text. An end of line marker is automatically added at the end to each line of text in a text file when you indicate that the end of the line is reached. The **fwrite()** function of C allows you to write an end of line marker at the end of text in a text file. However in case of a binary file, the data is not separated and hence no end of line marker is written at the end of the text when writing data to a binary file using **fwrite()** function. The reading process for text and binary files is also different. In case of a text file, the data is separated into lines of text as it is read according to the position of the end of the line marker. On the other hand, the data in a binary file is not separated into lines of text.

Streams

Stream is a sequence of data bytes, which is used to read and write data to a file. The streams that represent the input data of a program are known as input streams, whereas, the streams that represent the output data of program are known as output streams. Input streams interpret data from different devices such as keyboard and mouse, and provide input data to the program. However, output streams obtain data from the program and write that data on different devices such as memory or print them on the screen. Therefore, a stream only acts as an interface between a program and an input/output device. Programs use streams to read and write data independent of their source devices. Different stream classes are declared for different devices. The following figure shows the relationship between streams and input/output devices.

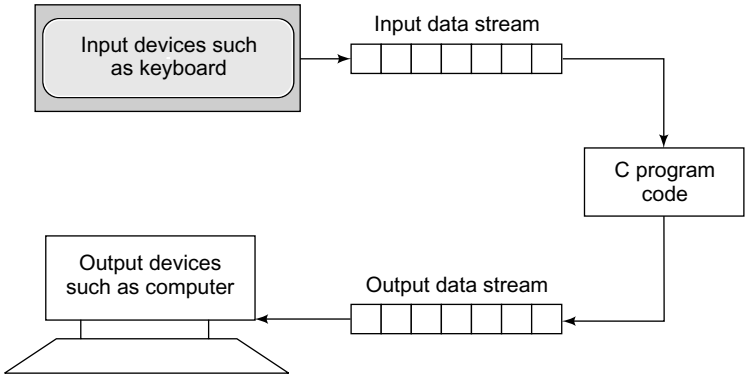


Fig. 13.1 *Relationship between streams and I/O devices*

You can read and write streams to a file using the **fread()** and **fwrite()** functions provided by C. As an example, consider the following code.

```
# include<stdio.h>
# include<conio.h>
void main()
{
struct employee
{
char name[25];
int phone;
} employee[3];
FILE *fptr;
int num,i;
fptr=fopen("hello.txt","ab+");//opening file in binary mode
clrscr();                    //to write records
if (fptr==NULL)
{
printf("The file does not exist");
exit(1);
}
else
{
printf("How many records do you want to enter\n");
scanf("%d", &num);
printf("Enter the names and phone numbers\n");
for(i=0;i<num;i++)
{
scanf("%s %d", &employee[i].name, &employee[i].phone);
}
printf("Adding records to a file\n");
i=0;
do
{
fwrite(&employee, sizeof(employee),1,fptr);
i++;
}
while(i<num);
fclose(fptr);
i=0;
fptr=fopen("hello.txt" , "rb"); //opening file in binary
                                // mode to read records
printf("records entered are\n");
do
{
fread(&employee, sizeof(employee), 1, fptr);
printf("%s, %d", employee[i].name, employee[i].phone);
i++;
}while(i<num);
fclose(fptr);
getch();
}}
```

Fig. 13.2 *C program using streams*

The above code shows how to open a file in the binary mode. It also shows how to read and write data to a binary file. The output of the above code is:

```
How many records do you want to enter
2
Enter the names and phone numbers
Jyotsna 2345
Richa 3457
Adding records to a file
records entered are
Jyotsna, 2345Richa, 3457
```

13.2 DEFINING AND OPENING A FILE

If we want to store data in a file in the secondary memory, we must specify certain things about the file, to the operating system. They include:

1. Filename.
2. Data structure.
3. Purpose.

Filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with the extension. Examples:

```
Input.data
store
PROG.C
Student c
Text.out
```

Data structure of a file is defined as **FILE** in the library of standard I/O function definitions. Therefore, all files should be declared as type **FILE** before they are used. **FILE** is a defined data type.

When we open a file, we must specify what we want to do with the file. For example, we may write data to the file or read the already existing data.

Following is the general format for declaring and opening a file:

```
FILE    *fp;
fp = fopen("filename", "mode");
```

The first statement declares the variable **fp** as a "pointer to the data type **FILE**". As stated earlier. **FILE** is a structure that is defined in the I/O library. The second statement opens the file named *filename* and assigns an identifier to the **FILE** type pointer **fp**. This pointer which contains all the information about the file is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The *mode* does this job. *Mode* can be one of the following:

r open the file for reading only.

w open the file for writing only.

a open the file for appending (or adding) data to it.

Note that both the filename and mode are specified as strings. They should be enclosed in double quotation marks.

When trying to open a file, one of the following things may happen:

1. When the mode is 'writing' a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2. When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the purpose is 'reading', and if it exists, then the file is opened with the current contents safe; otherwise an error occurs.

Consider the following statements:

```
FILE *p1, *p2;
p1   = fopen("data", "r");
p2   = fopen("results", "w");
```

The file **data** is opened for reading and **results** is opened for writing. In case, the **results** file already exists, its contents are deleted and the file is opened as a new file. If **data** file does not exist, an error will occur.

Many recent compilers include additional modes of operation. They include:

r+ The existing file is opened to the beginning for both reading and writing.

w+ Same as **w** except both for reading and writing.

a+ Same as **a** except both for reading and writing.

We can open and use a number of files at a time. This number however depends on the system we use.

13.3 CLOSING A FILE

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. It also prevents any accidental misuse of the file. In case, there is a limit to the number of files that can be kept open simultaneously, closing of unwanted files might help open the required files. Another instance where we have to close a file is when we want to reopen the same file in a different mode. The I/O library supports a function to do this for us. It takes the following form:

```
fclose(file_pointer);
```

This would close the file associated with the **FILE** pointer **file_pointer**. Look at the following segment of a program.

```
.....
.....
FILE   *p1, *p2;
p1   =  fopen("INPUT", "w");
p2   =  fopen("OUTPUT", "r");
.....
.....
fclose(p1);
fclose(p2);
.....
```

This program opens two files and closes them after all operations on them are completed. Once a file is closed, its file pointer can be reused for another file.

As a matter of fact all files are closed automatically whenever a program terminates. However, closing a file as soon as you are done with it, is a good programming habit.

13.4 INPUT/OUTPUT OPERATIONS ON FILES

Once a file is opened, reading out of or writing to it is accomplished using the standard I/O routines that are listed in Table 13.1.

The getc and putc Functions

The simplest file I/O functions are **getc** and **putc**. These are analogous to **getchar** and **putchar** functions and handle one character at a time. Assume that a file is opened with mode **w** and file pointer **fp1**. Then, the statement

```
putc(c, fp1);
```

writes the character contained in the character variable **c** to the file associated with **FILE** pointer **fp1**. Similarly, **getc** is used to read a character from a file that has been opened in read mode. For example, the statement

```
c = getc(fp2);
```

would read a character from the file whose file pointer is **fp2**.

The file pointer moves by one character position for every operation of **getc** or **putc**. The **getc** will return an end-of-file marker **EOF**, when end of the file has been reached. Therefore, the reading should be terminated when **EOF** is encountered.

Example 13.1

Write a program to read data from the keyboard, write it to a file called **INPUT**, again read the same data from the **INPUT** file, and display it on the screen.

A program and the related input and output data are shown in Fig. 13.3. We enter the input data via the keyboard and the program writes it, character by character, to the file **INPUT**. The end of the data is indicated by entering an **EOF** character, which is *control-Z* in the reference system. (This may be *control-D* in other systems). The file INPUT is closed at this signal.

Program

```
/*************************************************************************/
/*                    WRITING TO AND READING FROM A FILE                 */
/*************************************************************************/

#include <stdio.h>

main( )
{
    FILE *f1;
    char c;

    print("Data Input\n\n");
```

```
    f1 = fopen("INPUT", "w");       /*  Open the file INPUT                */
    while((c=getchar()) !=EOF)      /*  Get a character from keyboard      */
        putc(c,f1);                 /*  Write a character to INPUT         */
    fclose(f1);                     /*  Close the file INPUT               */
    printf("\nData Output\n\n");
    f1 = fopen("INPUT", "r");       /*  Reopen the file INPUT              */
    while((c=getc(f1)) != EOF)      /*  Read a character from INPUT        */
        printf("%c",c);             /*  Display a character on screen      */
    fclose(f1);                     /*  Close the file INPUT               */
}
```

Output

```
Data Input

This is a program to test the file handling
features on this system Z

Data Output

This is a program to test the file handling
features on this system
```

Fig. 13.3 *Character oriented read/write operations on a file*

The file INPUT is reopened for reading. The program then reads its content character by character, and displays it on the screen. Reading is terminated when **getc** encounters the end-of-file mark **EOF**.

Testing for the end-of-file condition is important. Any attempt to read past the end of file might either cause the program to terminate with an error or result in an infinite loop situation.

The fprintf and fscanf Functions

So far, we have seen functions which can handle only one character or integer at a time. Most compilers support two other functions, namely **fprintf** and **scanf**, that can handle a group of mixed data simultaneously.

The functions **fprintf** and **fscanf** perform I/O operations that are identical to the familar **printf** and **scanf** functions, except of course that they work on files. The first argument of these functions is a file pointer which specifies the file to be used. The general form of **fprintf** is

fprintf(*fp, "control string", list*);

where *fp* is a file pointer associated with a file that has been opened for writing. The *control string* contains output specifications for the items in the *list*. The *list* may include variables, constants and strings. Example:

```
fprintf(f1, "%s %d %f", name, age, 7.5);
```

Here, **name** is an array variable of type **char** and **age** is an **int** variable.

The general format of **fscanf** is

fscanf(*fp, "control string", list*);

This statement would cause the reading of the items in the *list* from the file specified by *fp*, according to the specifications contained in the *control string*. Example:

```
fscanf(f2, "%s %d", item, &quantity);
```

Like **scanf**, **fscanf** also returns the number of items that are successfully read. When the end of file is reached, it returns the value **EOF**.

Example 13.2

Write a program to open a file named INVENTORY and store in it the following data:

Item name	*Number*	*Price*	*Quantity*
AAA-1	111	17.50	115
BBB-2	125	36.00	75
C-3	247	31.75	104

Extend the program to read this data from the file INVENTORY and display the inventory table with the value of each item.

The program is given in Fig. 13.4. The filename INVENTORY is supplied through the keyboard. Data is read using the function **fscanf** from the file **stdin**, which refers to the terminal and it is then written to the file that is being pointed to by the file pointer **fp**. Remember that the file pointer **fp** points to the file INVENTORY.

After closing the file INVENTORY, it is again reopened for reading. The data from the file, along with the item values are written to the file **stdout**, which refers to the screen. While reading from a file, care should be taken to use the same format specifications with which the contents have been written to the file.

Program

```
/*********************************************************************************** /
/*       HANDLING OF FILES WITH MIXED DATA TYPES          */
/*                          (scanf and fprintf)                                    */
/*********************************************************************************** /

#include <stdio.h>

main( )
{
    FILE   *fp;
    int    number, quantity, i;
    float  price, value;
    char   item[10], filename[10];

    printf("Input file name\n");
    scanf("%s", filename);
    fp = fopen(filename, "w");

    printf("input inventory data\n\n");
    printf("Item nameNumber        Price        Quantity\n");
    for(i = 1; i <=3; i++)
    {
        fscanf(stdin,  "%s %d %f %d",
                        item, &number, &price, &quantity);
        fprintf(fp,  "%s %d %.2f %d",
                        item, number, price, quantity);
    }
    fclose(fp);
```

```
    fprintf(stdout, "\n\n");
    fp = fopen(filename, "r");
    printf("Item name Number   Price       Quantity       Value\n");
    for(i = 1; i<=3; i++)
    {
        fscanf(fp, "%s %d %f %d",item,&number,&price,&quantity);
        value = price * quantity;
        fprintf(stdout, "%–8s %7d %8.2f %8d %11.2f\n",
                        item, number, price, quantity, value);
    }
    fclose(fp);
}
```

Output

```
Input file name
INVENTORY
Input inventory data
Item name       Number      Price      Quantity
AAA-1           111         17.50      115
BBB-2           125         36.00      75
C-3             247         31.75      104
Item name       Number      Price      Quantity      Value
    AAA-1       111         17.50      115           2012.50
    BBB-2       125         36.00      75            2700.00
    C-3         247         31.75      104           3302.00
```

Fig. 13.4 *Operations on mixed data types*

13.5 ERROR HANDLING DURING I/O OPERATIONS

It is possible that an error may occur during I/O operations on a file. Typical error situations include:

1. Trying to read beyond the end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid filename.
6. Attempting to write to a write-protected file.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of the program or incorrect output. Fortunately, we have two status-inquiry library functions, **feof** and **ferror** that can help us detect I/O errors in the files.

The **feof** function can be used to test for an end of file condition. It takes a **FILE** pointer as its only argument and returns a nonzero integer value if all of the data from the specified file has been read, and returns zero otherwise. If **fp** is a pointer to file that has just been opened for reading, then the statement

```
if(feof(fp))
printf("End of data.\n");
```

would display the message "End of data." on reaching the end of file condition.

The **ferror** function reports the status of the file indicated. It also takes a **FILE** pointer as its argument and returns a nonzero integer if an error has been detected upto that point, during processing. It returns zero otherwise. The statement

```
if(ferror(fp) != 0)
printf("An error has occurred.\n");
```

would print the error message, if the reading is not successful.

We know that whenever a file is opened using **fopen** function, a file pointer is returned. If the file cannot be opened for some reason, then the function returns a **null** pointer. This facility can be used to test whether a file has been opened or not. Example:

```
if(fp == NULL)
printf("File could not be opened.\n");
```

Example 13.3

Write a program to illustrate error handling in file operations.

The program shown in Fig. 13.5 illustrates the use of the **null** pointer test and **feof** function. When we input filename as TETS, the function call

```
fopen("TETS", "r");
```

returns a **null** pointer because the file **TETS** does not exist and therefore the message "Cannot open the file" is printed out.

Similarly, the call **feof(fp2)** returns a nonzero integer when the entire data has been read, and hence the program prints the message "Ran out of data" and terminates further reading.

```
/*************************************************************************************/
/*                    ERROR HANDLING IN FILE OPERATIONS                              */
/*************************************************************************************/

#include <stdio.h>

main( )
{
    char    *filename;
    FILE    *fp1, *fp2;
    int     i, number;
    fp1 = fopen("TEST", "w");
    for(i = 10; i <=100; i + = 10)
        putw(i, fp1);

    fclose(fp1);

    printf("\nInput filename\n");

open_file:
    scanf("%s", filename);

    if((fp2 = fopen(filename, "r")) == NULL)
    {
     printf("Cannot open the file.\n");
```

```
        printf("Type filename again,\n\n");
        goto open_file;
      }
      else
      for(i = 1; i <=20; i++)
      { number = getw(fp2);
        if(feof(fp2))
          {
          printf("\nRan out of data.\n");
          break;
      }
          else
              printf("%d\n", number);
      }
          fclose(fp2);
}
```

Output

```
Input filename
TETS
Cannot open the file.
Type filename again.

TEST
10
20
30
40
50
60
70
80
90
100

Ran out of data.
```

Fig. 13.5 *Illustration of error handling*

13.6 RANDOM ACCESS TO FILES

So far we have discussed file functions that are useful for reading and writing data sequentially. There are occasions, however, when we are interested in accessing only a particular part of a file and not in reading the other parts. This can be achieved with the help of the functions **fseek, ftell**, and **rewind** available in the I/O library.

ftell takes a file pointer and returns a number of type **long**, that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following form:

```
n = ftell(fp);
```

n would give the relative offset (in bytes) of the current position. This means that n bytes have already been read (or written).

rewind takes a file pointer and resets the position to the start of the file. For example, the statement

```
rewind(fp);
n = ftell(fp);
```

would assign 0 to **n** because the file position has been set to the start of the file by **rewind**. Remember, the first byte in the file is numbered as 0, second as 1, and so on. This function helps us in reading a file more than once, without having to close and open the file. Remember that whenever a file is opened for reading or writing, a **rewind** is done implicitly.

fseek function is used to move the file position to a desired location within the file. It takes the following form:

fseek(*file ptr, offset, position*);

file ptr is a pointer to the file concerned, *offset* is a number or variable of type **long**, and *position* is an integer number. The *offset* specifies the number of positions (bytes) to be moved from the location specified by *position*. The *position* can take one of the following three values:

Value	*Meaning*
0	Beginning of file
1	Current position
2	End of file

The *offset* may be positive, meaning move forwards, or negative, meaning move backwards. The following examples illustrate the operation of the **fseek** function:

Statement	*Meaning*
fseek(fp,0L,0);	Go to the beginning. (Similar to rewind)
fseek(fp,0L,1);	Stay at the current position. (Rarely used)
fseek(fp,0L,2);	Go to the end of the file, past the last character of the file.
fseek(fp,m,0);	Move to (m+1)th byte in the file.
fseek(fp,m,1);	Go forward by m bytes.
fseek(fp,–m,1);	Go backward by m bytes from the current position.
fseek(fp,–m,2);	Go backward by m bytes from the end. (Positions the file to the mth character from the end.)

When the operation is successful, **fseek** returns a zero. If we attempt to move the file pointer beyond the file boundaries, an error occurs and **fseek** returns –1 (minus one). It is good practice to check whether an error has occurred or not, before proceeding further.

Example 13.4

Write a program that uses the functions **ftell** and **fseek**.

A program employing **ftell** and **fseek** functions is shown in Fig. 13.6. We have created a file **RANDOM** with the following contents:

```
Position    ————————→ 0  1  2 . . . . . . . . . . 25
Character
stored      ————————→ A B C . . . . . . . . . .  Z
```

We read the file twice. First, we read the contents of every fifth position and print its value along with its position on the screen. The second time, we read the contents of the file from the end and print the same on the screen.

During the first reading, the file pointer crosses the end-of-file mark when the parameter **n** of **fseek (fp,n,0)** becomes 30. Therefore, after printing the content of position 30, the loop is terminated. (There is nothing in the position 30.)

For reading the file from the end, we use the statement

fseek(fp, – 1L,2);

to position the file pointer to the last character. Since every read causes the position to move forward by one position, we have to move it back by two positions to read the next character. This is achieved by the function

fseek(fp, – 2L, 1);

in the **while** statement. This statement also tests whether the file pointer has crossed the file boundary or not. The loop is terminated as soon as it crosses it.

Program

```
/*********************************************************************************** /
/*              ILLUSTRATION OF fseek & ftell FUNCTIONS                            */
/*********************************************************************************** /

#include      <stdio.h>
main( )
{
    FILE   *fp;
    long    n;
    char    c;

    fp = fopen("RANDOM", "w");

    while((c = getchar( )) !=EOF)
        putc(c,fp);

    printf("No. of characters.entered = %ld\n", ftell(fp));

    fclose(fp);

    fp = fopen("RANDOM", "r");
    n = 0L;

    while(feof(fp) == 0)
    {
        fseek(fp, n, 0);        /* Position to (n+1)th character                  */
        printf("Position of %c is %ld\n", getc(fp), ftell(fp));
        n = n+5L;
    }
    putchar('\n');

    fseek(fp,–1L,2);            /* Position to the last character                 */
    do
        {
         putchar(getc(fp));
```

```
        }
    while(!fseek(fp,-2L,1));
    fclose(fp);
}
```

Output

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ^Z
No. of characters entered = 26
Position  of  A  is  0
Position  of  F  is  5
Position  of  K  is  10
Position  of  P  is  15
Position  of  U  is  20
Position  of  Z  is  25
Position  of     is  30

ZYXWVUTSRQPONMLKJIHGFEDCBA
```

Fig. 13.6 *Illustration of* ***fseek*** *and* ***ftell*** *functions*

13.7 COMMAND LINE ARGUMENTS

What is a command line argument? It is a parameter supplied to a program when the program is invoked. This parameter may represent a filename the program should process. For example, if we want to execute a program to copy the contents of a file named **X_FILE** to another one named **Y_FILE**, then we may use a command line like

```
C>PROGRAM  X_FILE  Y_FILE
```

PROGRAM is the filename where the executable code of the program is stored. This eliminates the need for the program to request the user to enter the filenames during execution. How do these parameters get into the program?

We know that every C program should have one **main** function and that it marks the beginning of the program. But what we have not mentioned so far is that it can also take arguments like other functions. In fact **main** can take two arguments called **argc** and **argv** and the information contained in the command line is passed on to the program through these arguments, when **main** is called up by the system.

The variable **argc** is an *argument counter* that counts the number of arguments on the command line. The **argv** is an argument vector and represents an array of character pointers that point to the command line arguments. The size of this array will be equal to the value of **argc**. For instance, for the command line given above, **argc** is three and **argv** is an array of three pointers to strings as shown below:

```
argv[0] ———→ PROGRAM
argv[1] ———→ X_FILE
argv[2] ———→ Y_FILE
```

In order to access the command line arguments, we must declare the **main** function and its parameters as follows:

```
main(argc, argv)
int    argc;
```

```
char *argv[ ];
{
    .....
    .....
}
```

The first parameter in the command line is always the program name and therefore **argv[0]** always represents the program name.

Example 13.5

Write a program that will receive a filename and a line of text as command line arguments and write the text to the file.

Figure 13.7 shows the use of command line arguments. The command line is

F12_6 TEXT AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG.

Each word in the command line is an argument to the **main** and therefore the total number of arguments is 9.

The argument vector **argv[1]** points to the string TEXT and therefore the statement

```
fp = fopen(argv[1], "w");
```

opens a file with the name TEXT. The **for** loop that follows immediately writes the remaining 7 arguments to the file TEXT.

Program also prints two outputs, one from the file TEXT and the other from the system memory. The *argument vector* **argv** contains the entire command line in the memory and, therefore, the statement:

```
printf("%s\n", i*5, argv[i]);
```

prints the arguments from the memory.

Program

```
/*****************************************************************************/
/*                    COMMAND LINE ARGUMENTS                                 */
/*****************************************************************************/

#include <stdio.h>

main(argc, argv)                    /*    main with arguments              */
int argc;                           /*    argument count                   */
char *argv[];                       /*    list of arguments                */
{
    FILE      *fp;
    int       i;
    char      word[15];

    fp = fopen(argv[1], "w");       /*  open file with name argv[1]        */
    printf("\nNo. of arguments in Command line = %d\n\n", argc);
    for(i = 2; i < argc; i++)
        fprintf(fp, "%s", argv[i]);  /*  write to file argv[1]             */
    fclose(fp);

/*  Writing content of the file to screen                                  */

    printf("Contents of %s file\n\n", argv[1]);
    fp = fopen(argv[1], "r");
    for(i = 2; i < argc; i++)
```

```
    {
        fscanf(fp, "%s", word);
        printf("%s", word);
    }
    fclose(fp);
    printf("\n\n");
/*  Writing the arguments from memory                                    */
    for(i = 0; i < argc; i++)
        printf("%*s\n", i*5,argv[i]);
}
```

Output

```
C>F12_6 TEXT AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG
No. of arguments in Command line = 9
Contents of TEXT file
AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG
C:\C\F12_6.EXE
TEXT
    AAAAAA
          BBBBBB
                CCCCCC
                      DDDDDD
                            EEEEEE
                                  FFFFFF
                                        GGGGGG
```

Fig. 13.7 *Use of command line arguments*

CASE STUDY

Appending Items to a File

A program to append additional items to the file INVENTORY and print the total contents of the file has been developed here. The program is shown in Fig. 13.8. It uses a structure definition to describe each item and a function **append** () to add an item to the file.

On execution, the program requests for the filename to which data is to be appended. After appending the items, the position of the last character in the file is assigned to **n** and then the file is closed.

The file is reopened for reading and its contents are displayed. Note that reading and displaying are done under the control of a **while** loop. The loop tests the current file position against **n** and is terminated when they become equal.

Program

```
/*****************************************************************************/
/*              APPENDING ITEMS TO AN EXISTING FILE                          */
/*****************************************************************************/

#include <stdio.h>

struct invent_record
```

```
{
    char    name[10];
    int     number;
    float   price;
    int     quantity;
}
main()
{
    struct  invent_record item;
    char    filename[10];
    int     response;
    FILE    *fp;
    long    n;

    printf("Type filename:");
    scanf("%s", filename);

    fp = fopen(filename, "a+");
    do
    {
        append(&item, fp);
        printf("\nitem %s appended \n",item.name);
        printf("\nDo you want to add another item \
(1 for YES /0 for NO)?");
        scanf("%d", &response);
    }   while (response == 1);
    n = ftell(fp);                          /*  Position of last character      */
    fclose(fp);

    fp = fopen(filename, "r");
    while(ftell(fp) < n)
    {
        fscanf(fp, "%s %d %f %d",
            item.name, &item.number, &item.price, &item.quantity);
        fprintf(stdout,"%–8s %7d %7.2f %8d\n",
            item.name, item.number, item.price item.quantity);
    }
    fclose(fp);
}
append(product, ptr)
struct invent_record *product;
FILE        *ptr;
{
    printf("Item name:");
    scanf("%s", product – > name);

    printf("Item number:");
    scanf("%d", &product->number);

    printf("Item price:")
    scanf("%f", &product –>price);

    printf("Quantity:");
    scanf("%d", &product->quantity);

    fprintf(ptr, "%s %d %.2f %d",
                product–>name,
                product–>number,
                product–>price,
```

```
                product–>quantity);
}
```

Output

```
Type filename:INVENTORY
Item name:XXX
Item number:444
Item price:40.50
Quantity:34

Item XXX appended.

Do you want to add another item(1 for YES/0 for NO)?1

Item name:YYY
Item number:555
Item price:50.50
Quantity:45

Item YYY appended.

Do you want to add another item(1 for YES/0 for NO)?0
AAA-1      111    17.50     115
BBB-2      125    36.00      75
C-3        247    31.75     104
XXX        444    40.50      34
YYY        555    50.50      45
```

Fig. 13.7 *Adding items to an existing file*

Review Questions and Exercises

13.1 Describe the use and limitations of the functions **getc** and **putc**.

13.2 What is the significance of EOF?

13.3 When a program is terminated, all the files used by it are automatically closed. Why is it then necessary to close a file during execution of the program?

13.4 Distinguish between the following functions:
- (a) getc and getchar
- (b) printf and fprintf.
- (c) feof and ferror.

13.5 How does an append mode differ from a write mode?

13.6 What are the common uses of **rewind** and **ftell** functions?

13.7 Explain the general format of **fseek** function?

13.8 What is the difference between the statements **rewind(fp)**; and **fseek(fp, 0L,0);**?

13.9 Write a program to copy the contents of one file into another.

13.10 Two files DATA1 and DATA2 contain sorted lists of integers. Write a program to produce a third file DATA which holds a single sorted, merged list of these two lists. Use command line arguments to specify the file names.

13.11 Write a program that will generate a data file containing the list of customers and their corresponding telephone numbers. Use a structure variable to store the name and telephone of each customer. Create a data file using a sample list.

13.12 Write an interactive, menu-driven program that will access the data file created in the above problem and do one of the following tasks:
(a) Determine the telephone number of a specified customer.
(b) Determine the customer whose telephone number is specified.

UNIT VI

Chapter 14

Data Structures

14.1 INTRODUCTION TO DATA STRUCTURES

In the field of computing, we have formulated many ways to handle data efficiently. So far, we have seen how to use variables to store data. However, variables are not feasible when handling a huge amount of data.

Consider a situation when we want to access information, such as student name, roll no, section name, sex, address, subjects, and marks obtained in each subject for students in a specific state of the country. With a single variable, it is an unfeasible task to store such a large amount of related data. Also, we need an organized medium of storage to handle any correlated information. Thus, the concept of data structure is introduced.

Data structure is a collection of organized data that are related to each other. Data structures can be classified into two types:

- Linear data structure
- Non-linear data structure

The linear data structures, such as array and list, and non-linear data structures, such as trees and graphs, are most popularly known data structures.

As we have seen earlier, an array is used to store data in consecutive memory locations. Arrays are considered as a good example for implementing simple data structures. Arrays can be effectively used for random access of fixed amount of data. We can also use arrays to create data structures, such as stack and queue. Stacks and queues are usually known as linear data structures, because data items in stacks and queues are arranged in a linear sequence.

14.2 STACKS

A stack is a linear data structure in which a data item is inserted and deleted at one end. A stack is called a Last In First Out (LIFO) structure because the data item that is inserted last into the stack is the first data item to be deleted from the stack. To understand this, imagine a pile of books. Usually, we pick up the topmost book from the pile. Similarly, the *topmost* item (i.e. the item which made the entry last) is the first one to be picked up/removed from the stack.

Stacks are extensively used in computer applications. Their most notable use is in system software (such as compilers, operating systems, etc.). For example, when one C function calls another function, and passes some parameters, these parameters are passed using the stack. In fact, even many compilers store the local variables inside a function on the stack.

In general, writing a value to the stack is called as a *push* operation, whereas reading a value from it is called as a *pop* operation. Like in the case of queues, a pop (i.e. read) operation in the case of a stack is also destructive. That is, once an item is popped from the stack, it is no longer available.

Figure 14.1 specifies what happens when the push and pop operations get executed. Note how the new items get added to the end of the stack, and also how elements get removed from there. We assume that initially the stack is empty.

Operation	Contents of the stack after the operation
Push (A)	A
Push (B)	A B
Pop	A
Push (C)	A C
Push (D)	A C D
Push (E)	A C D E
Pop	A C D
Pop	A C
Pop	A
Pop	Empty

Fig. 14.1 *How a stack works*

A typical implementation of the push and pop operations of a stack is shown in Fig. 14.2. As shown, the push operation checks it there is still some room left in the stack, and if there is any, it adds the received item to the top of the stack, and increments the stack counter by one. Similarly, the pop operation decrements the stack counter by one to see if stack is empty. If not, it returns the top of the stack.

Also note that we have implemented the stack as an array. Instead, it can be implemented using the dynamic memory management techniques. The variable *top* indicates the current top of the stack, and it moves up and down dynamically, depending on whether new items are added to the stack, or existing items are removed.

```
#define    MAX 100
int stack[MAX};

int top = 0;    /* indicates the top of the stack */

/* write an item to the stack
void push (int t)
{
    if (top >= MAX)
    {
        printf ("Stack is already full. Cannot add more items.\n");
        return;
    }

    stack[top] = t;
    top++;
}
```

```
/* read the tompost item from the stack */
int pop (void)
{
    top– –;
    if (top < 0)
    {
        printf (“Stack is already empty. No items to read.\n”);
        return;
    }

    return stack[top];
}
```

Fig 14.2 *Push and pop operations of a stack*

For a real-life use of stacks, imagine a magazines library. New magazines keep arriving in the library, and keep getting registered. Whenever a member walks in, she asks for the *latest* magazine that is available. The librarian takes a look at the topmost magazine, and offers it to the member. As and when any magazine is issued to a member, it is removed from the list of the available magazines. Of course, to keep things simple, we shall ignore the cases where the members do not want the latest magazine, but want something else. A stack best implements this arrangement. The program for this is listed in Fig. 14.3.

```
#include <stdio.h>
#include <string.h>

/* Global variables */
char magazines[1000][1000];   /* library can have 1000 magazines at the most at a time */
int top = 0;                  /* indicates the top of the stack */

/* function declarations */
void push (void);
void pop (void);


void main (void)
{
    int ch = 0;

    while (1)
    {
        printf (“** Main Menu **\n”);
        printf (“1. Add new magazine\n”);
        printf (“2. Issue the latest magazine\n”);
        printf (“3. Quit\n”);
        printf (“Your choice: ”);
        scanf(“%d”, &ch);

        switch (ch)
        {
            case 1: push (); break;
            case 2: pop (); break;
            case 3: return;
        }
    }
}
```

Fig. 14.3 *Magazines library implemented as a stack–Part I*

```
/* add a new magazine to the top of the stack */
void push (void)
{
    if (top >= 1000)
    {
        printf ("Stack is already full. Cannot add more magazines.\n");
        return;
    }

    printf("Magazine details: \n");
    gets (magazines[top]);

    if (!magazines[top])
        return;
    top++;
}

/* issue the tompost magazine from the stack */
void pop (void)
{
    top--;
    if (top < 0)
    {
        printf ("Stack is already empty. No magazines to issue.\n");
        return;
    }

    printf ("Issuing magazine: %s", magazines[top]);
}
```

Fig. 14.3 *Magazines library implemented as a stack—Part II*

14.3 QUEUES

A **queue** is very similar to the way we (are supposed to) queue up at train reservation counters or at bank cashiers' desks. A queue is a linear, sequential list of items that are accessed in the order *First In First Out (FIFO)*. That is, the first item inserted in a queue is also the first one to be accessed, the second item inserted in a queue is also the second one to be accessed, and the last one to be inserted is also the last one to be accessed. We cannot store/access the items in a queue arbitrarily or in any random fashion.

Suppose that we are creating a queue of items. For this purpose, let us assume two very simple functions, *write* and *read*. The *write* function adds a new item to the queue, whereas the *read* item reads one item from the queue. If we go on operating with these two functions with a few insertions and a few retrievals, the resulting values of the queue would look as shown in Fig. 14.4.

Operation	*Contents of the queue*
write (a)	a
write (b)	a b
write (c)	a b c
read (*returns a*)	b c
write (d)	b c d
read (*returns b*)	c d
read (*returns c*)	d
write (e)	d e

Fig. 14.4 *Queue operations*

As the figure shows, the operations on a queue are FIFO. Also, note that when one item is read from the queue, it is destroyed. Therefore, a *read* operation on a queue is destructive, unlike what happens with other data structures, such as linked lists. If the programmer needs to implement a non-destructive reading of a queue, the values, as they are read (and therefore, automatically removed) from the queue, must be stored somewhere else for later retrieval.

In order to implement the *write* and *read* operations of a queue, two pointers, *start* and *end* are required. One pointer (*start*) points at the current start of the queue (i.e. at the item that is the first in the queue at any point in time). The other pointer (*end*) points at the current end of the queue (i.e. at the next storage location available in the queue for a new item). Thus, insertions and retrievals from a queue, as discussed earlier, would have the effects on the pointers as shown in Fig. 14.5.

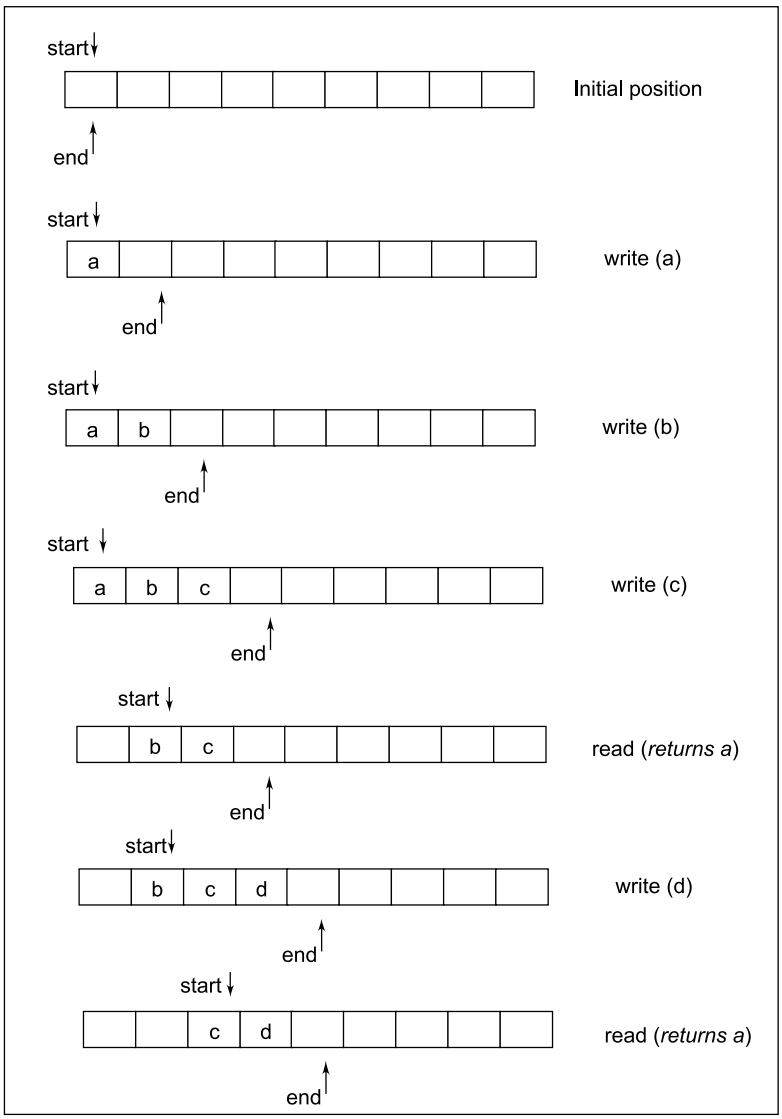


Fig. 14.5 *Pointer movements because of queue operations—Part I*

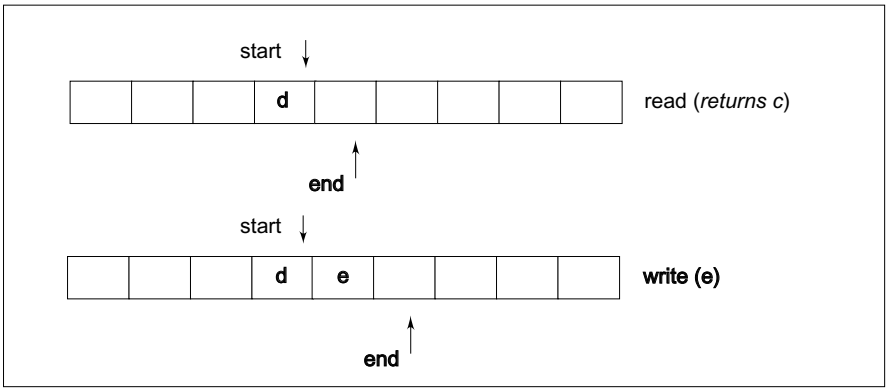


Fig 14.5 *Pointer movements because of queue operations—Part II*

Following program illustrates the functioning of a queue.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define    MAX   1000

/*Global variables */
char *arr[MAX];
int start, end;

/* Function declarations */
char *queue_retrieve (void);
void enter (void);
void queue_store (char *item);
void queue_review (void);
void item_delete (void);
```

Fig. 14.6 *Queue program—Part I*

```
/* Make an entry into the queue */
void enter (void)
{
    char str[400], *p;

    do
    {
        printf ("Please enter item number : %d", end+1);
        gets (str);

        if(!str)
            return;
        else
            p = (char *) malloc (strlen (str));
```

```
            if (!p)
            {
                printf ("Insufficient memory\n");
                return;
            }

            strcpy (p, str);

            if (*str)
                queue_store (p);
        }
        while (*str);
}


/* List queue items */
void queue_review (void)
{
    register int i;

    for (i=start; i<end; i++)
        printf ("%d  %s", i+1, arr[i]);
}

/* Delete an item from the queue */
void delete_item (void)
{
    char *p;

    p = queue_retrieve ();

    if (p == NULL)
        return;

    printf ("%s\n", p);
}
```

Fig. 14.6 *Queue program—Part II*

```
/* store an item */
void queue_store (char *p)
{
    if (end == MAX)
    {
        printf ("Queue is full. Cannot add new items.\n");
        return;
    }

    arr[end] = p;
    end++;
}

/* Retrieve an item from the queue */
char* queue_retrieve (void)
{
```

```
    if (start == end)
    {
        printf ("Queue is empty.\n");
        return NULL;
    }

    start++;

    return arr[start-1];
}
```

Fig. 14.6 *Queue program—Part III*

14.4 CIRCULAR QUEUES

A queue can also be circular in which case, it is called as a circular queue. When we discussed queues, you might have thought that, as the items from a queue get deleted, the space for that item is reclaimed. However, it is not so. Those queue positions continue to be empty. This problem is solved by circular queues.

Instead of using a linear approach, a circular queue takes a circular approach. That is why a circular does not have a beginning or end. The logical of a circular queue is shown in Fig. 14.7.

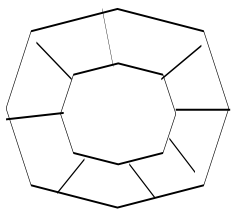


Fig. 14.7 *Circular queue*

The functions queue_store and queue_retrieve would change in the case of a circular queue, as follows:

```
void queue_store (char *p)
{
   /* The queue is full if (a) end is one less than start, or (b) end is at the end position of the queue,
            and start is at the beginning of the queue */

   if ((end+1 == start) || (end+1 == MAX && start == 0))
   {
printf ("Queue is full. Cannot add new items.\n");
      return;
   }
```

```
    arr[end] = p;
    end++;

    /* loop back to complete the circle of the queue if end is reached */
    if (end == MAX)
        end = 0;
}

char * queue_retrieve (void)
{
    /* loop back */
    if (start == MAX)
        start= 0;

    if (start == end)
    {
        printf ("Queue is empty.\n");
        return NULL;
    }

    start++;

    return arr[start-1];
}
```

14.5 APPLICATIONS OF STACKS

Stack operations are implemented in machines to do some basic tasks, such as evaluating expressions, handling dynamic memory allocation, etc.

Consider an arithmetic expression, a+b*c. In this expression, the addition operation is not evaluated first. This is because, operators are evaluated in the order of their precedence in the expression. So the entire equation is examined to determine whether there is any operator with higher precedence. After examining the expression, back tracking is done to evaluate the first operator to obtain the result. The back tracking operation can be best implemented by the stack operations. The various stack-oriented notations are:

- Infix
- Prefix
- Postfix

Infix notation

As we know, an ordinary mathematical expression is called infix notation. When using infix notation, the operators are placed between the operands in an expression. While denoting an expression in infix notation, we use parenthesis to specify the order in which the operations to be performed. Otherwise, the precedence rules will be followed to obtain an unambiguous result of the expression.

Example 14.1
An example of an infix notation is, (A/B)+(C-D)*E.

Prefix Notation

In the case of prefix notation, the operators are placed before the operands in a mathematical expression. This notation is also known as polish notation.

Example 14.2
The prefix notation for the equation, A/B+C-D*E, is +/AB*-CDE.

Conversion from Infix to Prefix Notation

The following algorithm is used to convert an infix expression to prefix expression.

Algorithm

1. First, initialize the stack to be empty and reverse the given input string.
2. For each character in the input string
   If the input string is a right parenthesis, push it onto the stack
   If the input string is an operand, append to the output.
   else
   If the stack is empty or the operator has higher priority than the operator on the top of the stack or the top of the stack is a right parenthesis,
   then
   Push the operator onto the stack
   else
   Pop the operator from the stack and append to the output.
3. If the input string is a left parenthesis, pop operators from the stack and append all the operators to the output until the right parenthesis is encountered. Pop the right parenthesis from the stack and discard it.
4. If the end of the input string is encountered, then iterate the loop until the stack is not empty

Pop the stack, and append the remaining input string to the output and reverse the output string.

Example 14.3
Consider the conversion of the infix expression, A*B/(C-D)+E*(F-G), to its equivalent prefix notation. First, to convert the infix expression to the prefix notation, reverse the given input string as follows:
)G-F(*E+)D-C(/B*A

Input String	Stack Operation	Postfix Notation
)	)	
G	)	G
-	)-	G
F	)-	GF
(	Empty	GF-
*	*	GF-

(*Contd*)

(*Contd*)

Input String	Stack Operation	Postfix Notation
E	*	GF-E
+	+	GF-E*
)	+)	GF-E*
D	+)	GF-E*D
-	+)-	GF-E*D
C	+)-	GF-E*DC
(	+	GF-E*DC-
/	+/	GF-E*DC-
B	+/	GF-E*DC-B
*	+/*	GF-E*DC-BA
A	+/*	GF-E*DC-BA
	Empty	GF-E*DC-BA*/+

Now, reverse the output string, GF-E*DC-BA*/+, as +/*AB-CD*E-FG to obtain the prefix notation. Thus, the given infix expression is converted to its equivalent prefix notation. Conversion of an infix to prefix expression is shown in Fig. 14.8.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
#define MAX 50
    char output[MAX];
    char stack[MAX];
    char input[MAX];
    char *s, *t;
    int top;
    int 1 ;
    /*Initializing Function*/
    void Initialize(void);
    void SetExpression(char *);
    char PopFromStack(void);
    void PushOnStack(char);
    int priority(char);
    void ConvertToPrefix(void);
    void main()
    {
    clrscr();
    Initialize();
    printf("\nEnter an infix expression: ");
    gets(input);
    SetExpression(input);
        strrev(s);
        ConvertToPrefix();
        strrev(output);
        printf("\nThe Prefix expression is: ");
    puts(output);
    getch();
}
void Initialize(void)
```

```
{
        top = -1;
        strcpy(output, "");
        strcpy(stack, "");
        l = 0;
}
void SetExpression(char *str)
{
        s = str;
        l = strlen(s);
        *(output + l) = '\0';
        t = output;
}
/*Adding operator to the stack */
void PushOnStack(char c)
{
            if (top == MAX - 1)
            printf("\nStack is full.\n");
            else
        {
            top++;
            stack[top] = c;
            }
    }
/*Popping an operator from the stack*/
char PopFromStack(void)
{
    if (top == -1)
        return -1;
    else
    {
            char item = stack[top];
            top--;
            return item;
            }
}
/* Returning the priority of the operator */
int priority(char c)
{
        if (c == '^') return 3;
        if (c == '*' || c == '/' || c == '%')
            return 2;
        else if (c == '+' || c == '-') return 1;
        else return 0;
}

/* Converting the Infix expression to Prefix*/
void ConvertToPrefix (void)
{
        char opr;
        while (*(s) )
        {
            if (*(s) == ' ' || *(s) == '\t')
        {
            s++;
```

```
                continue;
        }
                if (isdigit (*(s) ) || isalpha (*(s) ))
        {
        while (isdigit (*(s) ) || isalpha (*(s) ))
        {
                *(t) = *(s);
                s++ ;
                t++;
                }
        }
        if (*(s) == ')')
    {
                PushOnStack(*(s));
                s++;
                }
            if (*(s) == '*' || *(s) == '+' || *(s) == '/' || *(s) == '%' || *(s) == '-' || *(s)
            == '^')
    {
            if (top != -1)
            {
            opr = PopFromStack ();
            while(priority (opr)>priority (*(s)))
            {
            *(t) = opr;
            t++;
            opr = PopFromStack();
            }
            PushOnStack(opr);
            PushOnStack(*(s));
            }
                else PushOnStack ( *(s));
                s++ ;
            }
            if (*(s) == '(')
            {
                opr = PopFromStack ();
                while (opr != ')')
                {
                *(t) = opr;
                t++;
                opr = PopFromStack ();
                }
                s++;
            }
    }
            while (top != -1)
    {
            opr = PopFromStack ();
            *(t) = opr;
            t++;
    }
    t++;
}
```

Fig. 14.8

Evaluation of Prefix Expression

Stacks are also used to evaluate a prefix expression. To evaluate a prefix expression, consider the following steps:

- Reverse the given input string.
- If the input string is an operand, then push it onto the stack.
- If the input string is an operator, then the first two operands on the stack are evaluated using this operator by popping them from the stack and the result is also placed onto the stack.

Example 14.4
Consider the evaluation of a prefix expression, +/63*-432. To do this, reverse the input string as 234-*36/+ shown in Fig. 14.9.

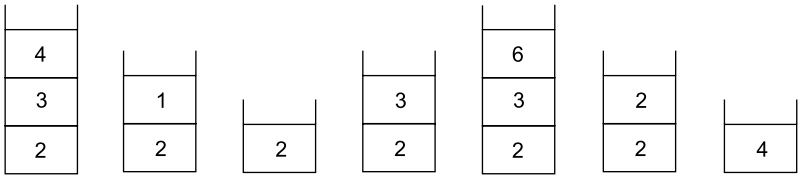


Fig. 14.9

Postfix Notation

When we use postfix notation for a mathematical expression, the operators are placed after the operands, i.e. the operators are preceded by the operands. Postfix notation is also known as Reverse Polish Notation (RPN). The advantage of using prefix and postfix notations is that the use of parenthesis and operator precedence rules is avoided completely.

Example 14.5
The postfix notation for the expression, A/B+C-D*E, is AB/CD-E*+.

Conversion from Infix to Postfix Notation

The following algorithm is used to convert an infix expression to an equivalent postfix expression:

Algorithm

1. First, initialize the stack to be empty.
2. For each character in the input string,
   If input string is an operand, append to the output.
   If the input string is a left parenthesis, push it onto the stack
   else
   If stack empty or the operator has higher priority than the operator on the top of the stack or the top of the stack is opening parenthesis
   then
   Push the operator onto the stack

else

Pop the operator from the stack and append to the output.

3. If the input string is a closing parenthesis, pop operators from the stack and append the operators to the output until an opening parenthesis is encountered. Pop the opening parenthesis from the stack and discard it.
4. If the end of the input string is encountered, then iterate the loop until the stack is not empty.

Pop the stack and append the remaining input string to the output.

Example 14.6

Consider the conversion of the infix expression, A*B/(C-D)+E*(F-G), to its equivalent postfix notation:

Input String	Stack Operation	Postfix Notation
A	Empty	A
*	*	A
B	*	AB
/	/	AB*
(	/(	AB*
C	/(	AB*C
-	/(-	AB*C
D	/(-	AB*CD
)	/	AB*CD-
+	+	AB*CD-/
E	+	AB*CD-/E
*	+*	AB*CD-/E
(	+*(	AB*CD-/E
F	+*(	AB*CD-/EF
-	+*(-	AB*CD-/EF
G	+*(-	AB*CD-/EFG
)	Empty	AB*CD-/EFG-*+

The given infix expression is converted to its equivalent postfix notation, as shown in Fig. 14.10.

```
    {
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>

#define MAX 50
    char output[MAX];
    char stack[MAX];
    char input[MAX];
    char *s, *t ;
    int top;
    int l ;
    /*Initializing Function*/
    void Initialize(void);
    void SetExpression(char *);
    char PopFromStack(void);
```

```
        void PushOnStack(char);
        int priority(char);
        void ConvertToPostfix(void);
        void main()
        {
        clrscr();
        Initialize();
        printf("\nEnter an infix expression: ");
        gets(input);
        SetExpression(input);
        ConvertToPostfix();
        printf("\nThe postfix expression is: ");
        puts(output);
        getch();
        }
        void Initialize(void)
        {
            top = -1;
            strcpy(output, "");
            strcpy(stack, "");
            l = 0;
        }
        void SetExpression(char *str)
        {
            s = str;
            l = strlen(s);
            *(output + l) = '\0';
            t = output;
    }
    /*Adding operator to the stack */
    void PushOnStack ( char c)
    {
            if (top == MAX - 1)
            printf ("\nStack is full.\n");
            else
            {
                top++;
                stack[top] = c;
            }
    }

    /* Popping an operator from the stack */
    char PopFromStack (void)
        {
            if (top == -1) /* Stack is empty*/
            return -1;
            else
        {
                char item = stack[top];
                top--;
                return item;
        }
    }

/* Returning the priority of the operator */
```

```
int priority (char c)
{
        if (c == '^') return 3 ;
        if (c == '*' || c == '/' || c == '%') return 2;
        else if (c == '+' || c == '-') return 1;
        else return 0;
}
/* Converting the infix expression to postfix */
void ConvertToPostfix (void)
{
    char opr;
    while (*(s) )
    {
    if (*(s) == ' ' || *(s) == '\t')
    {
    s++;
    continue;
    }
    if (isdigit (*(s) ) || isalpha (*(s) ))
    {
    while (isdigit (*(s) ) || isalpha (*(s) ))
    {
    *(t) = *(s);
    s++;
    t++;
    }
    }
    if (*(s) == '(')
    {
    PushOnStack ( *(s));
    s++;
    }
    if (*(s) == '*' || *(s) == '+' || *(s)== '/' || *(s) == '%' || *(s) == '-' || *(s)
        == '^')
        {
            if (top != -1)
        {
                opr = PopFromStack ();
                while (priority (opr) >=
                priority (*(s) ))
                {
                *(t) = opr;
                t++;
                opr =
                PopFromStack ();
                }
                PushOnStack ( opr);
                PushOnStack ( *(s) );
                }
                else PushOnStack ( *(s));
                s++;
        }
    if (*(s) == ')')
    {
                opr = PopFromStack ();
```

```
            while (opr != '(')
            {
                    *(t) = opr;
                    t++;
                    opr = PopFromStack ();
            }
                s++;
            }
        }
        while (top != -1)
        {
                opr = PopFromStack ();
                *(t) = opr;
                t++;
            }
            t++;
    }
```

Fig. 14.10

Evaluation of Postfix Expression

Stacks are used to evaluate the postfix expression. To evaluate the postfix expression, consider the following steps:

- If the input string is an operand, then push it onto the stack.
- If the input string is an operator, then the first two operands on the stack are evaluated using this operator by popping them from the stack and the result is also placed onto the stack.

Example 14.7
Consider the evaluation of postfix expression 68+92-/.

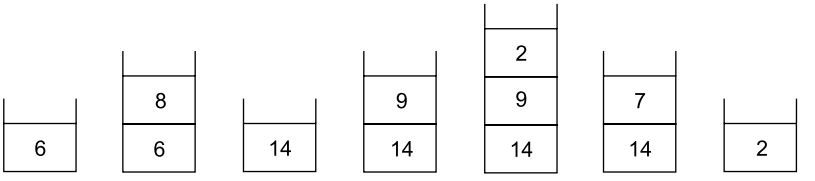


Fig. 14.11

POINTS TO REMEMBER

1. A queue is called as a First In First Out (FIFO) structure. In contrast, a stack is called as a Last In First Out (LIFO) structure.
2. The most notable use of stacks is in system software (such as compilers, operating systems, etc).
3. A pop (i.e. read) operation in the case of a stack is destructive. That is, once an item is popped from the stack, it is no longer available.

4. A queue is a linear, sequential list of items that are accessed in the order *First In First Out (FIFO)*.
5. A list/queue can also be circular in which case, it is called as a **circular linked list** or a **circular queue.**
6. Arrays and lists are called linear data structures.
7. The ordinary mathematical expression is called infix notation.
8. In prefix notation the operators are placed before the operands.
9. In postfix notation the operators are placed after the operands.

Review Questions and Exercises

14.1 Explain in detail the push and pop operations of stack with a real life example.

14.2 Describe how items in a queue are accessed.

14.3 Describe about circular queue.

14.4 Define infix and prefix notation.

14.5 Explain about the evaluation of postfix notation.

State whether the following statements are True or False

(a) Items in a stack exit in the same order in which they had entered the stack.

(b) Stacks are rarely used in computer applications.

(c) Writing a value to the stack is called as a *pop* operation, whereas reading a value from it is called as a *push* operation.

(d) Once an item is popped from the stack, it is no longer available.

(e) Pop operation increments the stack counter by one to see if the stack is empty.

(f) The stack can be implemented using the dynamic memory management techniques.

Answers to True and False Questions

(a) False (b) False (c) False (d) True
(e) False (f) False

UNIT VII

Chapter 15

Dynamic Memory Allocation and Linked Lists

15.1 INTRODUCTION

Most often we face situations in programming where the data is dynamic in nature. That is, the number of data items keep changing during execution of the program. For example, consider a program for processing the list of customers of a corporation. The list grows when names are added and shrinks when names are deleted. When list grows we need to allocate more memory space to the list to accommodate additional data items. Such situations can be handled more easily and effectively by using what is known as *dynamic data structures* in conjunction with *dynamic memory management* techniques.

Dynamic data structures provide flexibility in adding, deleting or rearranging data items at run time. Dynamic memory management techniques permit us to allocate additional memory space or to release unwanted space at run time, thus, optimizing the use of storage space. This chapter discusses the concepts of *linked lists*, one of the basic types of dynamic data structures. Before we take up linked lists, we shall briefly introduce the dynamic storage management functions that are available in C. These functions would be extensively used in processing linked lists.

15.2 DYNAMIC MEMORY ALLOCATION

C language requires the number of elements in an array to be specified at compile time. But we may not be able to do so always. Our initial judgement of size, if it is wrong, may cause failure of the program or wastage of memory space.

Many languages permit a programmer to specify an array's size at run time. Such languages have the ability to calculate and assign, during execution, the memory space required by the variables in a program. The process of allocating memory at run time is known as *dynamic memory allocation*. Although C does not inherently have this facility, there are four library routines known as "memory management functions" that can be used for allocating and freeing memory during program execution. They are listed in Table 15.1. These functions help us build complex application programs that use the available memory intelligently.

Table 15.1 Memory Allocation Functions

Function	Task
malloc	Allocates requested size of bytes and returns a pointer to the first byte of the allocated space.
calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
free	Frees previously allocated space.
realloc	Modifies the size of previously allocated space.

Memory Allocation Process

Before we discuss these functions, let us look at the memory allocation process associated with a C program. Figure 15.1 shows the conceptual view of storage of a C program in memory.

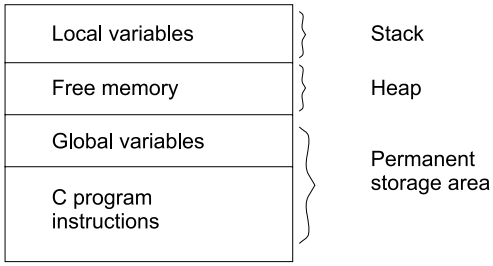


Fig. 15.1 *Storage of a C program*

The program instructions and global and static variables are stored in a region known as permanent storage area and the local variables are stored in another area called *stack*. The memory space that is located between these two regions is available for dynamic allocation during execution of the program. This free memory region is called the *heap*. The size of the heap keeps changing when program is executed due to creation and death of variables that are local to functions and blocks. Therefore, it is possible to encounter memory "overflow" during dynamic allocation process. In such situations, the memory allocation functions mentioned above return a NULL pointer (when they fail to locate enough memory requested).

Allocating a Block of Memory

A block of memory may be allocated using the function **malloc**. The **malloc** function reserves a block of memory of specified size and returns a pointer of type **void**. This means that we can assign it to any type of pointer. It takes the following form:

ptr = (*cast-type**) **malloc**(*byte-size*);

ptr is a pointer of type cast-type. The **malloc** returns a pointer (of *cast type*) to an area of memory with size *byte-size*.

Example:

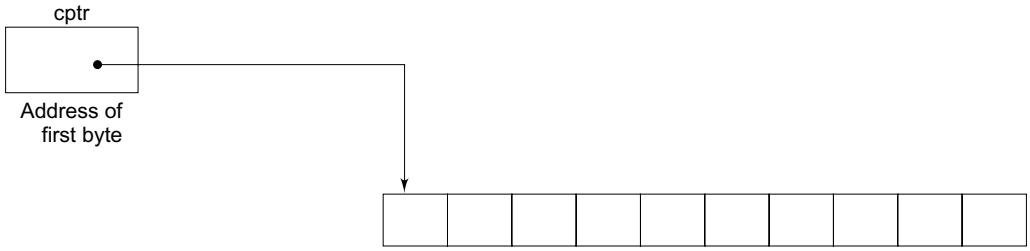
```
x = (int *) malloc(100 * sizeof(int));
```

On successful execution of this statement, a memory space equivalent to "100 times the size of an **int**" bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer **x** of type of **int**

Similarly, the statement

```
cptr = (char *) malloc(10);
```

allocates 10 bytes of space for the pointer **cptr** of type **char**. This is illustrated below:



Note that the storage space allocated dynamically has no name and therefore its contents can be accessed only through a pointer.

We may also use **malloc** to allocate space for complex data types such as structures. Example:

```
st_var = (struct store *)malloc(sizeof(struct store));
```

where **st_var** is a pointer of type **struct store.**

Remember, the **malloc** allocates a block of contiguous bytes. The allocation can fail if the space in the heap is not sufficient to satisfy the request. If it fails, it returns a NULL. We should therefore check whether the allocation is successful before using the memory pointer. This is illustrated in the program in Fig. 15.2.

Example 15.1

Write a program that uses a table of integers whose size will be specified interactively at run time.

The program is given in Fig. 15.2. It tests for availability of memory space of required size. If it is available, then the required space is allocated and the address of the first byte of the space allocated is displayed. The program also illustrates the use of pointer variable for storing and accessing the table values.

Program

```
/*******************************************************************************/
/*                        USE OF MALLOC FUNCTION                               */
/*******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#define NULL 0

main()
```

```
{
    int *p, *table;
    int size;
    printf("\nWhat is the size of table ?");
    scanf("%d", size);
    printf("\n")
    /* Memory allocation */
    if((table = (int *)malloc(size * sizeof(int))) == NULL)
    {
        printf("No space available \n");
        exit(1);
    }
    printf("\nAddress of the first byte is %u\n",table);
    /* Reading table values */
    printf("\nInput table values\n");
    for(p = table; p < table + size; p++)
        scanf("%d", p);
    /* Printing table values in reverse order */
    for (p = table + size –1; p >= table; p ––)
    printf("%d is stored at address %u\n", *p, p);
}
```

Output

```
What is the size of the table ? 5
Address of the first byte is 2262

Input table values
11 12 13 14 15

15  is  stored  at  address  2270
14  is  stored  at  address  2268
13  is  stored  at  address  2266
12  is  stored  at  address  2264
11  is  stored  at  address  2262
```

Fig. 15.2 *Memory allocation with* ***malloc***

Allocating Multiple Blocks of Memory

calloc is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. While **malloc** allocates a single block of storage space, **calloc** allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The general form of **calloc** is:

ptr = (*cast-type* *) **calloc** (*n,elem-size*);

The above statement allocates contiguous space for *n* blocks, each of size *elem-size* bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

The following segment of a program allocates space for a structure variable:

```
. . . .
. . . .
struct student
{
    char  name[25];
    float  age;
    long int id_num;
};
typedef struct student record;
record *st_ptr;
int class_size = 30;

st_ptr = (record *)calloc(class_size, sizeof(record));


. . . .

. . . .
```

record is of type **struct student** having three members: **name, age** and **id_num.** The **calloc** allocates memory to hold data for 30 such records. We must be sure that the requested memory has been allocated successfully before using the **st_ptr**. This may be done as follows:

```
if(st_ptr == NULL)
{
    printf("Available memory not sufficient");
    exit(1);
}
```

Releasing the Used Space

Compile-time storage of a variable is allocated and released by the system in accordance with its storage class. With the dynamic run-time allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited.

When we no longer need the data we stored in a block of memory, and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the **free** function:

```
free(ptr);
```

ptr is a pointer to a memory block which has already been created by **malloc** or **calloc**. Use of an invalid pointer in the call may create problems and cause system crash. The use of **free** function has been illustrated in Example 15.2.

Altering the Size of a Block

It is likely that we discover later, the previously allocated memory is not sufficient and we need additional space for more elements. It is also possible that the memory allocated is much larger than necessary and we want to reduce it. In both the cases, we can change the memory size already allocated with

the help of the function **realloc**. This process is called the *reallocation* of memory. For example, if the original allocation is done by the statement

ptr = **malloc**(size);

then reallocation of space may be done by the statement

ptr = **realloc**(ptr,*newsize*);

This function allocates a new memory space of size *newsize* to the pointer variable **ptr** and returns a pointer to the first byte of the new memory block. The *newsize* may be larger of smaller than the *size*. Remember, the new memory block may or may not begin at the same place as the old one. In case, it is not able to find additional space in the same region, it will create the same in an entirely new region and move the contents of the old block into the new block. The function guarantees that the old data will remain intact.

If the function is unsuccessful in locating additional space, it returns a NULL pointer and the original block is freed(lost). This implies that it is very necessary to test the success of operation before proceeding further. This is illustrated in the program of Example 15.2.

Example 15.2

Write a program to store a character string in a block of memory space created by **malloc** and then modify the same to store a larger string.

The program is shown in Fig. 15.3. The output illustrates that the original buffer size obtained is modified to contain a larger string. Note that the original contents of the buffer remains same even after modification of the original size.

Program

```
/*******************************************************************************/
/*                    USE OF realloc AND free FUNCTION                         */
/*******************************************************************************/

#include <stdio.h)
#include <stdlib.h>
#define NULL 0

main( )
{
    char *buffer;
    /* Allocating memory */
    if((buffer = (char *)malloc(10)) == NULL)
    {
        printf("malloc failed.\n");
        exit(1);
    }
    printf ("Buffer of size %d created \n",_msize(buffer));
    strcpy(buffer, "HYDERABAD");
    printf("\nBuffer contains: %s \n",buffer);
    /* Reallocation */
    if((buffer = (char *)realloc(buffer, 15)) == NULL)
    {
        printf("Reallocation failed. \n");
        exit(1);
    }
```

```
    printf("\nBuffer size modified. \n");
    printf("\nBuffer still contains: %s \n",buffer);
    strcpy(buffer, "SECUNDERABAD");
    printf("\nBuffer now contains: %s \n",buffer);
    /* Freeing memory */
    free(buffer);
}
```

Output

```
Buffer of size 10 created
Buffer contains: HYDERABAD
Buffer size modified
Buffer still contains: HYDERABAD
Buffer now contains: SECUNDERABAD
```

Fig. 15.3 *Reallocation and release of memory space*

15.3 LINKED LISTS

We know that a list refers to a set of items organized sequentially. An array is an example of list. In an array, the sequential organization is provided implicitly by its index. We use the index for accessing and manipulation of array elements. One major problem with the arrays is that the size of an array must be specified precisely at the beginning. As pointed out earlier, this may be a difficult task in many practical applications.

A completely different way to represent a list is to make each item in the list part of a structure that also contains a "link" to the structure containing the next item, as shown in Fig. 15.4. This type of list is called a *linked list* because it is a list whose order is given by links from one item to the next.

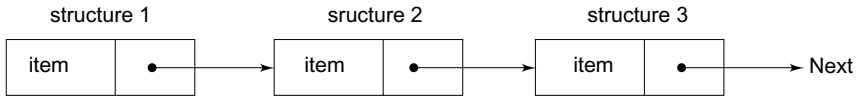


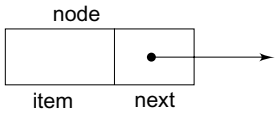
Fig. 15.4 *A linked list*

15.3.1 Self-Referential Structure

Each structure of the list is called a *node* and consists of two fields, one containing the *item*, and the other containing the *address* of the next item (a pointer to the next item) in the list. A linked list is therefore a collection of structures ordered not by their physical placement in memory (like an array) but by logical links that are stored as part of the data in the structure itself. The link is in the form of a pointer to another structure of the same type. Such a structure is represented as follows:

```
struct node
{
    int     item;
    struct node *next;
};
```

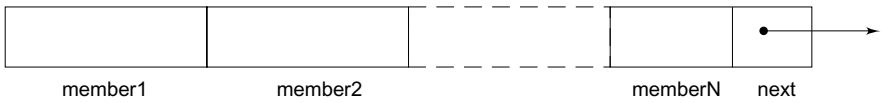
The first member is an integer item and the second a pointer to the next node in the list as shown below. Remember, the **item** is an integer here only for simplicity, and could be any complex data type.



Such structures which contain a member field that point to the same structure type are called *self-referential* structures.
A node may be represented in general form as follows:

```
struct tag-name
{
   type member1;
   type member2;
   . . . .
   . . . .
   struct tag-name *next;
};
```

The structure may contain more than one item with different data types. However, one of the items must be a pointer of the type **tag-name**.



15.3.2 Singly Linked List

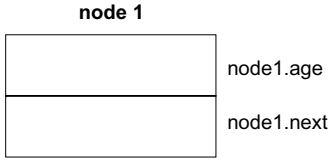
Let us consider a simple example to illustrate the concept of linking. Suppose we define a structure as follows:

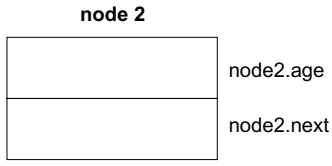
```
struct link_list
{
   float age;
   struct link_list *next;
};
```

For simplicity, let us assume that the list contains two nodes **node1** and **node2**. They are of type **struct link_list** and are defined as follows:

```
struct link_list node1, node2;
```

This statement creates space for two nodes each containing two empty fields as shown below:

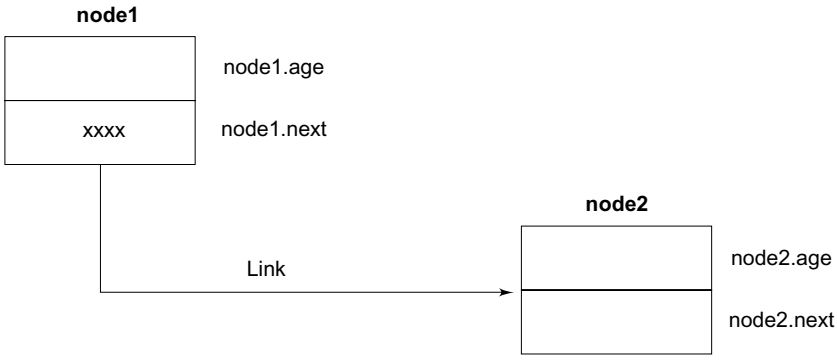




The **next** pointer of **node1** can be made to point to **node2** by the statement

node1.next = &node2;

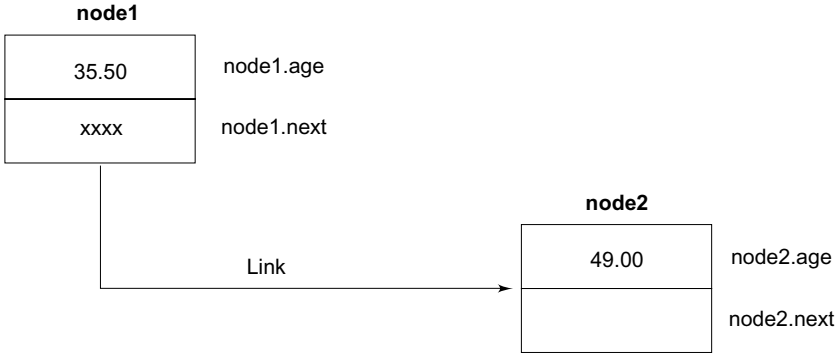
This statement stores the address of **node2** into the field **node1.next** and thus establishes a "link" between **node1** and **node2** as shown below:



"xxxx" is the address of **node2** where the value of the variable **node2.age** will be stored. Now let us assign values to the field age.

```
node1.age = 35.50;
node2.age  = 49.00;
```

The result is as follows:



We may continue this process to create a linked list of any number of values. For example,

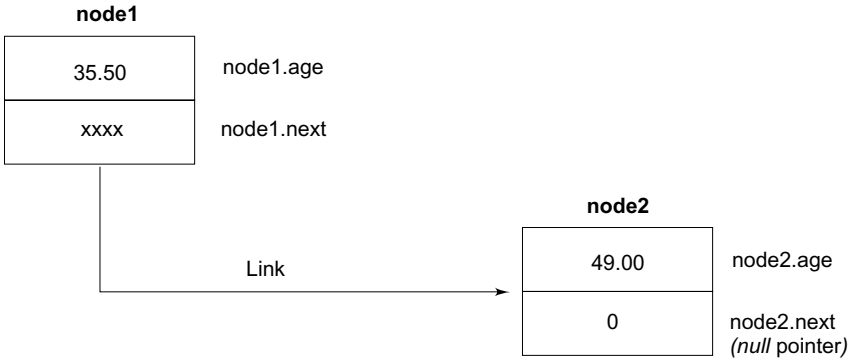
node2.next = &node3;

would add another link provided **node3** has been declared as a variable of type **struct link-list**.

No list goes on forever. Every list must have an end. We must therefore indicate the end of a linked list. This is necessary for processing the list. C has a special pointer value called null that can be stored in the **next** field of the last node. In our two-node list, the end of the list is marked as follows:

node2.next = 0;

The final linked list containing two nodes is shown below:



The value of the age member of **node2** can be accessed using the **next** member of **node1** as follows:

printf("%f\n", node1.next–>age);

15.4 ADVANTAGES OF LINKED LISTS

A linked list is a *dynamic data structure*. Therefore, the primary advantage of linked lists over arrays is that linked lists can grow or shrink in size during the execution of a program. A linked list can be made just as long as required.

Another advantage is that a linked list does not waste memory space. It uses the memory that is just needed for the list at any point of time. This is because it is not necessary to specify the number of nodes to be used in the list.

The third, and the more important advantage is that the linked lists provide flexibility in allowing the items to be rearranged efficiently. It is easier to insert or delete items by rearranging the links. This is shown in Fig. 15.5.

The major limitation of linked lists is that the access to any arbitrary item is little cumbersome and time consuming. Whenever we deal with a fixed length list, it would be better to use an array rather than a lined list. We must also note that a linked list will use more storage than an array with the same number of items. This is because each item has an additional link field.

15.5 TYPES OF LINKED LISTS

There are different types of linked lists. The one we discussed so far is known as *linear singly* linked list. The other linked lists are:

- Circular linked lists
- Two-way or doubly linked lists
- Circular doubly linked lists

The circular linked lists have no beginning and no end. The last item points back to the first item. The doubly linked lists uses double set of pointers, one pointing to the *next* item and other pointing to the preceding item. This allows us to traverse the list in either direction. Circular doubly linked lists employs both the forward pointer and backward pointer in circlular form. Figure 15.6 illustrates various kinds of linked lists.

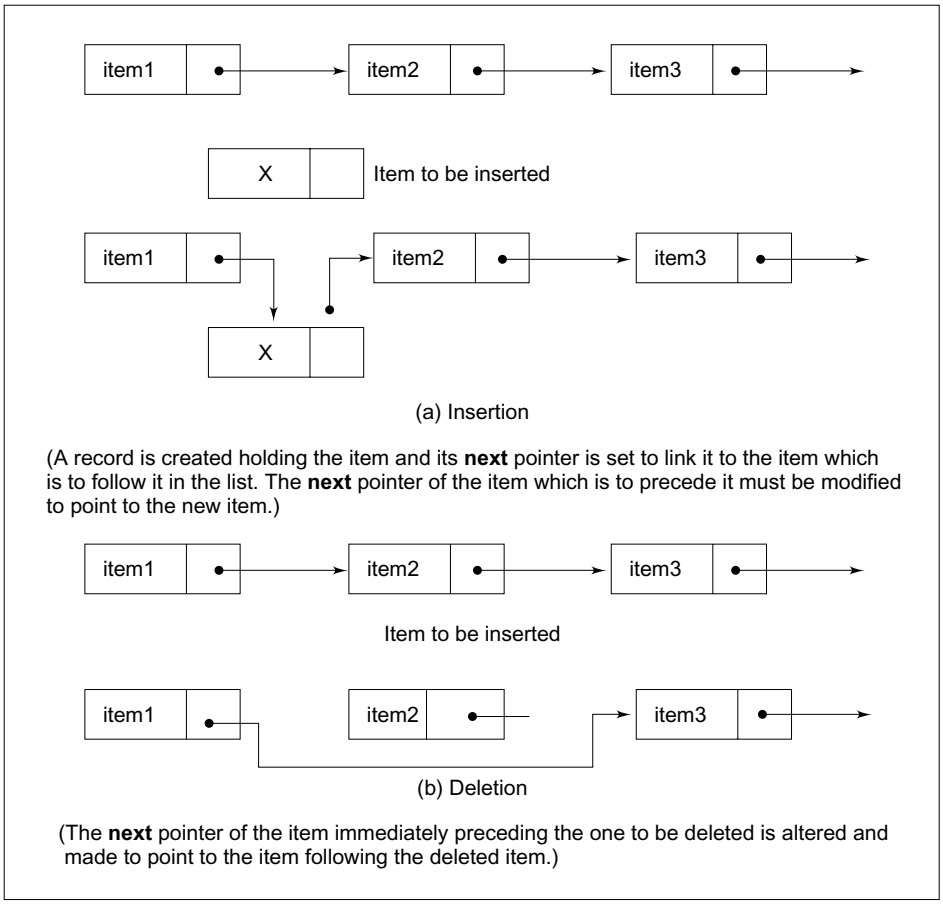


Fig. 15.5 *Insertion into and deletion from a linked list*

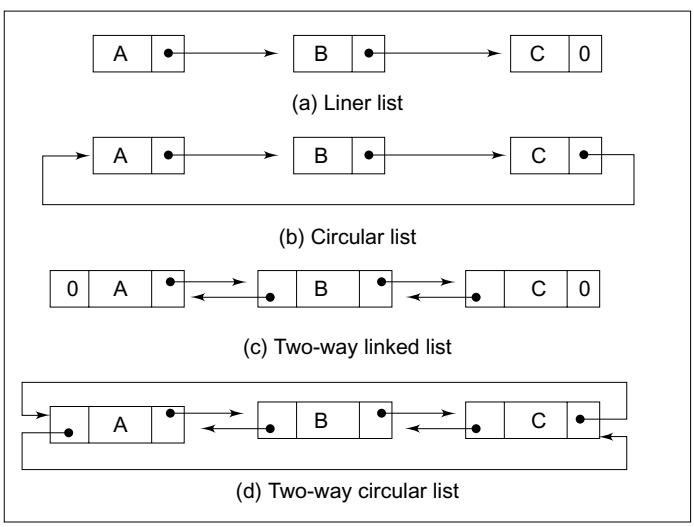


Fig. 15.6 *Different types of linked lists*

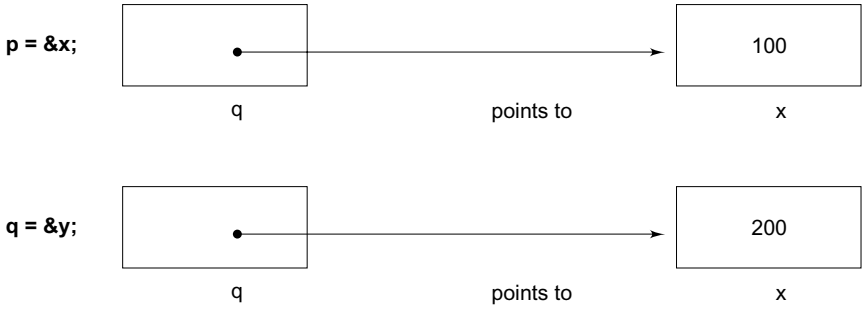
15.6 POINTERS REVISITED

The concept of pointers was discussed in Chapter 11. Since pointers are used extensively in processing of the linked lists, we shall briefly review some of their properties that are directly relevant to the processing of lists.

We know that variables can be declared as pointers, specifying the type of data item they can point to. In effect, the pointer will hold the address of the data item and can be used to access its value. In processing linked lists, we mostly use pointers of type structures.

It is most important to remember the distinction between the pointer variable **ptr**, which contain the address of a variable, and the referenced variable ***ptr**, which denotes the value of variable to which **ptr**'s value points. The following examples illustrates this distinction. In these illustrations, we assume that the pointers **p** and **q** and the variables **x** and **y** are declared to be of same type.

(a) Initialization

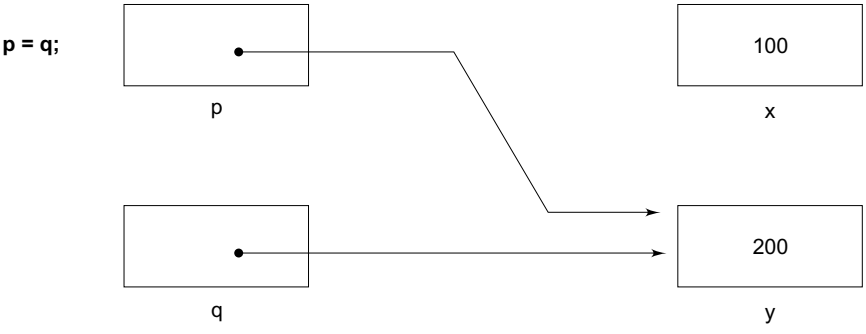


The pointer **p** contains the address of **x** and **q** contains the address of **y**.

***p = 100 and *q = 200 and p <> q**

(b) Assignment p = q

The assignment **p** = **q** assigns the address of the variable **y** to the pointer variable **p** and therefore **p** now points to the variable **y**.

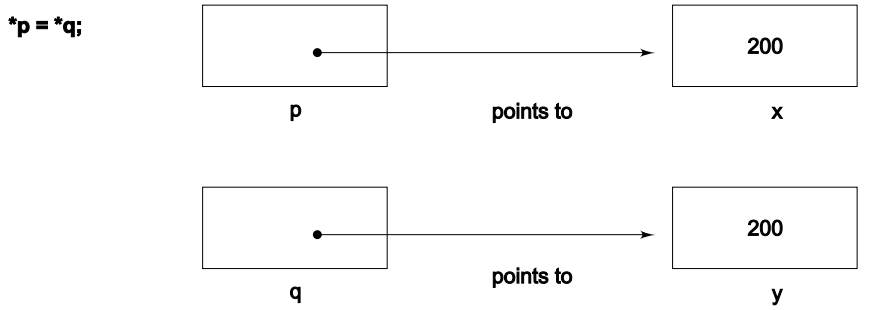


Both the pointer variables point to the same variable.

***p = *q = 200 but x <> y**

(c) Assignment *p = *q

This assignment statement puts the value of the variable pointed to by **q** in the location of the variable pointed to by **p**.



The pointer **p** still points to the same variable **x** but the old value of **x** is replaced by 200 (which is pointed to by **q**).

x = y = 200 but p <> q

(d) NULL pointers

A special constant known as NULL pointer (0) is available in C to initialize pointers that point to nothing. That is the statements

```
p  =  0; (or p = NULL;)        p  0
q  =  0; (q = NULL;)           q  0
```

make the pointers **p** and **q** point to nothing. They can be later used to point any values.

We know that a pointer must be initialized by assigning a memory address before using it. There are two ways of assigning memory addresses to a pointer.

1. Assigning an existing variable address (static assignment)

```
ptr = &count;
```

2. Using a memory allocation function (dynamic assignment)

```
ptr = (int *) malloc(sizeof(int));
```

15.7 BASIC LIST OPERATIONS

We can treat a linked list as an abstract data type and perform the following basic operations:

1. Creating a list
2. Traversing the list
3. Counting the items in the list
4. Printing the list (or sublist)
5. Looking up an item for editing or printing
6. Inserting an item
7. Deleting an item
8. Concatenating two lists

Creating a Linked List

In Sec. 15.3 we created a two-element linked list using the structure variable names **node1** and **node2**. We also used the address operator **&** and member operators . and – > for creating and accessing

individual items. The very idea of using a linked list is to avoid any reference to specific number of items in the list so that we can insert or delete items as and when necessary. This can be achieved by using "anonymous" locations to store nodes. Such locations are accessed not by name, but by means of pointers which refer to them. (For example, we must avoid using references like **node1.age** and **node1.next** –> **age.**)

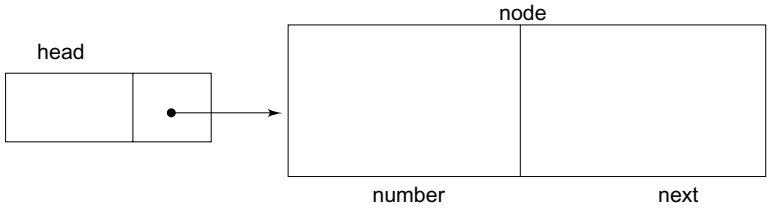
Annonymous locations are created using pointers and dynamic memory allocation functions such as **malloc**. We use a pointer **head** to create and access anonymous nodes. Consider the following:

```
struct linked_list
{
   int     number;
   struct linked_list  *next;
};
typedef struct linked_list node;
node  *head;
head = (node *) malloc(sizeof(node));
```

The **struct** declaration merely describes the format of the nodes and does not allocate storage. Storage space for a node is created only when the function **malloc** is called in the statement

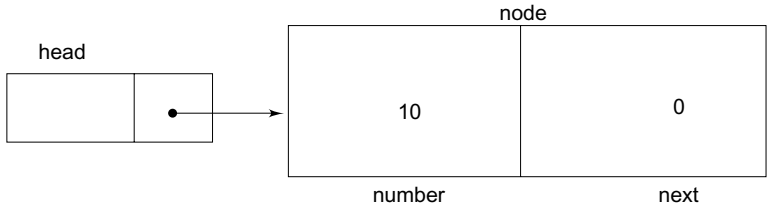
```
head = (node *) malloc(sizeof(node));
```

This statement obtains a piece of memory that is sufficient to store a node and assigns its address to the pointer variable **head**. This pointer indicates the beginning of the linked list.



The following statements store values in the member fields:

```
head → number  = 10;
head → next    = NULL;
```



The second node can be added as follows:

```
head–>next = (node *)malloc(sizeof(node));
head–>next–>number = 20;
head–>next–>next = NULL;
```

Although this process can be continued to create any number of nodes, it becomes cumbersome and clumsy if nodes are more than two. The above process may be easily implemented using both recursion and iteration techniques. The pointer can be moved from the current node to the next node by a sel-replacement statement such as

```
head = head –> next;
```

Example 15.3 shows creation of a complete linked list and printing of its contents using recursion.

Example 15.3

Write a program to create a linear linked list interactively and print out the list and the total number of items in the list.

The program shown in Fig. 15.7 first allocates a block of memory dynamically for the first using the statement

```
head = (node *)malloc(sizeof(node));
```

which returns a pointer to a structure of type **node** that has been type defined earlier. The linked list is then created by the function **create**. The function requests for the number to be placed in the current node that has been created. If the value assigned to the current node is –999, then NULL is assigned to the pointer variable **next** and the list ends. Otherwise, memory space is allocated to the next node using again the **malloc** function and the next value is placed into it. Note that the function **create** calls itself recursively and the process will continue until we enter the number –999.

The items stored in the linked list are printed using the function **print** which accept a pointer to the current node as an argument. It is a recursive function and stops when it receives a NULL pointer. Printing algorithm is as follows:

1. Start with the first node.
2. While there are valid nodes left to print
   (a) print the current item and
   (b) advance to next node.

Similarly, the function **count** counts the number of items in the list recursively and returns the total number of items to the **main** function. Note that the counting does not include the item –999 (contained in the dummy node).

Program

```
/*************************************************************************** /
/*                          CREATING A LINEAR LINKED LIST                    /*
/*************************************************************************** /
#include <stdio.h>
#include <stdlib.h>
#define NULL 0
struct linked_list
{
    int number;
    struct linked_list *next;
};
typedef struct linked_list node; /*   node type defined     */
```

```
main( )
{
    node   *head;
    void    create(node *p);
    int     count(node *p);
    void    print(node *p);

    head = (node *)malloc(sizeof(node));
    create(head);
    printf("\n");
    print(head);
    printf("\n");
    printf("\nNumber of items = %d \n", count(head));
}
void create(node  *list)

{
    printf("Input a number\n");
    printf("(type –999 at end): ");
    scanf("%d", &list–>number); /* create current node */

    if(list–>number == –999)
    {
        list–>next = NULL;
    }
    else        /* create next node */
    {
        list–>next = (node *)malloc(sizeof(node));
        create(list–>next);
    }
    return;
}
void print(node *list)
{
    if(list–>next != NULL)
    {
        printf("%d – –>",lis –>number); /* print current item */
        if(list–>next–>next == NULL)
            printf("%d", list–>next–>number);

    print(list–>next); /* move to next item */
    }
    return;
    }
int count(node *list)
{
    if(list–>next == NULL)
        return (0);
    else
        return(1 + count(list–>next));
}
```

Output

```
Input a number
(type –999 to end): 60
```

```
Input a number
(type –999 to end): 20
Input a number
(type –999 to end): 10
Input a number
(type –999 to end): 40
input a number
(type –999 to end): 30
Input a number
(Type –999 to end): 50
Input a number
(type –999 to end): –999
60 –>20 –>10 –>40 –>30 –>50 –>–999
Number of items = 6
```

Fig. 15.7 *Creating a linear lined list*

Inserting an Item

One of the advantages of linked lists is the comparative case with which new nodes can be inserted. It requires merely resetting of two pointers (rather than having to move around a list of data as would be the case with arrays).

Inserting a new item, say X, into the list has three situations:

1. Insertion at the front of the list.
2. Insertion in the middle of the list.
3. Insertion at the end of the list.

The process of insertion precedes a search for the place of insertion. The search involves in locating a node after which the new item is to be inserted. A general algorithm for insertion is as follows:

Begin

if the list is empty or
the new node comes before the head node *then*,
insert the new node as the head node,
else
If the new node comes after the last node, *then*,
insert the new node as the end node,
else
insert the new node in the body of the list.

End

Algorithm for placing the new item at the beginning of a linked list:

1. Obtain space for new node.

2. Assign data to the item field of new node.
3. Set the *next* field of the new node to point to the start of the list.
4. Change the head pointer to point to the new node.

Algorithm for inserting the new node X between two existing nodes, say, N1 and N2;

1. Set space for new node X.
2. Assign value to the item field of X
3. Set the *next* field of X to point to node N2
4. Set *next* field of N1 to point to X

Algorithm for inserting an item at the end of the list is similar to the one for inserting in the middle, except the *next* field of the new node is set to NULL (or set to point to a dummy or sentinal node, if it exists).

Example 15.4

Write a function to insert a given item *before* a specified node known as key node.

The function **insert** shown in Fig. 15.8 requests for the item to be inserted as well as the 'key node'. If the insertion happens to be at the beginning, then memory space is created for the new node, the value of new item is assigned to it and the pointer **head** is assigned to the next member. The pointer **new** which indicates the beginning of the new node is assigned to **head**. Note the following statements:

```
new–>number = x;
new–>next = head;
head = new;
```

```
/*******************************************************************************/
/*                              FUNCTION INSERT                                */
/*******************************************************************************/
node *insert(node *head)
{
    node *find(node  *p, int a);
    node *new;        /* pointer to new node */
    node *n1;         /* pointer to node preceding key node */
    int key;
    int x;            /* new item (number) to be inserted */

    printf("Value of new item?");
    scanf("%d", &x);
    printf("Value of key item? (type –999 it last) ");
    scanf("%d", &key);

    if(head–>number == key) /* new node is first */
    {
        new = (node*) malloc(sizeof(node));
        new–>number = x;
        new–>next = head;
        head = new;
    }
    else            /* find key node and insert new node */
    {               /* before the key node                */
        n1 = find(head, key);  /* find key node */

        if(n1 == NULL)
```

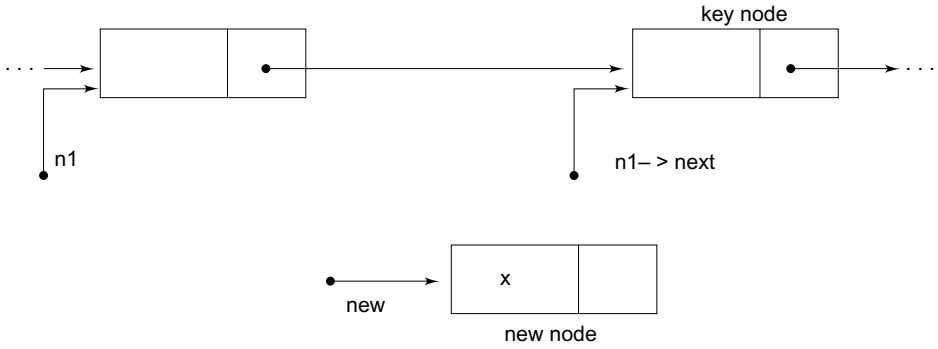
```
            printf("\n key is not found \n");
        else    /*  insert new node  */
        {
            new = (node *)malloc(sizeof(node));
            new–>number = x;
            new–>next = n1–>next;
            n1–>next = new;
        }
    }
    return(head);
}
node *find(node *list, int key)
{
    if(list–>next–>number == key) /*   key found */
        return(list);
else
    if(list–>next–>next == NULL) /* end */
        return(NULL);
    else
        find(list–>next, key);
}
```

Fig. 15.8 *A function for inserting an item into a linked list*

However, if the new item is to be inserted after an existing node, then we use the function **find** recursively to locate the 'key node'. The new item is inserted before the key node using the algorithm discussed above. This is illustrated below:

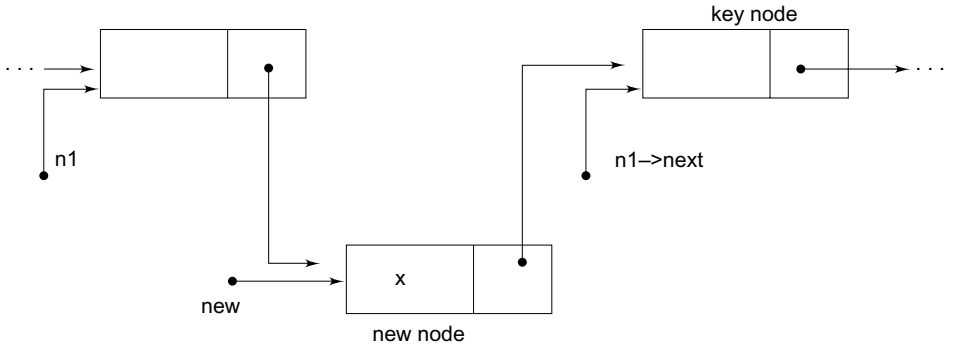
Before Insertion

```
new = (node *)malloc(sizeof(node));
new–>number = x;
```



After Insertion

```
new–>next = n1–>next;
n1–>next = new;
```



Deleting an Item

Deleting a node from the list is even easier than insertion, as only one pointer value needs to be changed. Here again we have three situations.

1. Deleting the first item
2. Deleting the last item
3. Deleting between two nodes in the middle of the list

In the first case, the head pointer is altered to point to the second item in the list. In the other two cases, the pointer of the item immediately preceding the one to be deleted is altered to point to the item following the deleted item. The general algorithm for delection is as follows:

Begin
    if the list is empty, *then*,
    node cannot be deleted,
    else
    if node to be deleted is the first node, *then*,
    make the head to point to the second node,
    else
    delete the node from the body of the list.

End

The memory space of deleted node may be released for reuse. As in the case of insertion, the process of deletion also involves search for the item to be deleted.

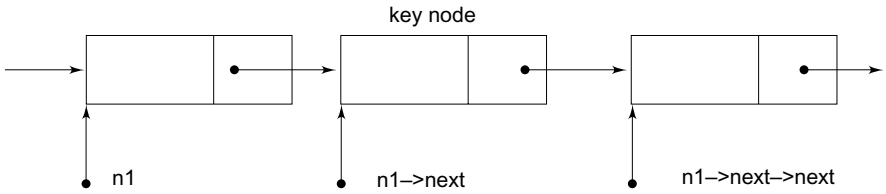
Example 15.5

Write a function to delete a specified node.

A function to delete a specified node is given in Fig. 15.9. The function first checks whether the specified item belongs to the first node. If yes, then the pointer to the second node is temporarily assigned the pointer variable **p**, the memory space occupied by the first node is freed and the location of the second node is assigned to **head**. Thus, the previous second node becomes the first node of the new list.

If the item to be deleted is not first one, then we use the **find** function to locate the position of 'key node' containing the item to be deleted. The pointers are interchanged with the help of a temporary pointer variable making the pointer in the preceding node to point to the node following the key node.

The memory space of key node that has been deleted is freed. The figure below shows the relative position of the key node.

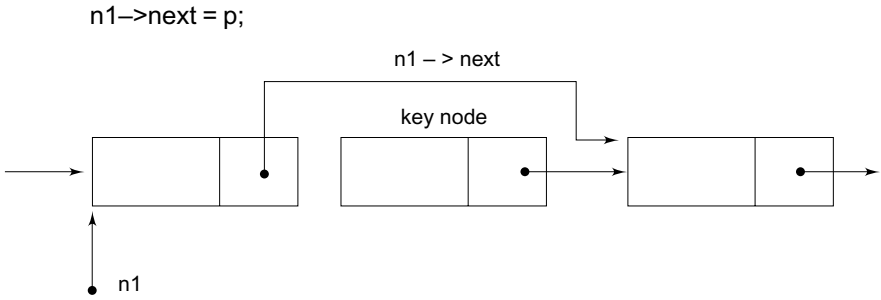



```
/*****************************************************************************/
/*                              FUNCTION  DELETE                             */
/*****************************************************************************/

node  *delete(node *head)
{
    node   *find(node *p, int a);
    int key;        /*  item to be deleted */
    node *n1;       /*  pointer to node preceding key node */
    node *p;        /*  temporary pointer */

    printf("\nWhat is the item (number) to be deleted?");
    scanf("%d", &key);

    if(head–>number == key)   /*  first node to be deleted) */
    {
        p = head–>next;       /* pointer to 2nd node in list */
        free(head);           /* release space of key node */
        head = p;             /* make head to point to 1st node */
    }
    else                      /* find key node and delete it */
    {
        n1 = find(head, key);

        if(n1 == NULL)
            printf("\n key not found\n");
        else                          /*  delete key node */
        {
            p = n1–>next–>next;       /*  pointer to the node
                                          following the key node */
            free(n1–>next);           /*  free key node */
            n1–>next = p;             /*  establish link */
        }
    }
    return(head);
}
            /*  USE FUNCTION Find() HERE     */
```

Fig. 15.9 *A function for deleting an item from linked list*

The execution of the following code deletes the key node.

```
p = n1–>next–>next;
free (n1–>next);
```



15.8 APPLICATION OF LINKED LISTS

Linked list concepts are useful to model many different abstract data types such as queues, stacks and trees.

If we restrict the process of insertions to one end of the list and deletions to the other end, then we have a model of a *queue*. That is, we can insert an item at the rear and remove an item at the front (see Fig. 15.10a). This obeys the discipline of "first in, first out" (FIFO). There are many examples of queues in real-life applications.

If we restrict insertions and deletions to occur only at one end of 1st, the beginning, then we model another data structure known as *stack*. Stacks are also referred to as *push-down* lists. An example of a stack is the "in" tray of a busy executive. The files pile up in the tray, and whenever the executive has time to clear the files, he takes it off from the top. That is, files are added at the top and removed from the top (see Fig. 15.10b). Stacks are sometimes referred to as "last in, first out" (LIFO) structure.

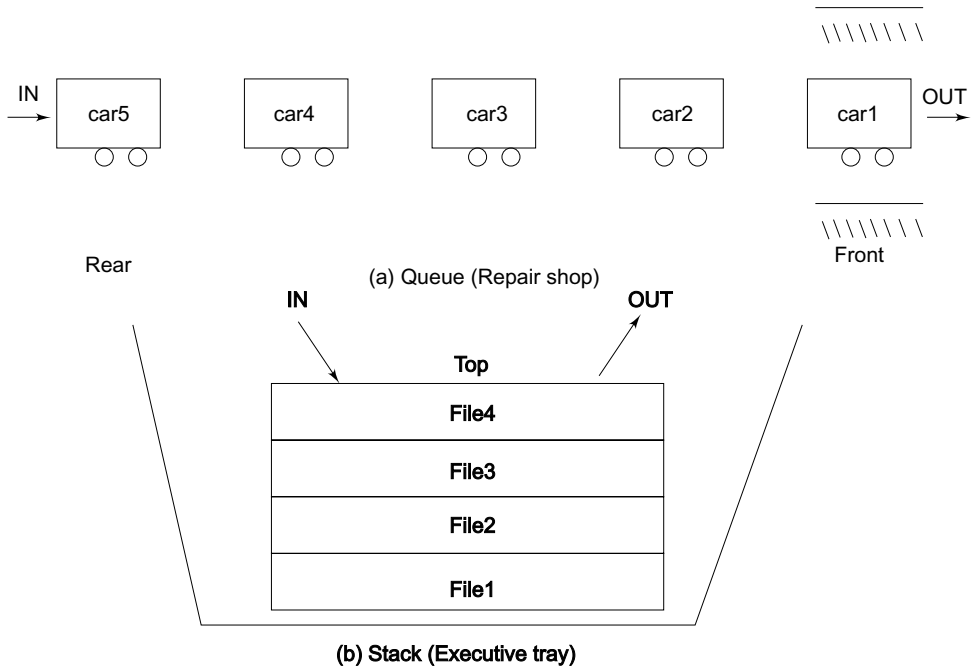


Fig. 15.10 *Applications of linked lists*

Lists, queues and stacks are all inherently one-dimensional. A *tree* represents a two-dimensional linked list. Trees are frequently encountered in everyday life. One example is the organizational chart of a large company. Another example is the chart of sports tournaments.

15.9 CIRCULAR LINKED LISTS

In a linked list, if we want to add any data item, we have to start from the beginning of the node because the last node has a null pointer. So, we cannot access any arbitrary item in a linked list. Unlike linked lists, circular linked lists can be used to add data items at any point without starting from the beginning point.

A singly linked circular list is a linked list in which the last node points to first node, as shown in Fig. 15.11.

Fig. 15.11

15.10 DOUBLY LINKED LISTS

A **double linked list**, also called as **two-way linked list**, consists of data as well as links to the next item, as well as the previous item. Figure 15.12 shows a double linked list.

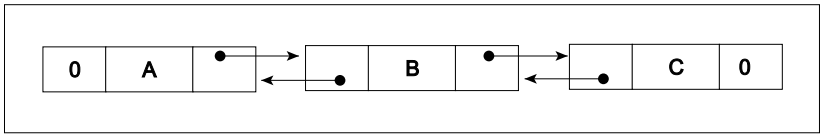


Fig. 15.12 *Double linked list*

As shown in the figure, each item has two links or pointers: one link points to the next item in the list, whereas the second link points back to the previous item in the list. Of course, the very first item in the list (shown here as A) has nothing to point *back to*, it being the very first item in the list. Hence, its *previous pointer* contains a dummy value of 0, which means *pointing to nothing*. Similarly, the last item in the list (shown here as C) has no further items to point to. Therefore, the *next pointer* of the last item in the list also contains a dummy value of 0. All other pointers point to the next or the previous item in the list, as appropriate.

Using two links rather than just one has two key advantages:

1. Note that a single linked list can be read only in one direction. This is because it has pointers only in one direction (which is the forward direction). Therefore, a single linked list cannot be read backwards, starting with the last item in the list. In contrast, a double linked list can be read in *either* direction, as follows:
   (a) We can start with the first item in the list, and read it up to the last item in the list in the forward direction. In our example, it means reading the three items in the order A-B-C.

(b) We can start with the last item in the list, and read it up to the first item in the list in the backward direction. In our example, it means reading the three items in the order C-B-A.

This property of double linked lists can be useful in practical situations. It not only simplifies the management of the list, but also can help in reading the records of a file or a database, if they are implemented as a double linked list.

2. If some failure causes one of the links (either forward or backward) to become invalid, it can be constructed with the help of the other link, which is still intact. Note that this is not possible in the case of a single linked list, because there is only one list, and if it is corrupted, there is no way of recreating it.

15.11 DOUBLY LINKED LIST OPERATIONS

Building a double linked list is similar to building a single linked list. However, here, two links need to be maintained, instead of just one. Therefore, the basic item structure of a double linked list must have sufficient space for maintaining two links. We shall consider the following operations in relation with a double linked list:

1. Inserting an item
2. Deleting an item
3. Displaying all the items in the list
4. Searching for an item in the list

The following simple structure is used for describing these operations:

```
struct linked_list
{
    int number;
    struct linked_list *next;
    struct linked_list *previous;
}
```

Inserting an Item

There are three situations in which a new item can be inserted into a double linked list:

1. Insert a new first item (i.e. push all the existing items in the list forward by one position): This operation is shown in Fig. 15.13. The original list is A-B-C. Since the list begins with A, the *previous pointer* of A is 0, as before.

Note that the item *New* is being inserted at the first position of the list. Therefore, the item that was previously the first item in the list (i.e. A) now points to *New*. Similarly, this being a double linked list, the *previous pointer* of *New* points back to A. The rest of the list in the forward direction is unchanged. That is, A points to B and B points to C. The *next pointer* of C continues to be 0.

2. Insert an item somewhere in the middle of the list: Consider the same linked list consisting of three items, A, B and C. Let us now suppose that a new item is to be added in between A and B. This item, as before, is *New*. Figure15.14 shows this operation. It consists of two steps:

(a) The *next pointer* of A should point to *New*. Similarly, the *previous pointer* of *New* should point to A.

(b) The *next pointer* of *New* should point to B. Similarly, the *previous pointer* of B should point to *New*. However, the links between B and C remain unchanged.

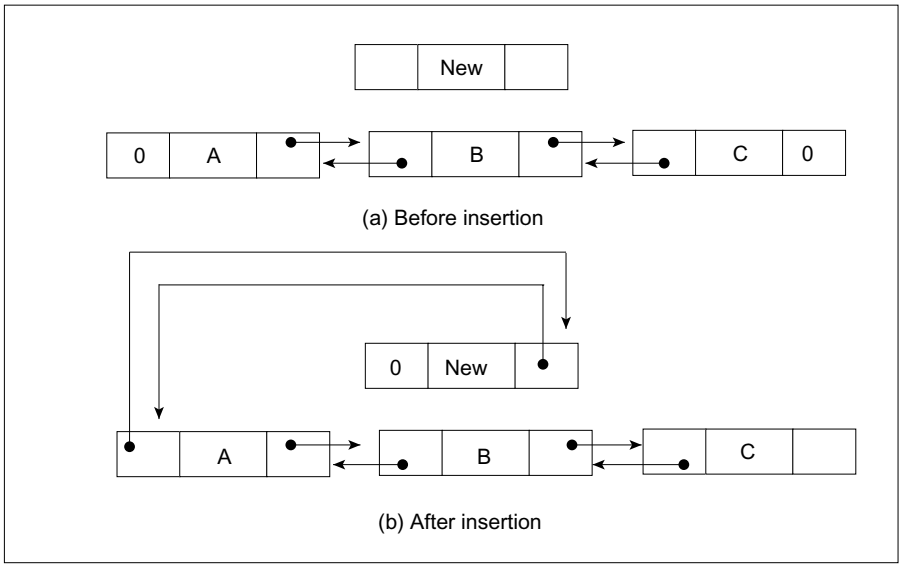


Fig. 15.13 *Insertion of a new first item in a double linked list*

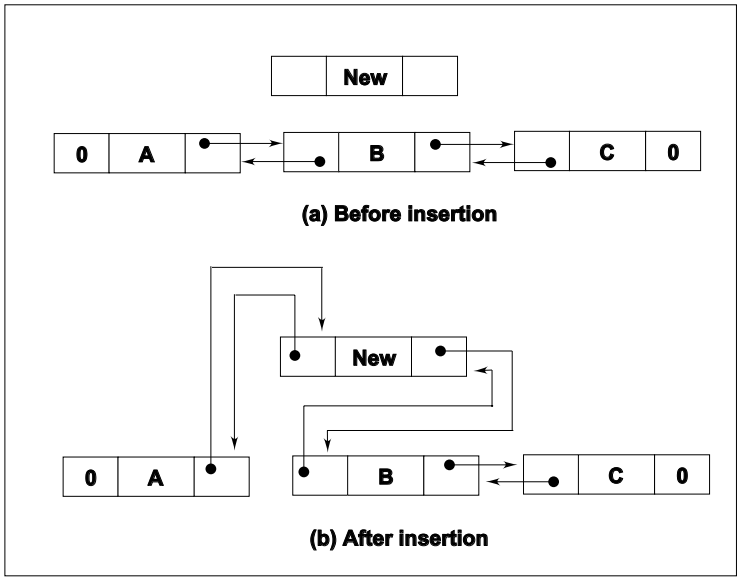


Fig. 15.14 *Insertion of a new item in the middle of a double linked list*

3. Insert an item at the end of the list: In this case, the item *New* will be added at the end of C. Thus, C's *next pointer* will not be 0 now. Instead, it will point to *New*. Similarly, the *previous pointer* of *New* will point to C. Finally, the *next pointer* of *New* will become 0. This is shown in Fig. 15.15.

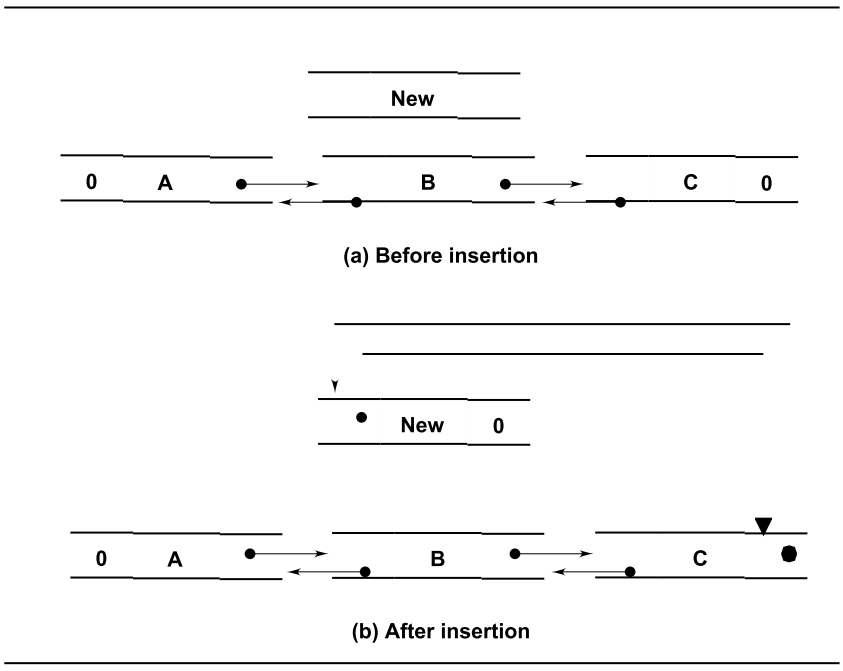


Fig. 15.15 *Insertion of a new item to the end of a double linked list*

Deleting an item

The deletion of an item also has three possible cases, very similar to those of the addition of an item to a double linked list. The item to be deleted can be the very first one, somewhere in the middle, or at the end.

1. Delete the first item in the list: This process is quite simple. Suppose our original list is A-B-C. To delete the first item (i.e. A), we simply need to set the *next pointer* of A to 0, and the *previous pointer* of B also to 0. Note that A being the first item in the list, its *previous pointer* is already 0. After this, B automatically becomes the first item in the list. This is shown in Fig. 15.28.

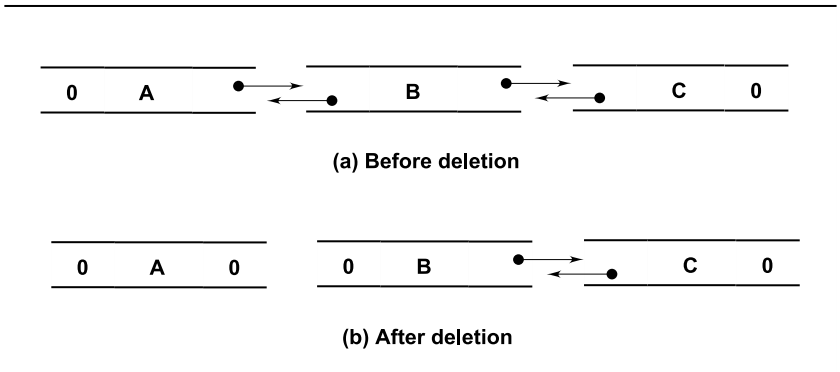


Fig. 15.16 *Deletion of the first item in a double linked list*

2. Delete an item somewhere in the middle of a list: The deletion of an item somewhere in the middle of a list involves the adjustment of the previous and the next pointers of the two items that are around the item to be deleted. Here, from the list A-B-C, the item B is being deleted. Consequently, the following changes are needed:

(a) The *next pointer* of A, which currently points to B, should now point to C.
(b) Similarly, the *previous pointer* of B was pointing to A. Instead, now, the *previous pointer* of C should point to A.

To do this, the *previous pointer* and the *next pointer* of B should be set to 0. This automatically results in the deletion of the item B from the list. This is shown in Fig.15.17.

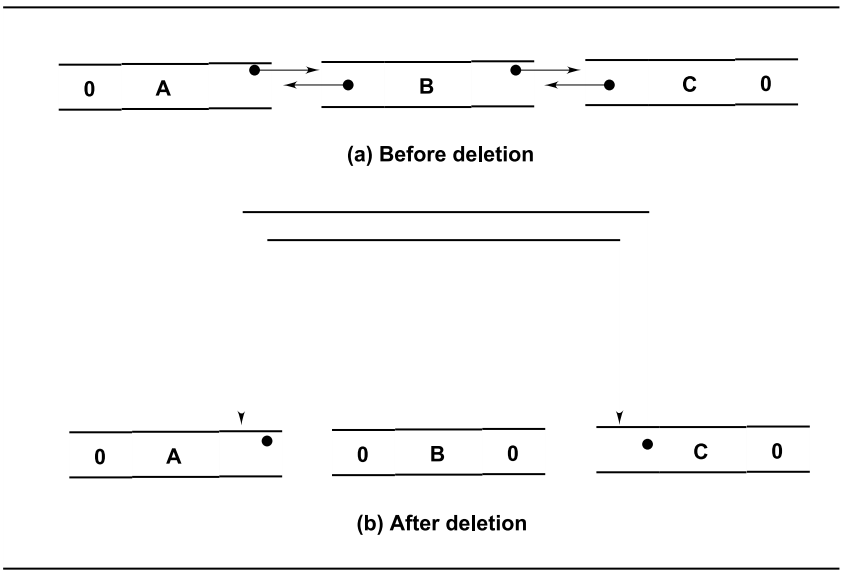


Fig. 15.17 *Deletion of an item from the middle of a double linked list*

Note that the deletion of items in a list is *logical*, and not *physical*. That is, the item to be deleted (i.e. B in this case) is not physically removed, but instead, the mere adjustments of pointers of A and C causes the removal of B automatically from the list.

3. Delete the last item in the list: The deletion of the last item in the list (i.e. C in our example) is the simplest. To achieve this, B's *next pointer*, which normally points to C, should now be set to 0. This achieves two objectives: B becomes the last item in the list, as well as C is automatically removed from the list. Similarly, C's *previous pointer* should be set to 0. This operation is shown in Fig. 15.18.

We shall illustrate the remaining two operations (i.e. displaying all the items in the list and searching for a specific item) with the help of a program. As a value addition, we will automatically sort the list while inserting a new item in the list. That is, a new item in the list is automatically added in its right sorted position.

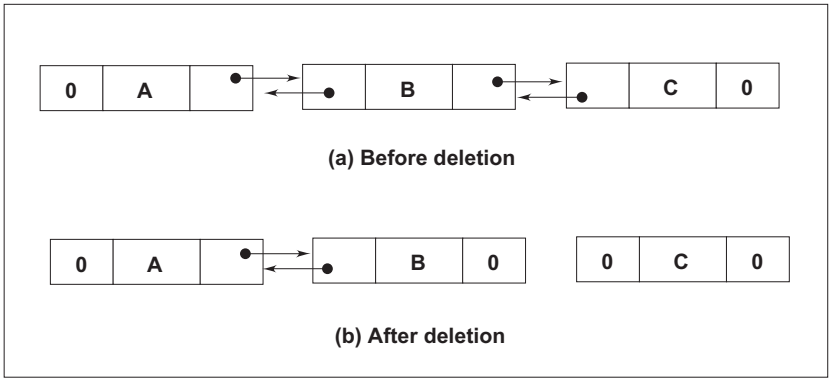


Fig. 15.18 *Deletion of the last item of a double linked list*

The program using double linked list is shown in Fig. 15.19.

```
/* This is a simple program that illustrates the use of a double linked list */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Global linked list variables */
struct linked_list
{
    int number;
    struct linked_list *next;   /* pointer to the next item */
    struct linked_list *previous; /* pointer to the previous item */
} list_entry;

struct linked_list *first;   /* pointer to the first entry in the list */
struct linked_list *last;    /* pointer to the last entry in the list */

/* Function declarations */
char user_choice (void);
void insert (void);
void item_insert (struct linked_list *i, struct linked_list **first, struct linked_list **last);
void item_delete (struct linked_list **first, struct linked_list **last);
struct linked_list* find (int n);
void list (void);
void search (void);
```

Fig. 15.19 *Double linked list program—Part I*

```
/* start of the program */
void main (void)
{
    first = last = NULL;   /* initialize first and last pointers */

    while (1)
```

```
    {
        switch (user_choice ())
        {
            case 1: insert ();
                    break;

            case 2: item_delete (&first, &last);
                    break;

            case 3: list ();
                    break;

            case 4: search ();
                    break;

            case 5: return;
        }
    }
}

/* User selects a particular operation to be performed */
char user_choice (void)
{
    char c;

    /* display the menu of available choices */
    printf ("1. Insert an item\n");
    printf ("2. Delete an item\n");
    printf ("3. Print all items\n");
    printf ("4. Search for an item\n");
    printf ("5. Exit\n");

    do
    {
        printf ("Enter your choice: ");
        c = getchar ();
    } while (c < '1' || c > '5');

    return c;
}
```

Fig. 15.19 *Double linked list program—Part II*

```
/* Accept and store an item at an appropriate position in the list */
void insert (void)
{
    struct linked_list *info;
    int n;   /* Used to accept user input */

    while (1)
    {
        info = (struct linked_list *) malloc (sizeof (list_entry));
        if (!info)
```

```
        {
            printf ("Insufficient memory");
            return ;
        }

        /* request user for the next number, 9999 to stop entry */
        printf ("Enter the number to be stored, 9999 to quit: ");
        scanf ("%d", &n);

        if (n == 9999)
            break;
        else
        {
            /* otherwise, add the number to the info structure */
            info->number = n;
            item_insert (info, &first, &last);
        }
    }
}
```

Fig. 15.19 *Double linked list program—Part III*

```
/* Insert an item into the list */
void item_insert (
    struct linked_list *i, /* new item to be added */
    struct linked_list **first, /* first element in the list */
    struct linked_list **last /* last element in the list */
)
{
    struct linked_list *old, *p;

    /* if this is the very first item to be inserted into the list */
    if (*last == NULL)
    {
        i–>next = NULL;
        i–>previous = NULL;
        *last = i;
        *first = i;
        return;
    }

    /* Begin at the start of the list */
    p = *first;
    old = NULL;

    while (p)   /* while there are more items in the list */
    {
        /* if number in the list is less than the one
          entered by the user */
        if (p–>number < i–>number)
        {
                old = p;
                p = p–>next;
```

```
        }

        else
        {
            /* if p is not the first item in the list, insert i somewhere middle in the list  */
            if (p–>previous)
            {
                p–>previous–>next = i;
                i–>next = p;
                i–>previous = p–>previous;
                p–>previous = i;
                return;
            }

            /* i would be the new first item */
            i–>next = p;
            i–>previous = NULL;
            i–>previous = i;
            *first = i;
            return;
        }
    }

    /* i should be the last item */
    old->next = i;
    i->next = NULL;
    i->previous = old;
    *last = i;
}
```

Fig. 15.19 *Double linked list program—Part IV*

```
/* Delete an item from the list */
void item_delete (struct linked_list **first, struct linked_list **last)
{
    struct linked_list *info;
    int s;

    /* accept the number that the user wants to delete */
    printf (“Enter number to be deleted: ”);
    scanf (“%d”, &s);

    /* find that number in the list */
    info = find (s);

    /* deletion process only if the item is available in the list */
    if (info)
    {
        /* if the first item of the list is to be deleted */
        if (*first == info)
        {
            *first = info->next;
```

```
            if (*first)
                (*first)–>previous = NULL;
            else
                *last = NULL;
        }
        else    /* item to be deleted is somewhere in middle */
        {
            info–>previous–>next = info–>next;

            if (info != *last)
                info–>next–>previous = info–>previous;
            else
                *last = info–>previous;
        }
        /* free up the allocated space */
        free (info);
    }
}

/* find an item in the list */
struct linked_list* find (int n)
{
    struct linked_list *info = first;

    /* while there are more items in the list */
    while (info)
    {
        /* item to be searched is found */
        if (n == info–>number) return info;

        /* otherwise, simply move to the next item in the list */
        info = info–>next;
    }

    /* if control comes here, it means item was not found */
    printf ("Number entered by you is not in the list\n");
    return NULL;
}
```

Fig. 15.19 *Double linked list program—Part V*

```
/* display all the items in the list */
void list (void)
{
    struct linked_list *info = first;

    /* as long as there are more items in the list */
    while (info)
    {
        /* display the current item in the list */
        printf ("%d\n", info–>number);

        /* move to the next item in the list */
```

```
            info = info–>next;
        }

        printf ("\n\n");
}

/* search for a specific item in the list */
void search (void)
{
        int s;
        struct linked_list *info;

        /* accept the number that the user wants to search */
        printf ("Enter number to be searched: ");
        scanf ("%d", &s);

        /* find that number in the list */
        info = find (s);

        /* display an appropriate result */
        if (info)
            printf ("Item found in the list\n");
        else
            printf ("Item not found in the list\n");
}
```

Fig. 15.19 *Double linked list program—Part VI*

15.12 DOUBLY LINKED CIRCULAR LISTS

In a circular or singly linked list, if we want to add data items in the middle of the list, we have to search the entire list. For example, consider a list shown in Fig.15.20. To add a new node Y before a node Z:

We can change the pointer field of Y to point to Z. But we do not know the address of the node X that precedes the node Z.

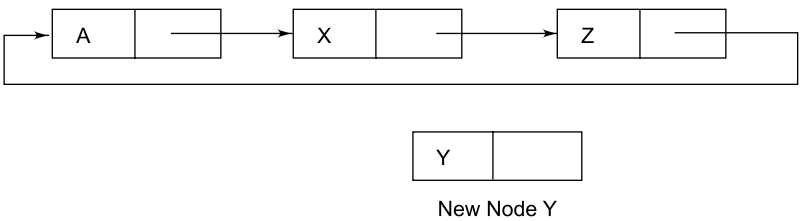


Fig. 15.20

To find the address of the node X, we have to search the entire list, which takes much time. To avoid this kind of inefficient process, a doubly linked circular list is introduced in which each node has two pointers such as a forward link and a backward link. The forward link points to the next node in the list and the backward link points to the previous or preceding node, as shown in Fig. 15.21.

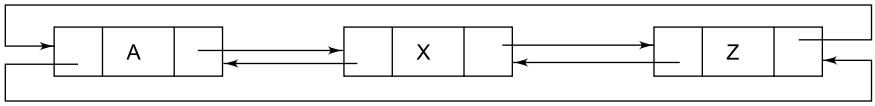


Fig. 15.21

In a doubly linked circular list, the backward link of the last node points to the first node and the forward link of the first node points to the last node. Unlike circular linked lists, we can easily insert a node in the middle of a doubly linked circular list, because it is not necessary to search the entire list to find the address of the preceding node. Fig. 15.22 shows how the new node Y is inserted before the node, X in the doubly linked circular list:

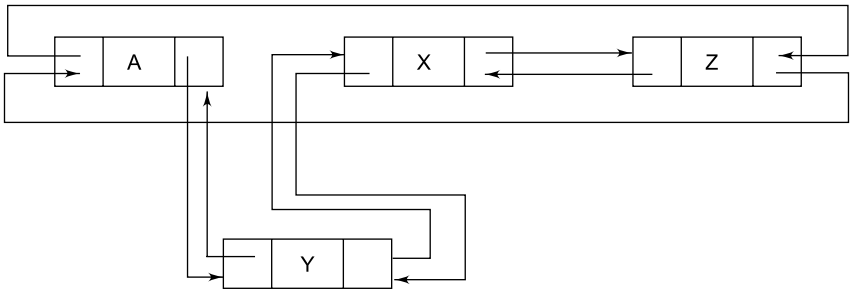


Fig. 15.22

To insert a new node Y, the forward link of the node A now points to the node Y and the backward link of the node, Y points to node A. Similarly the forward link of new node Y points to the node X and the backward link of the node, X points to the node, Y.
Fig. 15.23 illustrates the deletion of a node X from the doubly linked circular list:

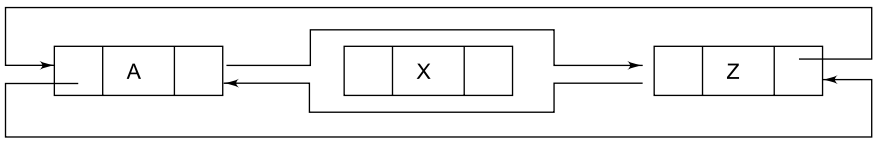


Fig. 15.23

In the above figure, the forward link points to the node Z and the backward link of Z points to node A so that the node X is automatically deleted from the list.

15.13 STACKS AND QUEUES USING LINKED LIST

In the previous chapter, we have seen the implementation of stacks and queues using arrays. In this section, we will discuss how stacks and queues are implemented using linked lists. Unlike arrays, when using linked lists, the memory is allocated dynamically when new data is inserted into the stacks and queues.

Stacks using Linked List

To implement a stack using linked list, we need to define a node, which in turn consists of data and a pointer to the next node, as mentioned in the previous chapter. The advantage of representing stacks using linked lists is that we can decide which end should be the top of a stack. In our example, we will select the front end as the top of the stack in which we can add and remove data. Consider a list, which consists of three data items, as shown in Fig. 15.24.

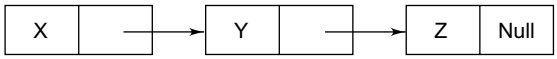


Fig. 15.24

Fig. 15.25 illustrates the push operation in the stack using linked list. This operation adds a new data item, named, A.

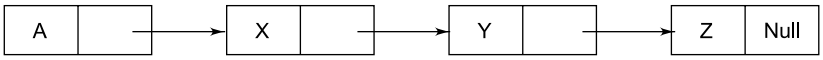


Fig. 15.25

When A is popped from the stack, then the stack will appear, as shown in Fig. 15.26.

Fig. 15.26

The program for implementing stack using linked list is shown in Fig. 15.27.

The linked list approach for a stack is as follows.

```
/* STACK IMPLEMENTATION USING LINKED LIST */

# include<stdio.h>
# include<malloc.h>

struct link
{
    int info;
    struct link *next;
} *start;


void display(struct link *);
struct link *push(struct link *);
struct link *pop(struct link *);
int main_menu();
```

```
void display(struct link *rec)
{
    while(rec != NULL)
    {
        printf(" %d ",rec–>info);
        rec = rec–>next;
    }
}

struct link * push(struct link *rec)
{
    struct link *new_rec;
    printf("\n Input the new value for next location of the stack:");

    new_rec = (struct link *)malloc(sizeof(struct link));
    scanf("%d", &new_rec->info);
    new_rec->next = rec;
    rec = new_rec;
    return(rec);
}

struct  link * pop(struct link *rec)
{
    struct link *temp;

    if(rec == NULL)
    {
        printf("\n Stack is empty");
    }
    else
    {
        temp = rec–>next;
        free(rec);
        rec = temp;
        printf("\n After pop operation the stack is as follows:\n");
        display(rec);
        if(rec  == NULL)
            printf("\n Stack is empty");
    }
    return(rec);
}

int main_menu ()
{
    int choice;
    do
    {
```

```
        printf("\n 1. Push ");
        printf("\n 2. Pop");
        printf("\n 3. Quit");
        printf("\n Input your choice :");
        scanf("%d", &choice);
        if(choice <1 || choice >3)
            printf("\n Incorrect choice-> Please retry");
    } while(choice <1 || choice >3);

    return(choice);
}

/* main function */

void main()
{
    struct link *start ;
    int choice;
    start = NULL;
    do
    {
        choice = main_menu();
        switch(choice)
        {
        case 1:
            start = push(start);
            printf("\n After push operation stack is as follows:\n");
            display(start);
            break;
        case 2:
            start = pop(start);
            break;
        default :
            printf("\n End of program");
        }
    } while(choice != 3);
}
```

Fig. 15.27

Queues using Linked List

Like stacks, when implementing a queue using linked list, a node must be defined. As we know, the front end of a queue is used to remove data and the rear end is used to add data. The advantage of representing a queue using linked list is that it allows us to select any end to add and remove data items.

Removing the first node in a linked list is much easier than that of the last node. This is because, to remove the last node, we need to locate the previous node, which takes much time. So we can select the end of the linked list as rear and beginning of the list as front. Consider a list, which consists of three data items, as shown in Fig. 15.28.

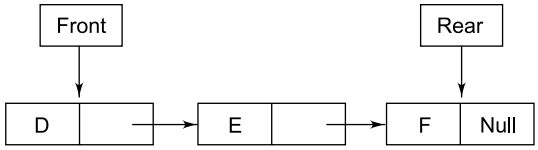


Fig. 15.28

When you add data, the new data will be added at the rear end of the list, as shown in Fig 15.29.

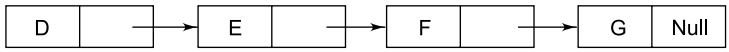


Fig. 15.29

Deleting the data in the queue is similar to that of the stack. The deletion of the queue is done at the front end, as shown in Fig. 15.30.

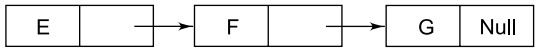


Fig. 15.30

CASE STUDIES

1. Insertion in a Sorted List

The task of inserting a value into the current location in a sorted linked list involves two operations:

(1) Finding the node before which the new node has to be inserted. We call this node as 'Key node'.
(2) Creating a new node with the value to be inserted and inserting the new node by manipulating pointers appropriately.

In order to illustrate the process of insertion, we use a sorted linked list created by the **create** function discussed in Example 15.3. Figure 15.31 shows a complete program that creates a list (using sorted input data) and then inserts a given value into the correct place using function **insert**.

Program

```
/*************************************************************************************** /
/*                INSERTING A NUMBER IN A SORTED LIST                                  */
/*************************************************************************************** /


#include <stdio.h>
#include <stdlib.h>
#define NULL 0

struct linked_list
{
    int  number;
```

```
    struct linked_list *next;
};
typedef struct linked_list node;

main( )
{
    int  n;
    node   *head;
    void    create(node *p);
    node   *insert(node *p, int n);
    void    print(node *p);

    head = (node *)malloc(sizeof(node));

    create(head);
    printf("\n");
    printf("Original list: ");
    print(head);
    print("\n\n");
    printf("Input number to be inserted:");
    scanf("%d", &n);

    head = insert(head,n);

    printf("\n");
    printf("New list:    ");
    print(head);
}

void create(node *list)
{
    printf("Input a number\n");
    printf("(type –999 at end): ");
    scanf("%d", &list–>number);

    if(lis–>number == –999)
    {
        list–>next = NULL;
    }
    else    /*   create next node  */
    {
        list–>next = (node *)(malloc(sizeof(node));
        create(List–>next);
    }
    return;
}

void print(node *list)
{
    if(list–> next != NULL)
    {
        printf("%d– –>", list – > number);

        if(list–>next –>next = = NULL)
            printf ("%d", list–> next –> number);
```

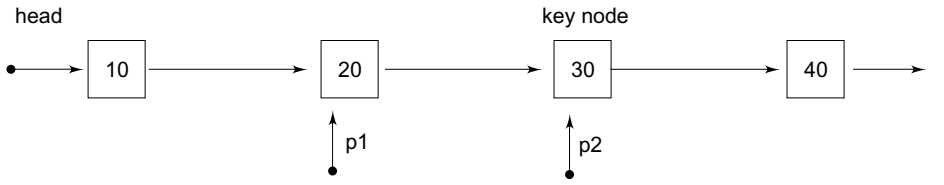
```
        print(list–>next);
    }
    return
}
node *insert(node *head,int x)
{
    node *p1,*p2,*p;
    p1 = NULL;
    p2 = head; /* p2 points to first node */

    for(; p2–>number < x  ;p2 = p2–>next)
{
    p1 = p2;

    if(p2–>next–>next == NULL)
        {
        p2 = p2–>next; /* insertion at end */
        break;
        }
    }
    /* key node found and insert new node */

    p = (node *)malloc(sizeof (node));/* space for
                                        new node */

    p–>number = x; /* place value in the new node */

    p–>next = p2; /*link new node to key node */

    if (p1 == NULL)
        head = p; /* new node becomes the first node */
    else
        p1–>next = p; /* new node inserted in middle */

    return (head);
}
```

Output

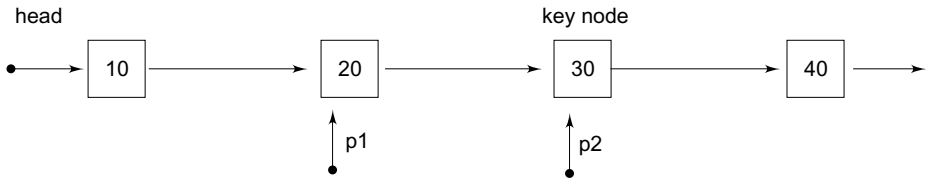
```
Input a number
(type –999 at end):   10
Input a number
(type –999 at end):   20
Input a number
(type –999 at end):   30
Input a number
(type –000 at end):   40
Input a number
(type –999 at end):   –999

Original list: 10– –>20– –>30– –>40– –>–999

Input number to be inserted: 25

New list: 10– –>20– –>25– –>30– –>40– –>–999
```

Fig. 15.31 *Inserting a number in a sorted linked list*

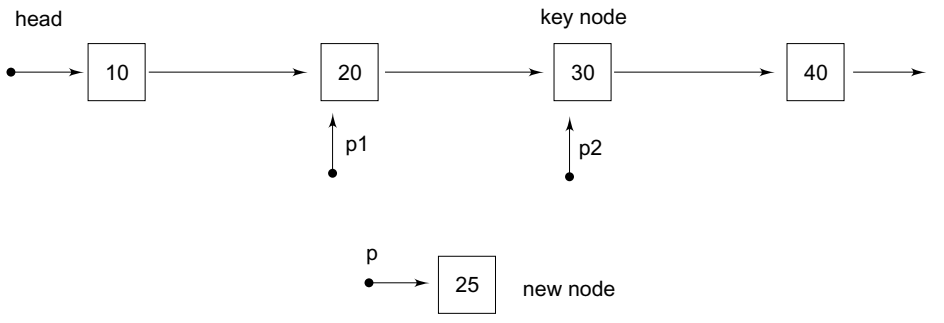
The function takes two arguments, one the value to be inserted and the other a pointer to the linked list. The function uses two pointers. **p1** and **p2** to search the list. Both the pointers are moved down the list with **p1** trailing **p2** by one node while the value **p2** points to is compared with the value to be inserted. The 'Key node' is found when the number **p2** points to is greater (or equal) to the number to be inserted.



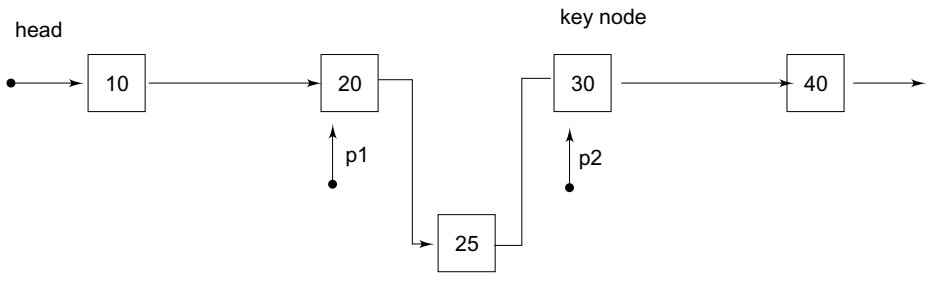
When key node is found



When key node is found



When new node is created



When new node is inserted

Once the key node is found, a new node containing the number is created and inserted between the nodes pointed to by **p1** and **p2**. The figures below illustrate the entire process.

2. Building a Sorted List

The program in Fig. 15.31 can be used to create a sorted list. This is possible by creating 'one item' list using the **create** function and then inserting the remaining items one after another using **insert** function.

A new program that would build a sorted list from a given list of numbers is shown in Fig. 15.32. The **main** function creates a 'base node' using the first number in the list and then calls the function **insert_sort** repeatedly to build the entire sorted list. It uses the same sorting algorithm discussed above but does not use any dummy node. Note that the last item points to NULL.

Program

```
/*****************************************************************************/
/*              CREATION OF SORTED LIST FROM A GIVEN LIST                    */
/*                              OF NUMBERS                                   */
/*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#define NULL 0

struct linked_list
{
    int number;
    struct linked_list *next;
};
typedef struct linked_list node;

main( )
{
    int  n;
    node *head = NULL;
    void print(node *p);
    node *insert_sort(node *p,int n);

    printf("Input the list of numbers.\n");
    printf("At end, type –999.\n");
    scanf("%d",&n);

    while (n != –999)
    {
        if(head == NULL)  /* create 'base' node */
        {
            head = (node *)malloc(sizeof(node));
            head–>number = n;
            head–>next = NULL;
    }
    else                  /* insert next item */
    {

        head = insert_sort(head,n);
```

```
    }

    scanf("%d", &n);
    }
    printf("\n");
    print(head);
    printf("\n");
}

node *insert_sort(node *list,int x)
{
    node *p1,*p2,*p;
    p1 = NULL;
    p2 = list; /* p2 points to first node */

    for (; p2->number < x; p2 = p2->next)
    {
        p1 = p2;

        if(p2->next == NULL)
        {
            p2 = p2->next;                  /* p2 set to NULL*/
            break;              /*  insert new node at end  */
        }
    }

    /* key node found */
    p = (node *)malloc(sizeof(node));/*space for new node*/
    p–>number = x;          /* place value in the new node */
    p–>next = p2;           /* link new node to key node */

    it (p1 == NULL)
        list = p;       /* new node becomes the first node */
    else
        p1–>next = p; /* new node inserted after 1st node */
    return (list);

}
void print(node *list)
{
    if(list == NULL)
        printf("NULL");
    else
    {
        printf("%d– –>",list–>number);
        print(list–>next);
    }

    return;

}
```

Output

Input the list of number.

```
At end, type –999.
80 70 50 40 60 –999

40– –>50– –>60– –>70– –>80– –>NULL

Input the list of number
At end, type -999.
40 70 50 60 80 -999

40– –>50– –>60– –>70– –>80– –>NULL
```

Fig. 15.32 *Creation of sorted list from a given list of numbers*

Review Questions and Exercises

15.1 What is dynamic memory allocation? How does it help in building complex programs?

15.2 What is the principal difference between the functions **malloc** and **calloc**.

15.3 Find errors, if any, in the following memory management statements:
(a) *ptr = (int *)malloc(m, sieof(int));
(b) table = (float *)calloc(100);
(c) node = free(ptr);

15.4 Why a linked list is called a dynamic data structure? What are the advantages of using linked lists over arrays?

15.5 Describe different types of linked lists.

15.6 Identify errors, if any, in the following structure definition statements:

```
struct
{
   char name[30]
   struct *next;
};
typedef struct node;
```

15.7 The following code is defined in a header file *list.h*

```
typedef struct
{
      char     name[15];
      int      age;
      float    weight;
} DATA;

struct linked_list
{
      DATA person;
      struct linked_list *next;
};

typedef    struct linked_list  NODE;
typedef    NODE   *NDPTR;
```

Explain how could we use this header file for writing programs.

15.8 In Example 15.3, we have used **print**() in recursive mode. Rewrite this function using iterative technique in **for** loop.

15.9 Write a menu driven program to create a linked list of a class of students and perform the following operations:
(a) Write out the contents of the list.
(b) Edit the details of a specified student.
(c) Count the number of students above a specified age and weight.
Make use of the header file defined in Exercise 15.7.

15.10 Write recursive and nonrecursive functions for reversing the elements in a linear list. Compare the relative efficiencies of them.

15.11 Write an interactive program to create a linear linked list of customer names and their telephone numbers. The program should be menu driven and include features for adding a new customer and deleting an existing customer.

15.12 Modify the above program so that the list is always maintained in the alphabetical order of customer names.

15.13 Develop a program to combine two sorted lists to produce a third sorted list which contains one occurrence of each of the elements in the original lists.

15.14 Write a program to create a circular linked list so that the input order of data item is maintained. Add functions to carry out the following operations on circular linked list.
(a) Count the number of nodes
(b) Write out contents
(c) Locate and write the contents of a given node

15.15 Write a program to construct an ordered doubly linked list and write out the contents of a specified node.

15.16 Write a function that would traverse a linear singly linked list in reverse and write out the contents in reverse order.

15.17 Given two ordered singly linked lists, write a function that will merge them into a third ordered list.

15.18 Describe double linked list in detail.

15.19 Explain the three situations in which a new item can be inserted into a double linked list.

15.20 Explain the deletion of an item from a double linked list.

15.21 Explain circular lists.

Chapter 16

Binary Trees and Graphs

16.1 BINARY TREES—REPRESENTATION AND TERMINOLOGY

The data structure of type tree comes in many forms. However, the **binary tree** is a bit special, because when they are in the sorted form, they facilitate quick search, insertion and deletion. Each item in a tree has two links: one to a left member, and another to a right member. This concept is shown in Fig. 16.1.

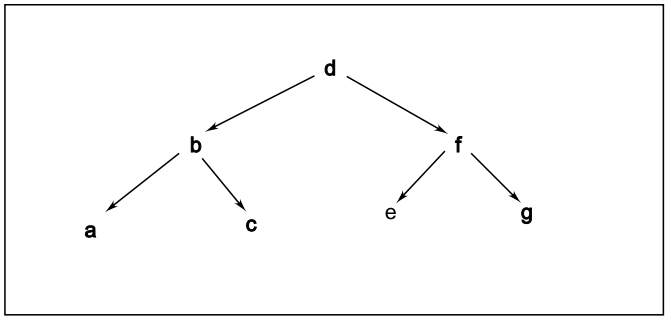


Fig. 16.1 *Binary tree*

In this figure, d is the *root* of the tree. Each data item (a through g) is called as a *node*. Any portion of a tree (e.g. a-b-c) is called as a *sub-tree*. A node that does not have any sub-tree is called as a *terminal node*. For example, a, c, e and g are terminal nodes, whereas b, d and f are not. The *height* of the tree is the number of the vertical node positions. In the sample tree just shown, the height is 3. Accessing a tree is called as the process of *tree traversal*.

Items of a tree can be accessed, inserted or deleted in any order. Interestingly, each sub-tree of a tree is itself a tree. Therefore, a tree is a recursive (i.e. self-repeating) data structure, and recursive programming techniques are popularly used with trees. Of course, tree-related programming need not always be recursive. Non-recursive tree programming is also possible, but is much harder to code and understand, as compared to the recursive version.

16.2 BINARY TREE TRAVERSAL

A tree can be traversed in three major ways:

- **In-order tree traversal:** In this type, the left sub-tree is traversed first, followed by the root, and finally by the right sub-tree. Thus, the in-order traversal of the sample tree shown earlier would yield the sequence a-b-c-d-e-f-g.
- **Pre-order tree traversal:** In this traversal, the root is visited first, followed by the left sub-tree, followed by the right sub-tree. Thus, the pre-order traversal of the sample tree shown earlier would yield the sequence d-b-a-c-f-e-g.
- **Post-order tree traversal:** In this type, the left sub-tree is traversed first, followed by the right sub-tree, and finally followed by the root. Thus, the post-order traversal of the sample tree shown earlier would yield the sequence a-c-b-e-g-f-d.

Although a tree need not always be in a sorted form, generally, all the practical applications require the use of a sorted tree. We shall also work with sorted trees. Also, we shall consider only in-order trees. Note that the other two tree types are simply variations of this type.

The actual view of a tree that has links between the various items is shown in the form of another tree in Fig. 16.2. This view needs to be kept in mind while working with trees in computer programs. Note how the *next* or *previous* pointers have a value 0 for the nodes that do not point to other nodes in the tree.

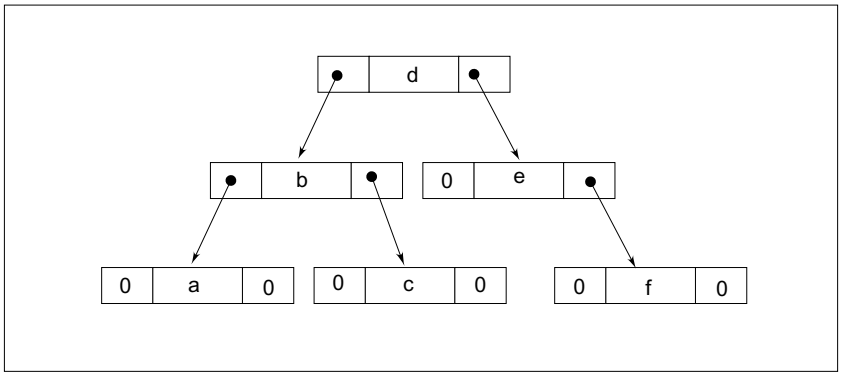


Fig. 16.2 *Another view of a tree*

We can represent each node in the tree in the form of a structure, as follows:

```
struct tree
{
   int number;        /* data to be stored */
   struct tree *left;  /* pointer to the left node */
   struct tree *right; /* pointer to the left node */
};
```

The following function, insert_node builds a sorted binary tree.

```
struct tree *insert_node (struct tree *root,   /* root of the tree */
struct tree *r,  /* node to be inserted */
int info   /* value of the node that is being inserted */)
{
   /* this is non-recursive portion of the code */

   /* if node to be inserted does not exist, create it */
   if (!r)
   {
      r = (struct tree *) malloc (sizeof (struct tree));

      if (!r)
      {
         printf ("Insufficient memory\n.");
         return;
      }

      /* set the left-right pointers of r to null, and set
        its information to that received from the calling
        function
      */
      r->left = NULL;
      r->right = NULL;
      r->info = info;

      /* root does not exist, so this is the first entry */
      if (!root)
         return r; /* r becomes the root now */

      /* if the value of the node to be inserted is less than
        that of the root, insert it before the root */
      if (info < root->info)
         root->left = r;
      else
      /* if the value of the node to be inserted is greater than
        that of the root, insert it after the root */
         root->right = r;

      return r;
   }
```

```
        /* these are recursive calls */

        /* insert the current node in the appropriate position, depending
           on its value */
        if (info < r–>info)
            insert_node (r, r–>left, info);
        else
            insert_node (r, r–>right, info);
    }
```

Once the tree is created, it can be traversed in either of the tree ways discussed earlier (in-order, pre-order and post-order). The in-order traversal looks as follows.

```
void in_order (struct tree *root)
{
    /* if root does not exist, it means this is an empty tree. So, do not display anything */
    if (!root)
        return;

    in_order (root–>left);
    print ("%d ", root–>number);
    in_order (root–>right);
}
```

Note that this also makes use of recursion. How does this program work? To understand this, refer to a sample tree shown in Fig. 16.3.

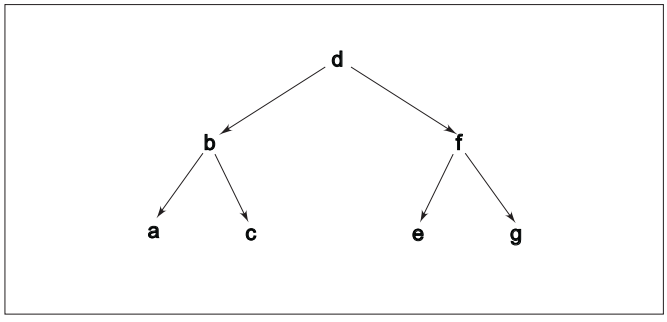


Fig 16.3 *Sample tree*

First, the program checks if the root exists. In this case, the root (d) exists. So, the program proceeds.

It now calls itself, passing the left node of itself (i.e. b). Since this is a recursive call, it would enter the in_order function again, and call itself once again, passing the new left node of itself (i.e. a). This also being a recursive call, will result into a call to itself passing the new left node of itself (i.e. null, this time). This call to itself will now cause the *if* condition (to check if the root is null) will become true. Therefore, the recursion ends here. The functio8ns are now executed in the reverse order (i.e. with values a and b). This will print a and b.

```
in_order (root->right);
```

This will cause a call to itself with the value c now, so it will print c.
At this stage, the traversal from d to the left sub-tree is over. So, the function will print d.

After this, the right sub-tree of the root (i.e. e-f-g) gets printed similar to the way the left sub-tree was printed earlier, using recursive calls. We will not explain it, as it is exactly similar to the way a-b-c were printed.

The pre-order traversal would look as follows.

```
void pre_order (struct tree *root)
{
    if (!root)
        return;

    printf ("%d", root–>number);
    pre_order (root–>left);
    printf ("%d", root–>right);
}
```

Finally, the function for a post-order traversal would be as follows.

```
void post_order (struct tree *root)
{
    if (!root)
        return;

    post_order (root->left);
    post_order (root->right);
    printf ("%d", root->number);
}
```

Depth-first and Breadth-first traversals of a binary tree

In the **depth-first traversal** technique, we traverse a single path of the tree as far as we can. That is, we visit a node with no successors or a node, all of whose successors have already been visited. We then resume at the last node on the path just traversed that has an unvisited successor, and begin traversing a new path starting with that node. Spanning trees created with depth-first traversal are usually very deep.

In the **breadth-first traversal** mechanism, we visit all successors of a visited node before we visit any successors of any of those successors. This is in contrast to the depth-first traversal, wherein we visit the successor of a visited node before visiting any other nodes at its level (i.e. its brothers). Breadth-first trees are usually not deep—instead, they are short and wide.

The following program finds the depth of a binary tree:

```
/*  Finding the depth of Binary Tree */
```

```
# include<stdio.h>
# include<malloc.h>

struct NODE
{
    char Info;
    struct NODE *Left_Child;
    struct NODE *Right_Child;
};

int depth = 0;

/* Function declarations */
void Output (struct NODE *, int );
int Depth (struct NODE *, int );
struct NODE *Create_Tree (char , struct NODE *);


void Output(struct NODE *T, int Level)
{
    int i;
    if (T)
    {
        Output(T–>Right_Child, Level+1);
        printf("\n");
        for (i = 0; i < Level; i++)
            printf("  ");
        printf("%c", T–>Info);
        Output(T–>Left_Child, Level+1);
    }
}

/* Find depth of the tree */

int Depth (struct NODE *Node, int Level)
{
    if (Node != NULL)
    {
        if (Level > depth)
            depth = Level;
        Depth (Node–>Left_Child, Level + 1);
        Depth (Node–>Right_Child, Level + 1);
    }
    return (depth);
}
/*  Create binary tree */
```

```
struct NODE *  Create_Tree (char Info, struct NODE *Node)
{

    if (Node == NULL)
    {
        Node = (struct NODE *) malloc(sizeof(struct NODE));
        Node–>Info = Info;
        Node–>Left_Child = NULL;
        Node–>Right_Child = NULL;
        return (Node);
    }

    /* Test for the left child */

    if (Info < Node–>Info)

        Node->Left_Child = Create_Tree (Info, Node->Left_Child);

    else

        /*  Test for the right child */

        if (Info > Node–>Info)

            Node–>Right_Child = Create_Tree (Info, Node–>Right_Child);
    return(Node);
}

/* Function main */

void main()
{
    int Number = 0;
    char Info ;
    char choice;
    int depth;
    struct NODE *T = (struct NODE *) malloc(sizeof(struct NODE));
    T = NULL;
    printf("\n Please enter your choice, 'b' to break:");
    choice = getchar();

    while(choice != 'b')
    {
        fflush(stdin);
        printf("\n Input information of the node: ");
        scanf("%c", &Info);
        T = Create_Tree(Info, T);
        Number++;
```

```
        fflush(stdin);
        printf("\n Input choice 'b' to break:");
        choice = getchar();
    }
    printf("\n Number of elements in the list is  %d", Number);
    printf("\n Tree is \n");
    Output(T, 1);

    depth = Depth(T, 0);
    printf("\n Depth of the above tree is:  %d", depth);
}
```

16.3 GRAPHS

A **graph** is a set of **nodes** (also called as **vertices**) and a set of **arcs** (also called as **edges**). A sample graph is shown in Fig. 16.4.

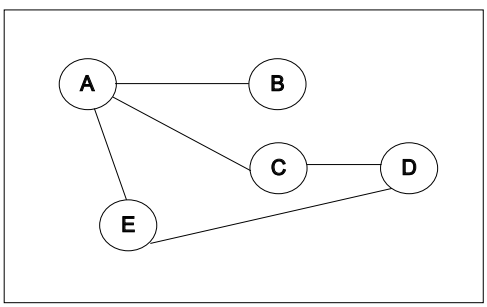


Fig. 16.4 *Graph*

The set of nodes or vertices in the above graph is denoted as: {A, B, C, D, E}. The set of arcs in this graph is {(A, B), (A, C), (A, D), (A, E), (C, D), (E, D)}.

A graph is **connected** if there is a path between any two nodes of the graph, i.e. if we can traverse from one node to any other node in the graph. Otherwise, the graph is **non-connected**. Figure 16.5 shows a connected and a non-connected graph..

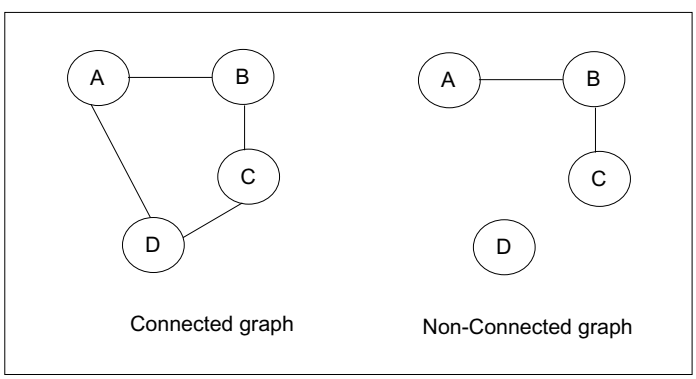


Fig. 16.5 *Connected and non-connected graph*

16.4 GRAPH REPRESENTATION IN C

A graph can be represented as an array or as a linked list in C. Consider the following graph (Fig. 16.6):

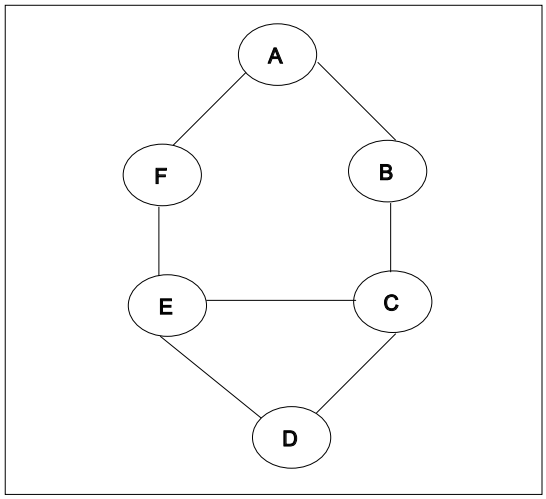


Fig. 16.6 *Sample graph*

Let us have a convention by which, we write:
1 if there is an edge between any two nodes, and
0 if there is no edge between any two nodes
Thus, we will have the following representation:

	A	B	C	D	E	F
A	0	1	0	0	0	1
B	1	0	1	0	0	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	1	0	1	1	0	1
F	1	0	0	0	1	0

Fig. 16.7 *Array representation of a graph*

As we can see, a '1' in the first row between A-B and A-F indicates that there is an edge between A-B and A-F. All other zeroes indicate that there is no edge between the said nodes (e.g. there is no edge between nodes A-C, indicated by a '0').

A graph can also be represented in the form of a linked list. There are two types of list nodes in such a case, a **header node** (also called as **graph node**), and a **list node.**

- A header node contains the actual information of the node, a pointer to the next node, as well as a pointer to an arc.
- A list node contains a pointer to a node and a pointer to the next arc.

These two node types can be represented as shown in Fig. 16.8.

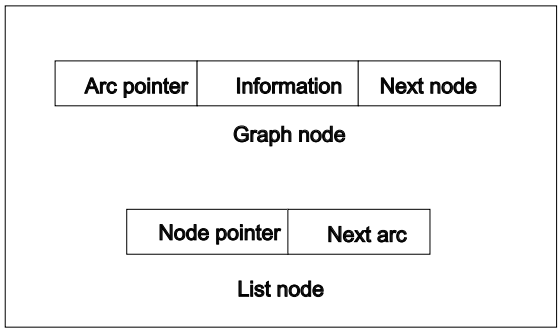


Fig. 16.8 *Types of nodes in a graph*

In C terms, we call Arc pointer as *arcptr*, and Node pointer as *nodeptr*. If p points to a list node representing an arc (A, B), then Nodeptr (p) points to the header node representing the graph node B. Next arc (p) points to the list node representing the next arc coming from the graph node A. The sample graph and it corresponding linked list representation is shown in Fig. 16.9.

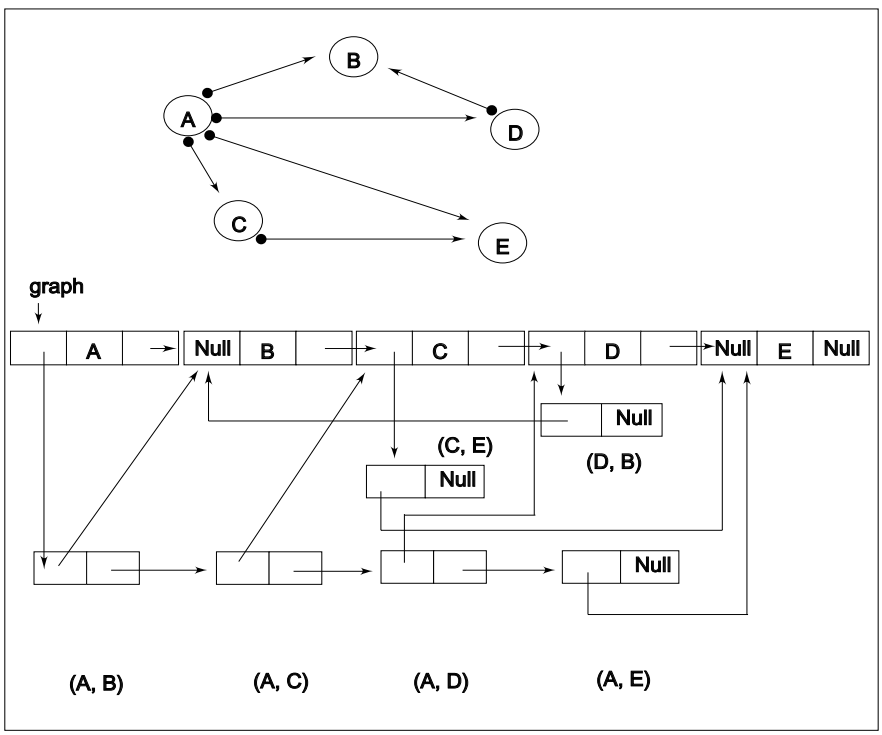


Fig. 16.9 *Sample graph and its linked list representation*

Note that we have shown arrows to indicate the direction of the arcs between the nodes. Thus, C-E is valid here, but E-C is invalid. A graph that contains such directions is called as a **directed graph**. A graph that does not have directions in this manner is called as an **undirected graph**.

Let us take a look at the various C operations that can be associated with this graph in the form of a linked list.

The basic definition of the *node type* is as follows:

```
#define  MAX 1000
struct nodetype
{
    int point;
    int info;
    int next;
};

struct nodetype node[MAX];
```

Joining Two nodes

The following function can be used to create an arc between two nodes. It accepts two pointers and joins them to form an arc.

```
Void join (int x, int y)
{
    int r1, r2;

    /* Search list of arcs originating from node [x] for an arc to node [y] */
    r1 = node [x].point;

    while (r1 >=0 && node[r1].point != y)
    {
        r2 = r1;
        r1 = node[r1].next;
    }

    if (r1 >=0)
        /* node [r1] represents an arc between x and y */
        node[r1].info = 1;

    /* if an arc between x and y does not exist, create it */
    r1.getnode ( );
    node [r1].point = y;
    node [r1].next = –1;
    node [r1].info = 1;

    if (r2 < 0)
        node [x].point = r1;
    else
        node [r2].next = r1;
}
```

The function to delete an arc can be written as follows:

```
void delete (int x, int y)
{
    int r1 = node [x].point;
    int r2 = –1;

    while (r1 >= 0 && node [r] != y)
    {
        r2 = r1;
        r1 = node [r1].next;
    }

    if (r1 >= 0)
    {
        /* r1 points to an arc from node [x] to node [y] */
        if (r2 < 0)
            node [x].point = node [r1].next;
        else
            node[r2].next = node [r1].next;

    free (node [r1]);
    return;
}

/* do nothing if the arc if not found */
}
```

The following function can be used to test if node y is adjacent to node x.

```
int isadjacent (int x, int y)
{
    int r = node [p].next;

    while (r >= 0)
        if (node [r].point == y)
            return 1;     /* found */
        else
            r = node [r].next;
    return 0;/* not found */
}
```

Finally, the following function can be used to find a particular node. It returns a pointer to a header node with information = x (the node being searched), if found, otherwise returns null.

```
int findnode (int graph, int x)
{
    int r = graph;
```

```
    while (r >= 0)
    {
        if (node [r].info == x)
            return r; /* found */
    else
        r = node [r].next;
    }
        return –1;   /* not found */
}
```

16.5 GRAPH TRAVERSAL

The process of traversing all the nodes or vertices on a graph is called graph traversal. There are various methods to traverse a node in the graph. The most popular methods are:

- Depth First Search (DFS)
- Breadth First Search (BFS)

Depth First Search

In the depth first search, we need to select a vertex as an arbitrary vertex to traverse a graph. We will determine all the adjacent vertices to the arbitrary vertex. Then, we need to select any one of the adjacency vertices from the adjacency list. Now all the vertices that are adjacent to the selected vertex are visited, then another vertex is selected and all the vertices adjacent to this vertex are visited, and so on. This process continues until all the vertices in the graph are visited. For example, consider the graph shown in Fig. 16.10.

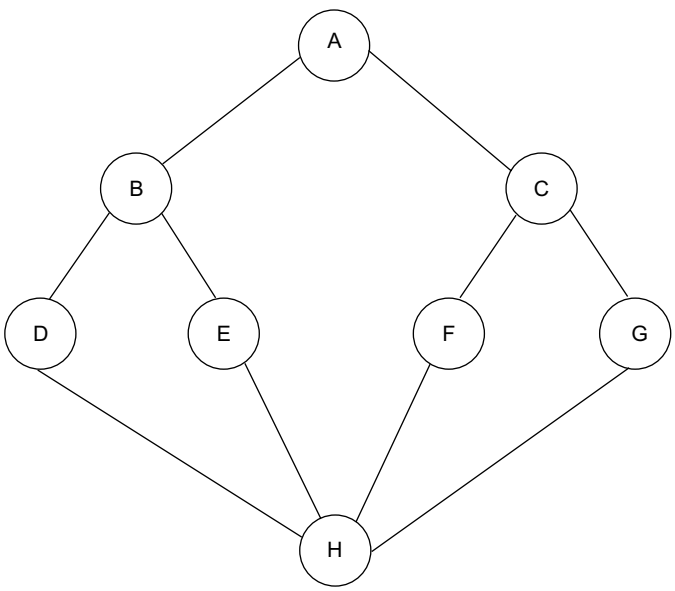


Fig. 16.10

Let us see how the DFS method can be used to traverse all the vertices of the graph shown in the above figure.

- Let us take the arbitrary vertex as A.
- The adjacency vertices for the vertex, A, are B, H and C. Select the adjacency vertex B.
- The adjacency vertices for the vertex, B, are A, D and E. A is already traversed or visited, so select the vertex D.
- The adjacency vertices for the vertex, D, are B and H. B is already traversed, so select the vertex H.
- The adjacency vertices for the vertex, H, are D, E, A, F, and G. D and A are already traversed so select the vertex E.
- The adjacency vertices for the vertex, E, are B and H. Both B and H are already visited so back track to traverse the vertex, H to traverse the unvisited vertices, F and G. Now, select the vertex F.
- The adjacency vertices for the vertex, F, are C and H. H is already traversed so select the vertex, C.
- The adjacency vertices for the vertex, C, are F and G. F is already visited. So traverse the vertex, G.
- All the adjacency vertices of the vertex, G is visited, so back track and determine whether all the adjacency vertices are visited.

The DFS method helps us to traverse the above graph in the sequence A, B, D, H, E, F, C, and G. The following program illustrates the depth-first approach.

```
/* DEPTH FIRST SEARCH TECHNIQUE */

#include<stdio.h>
#define size 50
#define T 1
#define F 0

struct Edge
{
    int terminal;
    struct Edge *next;
};
struct Vertex
{
    int visit;
    int vertex_no;
    char info;
    int path_length;
    struct Edge *Edge_Ptr;
};
void Table(int, int matrix [size][size], struct Vertex vert[size]);
struct Edge *Insert_Vertex (int , struct Edge *);
void DFS ( int , int *dist, struct Vertex vert [size]);
void Input(int, int a [size][size]);
void Output(int, int a [size][size]);

struct Edge * Insert_Vertex (int vertex_no, struct Edge *first)
{
    struct Edge *new1, *current;
```

```
    new1 = (struct Edge *) malloc(sizeof(struct Edge));
    new1–>terminal = vertex_no;
    new1–>next = NULL;
    if (!first)
        return (new1);
    for (current = first; current->next; current = current->next);
    current->next = new1;
    return (first);
}

/* Initializing entries */

void Table(int vertex_num, int matrix [size][size],
struct Vertex vert[size])
{
    int i, j;
    for (i = 0; i < vertex_num; i++)
    {
        vert [i].visit = F;
        vert [i].vertex_no = i +1;
        vert [i].info = 'A'+ i;
        vert [i].path_length = 0;
        vert [i].Edge_Ptr = NULL;
    }

    for (i =0; i < vertex_num ; i++)
        for (j =0; j < vertex_num ; j++)
            if (matrix [i][j] > 0)
                vert [i].Edge_Ptr = Insert_Vertex (j, vert [i].Edge_Ptr);
}

/* Computing path length */
void DFS ( int index, int *dist,
struct Vertex vert [size])
{
    struct Edge *Link;
    vert [index].visit = T;
    vert [index].path_length = *dist;
    *dist += 1;
    for ( Link = vert [index].Edge_Ptr; Link; Link = Link->next)
        if (vert [Link–>terminal].visit == F)
            DFS (Link–>terminal, dist, vert);
}

/* Input function to read adjacency matrix */

void Input(int number, int a [size][size])
{
    int i, j;
```

```
    printf("\n Input the adjacency matrix \n");

    for (i =0; i < number; i++)
    {
        for (j=0; j < number; j ++)
        {
            scanf("%d", &a [i][j]);
        }
        printf("\n");
    }
}

/* Output function */
void Output(int number, int a [size][size])
{
    int i, j;
    printf("\n Adjacency matrix \n");
    for (i = 0; i < number; i++)
    {
        for (j = 0; j < number; j ++)
        {
            printf("   %d", a [i][j]);
        }
        printf("\n");
    }
}

/* Function main */
void  main( )
{
    int i;
    int number, index, dist;
    int a [size][size];
    struct Vertex vert [size];
    struct Edge *List;
    printf("\n Input the number of vertices in the graph: ");
    scanf("%d", &number);
    Input(number, a);
    Output(number, a);

    Table(number, a, vert);
    printf("\n Input the starting vertex 0- %d:", number-1);
    scanf("%d", &index);
    dist = 0;
    DFS (index, &dist, vert);
    printf("\n Path length of the vertex from %c", vert[index].info);
    printf("\n Vertex Length Vertex Connectivity \n ");
    for (i = 0; i < number; i++)
```

```
    {
        printf("\n  %c  %d   ", vert[i].info, vert[i].path_length);
        for (List= vert[i].Edge_Ptr; List; List = List->next)
        {
            printf(" ");
            putchar(List->terminal+'A');
        }
    }
}
```

Breadth First Search

Like DFS, in the breadth first search, an arbitrary vertex is selected. However, we need to traverse all the adjacent vertices of the arbitrary vertex. Here, the unvisited vertices, which are adjacent to the visited vertices, are traversed. This process continues until all the vertices in a graph are visited.

For example, consider the above graph. To traverse the graph in BFS method, select an arbitrary vertex, say, A.

The adjacent vertices for the vertex, A, are B, C, and H. Select the vertex B.

The unvisited vertices for the vertex, B, are D and E.

Then, traverse the vertices, D and E.

Go back to the remaining vertices and visit the unvisited adjacent vertices of A.

Select the vertex, C. The unvisited adjacent vertices of the vertex, C, are F and G.

There are no more unvisited adjacent vertices for the vertices, H, D, E, F and G.

The traversal of BFS method is shown in Fig. 16.11(a)–(d).

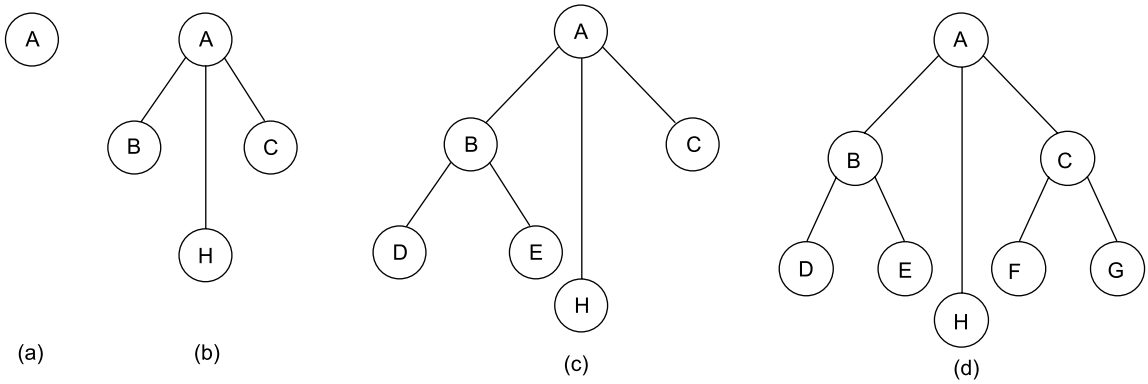


Fig. 16.11

The BFS method helps us to traverse the above graph in the sequence A, B, H, C, D, E, F and G.

The following program illustrates the breadth-first approach.

```
/* BREADTH FIRST SEARCH TECHNIQUE */

#include<stdio.h>
#define size 50
#define T 1
```

```
#define F 0

struct Edge
{
    int terminal;
    struct Edge *next;
};

struct Vertex
{
    int visit;
    int vertex_no;
    char info;
    int path_length;
    struct Edge *Edge_Ptr;
};
struct Q
{
    int info;
    struct Q *next;
};

void Table(int, int matrix [size][size], struct Vertex vert[size]);
struct Edge *Insert_Vertex (int, struct Edge *);
void BFS (int, struct Vertex vert [size]);
void Input(int, int mat [size][size]);
void Output(int number, int mat [size][size]);
struct Q *Insert_Queue(int vertex_no, struct Q *first);
struct Q *Delete_Queue(int *vertex_no, struct Q *first);

/* Insert vertex into connectivity list */

struct Edge * Insert_Vertex (int vertex_no, struct Edge *first)
{
    struct Edge *new1, *current;
    new1 = (struct Edge *) malloc(sizeof(struct Edge));
    new1->terminal = vertex_no;
    new1->next = NULL;
    if (!first)
        return (new1);
    for (current = first; current–>next; current = current–>next);
    current–>next = new1;
    return (first);
}

/* Insert vertices into queue */

struct Q * Insert_Queue(int vertex_no, struct Q *first)
```

```
{
    struct Q *new1, *current;
    new1 =(struct Q *) malloc(sizeof(struct Q));
    new1–>info = vertex_no;
    new1–>next = NULL;
    if (!first)
        return (new1);
    for (current = first; current–>next; current = current->next);
    current–>next = new1;
    return (first);
}

struct  Q * Delete_Queue(int *vertex_no, struct Q *first)
{
    struct Q *previous;
    if (!first)
        return (NULL);
    *vertex_no = first–>info;
    previous = first;
    first = first–>next;
    free(previous);
    return (first);
}

/* Initializing entries */

void Table(int vertex_num, int matrix [size][size],
struct   Vertex vert[size])
{
    int i, j;
    for (i = 0; i < vertex_num; i++)
    {
        vert [i].visit = F;
        vert [i].vertex_no = i+1;
        vert [i].info = 'A'+ i;
        vert [i].path_length = 0;
        vert [i].Edge_Ptr = NULL;
    }

    for (i =0; i < vertex_num ; i++)
        for (j =0; j < vertex_num ; j++)
            if (matrix [i][j] > 0 )
                vert [i].Edge_Ptr = Insert_Vertex (j, vert [i].Edge_Ptr);
}

/* Computing path length */

void BFS ( int index, struct Vertex vert [size])
```

```
{
    struct Q  *queue = NULL;
    struct Edge *Link;
    vert [index].visit = T;
    queue = Insert_Queue(index, queue);
    while(queue)
    {
        queue = Delete_Queue(&index, queue);
        for ( Link = vert [index].Edge_Ptr; Link; Link = Link->next)
        {
            if (vert [Link–>terminal].visit == F)
            {
                vert[Link–>terminal].visit = T;
                vert[Link–>terminal].path_length=vert[index].path_length+1;
                queue = Insert_Queue(Link->terminal, queue);
            }
        }
    }
}

/* Input function to read adjacency matrix */

void Input(int number, int mat [size][size])
{
    int i, j;
    printf("\n Input the adjacency matrix \n");
    for (i =0; i < number; i++)
    {
        for (j=0; j < number; j ++)
        {
            scanf("%d", &mat [i][j]);
        }
        printf("\n");
    }
}

/* Output function to display adjacency matrix */

void Output(int number, int mat [size][size])
{
    int i, j;
    printf("\n Adjacency matrix \n");
    for (i =0; i < number; i++)
    {
        for (j=0; j < number; j ++)
        {
            printf("   %d", mat [i][j]);
        }
```

```
            printf("\n");
        }
    }

    /* Function main */
    void  main()
    {
        int i, number, index;
        int mat [size][size];
        struct Vertex vert [size];
        struct Edge *List;
        printf("\n Input the number of vertices in the graph: ");
        scanf("%d", &number);
        Input(number, mat);
        Output(number, mat);
        Table(number, mat, vert);
        printf("\n Input the starting vertex 0- %d :",number-1);
        scanf("%d", &index);
        BFS (index, vert);
        printf("\n Path length of the vertex from %c", vert[index].info);
        printf("\n Vertex Length Vertex Connectivity \n ");
        for (i = 0; i < number; i++)
        {
            printf("\n   %c    %d   ",vert[i].info, vert[i].path_length);
            for (List= vert[i].Edge_Ptr; List; List = List->next)
            {
                printf(" ");
                putchar(List–>terminal+'A');
            }
        }
    }
```

Shortest path problem

As we have seen, a graph is traversed through its edges. Let us consider a situation where we need to determine the minimum distance between two places when traveling from one place to another. Here, the distance between the two locations is assigned as weights to the edges.

For example, consider a situation where we want to travel from Chennai to New York. We can travel from Chennai to New York through London at a cost of $1500 and through Singapore at a cost of $2500. To travel from Chennai to New York we usually want to travel through London, because the cost of traveling is less. Like this, there are many applications in which we require to find the shortest path, i.e., the path having the minimum weight between the two vertices. In our traveling example, the vertices refer to the places Chennai and New York and weight refers to the cost of traveling.

Consider a graph in which S is the source vertex and E is the destination vertex.

A path in a graph refers to the sequence of vertices from the source vertex to the destination vertex such that we follow the edges between two successive vertices. The sum of weights of the edges is called the length of the path. The minimum length determines the shortest path between the source and destination vertices. For example consider a weighted graph shown in Fig. 16.12.

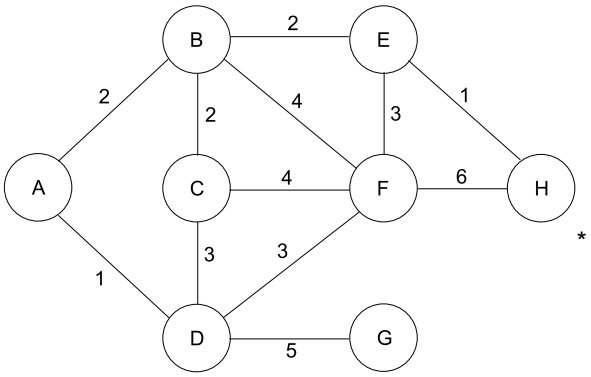


Fig. 16.12

The above graph has 8 nodes in which A is the source vertex and H is the destination vertex. First, we need to determine the paths that exist between the two nodes, A and H. They are:

- ADCFH and the length of this path is 14
- ABEFH and the length of this path is 13
- ABEH and the length of this path is 5.

From this, we can find the shortest path between the source vertex, A, and the destination vertex, H. The path is ABEH, which constitutes the minimum length of 5.

A program for finding the shortest path for a graph is as follows:

```
/* SHORTEST PATH */

#include <stdio.h>
#define size 10
#define infinity 9999

int a[size][size];
int m[size][size];
int i,k,j;
int n;

void Input ();
void Short ();
void Output ();

/* Input function */

void Input()
   {
    printf("\n Input the number of vetices: ");
    scanf("%d", &n);
    printf("\n Please enter the adjacency matrix\n");
    for (i = 0; i < n; i++)
  {
    for (j = 0; j < n; j++)
```

```
    {
       scanf("%d", &a[i][j]);
    }
      printf("\n");
    }
     printf("\n Adjacency matrix: \n");
    for ( i = 0; i < n; i++)
     {
       for ( j = 0; j < n; j++)
        {
          printf("  %i", a[i][j]);
        }
       printf("\n");
     }
     }

/* Output function */

void Output()
    {
      for ( i = 0; i < n; i++)
    {
       for ( j = 0; j < n; j++)
        {
          printf("   %d", m[i][j]);
        }
       printf("\n");
     }
     }

/* Shortest path function */

void  Short ()
    {
    /* Initialization of matrix m */

      for ( i = 0; i < n; ++i)
    {
       for ( j = 0; j < n; ++j)
        {
          if (a[i][j] == 0)
          m[i][j] = infinity;
          else
          m[i][j] = a[i][j];
        }
     }
    printf("\n Adjacency matrix after replacing zeros by very large value");
    Output();

/*  Shortest path evaluation start from here */
```

```
    for ( k = 0; k < n; k++)
   {
    for ( i = 0; i < n; i++)
   {
     for ( j = 0; j < n; j++)
      {
        if ( m [i][j] <= m [i][k] + m [k][j] )
        m [i][j] = m [i][j];
        else
        m [i][j] = m[i][k] + m [k][j];
      }
   }
   printf("\n STEP %d \n", k);
   Output();
    }
  }

/* Function main */

  void main()
     {
    Input();
    Short();
    printf("\n The shortest path matrix is: \n");
    Output();
     }
```

POINTS TO REMEMBER

1. Each item in a binary tree has two links: one to a left member, and another to a right member.
2. A tree is a recursive (i.e. self-repeating) data structure, and recursive programming techniques are popularly used with trees.
3. A tree can be traversed in three major ways: In-order tree traversal, Pre-order tree traversal, Post-order tree traversal.
4. A **graph** is a set of **nodes** (also called as **vertices**) and a set of **arcs** (also called as **edges**).
5. A graph can be represented as an array or as a linked list in C.
6. If a graph is to be represented as a linked list then there are two types of list nodes, a **header node** (also called as **graph node**), and a **list node**.

Review Questions and Exercises

16.1 Explain Binary tree in detail.

16.2 Discuss the graph representation in C in brief.

16.3 Discuss the various C operations that can be associated with the graph in the form of a linked list. Explain this with an example.

State whether the following statements are True or False

(a) Each data item (a through g) in a tree is called as a sub-tree.
(b) Any portion of a tree (e.g. a-b-c) is called as a node.
(c) A node that does not have any sub-tree is called as a terminal node.
(d) Accessing a tree is called as the process of tree traversal.
(e) Items of a tree must be accessed, inserted or deleted in a specific order.
(f) A tree is a recursive data structure.
(g) Set of nodes of a graph can also be called as edges.
(h) A graph is **connected** if there is a path between any two nodes of the graph.
(i) If a graph is represented as a linked list then there are three types of list nodes.
(j) A header node contains the actual information of the node, a pointer to the next node, as well as a pointer to an arc.
(k) A list node contains a pointer to a node and a pointer to the previous arc.

Answers to True and False Questions

(a) False	(b) False	(c) True	(d) True	(e) True
(f) True	(g) False	(h) True	(i) False	(j) True
(k) False	(l) False	(m) True		

UNIT VIII

Chapter 17

Sorting and Searching Techniques

17.1 INTRODUCTION

Information retrieval is one of the key features expected out of a computerized system. Given a telephone number getting the name and address of the subscriber, given a policy number showing the policy details are all some of the widely used retrieval practices. Basically, the process of retrieval involves searching a collection of well-organized data in a faster manner which must be capable of throwing the result almost instantaneously to the querying person. In order to be faster, the retrieval process has to be very efficient. Also, the data must be so arranged that it assists the retrieval process. The retrieval as we have been talking so far is called searching while the organizing of data is achieved through sorting. We will discuss these two aspects in this chapter.

17.2 SORTING

As we saw in the last section, the term sorting means bringing some orderliness in the data, which are otherwise not ordered. Consider a simple scenario—you are going to the railway station to board a train. The train, which you are supposed to board, is at platform 10. Assume that the railway station does not follow an ordered platform numbering scheme. Because of this the platforms are numbered in the fashion—8, 5, 2, 1, etc. Will it not take more time for you to locate the platform, which you are looking for? On the other hand, if the platform numbers are arranged like 1, 2, 3, etc. finding the 10^{th} platform is an easier task. Thus, an ordered data makes the searching process efficient and easier. The term sorting refers to the process of bringing orderliness in the data.

Many sorting methods are advocated. Each method has its own advantages and disadvantages. Often a designer or a developer is puzzled as to which method is best and efficient. The efficiency could be measured in terms of memory usage, time taken to sort, CPU usage, etc. We are more concerned about the quickness of the algorithm. The next section answers this question.

17.2.1 Sorting Efficiency

The efficiency of sorting algorithms is measured using the O-notation (called Big O). The letter O stands for "order of". Let us understand what we mean by this notation. Consider an algorithm, which can sort n numbers in 50 seconds on a Pentium II PC. The same algorithm can run in 40 seconds on a Pentium III PC. Here the reduction in the time is not due to an increase in the efficiency of the algorithm but due to the configurational change in the PC in which it is getting executed. Hence the reduction in time is not attributable to the efficiency of the algorithm but to that of the machine. Hence we need an altogether different scale to measure the efficiency of an algorithm. The answer is the O-notation. It is intended to measure the performance of the algorithm based on the number of inputs it receives.

Let us understand some mathematics behind the O-notation. Consider a general quadratic equation of the form $an^2 + bn + c = 0$. In this form, let us assume the values of a, b and c to be 0.5, 0.25 and 0.12 respectively. The equation now becomes:

$$f(n) = 0.5n^2 + 0.25n + 0.12$$

What will happen to this function, as n gets larger and larger? This would mean that n would contribute more to the expression yielding higher values. You can also eventually see that the term n^2 will significantly contribute to the value of $f(n)$. Hence the terms n and the constant are not very important. The O-notation depends on this.

The notation identifies the most contributing term in the expression and throws out the other terms. We categorize the O-notation based on these dominant terms to create various classes of functions to measure an algorithm's efficiency. Some of the widely used classes are O(1) – Constant , O(n) – linear, $O(n^k)$—polynomial, $O(2^n)$—Exponential and O(log n) – logarithmic. The corresponding tabular form is shown in Table 17.1.

Table 17.1 O-notation analysis

n	O(1)	O(n)	O(log n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	2	1	2	4	8
4	1	4	2	8	16	64
8	1	8	3	24	64	512
16	1	16	4	64	256	4096

Let us now understand what Table 17.1 means?

- **O(1)**—this means that an algorithm will run for a constant amount of time irrespective of the size of input it receives. Whatever be the size of the input you give, the algorithm will run for the same amount of time to give the result. Some examples are push to stack, pop from stack.
- **O(n)**—this means that the amount of time the algorithm will take to complete is directly proportional to the size of the input. If the size of the input is large, the time taken to complete the process will increase. Consider a typical case of finding the sum of numbers in an array. If the array size increases, the time taken to compute the sum also increases.
- **$O(n^2)$**—this means that the amount of time the algorithm will take to complete increases by n^2 if you increase the size of the input by n. Certain sorting algorithms (which we will see shortly) like bubble sort, insertion sort have this.

- **$O(2^n)$**—this means that the amount of time the algorithm will take to complete doubles each time you increase the input.

You will agree to the fact that an algorithm with O(log n) has the best efficiency factor. However, it is always not possible to get an algorithm with such efficiency. You will understand this statement as you read the chapter.

The sorting algorithms, which we will discuss in this chapter, are divided into two categories based on the O-notation efficiency. These are $O(n^2)$ and O(n log n). As you can see we do not have a category of sorting algorithms with O(log n) efficiency. The algorithms, which we will discuss, are:

- O(n) – bubble sort, insertion sort, selection sort and shell sort
- O(n log n) – heap sort, merge sort and quick sort

17.2.2 Exchange Sorting—Bubble Sort

The simplest and the oldest sorting technique is the bubble sort. However this is the most inefficient algorithm. Let us understand this method.

The method takes two elements at a time. It compares these two elements. If the first element is less than the second element, they are left undisturbed. If the first element is greater than the second element, then they are swapped. The procedure continues with the next two elements, goes and ends when all the elements are sorted. Consider the list 74, 39, 35, 97, 84.

Pass 1: (first element is compared with all other elements) Note that the first element may change during the process.

(1) Compare 74 and 39. Since 74 > 39, they are swapped. The array now changes like 39, 74, 35, 97, 84.
(2) Compare 39 and 35. Since 39 > 35, they are swapped. The array now changes like 35, 39, 74, 97, 84.
(3) Compare 35 and 97. Since 35 < 97, they are undisturbed. The array is now 35, 39, 74, 97, 84.
(4) Compare 35 and 84. Since 35 < 84, they are undisturbed. The array is now 35, 39, 74, 97, 84.

At the end of this pass, the first element will be in the correct place. The second pass begins with the second element and is compared with all other elements. Note that the number of comparisons will be reduced by one since the first element is not compared any more.

Pass 2: (second element is compared)

(1) Compare 39 and 74. Since 39 < 74, they are undisturbed. The array is now 35, 39, 74, 97, 84.
(2) Compare 39 and 97. Since 39 < 97, they are undisturbed. The array is now 35, 39, 74, 97, 84.
(3) Compare 39 and 84. Since 39 < 84, they are undisturbed. The array is now 35, 39, 74, 97, 84.

This pass did not bring any change in the array elements. The next pass starts.

Pass 3: (third element is compared)

(1) Compare 74 and 97. Since 74 < 97, they are undisturbed. The array is now 35, 39, 74, 97, 84.
(2) Compare 74 and 84. Since 74 < 84, they are undisturbed. The array is now 35, 39, 74, 97, 84.

This pass also did not bring any change in the array elements. The next pass starts.

Pass 4: (fourth element is compared)

(1) Compare 97 and 84. Since 97 > 84, they are swapped. The array is now 35, 39, 74, 84, 97.

The sorting ends here. Note that the last number is not compared with any more numbers.

1. C program for bubble sort

Let us assume that an array named arr[] will hold the array to be sorted and n is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature. The required C program is shown in Fig 17.1.

```
#include <stdlib.h>
#include <stdio.h>
#define MAXSIZE 500
void bubblesort(int arr[], int maxsize);
int arr[n],n;
int main()
{
    int i;
    printf("\nHow many arr you want to sort: ");
    scanf("%d",&n);
    printf("\nEnter the values one by one: ");
    for (i = 0; i < n; i++)
    {
    printf ("\nEnter element %i :",i);
    scanf("%d",&arr[i]);
    }
    printf("\nArray before sorting:\n");
    for (i = 0; i < n; i++)
    printf("[%i], ",arr[i]);
    printf ("\n");
    bubblesort(arr, n);
    printf("\nArray after sorting:\n");
    for (i = 0; i < n; i++)
    printf("[%i], ", arr[i]);
    }
    void bubblesort(int arr[], int n)
    {
    int i, j, temp;
    for (i = 0; i < n - 1; i++)
    {
        for (j = i; j < n; j++)
        {
            if (arr[i] > arr[j])
            {
                        temp = arr[i];
                arr[i] = arr[j];
                    arr[j] = temp;
            }
        }
    }
}
```

Fig. 17.1 *Bubble sort—C program*

At the end of the execution, the array arr[] will be sorted in the ascending order. If you need to perform a descending order sorting, replace the > symbol with < symbol in the comparison statement.

2. Efficiency of bubblesort

Let us now analyze the efficiency of the bubble sort. You can see from the algorithm that each time (each pass), the number of elements scanned for comparison reduces by 1. Thus, we have:

Number of comparisons in the first pass = (n – 1)
Number of comparisons in the second pass = (n – 2)
Number of comparisons in the last pass = 1

Thus, the total number of comparisons at the end of the algorithm would have been:

$$(n - 1) + (n - 2) + \ldots + 2 + 1 = n\,(n - 1) / 2 = n^2 / 2 + O(n) = O(n^2)$$

Hence the order of the bubble sort algorithm is $O(n^2)$. A sample output showing the result of the program is given below:

```
How many elements you want to sort: 5

Enter the values one by one:
Enter element 0 :74

Enter element 1 :39

Enter element 2 :35

Enter element 3 :97

Enter element 4 :84

Array before sorting:
[74], [39], [35], [97], [84],

Array after sorting:
[35], [39], [74], [84], [97],
```

17.2.3 Exchange Sorting—Quick Sort

This method, invented by Hoare, is considered to be a fast method to sort the elements. The method is also called partition exchange sorting. The method is based on divide-and-conquer technique, i.e. the entire list is divided into various partitions and sorting is applied again and again on the partitions.

In this method, the list is divided into two, based on an element called the pivot element. Usually, the first element is considered to be the pivot element. Now, move the pivot element into its correct position in the list. The elements to the left of the pivot are less than the pivot while the elements to the right of the pivot are greater than the pivot. The process is reapplied to each of these partitions. This process proceeds till we get the sorted list of elements. Let us understand by an example. Consider the list 74, 39, 35, 32, 97, 84.

(1) We will take 74 as the pivot and move it to position so that the new list becomes—39, 35, 32, 74, 97, 84. Note that the elements to the left are lesser than 74. Also, these elements (39, 35 and 32) are yet to be sorted. Similarly, the elements 97 and 84, which are right to the pivot, are yet to be sorted.
(2) Now take the partitioned list—39, 35, 32. Let us take 39 as the pivot. Moving it to the correct position gives—35, 32, 39. The partition to the left is 35, 32. There is no right partition. Reapply the process on the partition, we get 32, 35. Thus, we have 32, 35, 39.
(3) Apply the process to the right partition of 74 which is 97, 84. Taking 97 as the pivot and positioning it, we get 74, 84, 97.
(4) Assembling all the elements from each partition, we get the sorted list.

1. C program for quick sort

The quick sort is a very good example for recursive programming. Let us assume that an array named elements[] will hold the array to be sorted and maxsize is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature. The required C program is shown in Fig. 17.2.

```
#include <stdlib.h>
#include <stdio.h>

#define MAXSIZE 500

void quickSort(int elements[], int maxsize);
void sort(int elements[], int left, int right);

int elements[MAXSIZE];

int main()
{
    int i, maxsize;
    printf("\nHow many elements you want to sort: ");
    scanf("%d",&maxsize);

    printf("\nEnter the values one by one: ");
    for (i = 0; i < maxsize; i++)
    {
    printf ("\nEnter element %i :",i);
    scanf("%d",&elements[i]);
    }

    printf("\nArray before sorting:\n");
    for (i = 0; i < maxsize; i++)
        printf("[%i], ",elements[i]);
        printf ("\n");
        quickSort(elements, maxsize);

        printf("\nArray after sorting:\n");
        for (i = 0; i < maxsize; i++)
```

```
        printf("[%i], ", elements[i]);
    }
    void quickSort(int elements[], int maxsize)
    {
    sort(elements, 0, maxsize - 1);
    }
    void sort(int elements[], int left, int right)
    {
    int pivot, l, r;
    l = left;
    r = right;
    pivot = elements[left];
    while (left < right)
    {
    while ((elements[right] >= pivot) && (left < right))
    right—;
    if (left != right)
    {
    elements[left] = elements[right];
    left++;
    }
    while ((elements[left] <= pivot) && (left < right))
    left++;
    if (left != right)
    {
    elements[right] = elements[left];
    right—;
    }
    }
    elements[left] = pivot;
    pivot = left;
    left = l;
    right = r;
    if (left < pivot)
    sort(elements, left, pivot - 1);
    if (right > pivot)
    sort(elements, pivot + 1, right);
}
```

Fig. 17.2 *Quicksort C program*

A sample output showing the result of the program is given below:

```
How many elements you want to sort: 6

Enter the values one by one:
Enter element 0 :74

Enter element 1 :39

Enter element 2 :32

Enter element 3 :35

Enter element 4 :97

Enter element 5 :84

Array before sorting:
[74], [39], [32], [35], [97], [84],

Array after sorting:
[32], [35], [39], [74], [84], [97],
```

2. Efficiency of quick sort

Let us now analyze the efficiency of the quick sort. The selection of the pivot plays a vital role in determining the efficiency of the quick sort. The main reasons for this are—the pivot may partition the list into two so that one partition is much bigger than the other and the partition may be in an unsorted fashion (which is possible to the maximum). We will analyze the quick sort from the comparison point of view. We will consider two possible cases here.

Case I:

In this case we will make two vital assumptions. These are:

1. The pivot, which we choose, will always be swapped into exactly the middle of the partition. To put it simple, the pivot (after swapping into its correct position) will have an equal number of elements both to its left and right.
2. The number of elements in the list is a power of 2. This means, if there are x elements in the array, we say that $x = 2^y$. This can be rewritten as $y = \log_2 x$.

In this situation, the first of the pass will have x comparisons. When the first pass is completed, the list will be partitioned into two equal halves, each with x/2 elements (which is obvious since we assumed that the pivot will partition the list exactly into two). In the next pass, we will have four partitions, each with equal number of elements (which will be x/4). Proceeding in this way, we will get the following table.

Table 17.2

Pass	Number of comparisons
1	x
2	2 * (x / 2)
3	4 * (x / 4)
4	8 * (x / 8)
X	X * (x / x)

Thus, the total number of comparisons would be O(x) + O(x) + ⋯ + y terms. This is equivalent to O(x * y). Let us substitute $y = \log_2 x$, so that we get O(x log x). Thus the efficiency of quick sort is O(x log x).

Case II:

Let us now forgo one of our assumptions, which we made earlier. We will assume that the pivot partitions the list into two so that one of the partitions has no elements while the other has all the other elements. This is the worst case possible. Here, the total number of comparisons at the end of the sort would have been:

$(x - 1) + (x - 2) + \ldots + 2 + 1 = \frac{1}{2}(x - 1) * x = \frac{1}{2}(x^2) - \frac{1}{2}(x) = O(x^2)$. Thus, the efficiency of the quick sort in its worst case is $O(x^2)$.

17.2.4 Selection Sort

We will discuss the straight selection sort (also called push-down sorting) in this section. As the name suggests, the first element of the list is selected. It is compared repeatedly with all the elements. If any element is found to be lesser than the selected element, these two are swapped. This procedure is repeated till the entire array is sorted. Let us understand this with a simple example. Consider the list 74, 39, 35, 32, 97, 84. Table 17.3 gives the status of the list after each pass is completed.

Table 17.3

Pass	List after pass
1	32, 39, 35, 74, 97, 84
2	32, 35, 39, 74, 97, 84
3	32, 35, 39, 74, 97, 84
4	32, 35, 39, 74, 97, 84
5	32, 35, 39, 74, 84, 97

1. C program for selection sort

Let us assume that an array named elements[] will hold the array to be sorted and maxsize is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature. The required C program is shown in Fig. 17.3.

```
#include <stdlib.h>
#include <stdio.h>
#define MAXSIZE 500
void selection(int elements[], int maxsize);
int elements[MAXSIZE],maxsize;
int main()
{
  int i;
  printf("\nHow many elements you want to sort: ");
  scanf("%d",&maxsize);
  printf("\nEnter the values one by one: ");
  for (i = 0; i < maxsize; i++)
  {
  printf ("\nEnter element %i :",i);
  scanf("%d",&elements[i]);
  }
  printf("\nArray before sorting:\n");
  for (i = 0; i < maxsize; i++)
  printf("[%i], ",elements[i]);
  printf ("\n");
  selection(elements, maxsize);
  printf("\nArray after sorting:\n");
  for (i = 0; i < maxsize; i++)
  printf("[%i], ", elements[i]);
  }
  void selection(int elements[], int array_size)
  {
  int i, j, k;
  int min, temp;
  for (i = 0; i < maxsize-1; i++)
  {
  min = i;
  for (j = i+1; j < maxsize; j++)
  {
  if (elements[j] < elements[min])
min = j;
}
temp = elements[i];
elements[i] = elements[min];
elements[min] = temp;
}
}
```

Fig. 17.3 *C program for selection sorting*

2. Efficiency of selection sort

Let us now analyze the efficiency of the selection sort. You can easily understand from the algorithm that the first pass of the program does (maxsize – 1) comparisons. The next pass does (maxsize – 2) comparisons and so on. Thus, the total number of comparisons at the end of the sort would be :

$$(\text{maxsize} - 1) + (\text{maxsize} - 2) + \ldots + 1 = \text{maxsize} * (\text{maxsize} - 1) / 2 = O(\text{maxsize}^2)$$

Note that this efficiency is same as the worst case of quick sort.

17.2.5 Merge Sort

The merge sort technique sorts a given set of values by combining two sorted arrays into one larger sorted array.

Consider the sorted array, **A,** which contains **p** elements, and the sorted array **B,** containing **q** elements. The merge sort technique combines the elements of **A** and **B** into a single sorted array **C** with **p** + **q** elements.

The total number of comparisons in the merge sort technique to sort *n* data-items of an array is log *n*. The merge sort technique requires at most log *n* passes, so the complexity of the merge sort is $O(n \log n)$. Consider an example, to merge two sorted arrays into a single sorted array by using the merge sort technique. The data items stored in an array *A*, are:

$$A=\{56,78\}$$

The data items stored in an array, *B*, are:

$$B=\{45,67,89\}$$

The data items of the *C* array after merging the two sorted arrays, *A* and *B*, are:

$$C=\{45, 56, 67, 78, 89\}$$

The first data item of the *A* array is compared with the first data item of the *B* array. If the first data item of *A* is smaller than the first data item of *B*, then that data item from *A* is moved to the new array, *C*. If the data item of *B* is smaller than the data item of *A*, then it is moved to the array, *C*. This comparing of data items continues until one of the arrays ends.

The following code shows the implementation of merge sort technique.

```
#include<stdio.h>
#include<conio.h>
void smerge_sort(int *a, int *b, int *c, int n1, int n2, int *n);
void main()
{
 int n1=0, n2=0, n=0, i=0;
 int a[30], b[30], c[60];
 clrscr();
 printf("\nEnter two sorted arrays\n");
 printf("\n\t\tDetails of 1st array:");
 printf("\nNumber of elements n1: ");
 scanf("%d",&n1);
 printf("Enter elements: ");
 for(i=0;i<n1;i++)
```

```
        scanf("%d",&a[i]);
    printf("\n\t\tDetails of 2nd array:");
    printf("\nNumber of elements n2: ");
    scanf("%d",&n2);
    printf("Enter elements: ");
    for(i=0;i<n2;i++)
    scanf("%d",&b[i]);
    smerge_sort(a, b, c, n1, n2, &n);
    printf("\nResultant array after merge sort.\n");
    for(i=0;i<n;i++)
    printf("%d ",c[i]);
    getch();
   }
   void smerge_sort(int *a, int *b, int *c, int n1, int n2, int *n)
   {
    int i=0, j=0, k=0;
    while(i<n1 && j<n2)
    {
    if(a[i]<b[j])
    {
        c[k]=a[i];
        i++;
    }
    else
    {
        c[k]=b[j];
        j++;
    }
    k++;
    }
    while(i<n1)
    {
    c[k]=a[i];
    i++;
    k++;
    }
    while(j<n2)
    {
    c[k]=b[j];
    j++;
    k++;
    }
    *n=k;
   }
```

Fig. 17.4 *C program for merge sort*

The output of the above code is:

```
Enter two sorted arrays
                Details of 1st array:
Number of elements n1:5
Enter  elements: 5 10 14 19 30
                Details of 2nd array:
```

```
Number of elements n2: 6
Enter elements: 2 23 28 44 49 60
Resultant array after merge sort.
2   5   10   14   19   23   28   30   44   49   60
```

17.2.6 Simple insertion sort

We will discuss the insertion sort in this section. An example of an insertion sort occurs while playing cards. To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence. This is illustrated with a simple example in Fig. 17.5. Consider the list – 23, 17, 5, 14.

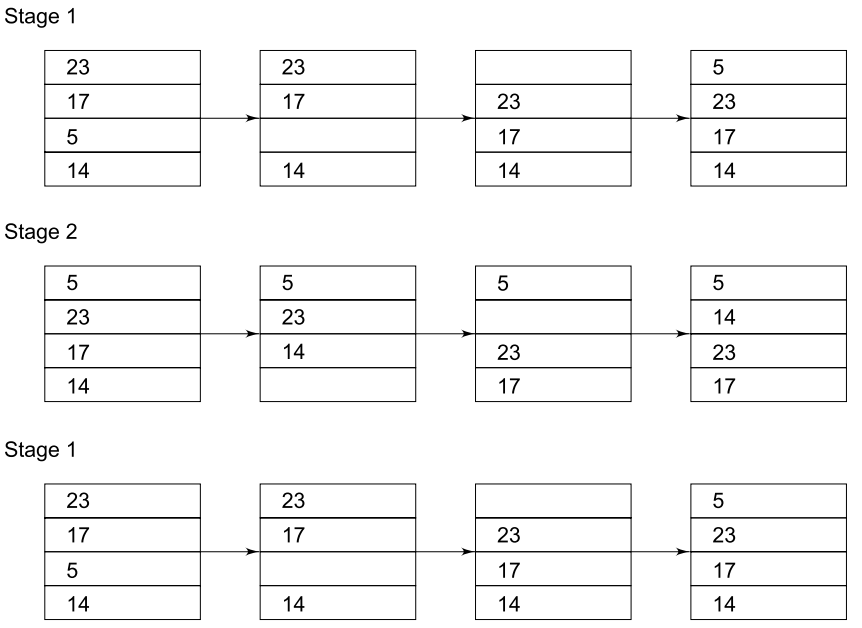


Fig. 17.5

Look at how the sorting proceeds in various stages. At each stage pushing the remaining elements down creates an empty space.

1. C program for insertion sort

Let us assume that an array named elements[] will hold the array to be sorted and maxsize is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature. The required C program is shown in Fig 17.6.

```
#include <stdlib.h>
#include <stdio.h>

#define MAXSIZE 500

void insertionsort(int elements[], int maxsize);
```

```
int elements[MAXSIZE],maxsize;
int main()
{
  int i;
  printf("\nHow many elements you want to sort: ");
  scanf("%d",&maxsize);
  printf("\nEnter the values one by one: ");
  for (i = 0; i < maxsize; i++)
  {
  printf ("\nEnter element %i :",i);
  scanf("%d",&elements[i]);
}
printf("\nArray before sorting:\n");
for (i = 0; i < maxsize; i++)
  printf("[%i], ",elements[i]);
printf ("\n");
insertionsort(elements, maxsize);
printf("\nArray after sorting:\n");
for (i = 0; i < maxsize; i++)
  printf("[%i], ", elements[i]);
}
void insertionsort(int elements[], int maxsize)
{
  int i, j, index;
  for (i=1; i < maxsize; i++)
  {
  index = elements[i];
  j = i;
  while ((j > 0) && (elements[j-1] > index))
  {
  elements[j] = elements[j-1];
  j = j - 1;
  }
  elements[j] = index;
  }
}
```

Fig. 17.6 *C program for insertion sort*

2. Efficiency of insertion sort

Let us now analyze the efficiency of the selection sort. Assuming there are n elements in the array, we must pass through n – 1 entry. For each entry, we may need to examine and shift a maximum of n – 1 other entries, resulting in a efficiency of $O(n^2)$.

17.2.7 Shell sort

We will discuss the shell sort in this section. The shell sort is based on the insertion sort. The given list of elements is broken down in few smaller lists. After this, the insertion sort is applied on such smaller lists. These smaller lists are again recombined. Another partition is made and the procedure is repeated. The breaking down of the list into smaller lists is by selecting elements at specific locations. This is determined by a factor called increment. Consider the list – 23, 12, 6, 34, 13, 7, 44. We may initially take the increment as 3. This would break the list as 23, 34, 44; 12,13 and 6, 7. The first list is done using elements[0], elements[3] and elements[6] where the increment is 3. This list is sorted using insertion sort. The next list is elements[1] and elements[4] where the increment is the same 3. The final list is elements[2] and elements[5] with the same increment of 3. After sorting them, the sublists are recombined.

As a next stage, the increment is changed (reduced by a suitable amount) and the process is repeated. The process continues till the increment is 1. The choice of increment is left to our choice. We can increments such as 7, 5, 3 and 1 or 6, 4, 1, etc.

1. C program for shell sort

Let us assume that an array named elements[] will hold the array to be sorted and maxsize is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature. The required C program is shown in Fig. 17.7. Note that we have taken the increments as 7, 2 and 1. The successive increments are deduced by the formula increment / 2.

```
#include <stdlib.h>
#include <stdio.h>
#define MAXSIZE 500
void shellsort(int elements[], int maxsize);
int elements[MAXSIZE],maxsize;
int main()
{
  int i;
  printf("\nHow many elements you want to sort: ");
  scanf("%d",&maxsize);
  printf("\nEnter the values one by one: ");
  for (i = 0; i < maxsize; i++)
  {
  printf ("\nEnter element %i :",i);
  scanf("%d",&elements[i]);
  }
  printf("\nArray before sorting:\n");
  for (i = 0; i < maxsize; i++)
    printf("[%i], ",elements[i]);
    printf ("\n");
    shellsort(elements, maxsize);
```

```
    printf("\nArray after sorting:\n");
    for (i = 0; i < maxsize; i++)
    printf("[%i], ", elements[i]);
  }
void shellsort(int elements[], int maxsize)
{
  int i, j, increment, temp;
  increment = 7;
  while (increment > 0)
  {
  for (i=0; i < maxsize; i++)
  {
  j = i;
  temp = elements[i];
  while ((j >= increment) && (elements[j-increment] > temp))
    {
    elements[j] = elements[j - increment];
    j = j - increment;
    }
    elements[j] = temp;
  }
  if (increment/3 != 0)
    increment = increment/3;
  else if (increment == 1)
    increment = 0;
  else
    increment = 1;
  }
}
```

Fig. 17.7

17.3 SEARCHING

All of us use searching in one way or the other in our daily life. Any information gathering process or retrieval process is basically a searching process. We already saw that searching is more efficient if the data is available in an ordered fashion (sorted). We saw in the last few sections some of the widely used methods of sorting. Let us now look into two of the widely used search methods—the linear search and binary search techniques.

17.3.1 Linear Search

The linear search or the sequential searching is most simple searching method. It does not expect the list to be sorted also. The key, which is to be searched, is compared with each element of the list one by one. If a match exists, the search is terminated. If the end of the list is reached, it means that the search has failed and the key has no matching element in the list.

1. C program for linear search

Figure 17.8 shows the C program for linear search

```
#include <stdlib.h>
#include <stdio.h>

#define MAXSIZE 500

int linearsearch(int elements[], int maxsize, int key);

int elements[MAXSIZE], maxsize, key;
int main()
  {
  int i, pos;
  printf("\nHow many elements you have in the list: ");
  scanf("%d",&maxsize);
  printf("\nEnter the key to be searched: ");
  scanf("%d",&key);
  printf("\nEnter the values one by one: ");
  for (i = 0; i < maxsize; i++)
  {
  printf ("\nEnter element %i :",i);
  scanf("%d",&elements[i]);
  }
  pos = -1;
  pos = linearsearch(elements, maxsize, key);
  if (pos != -1)
    printf("\nThe key %d is found at position %d", key, pos);
  else
    printf("\nThe key is not found in the list");
  }
  int linearsearch(int elements[], int maxsize, int key)
  {
    int i, j, temp;
    for (i = 0; i < maxsize - 1; i++)
    {
    if (key == elements[i])
    return i;
  }
  return -1;
}
```

Fig. 17.8

2. Efficiency of linear search

To find the number of key comparisons for a successful match, we can add the number required for each comparison and divide by the total number of elements in the list. This would give:

$$\frac{1+2+3+\ldots+n}{n} = \frac{n(n+1)}{2n}$$

17.3.2 Binary Search

The binary search is yet another simple searching method. However, in the binary search, the algorithm expects the list to be sorted. You can apply any one of the sorting methods before using the binary search algorithm.

How does the algorithm operates? The given list is divided into two equal halves. The given key is compared with the middle element of the list. Now three situations may occur:

(a) The middle element matches with the key—the search will end peacefully here.
(b) The middle element is greater than the key—then the value which we are searching is (possibly) in the first half of the list
(c) The middle element is lower than the key—then the value which we are searching is (possibly) in the second half of the list

Now the list is searched either in the first half or in the second half as got by the result of conditions (b) and (c) given above. This process is repeated till we get the key or the search fails because the key does not exist.

1. C program for Iterative binary search

Figure 17.9 gives the C program for interative binary search.

```
#include <stdlib.h>
#include <stdio.h>

#define MAXSIZE 500

int binsearch(int elements[], int maxsize, int key);
void bubblesort(int elements[], int maxsize);

int elements[MAXSIZE], maxsize, key;

int main()
{
  int i, pos;
  printf("\nHow many elements you have in the list: ");
  scanf("%d",&maxsize);
  printf("\nEnter the key to be searched: ");
  scanf("%d",&key);

  printf("\nEnter the values one by one: ");
  for (i = 0; i < maxsize; i++)
  {
  printf ("\nEnter element %i :",i);
  scanf("%d",&elements[i]);
  }

  bubblesort(elements, maxsize);

  pos = -1;

  pos = binsearch(elements, maxsize, key);
  if (pos != -1)
    printf("\nThe key %d is found at position %d", key, pos);
  else
```

```
    printf("\nThe key is not found in the list");
  }
  void bubblesort(int elements[], int maxsize)
  {
    int i, j, temp;
    for (i = 0; i < maxsize - 1; i++)
    {
      for (j = i; j < maxsize; j++)
      {
      if (elements[i] > elements[j])
      {
            temp = elements[i];
        elements[i] = elements[j];
            elements[j] = temp;
      }
    }
  }
}
int binsearch(int elements[], int maxsize, int key)
{
  int i, first, middle, last;
  first = 0;
  last = maxsize - 1;
  while (last >= first)
  {
    middle = (first + last ) / 2;
    if (key > elements[middle])
      first = middle + 1;
    else if (key < elements[middle])
      last = middle - 1;
    else
      return middle;
    }
  return -1;
}
```

Fig. 17.9

In the above program, a bubble sort function is also added to get a sorted list before applying binary search on the list. Instead of the bubble sort, you can adopt any other sort routine, which we have already discussed in previous sections.

In the program, you can infer that we have used the variable 'middle' to locate our key. All the comparisons are based on the element, which is at the position pointed by the variable 'middle'. This variable is altered continuously till we get the result. If the search fails, the function returns a negative value.

2. C program for Recursive binary search

We can write a recursive version of the binary search also very easily. Figure 17.10 gives program for such a version of binary search.

```
#include <stdlib.h>
#include <stdio.h>

#define MAXSIZE 500

int binsearch(int elements[], int maxsize, int key, int first, int last);
void bubblesort(int elements[], int maxsize);

int elements[MAXSIZE], maxsize, key;

int main()
{
  int i, pos;
  printf("\nHow many elements you have in the list: ");
  scanf("%d",&maxsize);
  printf("\nEnter the key to be searched: ");
  scanf("%d",&key);

  printf("\nEnter the values one by one: ");
  for (i = 0; i < maxsize; i++)
  {
  printf ("\nEnter element %i :",i);
  scanf("%d",&elements[i]);
  }

  bubblesort(elements, maxsize);

  pos = -1;
  pos = binsearch(elements, maxsize, key, 0, maxsize - 1);
  if (pos != -1)
    printf("\nThe key %d is found at position %d", key, pos);
  else
    printf("\nThe key is not found in the list");
  }

  void bubblesort(int elements[], int maxsize)
  {
  int i, j, temp;
    for (i = 0; i < maxsize - 1; i++)
    {
      for (j = i; j < maxsize; j++)
      {
        if (elements[i] > elements[j])
        {
                temp = elements[i];
          elements[i] = elements[j];
              elements[j] = temp;
        }
```

```
    }
  }
}
int binsearch(int elements[], int maxsize, int key, int first, int last)
{
int i, middle;
middle = -1;
if (first <= last)
{
middle = (first + last ) / 2;
if (key > elements[middle])
middle = binsearch(elements, maxsize, key, middle + 1, last);
else if (key < elements[middle])
middle = binsearch(elements, maxsize, key, first, middle - 1);
}
return middle;
}
```

Fig. 17.10

17.3.3 Indexed Sequential search

Yet another searching technique is the indexed sequential search. This is widely employed in various file handling and database systems. The method is based on a "short representation" of the list. This "short representation" is called as index. How this index is formed? Many ways do exist to form this index. One of the widely used ways is to extract every eighth element from the list and make that as a part of the index.

Whenever a particular key is to be searched, it is first searched in the index. The appropriate location from the index is taken and that location is sequentially searched. Figure 17.11 explains this scenario. In the figure, every fifth record is taken as an example.

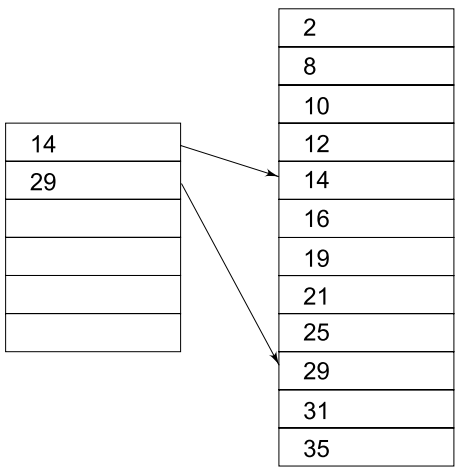


Fig. 17.11

In Fig. 17.11, assume that we are searching for 16. The index is first searched. Note that you may not get an exact match for the key in the index. You will get only the range, which you may use for searching the main list. In this example, we get the range that 16 is between 14 and 29. You may perform a sequential search in this range to get at the key value.

POINTS TO REMEMBER

1. Sorting means rearranging a given list in an orderly manner. Searching is the process of locating a particular key from a list.
2. The efficiency of a sorting algorithm is expressed in terms of the O-notation.
3. A sorting algorithm with O(log n) is supposed to be a highly efficient sorting algorithm.
4. The efficiency of bubble sort, insertion sort, selection sort and shell sort is O(n). The efficiency of heap sort, merge sort and quick sort O(n log n).
5. Quick sort designed by Hoare is a very efficient sorting algorithm,.
6. Binary search may be combined with the indexed sequential searching to give better efficiency.
7. Always remember that searching in a sorted list is more efficient than searching in an unsorted list.

Review Questions

17.1 Discuss bubble sort, write a C program and show its efficiency.
17.2 Discuss the quick sort. Derive the efficiency of the quick sort in the best and worst cases.
17.3 What is a heap? Show how would you perform a heap sort using a heap? Work out the efficiency factor of the heap sort.
17.4 Show that the efficiency of quick sort in its worst case is the same as that of simple insertion sort.
17.5 Show that shell sort is based on the insertion sort.
17.6 Discuss the linear search method.
17.7 Explain the binary searching with a C program.
17.8 What is an index? How can an index speed up the search process?

State whether the following statements are True or False

(a) Sorting and searching are always tightly interrelated.
(b) Quick sort's efficiency relies on the pivot.
(c) Bubble sort even though simple is very efficient.
(d) Selection sort is based on binary tree.
(e) Shell sort is based on quick sort.
(f) Quick sort uses a divide-and-conquer approach.
(g) A sorting method whose efficiency is $O(2^n)$ means that the amount of time the algorithm will take to complete doubles each time you increase the input.
(h) A sorting method with O(log n) is the most efficient method.
(i) Sorting is mandatory for linear searching.
(j) Binary search can be done on an ordered list only.
(k) Indexed sequential search can be applied to file handling only and not to list.
(l) The choice of constructing the index by picking every third number from the main list can be changed.

Answers to True and False Questions

(a) False	(b) True	(c) False	(d) False	(e) False
(f) True	(g) True	(h) True	(i) False	(j) True
(k) False	(l) True			

Appendix A

ASCII Values of Characters

ASCII Value	Character	ASCII Value	Character	ASCII Value	Character	ASCII Value	Character
000	NUL	032	blank	064	@	096	'
001	SOH	033	!	065	A	097	a
002	STX	034	”	066	B	098	b
003	ETX	035	#	067	C	099	c
004	EOT	036	$	068	D	100	d
005	ENQ	037	%	069	E	101	e
006	ACK	038	&	070	F	102	f
007	BEL	039	‘	071	G	103	g
008	BS	040	(	072	H	104	h
009	HT	041	)	073	I	105	i
010	LF	042	*	074	J	106	j
011	VT	043	+	075	K	107	k
012	FF	044	'	076	L	108	l
013	CR	045	–	077	M	109	m
014	SO	046	.	078	N	110	n
015	SI	047	/	079	O	111	o
016	DLE	048	0	080	P	112	p
017	DC1	049	1	081	Q	113	q
018	DC2	050	2	082	R	114	r
019	DC3	051	3	083	S	115	s
020	DC4	052	4	084	T	116	t
021	NAK	053	5	085	U	117	u
022	SYN	054	6	086	V	118	v
023	ETB	055	7	087	W	119	w
024	CAN	056	8	088	X	120	x
025	EM	057	9	089	Y	121	y
026	SUB	058	:	090	Z	122	z
027	ESC	059	;	091	[	123	{
028	FS	060	<	092	\	124	|
029	GS	061	=	093	]	125	}
030	RS	062	>	094	↑	126	~
031	US	063	?	095	-	127	DEL

Note: The first 32 characters and the last character are control characters; they cannot be printed.

Appendix B

Multiple Choice Questions

UNIT-I

1. a<<1 is equivalent to
   (a) multiplying by 2 (b) dividing by 2
   (c) adding 2 (d) none of the above
2. The operation of a staircase switch best explains the
   (a) or operation (b) and operation
   (c) exclusive nor operation (d) exclusive or operation
3. Which of the following is/are syntactically correct?
   (a) for(); (b) for(;); (c) for(,); (d) for(;;);
4. The expression 4+6/3*2–2+7%3 evaluates to
   (a) 3 (b) 4 (c) 6 (d) 7
5. Any C program
   (a) must contain at least one function (b) need not contain any function
   (c) needs input data (d) none of the above
6. Using goto inside for loop is equivalent to using
   (a) continue (b) break
   (c) return (d) none of the above
7. The program fragment
   int a=5, b=2;
   printf("%d",a+++++b);
   (a) prints 7 (b) prints 8
   (c) prints 9 (d) none of the above

8. printf("ab" , "cd","ef"); prints
   (a) ab abcdef (c) abcdef, followed by garbage value
   (d) none of the above
9. Consider the following program segment.

```
i=6720; j=4;
while((i%j)==0)
{
i=i/j;
j=j+1;
}
```

   On termination, j will have the value
   (a) 4 (b) 8 (c) 9 (d) 6720

UNIT- II

1. C preprocessor
   (a) takes care of conditional compilation (b) takes care of macros
   (c) takes care of include files (d) acts before compilations
2. A preprocessor command
   (a) need not start on a new line (b) need not start on the first column
   (c) has # as the first character (d) comes before the first executable statement
3. The following program

```
main()
{
int a=4;
change(a);
printf("%d",a);
}
change(int a)
{ printf("%d",++a); } outputs
```

   (a) 5 5 (b) 4 5 (c) 5 4 (d) 4 4
4. The output of the following program is

```
main()
{
    static int x[]={1, 2, 3, 4, 5, 6, 7, 8};
    int i;
    for(i=2;i<6;i++)
    x[x[i]]=x[i];
for(i=0; i<8;i++)
printf("%d",x[i]);
}
```

(a) 1 2 3 3 5 5 7 8 (b) 1 2 3 4 5 6 7 8
(c) 8 7 6 5 4 3 2 1 (d) 1 2 3 5 4 6 7 8

5. The order in which actual parameters are evaluated in a function call
   (a) is from the left (b) is from the right
   (c) is compiler dependent (d) none of the above
6. The default parameter passing mechanism is
   (a) call by value (b) call by reference
   (c) call by value result (d) none
7. C does no automatic array bound checking. This is
   (a) true (b) false
   (c) C's asset (d) C's shortcoming
8. If a two-dimensional array is used as a formal parameter, then
   (a) both the subscripts may be left empty (b) the first(row) subscript may be left empty
   (c) the first subscript must be left empty (d) both the subscripts must be left empty
9. If storage class is missing in the array definition, by default it will be taken to be
   (a) automatic (b) external
   (c) static
   (d) either automatic or external depending on the place of occurrence
10. Consider the declaration
    static char hello[]="hello"; The output of printf("%s\n",hello);
    will be the same as that of
    (a) puts("hello"); (b) puts(hello);
    (c) printf("%s\n","hello"); (d) puts("hello\n");
11. The array name can be a pointer to
    (a) another array (b) another variable
    (c) to that array only (d) none
12. An array of pointers to a table of strings saves
    (a) time (b) memory
    (c) CPU utilization (d) none of the above
13. The following program

```
main()
{
inc(); inc(); inc();
}
inc()
{
static int x;
printf("%d",++x);
}
```

prints

(a) 0 1 2 (b) 1 2 3
(c) 3 consecutive, but unpredictable numbers (d) 1 1 1

UNIT- III

1. puts(argv[0])prints
   (a) the name of the source code file (b) the number of command line arguments
   (c) argv (d) the name of the executable code file
2. The address operator & , cannot act on
   (a) R-values (b) arithmetic expressions
   (c) members of a structure (d) local variables
3. The argument used to print the number of command line arguments is
   (a) printf("%d",argv); (b) printf("%d",argv[0]);
   (c) printf("%d",argc); (d) none
4. In command line arguments main() function takes _____ number of arguments
   (a) 1 (b) 2 (c) 3 (d) 4

UNIT- IV

1. Structures may contain
   (a) multiple data items (b) single data items
   (c) a only (d) a and b
2. The size of a structure and a union is same when they contain
   (a) single member (b) any number of members
   (c) a and b (d) none
3. The operator used to find the size of any variable
   (a) sizeof (b) Sizeof
   (c) sizeOf (d) all the above
4. The operator that is used to access the members of the structure using pointer variable
   (a) . (b) -> (c) * (d) none of the above
5. The operator used to access the member of the structure
   (a) . (b) -> (c) * (d) none of the above
6. The operator -> is same as the combination of the operators
   (a) * and . (b) & and .
   (c) * and & (d) none of the above
7. Bitfields are used to
   (a) save time (b) save memory
   (c) change order of allocation of memory (d) none of the above
8. Union can store _____ number of values at a time
   (a) all its members (b) only 1
   (c) 2 (d) cannot hold value

UNIT- V

1. If a file is opened in r+ mode then
   (a) reading is possible (b) writing is possible
   (c) it will be created if it does not exist (d) appending is possible
2. If a file is opened in w+ mode then
   (a) reading is possible (b) writing is possible
   (c) it will be created if it does not exist (d) appending is possible
3. If a file is opened in r mode then
   (a) reading is possible (b) writing is possible
   (c) it will be created if it does not exist (d) appending is possible
4. If a file is opened in a mode then
   (a) reading is possible (b) writing is possible
   (c) it will be created if it does not exist (d) appending is possible
5. ftell
   (a) is a function (b) gives the current file position indicator
   (c) can be used to find the size of a file (d) none of the above
6. The fseek function
   (a) needs 2 arguments (b) makes rewind function unnecessary
   (c) takes 3 arguments (d) none of the above
7. The rewind function takes ____ number of arguments
   (a) 1 (b) 2 (c) 3 (d) 0
8. fseek(fptr,0,0) is equivalent to
   (a) ftell (b) rewind
   (c) a and b (d) none of the above
9. ferror function is used to find _____ errors.
   (a) logical (b) file opening (c) data (d) all the above
10. The contents of the file are lost if it is opened in _____ mode .
    (a) a (b) w (c) w+ (d) a+
11. The contents of the file are safe if it is opened in _____ mode.
    (a) a (b) r (c) a+b (d) all the above
12. The valid binary modes of operation are
    (a) ab (b) rb+ (c) wb+ (d) ab+
13. The rewind function is used to
    (a) reset the file pointer (b) point it to the end of the file
    (c) stay at current position (d) none of the above
14. feof function checks for
    (a) file opening error (b) data error
    (c) end of file (d) file closing error

15. The value returned by fopen() function when the file is not opened
    (a) 0 (b) garbage value
    (c) NULL (d) none of the above
16. The fcloseall() function performs
    (a) closing of all the files
    (b) closes all the files that are opened by that program
    (c) closes only specified files
    (d) none of the above
17. The function that is not used for random access to files is
    (a) rewind (b) ftell (c) fseek (d) fprintf

UNIT- VI

1. The prefix equivalent for the postfix ab+cd+* is
   (a) a+b*c+d (b) +ab*+cd (c) *+ab+cd (d) *++abcd
2. The postfix equivalent for the prefix *++abcd is
   (a) ab+c+d* (b) abcd++* (c) ab+cd+* (d) ab+c*d+
3. The infix equivalent to the postfix expression abc+d-*e%f/ is
   (a) a+b*c-d%f/f (b) a*(b+c-d)%e/f (c) a*b+c-d%e/f (d) a*(b-c+d)%e/f
4. Evaluate the expression 2*3/5+6-4
   (a) 1 (b) 2 (c) 3 (d) 4
5. The value of the prefix expression +/*2-5 6 4 3 is
   (a) 1 (b) 2 (c) 3 (d) 4
6. The value of the postfix expression 1 4 +3 / 2 * 6 4 % - is
   (a) 1 (b) -1 (c) 0 (d) 4
7. Towers of Hanoi is an application of
   (a) stack (b) queue (c) linked list (d) dequeue
8. The data structure used in railway reservation is
   (a) stacks (b) queues (c) priority queues (d) binary tree
9. The data structure applicable for a fully packed bus is
   (a) stacks (b) queues (c) priority queues (d) binary tree
10. The recursive functions are evaluated using
    (a) stacks (b) queues (c) priority queues (d) binary tree
11. The nested loops are evaluated using
    (a) stacks (b) queues (c) structures (d) binary tree
12. The data structure used in resource sharing systems is
    (a) stacks (b) queues (c) arrays (d) binary tree
13. Which of the following is not a linear data structure?
    (a) stacks (b) queues (c) linked list (d) binary tree
14. In evaluation of postfix expression the data structure used is
    (a) stacks (b) queues (c) arrays (d) binary tree

UNIT- VII

1. Linked list uses _____ type of memory allocation
   (a) static (b) random (c) dynamic (d) compile time
2. Binary tree can be implemented using
   (a) arrays (b) double linked list (c) a and b (d) b only
3. In a complete binary tree, if the parent is at *n*th position then the children will be at
   (a) n+1, n+2 (b) 2n,2n-1 (c) 2n,2n+1 (d) 2n+1,2n–1
4. The number of non-leaf nodes in a complete binary tree of height 5 is
   (a) 16 (b) 32 (c) 31 (d) 15
5. The number of leaf nodes in a complete binary tree of height 5 is
   (a) 16 (b) 32 (c) 31 (d) 15
6. The number of nodes in a complete binary tree of height 5 is
   (a) 16 (b) 32 (c) 31 (d) 15
7. The number of edges in a minimum cost spanning tree of n nodes is
   (a) n (b) n+1 (c) n-1 (d) 2n
8. Traveling salesman problem is an application of
   (a) spanning trees (b) binary tree
   (c) greedy method (d) divide and conquer
9. The number of extra pointers required to reverse a singly linked list is
   (a) 1 (b) 2 (c) 3 (d) 4
10. The number of extra pointers required to reverse a double linked list is
    (a) 1 (b) 2 (c) 3 (d) 4
11. The functions used for memory allocation are
    (a) malloc (b) calloc (c) a and b
    (d) none of the above
12. Linked lists use _____ type of structures.
    (a) nested (b) self-referential (c) simple (d) unions
13. _____ cannot be used to represent linked lists.
    (a) Arrays (b) Structures (c) Unions (d) All the above
14. Binary trees cannot be implemented using
    (a) arrays (b) unions
    (c) single linked list (d) all the above
15. calloc(m,n) is equivalent to
    (a) malloc(m*n,0) (b) memset(0,m*n)
    (c) ptr=malloc(m*n) (d) malloc(m/n)
16. Prim's and Krushkals algorithms are used for finding solution to
    (a) BFS (b) DFS
    (c) traveling salesman problem (d) none of the above

UNIT- VIII

1. The time complexity of binary search in average case is
   (a) O(n) (b) O(n2) (c) O(nlog n) (d) O(logn)
2. The time complexity of bubble sort in best case is
   (a) O(n) (b) O(n2) (c) O(nlog n) (d) O(logn)
3. The time complexity of selection sort in best case is
   (a) O(n) (b) O(n2) (c) O(nlogn) (d) O(logn)
4. The time complexity of insertion sort in best case is
   (a) O(n) (b) O(n2) (c) O(nlog*n*) (d) O(logn)
5. The time complexity of quick sort in best case is
   (a) O(n) (b) O(n2) (c) O(nlogn) (d) O(logn)
6. The time complexity of heap sort in best case is
   (a) O(n) (b) O(n2) (c) O(nlogn) (d) O(logn)
7. The time complexity of merge sort in best case is
   (a) O(n) (b) O(n2) (c) O(nlogn) (d) O(logn)
8. The best sorting technique among the following is
   (a) quick (b) heap (c) merge (d) bubble
9. In worst case quick sort behaves like
   (a) insertion (b) heap (c) selection (d) bubble
10. The time complexity of bubble sort in worst case is
    (a) O(n) (b) O(n2) (c) O(nlogn) (d) O(logn)
11. The time complexity of selection sort in worst case is
    (a) O(n) (b) O(n2) (c) O(nlogn) (d) O(logn)
12. The time complexity of insertion sort in worst case is
    (a) O(n) (b) O(n2) (c) O(nlogn) (d) O(logn)
13. The time complexity of quick sort in worst case is
    (a) O(n) (b) O(n2) (c) O(nlogn) (d) O(logn)
14. The time complexity of heap sort in worst case is
    (a) O(n) (b) O(n2) (c) O(nlogn) (d) O(logn)
15. The time complexity of merge sort in worst case is
    (a) O(n) (b) O(n2) (c) O(nlogn) (d) O(logn)
16. Quick sort is an application of
    (a) partition exchange sort (b) partition sort
    (c) greedy method (d) divide and conquer
17. Merge sort is an application of
    (a) greedy method (b) divide and conquer
    (c) a and b (d) none

18. The space complexity of quick sort in average case is
    (a) 0 (b) O(n) (c) O(nlogn) (d) O(logn)
19. The space complexity of bubble sort in worst case is
    (a) 0 (b) O(n) (c) O(nlogn) (d) O(logn)
20. Binary search is effective only when the elements are in
    (a) ascending order (b) descending order
    (c) a and b (d) jumbled order

Appendix C

Solved Question Papers
C Programming and Data Structures
May/June 2008

SET- 1

1. (a) Define algorithm.

Ans: In solving a problem or to develop a program, we should have two criteria.

1. algorithm
2. flow chart

Algorithm: An algorithm is a method of representing the step-by-step procedure for solving a problem. An algorithm is very useful for finding the right answer to a simple problem or a difficult problem by breaking the problem into simple cases.

Algorithm for in-order traversal (recursive):

Step 1: check if tree is empty by verifying root pointer R. Print tree empty if R=NULL
Step 2: if leftptr(R)!=NULL then call in-order (leftptr(R))
Step 3: print data(R).
Step 4: if rightptr(R)!= NULL then call in-order(rightptr(R))
Step 5: return.

(b) What is the use of flow chart?

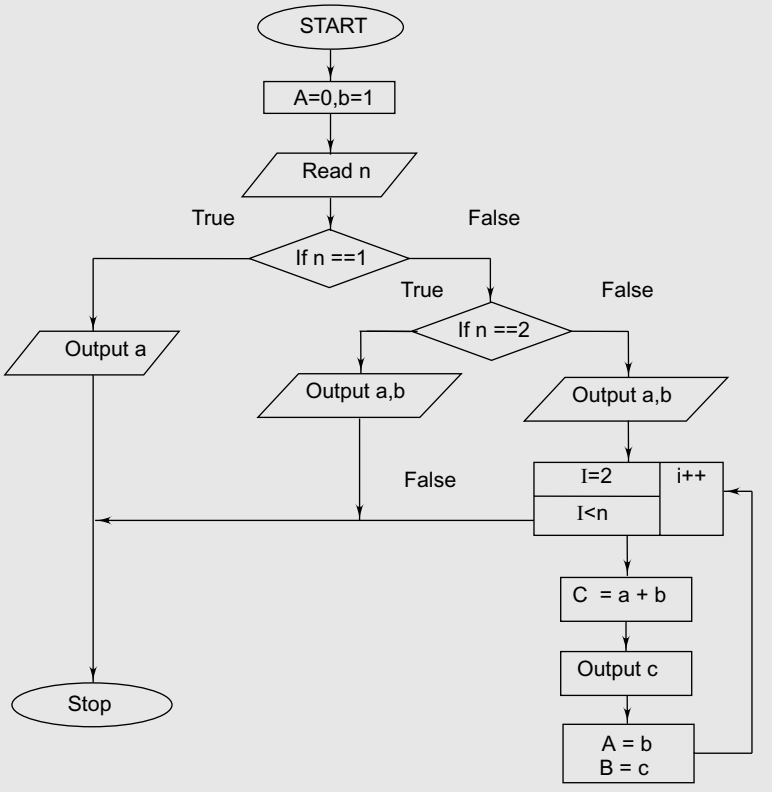
Ans: Flow chart is a diagrammatic representation of an algorithm. It is built using different types of boxes and symbols. The operation to be performed is written in the box. All symbols are interconnected processing to indicate the flow of information and processing.

Uses of flow chart:

1. Flow chart helps to understand the program easily.
2. As different boxes are used to specify the type of operation performed, it will be easy to understand the complex programs.

Example for flow chart:

/* To print the Fibonacci series for 1 to n value*/



(c) What are the different steps followed in the program development?

Ans: A 'C' program has to pass through many phases for its successful execution and to achieve the desired output.

The various steps involved in the program development are as follows:

1. Editing phase
2. Compilation phase
3. Linking phase
4. Execution phase

Editing Phase

In the editing phase, the C program is entered into a file through a text editor. Unix operating system provides a text editor which can be accessed through the command 'Vi'. Modern compiler vendors provide Indegrated Development Environment (IDS). The file is saved on the disk with an extension of 'C'. Corrections in the program at later stages are done through these editors. Once the program has been written, it requires to be translated into machine language.

Compilation Phase

This phase is carried out by a program called as compiler. Compiler translates the source code into the object code. The compilation phase cannot proceed successfully until and unless the source code is error-free. The compiler generates messages if it encounters syntactic errors in the source code. The error-free source code is translated into object code and stored in a separate file with an extension '.obj'.

Linking Phase

In this phase, the linker links all the files and functions with the object code under execution. For example, if a user uses a '`printf`' function in his\ her program, the linker links the user programs, object code with the object code of the `printf` function. Now the object code is ready for next phase.

Execution Phase

In this phase, the executable object code is loaded into the memory and the program execution begins. We may encounter errors during the execution phase even though compilation phase is successful. The errors may be run-time errors and logical errors.

2. (a) Give some important points while using return statement.

Ans: **Return statement:** This statement returns control to the calling function. There can be a number of return statements in the program but each statement can return only one value.

The return statement terminates the execution of a function and returns control to the calling function. A return statement can also return a value to the calling function. The value of expression if present is returned to the calling function. If expression is omitted, the return value of the function is undefined.

The type of variable that is to be returned is to be declared in the function in which return statement is used.

A return statement with a value should not use ( ) [parentheses] around the value. The return value expression must start on the same line as the return keyword in order to avoid semicolon insertion.

If a return value is not required, declare the function to have `void` return type; otherwise the default return type is `int`.If the function was declared with return type `void`, a return statement containing an expression generates a warning and the expression is not evaluated. Example for the return statement

/* Write a program to find the factorial of given number*/

Program:

```
#include<stdio.h>
#include<conio.h>
int fact(int n)   //starting of the sub program
 {
```

```
   int f=1,i;
   if((n==0)||(n==1))  // check the condition for n value
   return(1);
   else
 for(i=1;i<=n;i++)  // perform the looping operation for calculating
 the factorial
   f=f*i;
  return(f);
}
void main()
{
   int n;
   clrscr();
   printf("enter the  number :");
   scanf("%d",&n);
   printf("factoria of number%d",fact(n));
   getch();
}
```

(b) Write a short note on scope of a variable.

Ans: Scope of a variable means whether the variable declared is a global or local variable.

Local variable means it is valid only to the block in which it is declared. Global variable means it is valid throughout the program and its value is not changed throughout the program. The scope of a variable for different storage classes in C language is different.

The scope of variable for `register` storage class is local to the block in which the variable is defined. The scope of variable for `automatic` storage class is local to the block. The scope of variable for `static` storage class is local to the block in which the variable is defined. The scope of variable for `external` storage class is global.

If the variable is declared in the main function or within any function, that variable is called local variable and if the variable is declared before main, then it is called as global variable. By the scope of variable, we can know about the functioning of the variable in a program.

Example:

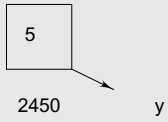
```
#include<stdio.h>
  int a;          /*  global variable*/
     Void main ( )
  {
Int  n,m;   /*  local variable*/
  Printf("none");
   }
```

3. (a) Write a short note on pointers.

Ans: Pointers: Pointer is a variable used to point the address of the another variable. When you declare a variable, storage is allocated to it and the value assigned to it is stored in that storage location.

Example: int $y=5$

When this statement is encountered, memory is allocated to variable y. the value 5 is stored at this address



Similarly pointers are special variables which store address of some other variable or values instead of storing normal values.

Declaration of pointers:
Pointers can be declared as follows

```
Data type   *ptr;
```

Here '*' indicates that `ptr` is a pointer variable representing the value stored at a particular address.

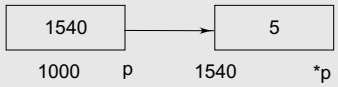
Example:
```
int  *p;
char  *q;
```

Here *p* is a pointer variable pointing to address location where an `integer` type data is stored. *q* is a pointer variable pointing to address location where `character` type data is stored.

Initialization of pointer:
Pointer variable can be initialized during declaration as given below

```
int  *p;
  *p=5;
```

On compilation of the above statement, the value 5 is stored in an address pointed by *p*. Suppose *p* is a pointer to an address 1540, then 5 is stored in location 1540.

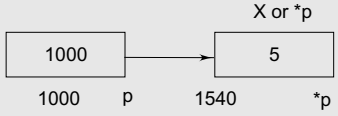


Here, *p* is 1540 and **p* gives value 5.

Assigning of address of variable to pointer variables:
A pointer variable is used to print the addresses of other variables.
Example:

```
int *p;
int  x=5;
p=&x;
```

Here, address of variable x is assigned to pointer p.



**p* gives values at location 1000 i.e., 5 which is same as value of x.

Here, the '&' used is same as the one we use in `scanf` function.'&' is called 'address of operator'.

Example:

```
int   *p=&x;
```

This statement will transfer the address of *x* to *p*.

The following program is an example for pointers.

Program:

/* Write a C program that displays the position or index in the string S where the string T begins, or -1 if S does not contain T */

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
void main()
{
     char s[30], t[20];
     char *found;
     clrscr();

     /* Entering the main string */
      puts("Enter the first string: ");
     gets(s);

     /* Entering the string whose position or index to be displayed */
     puts("Enter the string to be searched: ");
     gets(t);

     /*Searching string t in string s */
     found=strstr(s,t);
     if(found)
            printf("Second String is found in the First String at %d
            position.\n",found-s);
     else
            printf("-1");
     getch();
 }
```

Output:

1. Enter the first string:
   kali
   Enter the string to be searched:
   li
   Second String is found in the First String at 2 position.
2. Enter the first string:
   nagaraju
   Enter the string to be searched:
   raju
   Second String is found in the First String at 4 position
3. Enter the first string:
   nagarjuna
   Enter the string to be searched:
   ma
   -1

4. Write a program using structures to display the following information for each customer name, account number, street, city, old balance, current payment, new balance, account, status.

Ans:

Program:

```
     # include<stdio.h>
     #include<conio.h>
       Void main ()
         {
          Struct organization
            {
             Char name, street, city, status, account;
            Int  account number, oldbalance, currentpayment, newbalance;
             };
   Struct organization cust1={"ramu",101,"subhashchandrabose street",
                              "vijayawada", 40000, 10000, 50000,
                                   "sbaccount","new"};
   Struct organization cust2={"raju",100,"subhashchandrabose street",
                                     "vijayawada", 60000, 10000, 70000,
                                       "sbaccount","new"};
   Printf("the information of the particular customer1
        is=%c%c%c%d%d%d%d%c%c ", cust1.name,cust1.street,cust1.city,
  cust1.oldbalance,cust1.currentpayment,cust1.newbalance,cust1.account,cust1.status);
          }
```

Output:

The information of the customer1 is :
ramu,101, subhashchandrabose street, vijayawada,
40000,10000,50000, sbaccoount new

5. Write a program for indexed sequential file for the employee database for the following operation

(a) add record
(b) delete record
(c) search record based on the department.

Ans:

Program:

/* Write a program to add, delete, search and retrieve records from a file*/

```
#include<stdio.h>
Typedef  struct
     {
     unsigned int empno;
     Char name[50];
     Int salary;
```

```
} st_rec;
Void main()
  {
        int ch;
     St_rec rec;
      FILE *fp;
     Fp= fopen("employee.rec","wb");
      Printf("\n enter the empno");
      Printf("type ctrl+z to stop\n");
      /*read from keyboard and write to file*/
     While(scanf("%u%c%d",&rec.empno,&rec.name,&rec.salary)!EOF)
      Fwrite(&rec,sizeof(rec),1,fp);
       Fclose(fp);
       Printf("\n");
       /* read from file and write on screen*/
      Fp=fopen("employee.rec","rb");
       /* while loop terminates when fread returns 0*/
       While (fread(&rec,sizeof(rec),1,fp))
     Printf("%5u%-10c%3d\n",rec.empno,rec.name,rec.salary);
         int addrecord ();
         int removerecord();
         int searchrecord ();
         Switch(ch)
           {
           Case 'a':  int addrecord()
                          {
                  While(scanf("%u%c%d",&rec.empno,&rec.name,&rec.salary)!EOF)
                fwrite(&rec,sizeof(rec),1,fp);
                      fclose(fp);
                    }
                  break;
          Case 'd': int removerecord(const char*employeerecord.index);
                      Break;
            Case 's' : int searchrecord()
                           {
                            Fp=fopen("employee.rec","rb");
                           /* while loop terminates when fread returns 0*/
                           While (fread(&rec,sizeof(rec),1,fp))
                          Printf("%5u%-10c%3d\n",rec.empno,rec.
                             name,rec.salary);
                              }
                              fclose(fp);
               }
        fclose(fp);
}
```

6. What do you mean by sorting? Mention the different types of sorting. Give some examples and explain any one in detail.

Ans: Sorting: Sorting means arranging a set of data in some order. There are different methods that are used to sort the data in ascending or descending order. Searching and sorting techniques play a vital role for maintaining database applications. Sorting is a process of arranging the records either in ascending or descending order i.e., bringing some orderliness in the data. Sorting methods are very important in data structures. Sorting can be performed on any one or combination of one or more attributes present in each record. It is very easy and efficient to perform searching if data is stored in sorted order. The efficiency of sorting technique depends on utilisation of memory, sorting time, utilisation of processor, etc.

There are many sorting techniques used. Some of them are.

1. bubble sort
2. insertion sort
3. selection sort
4. merge sort
5. quick sort
6. heap sort etc.,

Let us explain the quick sort

Quick sort: This method is invented by Hoare is considered to be a fast method to sort the elements. The method is also called partition exchange sorting. The method is based on divide and conquer technique. i.e., the entire list is divided into various partitions and sorting is applied again and again on the partition.

In this method, based on an element called pivot element the list is divided into two. Usually, the first element is considered to be the pivot element. Now move the pivot element to its correct position in the list. The elements to the left of the pivot element are less than pivot while the elements to the right of pivot are greater. The process is reapplied to each of these partitions till we get the sorted list of elements.

Algorithm for quick sort:

1. start
2. if lowerbound < upperbound repeat through steps 3 to 13 otherwise
   go to step 14
   begin
3. assign lowerbound to i, upperbound to j, i to pivot
4. if i<j repeat through steps 5 to 10 otherwise go to step
   begin
5. if a[i]<=k[pivot] and i< upperbound repeat through step 6 otherwise
   go to step 7
   begin
6. assign i+1 to i
   end if
7. if k[j] > k[pivot] repeat through step 8 otherwise go to step 9
   begin

8. assign j-1 to j
   end if
9. if i< j go to step 10 otherwise go to step 4
   begin
10. call function to swap k[i] and k[j]
   end if
   end if
11. call function to swap k[pivot] and k[j]
12. call function qsort(x,lowerbound,j-1)
13. call function qsort(x,j+1,upperbound)
   end if
14. stop

The time complexity of the quick sort algorithm is

In worst case = $O(n^2)$
In average case= $\log_2 n$
In best case = $\log_2 n$

Program :

```
#include<stdio.h>
main()
{
  int x[10],i,n;
  clrscr();
  printf("enter no of elements:");
  scanf("%d",&n);
  printf("enter %d elements:",n);
  for(i=1;i<=n;i++)
  scanf("%d",&x[i]);
  quicksort(x,1,n);
  printf("sorted elements are:");
  for(i=1;i<=n;i++)
  printf("%3d",x[i]);
  getch();
}
quicksort(int x[10],int first,int last)
{
  int pivot,i,j,t;
  if(first<last)
{
     pivot=first;
   i=first;
   j=last;
   while(i<j)
   {
```

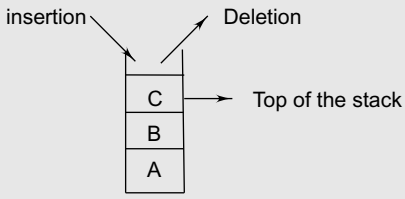
```
      while(x[i]<=x[pivot] && i<last)
       i++;
      while(x[j]>x[pivot])
      j--;
      if(i<j)
        {
              t=x[i];
              x[i]=x[j];
              x[j]=t;
       }
 }
 t=x[pivot];
 x[pivot]=x[j];
 x[j]=t;
 quicksort(x,first,j-1);
 quicksort(x,j+1,last);
 }
}
```

***** OUTPUT *****

enter no of elements:6
enter 6 elements:
23
12
45
34
21
87
sorted elements are: 12 21 23 34 45 87

7. What are the advantages and disadvantages of stack? Write a program to illustrate stack operation?

Ans: A stack is an ordered collection of items into which new items may be inserted and items may be deleted at one end, called top of the stack. A stack is also known as LIFO structure because the element which is inserted last is the first element to be deleted.



By the definition of stack, there is no upper limit on the number of items that can be kept in a stack. Pushing an element results in a larger collection of items in the stack and popping an element leads to a lesser collection of elements.

Representation of stack:

Advantages of stack:

1. To convert infix expression to postfix expression.
2. stack is used for recursion
3. Stack is used to evaluate postfix expressions.

Disadvantages:

1. In LIFO, if we want to delete the first element then all the elements are to be deleted.
2. As insertion and deletion are done from only one end, the other end is closed.

There are two operations that are performed in the stack. They are

1. **push( ):**
   Inserting an element into a stack is called as pushing operation
2. **pop ( ):**
   Deleting an element from the stack is called as popping operation.

The following program illustrates the operations in a stack

Program:

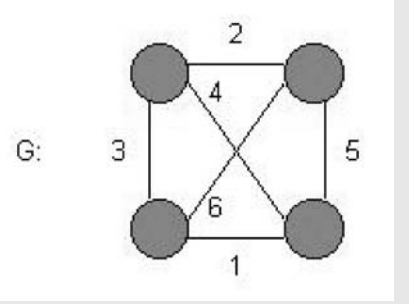
```
/* Write a 'C' program that implements stack and its operation by using the arrays */
# include <stdio.h>
# define size 4
int choice,top=0,a[size],item;
main()
{
clrscr();
while(1)
{
    printf(" *** MENU ***\n 1. PUSH\n 2. POP\n 3.
             TRAVERSE\n  4. EXIT\n");
    printf("enter your choice from menu:");
    scanf("%d",&choice);
    switch(choice)
    {
          case 1:push();
                 break;
          case 2:pop();
                 break;
          case 3:traverse();
                 break;
          case 4:exit();
          default:printf("wrong choice\n");
    }
}
```

```
getch();
}
push()
{
          if(size==top)
          printf("*** stack is full ***\n");
          else
          {
                printf("enter the item to be pushed into the stack:");
                scanf("%d",&item);
                top++;
                a[top]=item;
          }
}

pop()
{
          if(top==0)
          printf("*** stack is empty ***\n");
          else
          {
                item=a[top];
                top--;
                printf("the deleted item from stack is %d\n",item);
          }
traverse()
{
          int i;
          if(top==0)
          printf("**** stack is empty ****");
          else
          {
                printf("*** stack display ***\n");
                for(i=1;i<=top;i++)
                if(i==top)
                printf("%d at %d ->top\n",a[i],i);
                else
                printf("%d at %d\n",a[i],i);
          }
{
```

8. (a) What is a network?
(b) What is a spanning tree?
(c) Define minimal spanning tree.
(d) What are the various traversals in a tree?

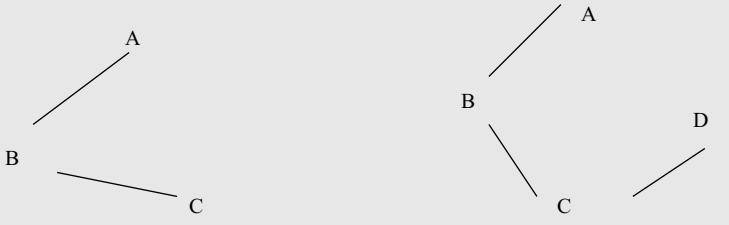
Ans: (a) A network means connection / communication between two mediums.
According to C language, a network means the connection between two or more nodes (or)

communication from one graph to another graph. Graph node is connected with another node called Edge. The following graph explains and shows how the network operation is performed.



(b) Spanning trees are usually represented as graphs. Hence, spanning trees are undirected trees, residing with only those edges which are sufficient to connect all the nodes with a tree. There are two important considerations regarding a spanning tree.

1. Each pair of nodes in the spanning trees should possess a single path.
2. The number of edges in the spanning tree is one less than the total number of nodes [if a tree consists of 'n' nodes, then the edges should be (n-1)].
3. Hence, any tree passing successfully through above-mentioned considerations can be termed as a spanning tree. Here are few examples of spanning trees.



There are three major concepts associated with each spanning tree. These are

1. depth first search spanning tree
2. breadth first search spanning tree
3. cost of spanning tree.

The spanning tree obtained by considering the depth first search mechanism is called as **depth first search spanning tree.** The spanning tree obtained by considering the breadth first search mechanism is called as **breadth first search spanning tree**. When there is a cost associated with each edge at the spanning tree, then the cost of spanning tree is nothing but the sum of costs of all the edges of the spanning tree.

Based on the cost features of spanning tree, it has been successfully applied in various studies related to electrical circuits and determining of shortest paths in the bulk routes of networks.

(c) The minimal spanning tree (MST) of a weighted graph is a spanning tree whose sum of edge weights is minimal. The minimum spanning tree describes the cheapest network to connect all of a given set of vertices.

Kruskal's algorithm for minimal spanning tree works by inserting edges in order of increasing cost (adding as edges to the tree those which connect two previously disjoint components).

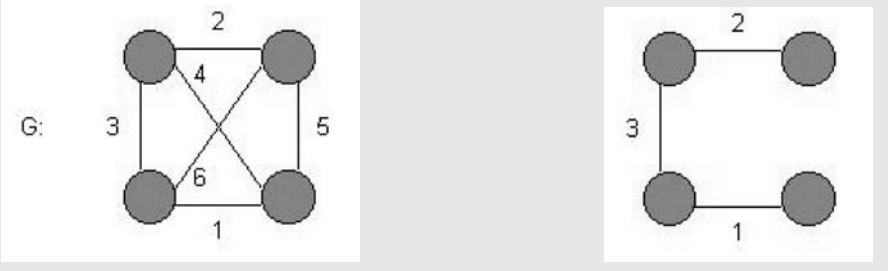
The minimal spanning tree (MST) of a graph defines the cheapest subset of edges that keeps the graph in one connected component. Telephone companies are particularly interested in minimal spanning trees, because the minimal spanning tree of a set of sites defines the wiring scheme that connects the sites using as little wire as possible. It is the mother of all network design problems.

Minimal spanning trees prove important for several reasons:

- They can be computed quickly and easily, and they create a sparse subgraph that reflects a lot about the original graph.
- They provide a way to identify clusters in sets of points. Deleting the long edges from a minimal spanning tree leaves connected components that define natural clusters in the data set.
- They can be used to give approximate solutions to hard problems such as Steiner tree and travelling salesman.
- As an educational tool, minimal spanning tree algorithms provide graphic evidence that greedy algorithms can give provably optimal solutions.

Example of minimal spanning tree

A weighted graph G **The minimal spanning tree from weighted graph G**



Using Kruskal's Algorithm, to find the mimimum cost of the tree.

MST Problem:

Given a connected graph G=(V,E) and a weight d:E->R+, find a minimum spanning tree T.

Kruskal's Algorithm

1. Set i=1 and let $E_0=\{\}$
2. Select an edge e_i of minimum value not in E_{i-1} such that T;=<E_{i-1} cup $\{e_i\}$ >is acyclic and define $E_i=E_{i-1}$ cup $\{e_i\}$. If no such edge exists, Let T=<E_i>and stop.
3. Replace i by i+1. Return to Step 2.

The time required by Kruskal's algorithm is O(|E|log|V|).

(d) Tree traversal are of three types:

1. In-order traversal
2. Pre-order traversal
3. Post-order traversal

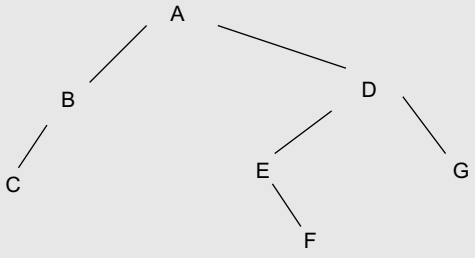
In-order traversal:

In-order traversal is given by the following steps.

1. traverse the left sub-tree
2. process the root node
3. traverse the right sub-tree

Algorithm for in-order traversal (recursive):

Step 1: check if tree is empty by verifying root pointer R. print tree empty if R=NULL
Step 2: if leftptr(R)!=NULL then call in-order (leftptr(R))
Step 3: print data(R).
Step 4: if rightptr(R)!= NULL then call in-order(rightptr(R))
Step 5: return.



So the **in-order** of the binary tree in the figure above is C—B—A—E—F—D—G.

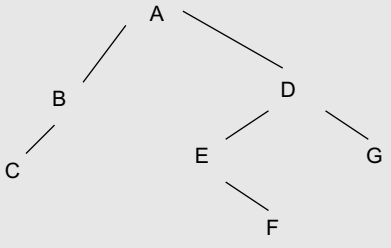
Pre-order traversal:

Pre-order traversal is defined as follows:

1. process the root node
2. traverse the left sub-tree in pre-order
3. traverse the right sub-tree in pre-order

Algorithm for pre-order traversal:

Step 1: check if tree is empty by verifying root pointer R
If R=NULLL, then print "tree empty"
Step 2: print data(R)
Step 3: if leftptr(R)!=NULL then call pre-order(leftptr(R))
Step 4: if rightptr(R)!=NULL call pre-order(rightptr(R))
Step 5: return

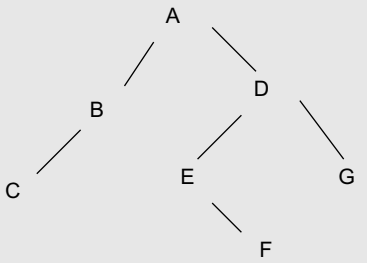


Pre-order traversal is A B C D E F G

Post-order traversal:

Post-order traversal is defined as follows:

1. traverse the left sub-tree
2. traverse the right sub-tree
3. process the root node



Post-order traversal for the above tree is C-B-F-E-G-D-A

C Programming and Data Structures
May/June 2008

SET- 2

1. (a) Explain the working of unary operator with example.
(b) Explain the working of binary operator with example.
(c) Explain the working of assignment operator with example.
(d) Explain the working of ternary operator with example.

Ans: (a) C supports a rich set of operators, such as +,-,=,*,& and <.

1. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.
2. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.
3. C operators can be classified into a number of categories. They include:
    1. arithmetic operators.
    2. relational operators.
    3. logical operators
    4. assignment operators
    5. increment or decrement operators.
    6. conditional operators
    7. bitwise operators
    8. special operators.

    C provides all the basic arithmetic operators. These operators are +, -, * and /. All work the same way as they do in other languages. They can operate on any built-in data type allowed in c.

Unary Operators

Unary operators include

1. addition or unary plus
2. subtraction or unary minus
3. increment operator (++)
4. decrement operator (--)
5. address operator
6. size operator

The following table summarizes these operators.

Operator	description or action
+	plus
–	minus
++	increment
--	decrement
&	address operator
Size of	gives the size of operator

The following is about the working of various unary operators:

1. Plus Operator (+):

This operator is used to find the algebraic sum of two operands.

Example: a + b
If a = 9,b = 11
Then a + b=20 i.e (9+11).
Here a, b are operands.
+ is unary operator.

Program:

/* write a program to calculate the sum of two operands */

```
#include<stdio.h>
#include<conio.h>

Main ()
 {
        Int a,b,s;
        Clrscr();
        Printf("enter the values of a,b:");
Scanf("%d%d",&a,&b);
S=a+b;
Printf("%d=sum",s);
Getch();
}
```

Output: enter the values of a,b
9,11
Sum =20.

2. Minus Operator

This operator is used to indicate change in the algebraic sign of a value.

Example:
Int x = -50;
Int y = -x;

This assigns the value of -50 to *x* and the value of 50 to *y* through *x*. The (-) sign used in this way is called unary operator because it takes only one operand.

Program:

/* write a program to subtract two numbers */

```
#include<stdio.h>
#include<conio.h>

Main ()
 {
        Int a,b,s;
        Clrscr();
         Printf("enter the values of a,b:");
Scanf("%d%d",&a,&b);
S=a-b;
Printf("%d=subtraction",s);
Getch();
}
```

Output: enter the values of a,b
11,9
Subtraction=2.

3. Increment Operator(++)

The operator ++ adds one to its operand whereas the operator--subtracts one from its operand.

Example:

X=x+1 can be written as x++

Consider, z=x*y++

Where x=20, y=10

In the above example, the current value of *y* is used for product and after performing the operation, the value of *y* is incremented.

Result is z = 20*10=200

Now consider, x=20, y=10

Z=x*++y

Here, first the value of *y* is incremented then it is used for the product.

Result is, z=20*11=220.

4. Sizeof and '&' Operator

The sizeof operator gives the bytes occupied by a variable. The number of bytes occupied varies from variable to variable.

The '&' operator prints the address of the variable in the memory.

The program below explains these operators.

```
#include<stdio.h>
#include<conio.h>
```

```
Main ()
  {
       Int x=2;
       Float y=2;
       Printf("sizeof (x)=%d bytes", size of(x));
       Printf("sizeof (y)=%f bytes",size of(y));
       Printf("\n address of x=%u and y=%u",&x,%y);
}
```

Output:

```
                  Size of(x)= 2
               Size of(y)= 4
               Address of x = 4066 and y=25096.
```

(b) Simple binary operators:

Expression:

Expression + expression
Expression - expression
Expression * expression
Expression / expression
Expression % expression
Expression == expression
Expression != expression
Expression < expression
Expression > expression
Expression <= expression
Expression >= expression
Expression && expression
Expression || expression
Expression & expression
Expression / expression
Expression ^ expression
Expression << expression
Expression >> expression

Example : p+-3 is same as = p-3

(c) Assignment operator:

Assignment operators are used to assign the result of an expression to a variable. We have seen the usual assignment operator '='.

In addition, C has a set of 'shorthand' assignment operators of the form

V op =exp;

Where 'V' is a variable, 'exp' is an expression and ‘op’ is a ‘C’ binary arithmetic operator.
The operator op= is known as the shorthand assignment operator.
The assignment statement

V op = exp;

Example:

X+=y+1;

This is same as the statement, x=x+(y+1);

Statement with simple assignment operator	statement with shorthand operator
A=a+1	a+=1
A=a-1	a-=1
A=a*(n+1)	a*=n+1
A=a/(n+1)	a/=n=1
A=a%b	a%=b

The use of shorthand assignment operators has three advantages:

1. What appears on the left hand side need not be repeated. Therefore, it becomes easier to write.
2. The statement is more concise and easier to red.
3. The statement is more efficient.

(d) The ternary operator is also called as conditional operator.

Syntax:

Exp1?exp2:exp3;

Where exp1, exp2 and exp3 are expressions.

Operation of Ternary Conditional Operator:

The value of exp1 is evaluated first. If it is true, value of exp2 is evaluated, otherwise exp3 is evaluated.

Example:

```
Int a=5,b=10,c=15;
Y=(a>b)?b:c;
```

In the above statement, the expression a>b is evaluated. Since it is false, value of c will be assigned to y.

So, the value of y will be 15.

Y=(a<b)?b:c;

Here a<b is true so value of b is assignment to y.

So, value of y will be 100.

Program:

```
/* Write a program to find the largest of three input numbers using ternary operator */
#include<stdio.h>
#include<conio.h>

Main ()
  {
       Int a,b,c,largest;
       Clrscr();
       Printf("enter the values of a,b,c:\n");
      Scanf("%d%d%d",&a,&b,&c);
```

```
    Largest=(a>b)?(a>c)?(b<c)? a:b:c;
    Printf("the largest number is: %d", largest);
    Getch();
}
```

2. What is the advantage of using functions. Write a 'C' program to explain about built-in functions with an example.

Ans: A function is a self-contained program segment that performs specific and well defined tasks. Use of functions ease the complex tasks by dividing the program into modules.

There are two functions in 'C' language

1. programmer defined functions.
2. library functions.

Advantages of using functions:

1. When programs are very lengthy, handling them becomes difficult. It is also difficult to debug and understand such lengthy programs. In order to overcome these drawbacks, complex programs can be subdivided into smaller modules. These modules are implemented using functions. Hence these subprograms or functions make programs easy to understand, debug and test.
2. The part of program which has to be repeated and used many times can be written as a function and can be called wherever needed to avoid repetition. This helps in reducing the size of the program.
3. A function can be shared by many programs.
4. It facilities top-down modular programming.

Before writing a program on built-in functions, let us know what is meant by built-in functions.

Built-in functions

1. Built-in functions are commonly required functions that are not grouped together and are stored in a library.
2. These functions are not modifiable.
3. These are not written by the programmers.
4. White space between the function names and parenthesis ( ) are ignored in case of built-in functions.
5. They are predefined.
6. Built-in functions cannot be overloaded.
7. Programmers overhead is reduced here, as they are available as 'ready to use' functions.
8. Already available for the programmers to call.
9. Their definitions appear in standard libraries.
10. Declarations of the built-in functions are not required
11. Example: cos(x), sqrt(x), sin(x), etc.,

The following is a C program to explain about built-in functions.

Program:

/* a 'C' program to find the square root of x */

```
#include<stdio.h>
#include<conio.h>
#include<math.h>

Void square(int);
    Main ()
       {
           Int x,y;
           Printf("enter the value of x");
           Scanf("%d",&x);
           Square(x);
           Y=sqrt(x);    /*  call to a built-in function   */
           Printf("%d",y);
       }
         Void square(int a)
           {
                Int p;
                P=a*a;
                Printf("%d",p);
            }
```

The built-in function sqrt(x) is available in the standard library header file "math.h", hence it must be included in the program.

3. What is the advantage of using arrays? Give the syntax for declaration, accessing and printing one-dimensional array.

Ans:

An array is a collective name given to a group of similar elements whereas a variable is any entity that may change during program execution.

The following are the advantages of using arrays.

1. An array is a variable that is capable of handling many values, whereas, an ordinary variable can hold a single value at a time. For example, an array initialization would be,

   int number[8];

   for which there are eight storage locations reserved where eight different values can be stored belonging to the same type. An ordinary variable initialization would be,

   int number;

   and only one storage location is reserved and can hold only one value at a time.
2. Elements of an array can be initialized at the time and place of their declaration. Consider an array with its elements.

   Int a[4] = {5,6,7,8};

In the above example, four elements are stored in an array *a*. The array elements are stored in contiguous memory locations. The array elements are read from '0'.i.e., *a*[0] is assigned the value '5'. Similarly, *a*[1] is assigned the value 6. The number 4 in *a*[4] represents the total number of elements.

The condition that needs to be satisfied is that all the elements of the array must be of the same datatype. Types of the declared array.

This is about the advantages of using arrays. The following is about the syntax for declaration, accessing and printing one-dimensional array.

One-dimensional Array

It is the simplest form of an array in which the list of elements are stored in contiguous memory locations and accessed using only one subscript.

Example:

```
Int abc[10];
```

Here abc is an array of 10 integers.

Declaration of one-dimensional array:

The general format of declaring array is as follows: dataType array_name [size];
where dataType is the data type of array elements, array_name is name of array variable and size specifies the number of elements in the array. The size of array is fixed at compile time and it cannot be changed during execution of program.

Example:

```
Int  a[50];
```

Here, array *a* can hold 50 integers.

The total size of the array 2*50=100 byte's. This is because each integer element requires 2 bytes for storage.

Initializing and accessing:

We can initialize array elements at the place of their declaration itself. The general format of this initialization is,

```
Type array_name [size]={list of elements separated by commas}
```

Example:

```
Int a[4]={5,10,1,55};
```

Here the size of this array is 4, 5 is assigned to first element, i.e., a[0]=5,10 is assigned to second element i.e., a[1]=10, 1 is assigned to third element, i.e., a[2]=1, 55 is assigned to fourth element a[3]=55.

Suppose the list of elements specified are less than the size of array then only that many elements are initialized. The remaining elements are assigned garbage values.

Example:

```
Int xyz[5]={10,5,20};
```

Here, the first element is assigned value 10, second element is assigned value 5 and third element is assigned value 20. Since, the list of elements is less than the size of the array, the remaining two elements are assigned garbage values. Index in general runs from 0 to n-1 i.e., for an array a[n], we have values from a[0],a[1],…..a[n-1].

Xyz[0], xyz[1]=5, xyz[2]=20,

Xyz[3] is assigned garbage value.
Xyz[4] is assigned garbage value.
Consider the following statement
Int xyz[ ] = [5,6,7,8,9];

In the above statement, we did not specify the size of the array. In such cases, compiler fixes the size of array based on number of values in the list. In the above statement, since we have five values in the list, size of the array is 5.

If we want to initialize elements of a large sized array, (suppose of 250 elements), then it is difficult to initialise such an array at the time of declaration. In such cases, declarations must be done explicitly. In 'C', we use loops (while, for, do-while) to initialise the elements. There is no short cut to initialize array of array size in C.

The following is a program on one-dimensional array.

/* Program showing one-dimensional array */

```
#include<stdio.h>
#include<conio.h>

      Main ( )
        {
             Int i;
            Float x[10], value, total;
      /* reading values into array */
       Printf("enter 10 real numbers\n");
        For(i=0;i<10;i++)
            {
              Scanf("%f",&value);
               X[i]=value;
             }
  /*   computation of total          */
                  Total = 0.0;
               For(i=0;i<10;i++)
                 Total= total+x[i]*x[i];
/* printing of x[i] values and total */
              Printf("\n");
               For(i=0;i<10;i++)
              Printf("x[%2d]=%5.2 f\n", i+1,x[i]);
               Printf("\n total =%2f \n", total);
      }
```

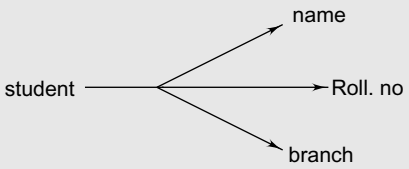
Output of 10 real numbers: 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9,10.10.

4. Define structure and write the general format for declaring and accessing members.

Ans: Structure:

When you want to access data consisting of different types of elements, it cannot be done using arrays, as array is a collection of similar data items. Suppose that you want to store information

regarding a student. Each student will have a name, roll number and branch of study. This entire information cannot be stored in a single array as student information consists of data of different types [name is a string, roll number is an integer and branch is a string]. To handle such string of data, C provides a derived data type known as structure.



Structure definition :

A structure is a collection of data items of different datatypes under a single name. Or, a structure is a collection of heterogeneous data items.

A structure is declared as follows

```
Struct < tag >
   {
     Data type member1;
     Data type member2;
     ……..
          ……..
          Data type member n;
                        };
```

* In this declaration struct is a keyword.
* <tag > is the name given to the structure declared.
* Data type can be integer, float, char or any other data types.
* member1, member2,…..member *n* are members of structures. Individual members can be variable, pointers, arrays or other structures.
* Individual members cannot be initialized within structure definition. After structure has been declared, individual structure type variable are declared as follows.
  Struct <tag> variable1,variable2……variable *n*;
* Variable 1, variable2,……variable *n* are structure variables of type tag.
* After closing bracket semicolon ';' must be specified.
* Members of structure themselves do not occupy any space. Memory is allocated only after declaring structure variables.

  Structure can be declared as,

```
Struct  tag
        {
            Type 1  data 1;
            Type 2 data 2;
```

```
                    ……….
                    ……….
                   Type n  data n;
                } variable1, variable2,……..variable n;
```

In the case of above declaration, tag is optional

Example:

```
Struct
   {
       Char name[26];
       Int age;
       Char gender[6];
       Char occupation[25];
    } p1,p2;
```

In the above declaration, there is no need to specify structure name.

Accessing members of structure:

Structure members can be accessed by writing variable name.member. Dot(.) operator links structure variable to the data member. It is the highest precedence operator and associatively is left to right.

Assigning values to structure:

One way is to assign values at the time of declarating structure variables.

```
Struct person p1={“ravi”,“21”,“male”,“student”}
```

Another way is to assign values to each member separately as shown below.

```
Strcpy(p1.name,“ravi”);
P1.age=21;
Strcpy(p1.gender,“male”);
Strcpy(p1.occupation,“student”);
```

Here is an example demonstrating the structure with initialized values.

/* structure with initialized values */

```
#include<stdio.h>
#include<conio.h>
Void main ( )
 {
    Struct organization
      {
          bb   Char    name[20];
              Char   designation[20];
            Int  sal;
       };
```

Struct organization emp1={“ramu”,“secretary”,“8000”};

Struct organization emp2={“raju”,“manager”,“18000”};

Printf("details of employee one =%c%c%d",emp1.name ,emp1.designation, emp1.sal);
Printf("details of employee one =%c%c%d",emp2.name ,emp2.designation, emp2.sal);
}.

5. Write a program to read the following data, to find the value of each item and display the contents of the file.

Item	code	price	quantity
Pen	101	Rs. 20	5
Pencil	103	Rs. 3	100

Ans:

Program:

```
#include<stdio.h>
#include<conio.h>
   Void main( )
     {
        Struct shop
          {
            Char itemname[30];
             int code;
             Float price;
             Int quantity[20] ;
                };
              Struct shop item1={"pen",101,20,5};
              Struct shop item2={"pencil",103,3,100};
              Printf("value of each item1(pen)
                is=%f",item1.price);
              Printf("value of each item2(pencil)
                is=%f",item2.price);
              Printf(display the contents of the file is
              item1=%c%d%f%d",item1.itemname,item1.code,
              item1.price,item1.quantity);
          }
```

6. Compare the advantage and disadvantage of Bubble, insertion and selection sort.

Ans:

Bubble sort (exchange sort) Bubble sort algorithm for sorting is one of the simplest algorithm, which organizes the sorting procedure by uniformly checking the two adjacent elements and accordingly adjusting their positions. It is inefficient because the number of iterations increases with the increase in number of elements to be sorted. Hence, if there are *n* elements to be sorted, then the number of iterations required are *n*-1. Bubble sort is the simplest and oldest sorting technique. This method takes two elements at a time. It compares these two elements. If first element is less than second one, the elements are left undisturbed. If the first element is greater than second one, then they are swapped. The procedure continues with the next two elements and ends when all the elements are sorted. It is to be noted that bubble sort is an inefficient algorithm. The order of bubble sort algorithm is O(n2).

Example :

1) 5 3 11 14 12
2) 3 5 11 14 12
3) 3 5 11 14 12
4) 3 5 11 14 12
5) 3 5 11 12 14

Program:

/* Program to sort array elements using bubble sort technique */

```
#include<stdio.h>
main()
{
int i,j,t,a[5],n;
clrscr();
printf("enter the range of array:");
scanf("%d",&n);
printf("enter elements into array:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
for(i=0;i<n-1;i++)
for(j=i+1;j<n;j++)
if(a[i]>a[j])
{
t=a[i];
a[i]=a[j];
a[j]=t;
}
printf("the sorted order is:");
for(i=0;i<n;i++)
printf("\t%d",a[i]);
getch();
}
```

* This sorting method requires many comparisons, thus take more time than quick sort, insertion sort, etc.

Insertion Sort

Insertion sort is similar to playing cards. To sort the cards in your hand, you extract a card, shift the remaining cards and then insert the extracted card in its correct place. The efficiency of insertion sort is O(n2).

As the name suggests, insertion sort is a technique, which inserts an element at an appropriate location by comparing each element with the corresponding elements present at its left and accordingly moves rest of the given array.

Consider insertion sort mechanism with an example

1) 9 6 14 2 1
2) 6 9 14 2 1
3) 6 9 14 2 1
4) 2 6 9 14 1
5) 1 2 6 9 14.

Program :

/* Write a C program that implements the insertion sort method */

```
#include<stdio.h>
main()
{
      int i,j,t,a[10],n,p=0;
     clrscr();
     printf("enter the range of array:");
     scanf("%d",&n);
     printf("enter elements into array:");
     for(i=0;i<n;i++)
     scanf("%d",&a[i]);
     for(i=1;i<n;i++)
     {
     t=a[i];
         for(p=i;p>0 && a[p-1]>t;p--)
         a[p]=a[p-1];
         a[p]=t;
     }
     printf("the sorted order is:");
     for(i=0;i<n;i++)
     printf("\t%d",a[i]);
     getch();
 }
```

Insertion sort requires more comparisons than quick sort but is easy to implement.

Selection Sort

A selection sort algorithm sorts the list by selecting the largest element in the unsorted list and places it at the appropriate position. In each pass the largest element in the list is placed at appropriate position. If there are *n* elements in the list, then

In the first pass all the elements are compared and largest element is placed at the last position.

In the second pass, first *n*-1 elements are compared and process continues for list of *n*-2 elements, then for *n*-3 elements and so on until first element.

If there are *n* elements, we require *n*-1 passes.

The time complexity of selection sort is $O(n^2)$ in all cases.

Program:

/* Write a 'C' program that implements the selection sort method */

```
#include<stdio.h>
#include<conio.h>
Void main()
    {
        Int a[20],I,j,temp,t,n,max;
         Clrscr();
 Temp=0;
 Printf("\n enter the no of elements in the list:");
Scanf("%d",&n);
Printf("\n enter the elements of the list\n");
For(i=0;i<n;i++)
Scanf("%d",&a[i]);
For(i=n;i>1;i--)
 {
        Max=0;
        For(j=1;j<I;j++)
         If(a[j]>a[max])
          Max=j;
         Temp=a[i-1];
         A[i-1]=a[max];
         A[max]=temp;
  }
Printf("\n elements in the ascending order are as follows:\n");
For(i=0;i<n;i++)
Printf("%d\n",a[i]);
Getch();
}
```

Advantage:

Selection sort minimizes data movement by putting one entry at its final position at every pass. Hence this algorithm is useful for continuous list of large elements where movement of entries is expensive.

Disadvantage:

If the elements are stored using linked lists, then insertion sort is more efficient than selection sort. It requires more comparisons than quick sort but is easy to implement than quick sort.

7. Define abstract data type. Explain with an example.

Ans: An abstract data type (ADT) is a general way to group and operate on data, independent of any particular language. Two 'classic' ADTs are List and Stack. These concepts exist as general

Computer Science. Sci. structures, and can be implemented in many languages. The key features of how they store and operate on data are defined by the ADTs, and implemented in code in various languages. In OO languages like Java, an ADT would typically be implemented by a class or group of classes. In non-OO languages like C, it would be implemented by a data structure and a set of functions to be called against it.

The examples of an abstract data type can be implemented in stack, queue, linked list, etc. Let us discuss one of these topics.

Stack:

A stack is a data structure in which addition of new element or deletion of an existing element always takes place at the same end. This end is called as top of stack. A stack is generally implemented with two basic operations. They are push and pop. Here, push operation allows adding an element at the top of the stack.

Pop operation allows removal of an element from the top of the stack. Before making use of stack in a program, it is very important to decide how to represent a stack. There are several ways of representing a stack such as by using arrays, by using pointers, by using linked list, etc.

The following program explains the stack by using single linked list.

Program:

/* Write a program to implement the stack operation by using single linked list*/

```
#include<malloc.h>
#include<stdio.h>
struct node
{
int info;
struct node *next;
};

typedef struct node *nodeptr;
push(nodeptr);
nodeptr pop(nodeptr);

main()
{
nodeptr stack;
int choice;
clrscr();
stack=NULL;

while(1)
{
printf("\n\t1\tpush\n\t2\tpop\n\t3\tdisplay\n\t4\tEXIT\n");
```

```
printf("ENTER YOUR CHOICE");
scanf("%d",&choice);

switch(choice)
{
case 1: stack = push(stack);
 break;
case 2: stack = pop(stack);
 break;
case 3: display(stack);
 break;
case 4: exit();
}

}
   getch();
}

push(nodeptr stack)
{
nodeptr temp;
int info;
printf("ENTER THE ELEMENT TO BE PUSHED");
scanf("%d",&info);
     temp=(nodeptr)malloc(sizeof(nodeptr));
     temp->info=info;
     temp->next=stack;
     stack=temp;
     return(stack);

display(stack);
}

display(nodeptr stack)
{
 printf("TOP");
 while(stack != NULL)
 {
     printf("%d",stack->info);
     stack=stack->next;
     printf("->");
 }
 printf("NULL\n");
}
nodeptr pop(nodeptr stack)
{
```

```
nodeptr temp;
if(stack == NULL)
printf("POP OPERATION IS NOT POSSIBLE");
else
{
temp=stack;
printf("THE POPED ITEM IS %d\n",stack->info);
printf("press any key to continue");
getchar();
stack=stack->next;
}
free(temp);
return(stack);
}

/*
INPUT / OUTPUT
   1  PUSH
   2  POP
   3  DISPLAY
   4  EXIT
ENTER YOUR CHOICE      1
ENTER THE ELEMENT TO BE PUSHED 34
   1  PUSH
   2  POP
   3  DISPLAY
   4  EXIT
ENTER YOUR CHOICE       3
TOP 34->NULL
   1  PUSH
   2  POP
   3  DISPLAY
   4  EXIT
ENTER YOUR CHOICE       2
THE POPED ITEM IS 34
PRESS ANY KEY TO CONTINUE
   1  PUSH
   2  POP
   3  DISPLAY
   4  EXIT
ENTER YOUR CHOICE       4
      EXIT.
```

8. (a) What is a network?

(b) What is a spanning tree ?

(c) Define minimal spanning tree.

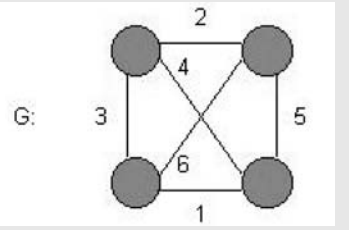
(d) What are the various traversals in a tree ?

Ans:

(a) What is a network ?

Ans: A network means connection / communication between two medias.

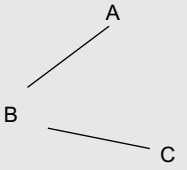
According to 'C' language, the network means the connection between two or more nodes (or) communication from one graph to another graph. Graph node is connected with another node called edge. The following graph explains and shows how the network operation is performed.

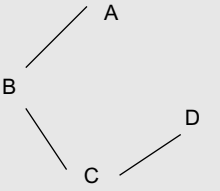


(b) What is a spanning tree ?

Ans: Spanning trees are usually represented as graphs. Hence, spanning trees are undirected trees, residing with only those edges which are sufficient to connect all the nodes with a tree. There are two important considerations regarding a spanning tree.

1. Each pair of nodes in the spanning trees should possess single path.
2. The number of edges in the spanning tree is one less than the total number of nodes [if a tree consists of 'n' nodes, then the edges should be(n-1)].
3. Hence any tree passing successfully through above mentioned considerations can be termed as spanning trees. Here are few examples of spanning trees.





There are three major concepts associated with each spanning tree. There are

1. Depth first search spanning tree
2. Breadth first search spanning tree
3. Cost of spanning tree.

The spanning tree obtained by considering the depth first search mechanism is called as **depth first search spanning tree**. The spanning tree obtained by considering the breadth first search mechanism is called as breadth first search spanning tree.

When there is a cost associated with each edge at the spanning tree, then cost of spanning tree is nothing but the sum of costs of all the edges of the spanning tree.

Based on the cost features of spanning tree, it has been successfully applied in various studies related to electrical circuits and determining of shortest paths in the bulk route of networks.

(c) Define minimal spanning tree.

Ans:

The minimum spanning tree (MST) of a weighted graph is a spanning tree whose sum of edge weights is minimal. The minimum spanning tree describes the cheapest network to connect all of a given set of vertices.

Kruskal's algorithm for minimum spanning tree works by inserting edges in order of increasing cost (adding as edges to the tree those which connect two previously disjoint components).

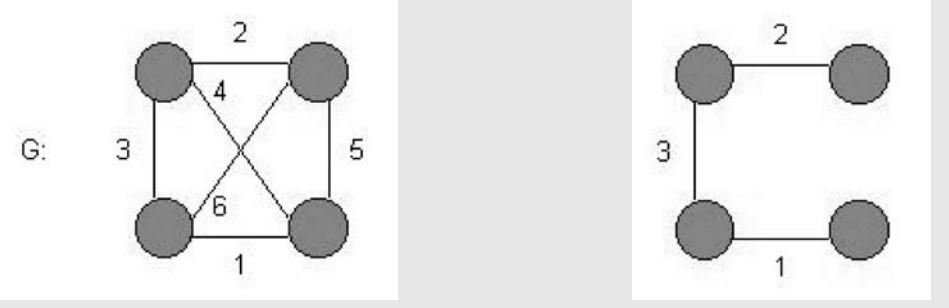
The minimum spanning tree (MST) of a graph defines the cheapest subset of edges that keeps the graph in one connected component. Telephone companies are particularly interested in minimum spanning trees, because the minimum spanning tree of a set of sites defines the wiring scheme that connects the sites using as little wire as possible. It is the mother of all network design problems.

Minimum spanning trees prove important for several reasons:

- They can be computed quickly and easily, and create a sparse subgraph that reflects a lot about the original graph.
- They provide a way to identify clusters in sets of points. Deleting the long edges from a minimum spanning tree leaves connected components that define natural clusters in the data set.
- They can be used to give approximate solutions to hard problems such as Steiner tree and traveling salesman.
- As an educational tool, minimum spanning tree algorithms provide graphic evidence that greedy algorithms can give provably optimal solutions.

Example of minimum spanning tree

A weighted graph G **The minimum spanning tree from weighted graph G**



Using Kruskal's Algorithm algorithm to find the mimimum cost of the tree.

MST Problem:

Given a connected graph G=(V,E) and a weight d:E->R+, find a minimum spanning tree T.

Kruskal's Algorithm

1. **Set i=1 and let E0={}**
2. **Select an edge ei of minimum value not in Ei-1 such that Ti=<Ei-1 cup {ei} >is acyclic**

and define Ei=Ei-1 cup {ei}. If no such edge exists, Let T=<Ei>and stop.

3. **Replace i by i+1. Return to Step 2.**

The time required by Kruskal's algorithm is O(|E|log|V|).

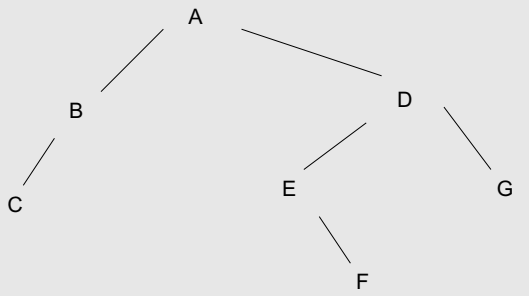
(d) What are the various traversals in a tree ?

Ans:

Tree traversal are of three types:

1. In-order traversal
2. Pre-order traversal
3. Post-order traversal

Example



In-order traversal:

In-order traversal is given by following steps.

1. Traverse the left sub tree
2. Process the root node
3. Traverse the right sub-tree

So the **in-order** of the binary tree in the figure above is C—B—A—E—F—D—G.

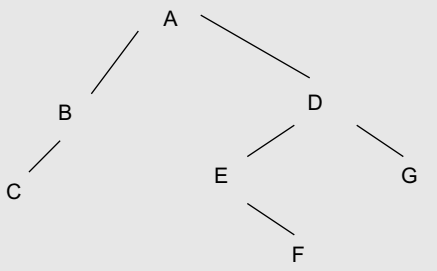
Pre-order traversal

Pre-order traversal is defined as follows:

1. Process the root node
2. Traverse the left sub-tree in pre-order
3. Traverse the right sub-tree in pre-order

Pre-order traversal is A B C D E F G

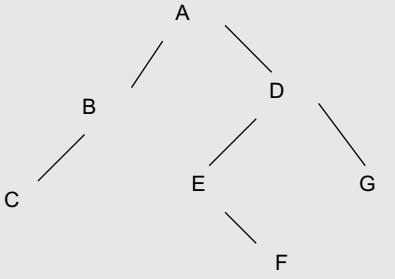
Example



Post-order traversal:

Post-order traversal is defined as follows:

1. Traverse the left sub-tree
2. Traverse the right sub-tree
3. Process the root node



Algorithm for post-order traversal :

Step 1: check if tree is empty by verifying root pointer R. print tree empty if R=NULL, then print "tree empty"
Step 2: if leftptr(R)!=NULL then call post-order (leftptr(R))
Step 3: print data(R).
Step 4: if rightptr(R)!= NULL then call post-order(rightptr(R))
Step 5: return.

C Programming and Data Structures
May/June 2008

SET- 3

1. What are different types of control statements available in 'C'. Explain them with an example ?

Ans: Control statements are also called as decision making statements. They are

1. If statement
2. Switch and break statements
3. Conditional operator statement
4. Go to statement

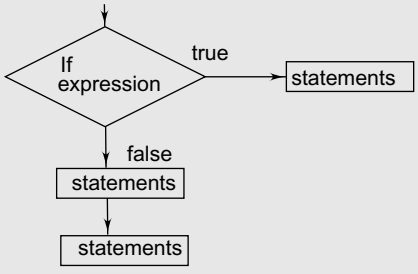
Let us discuss these statements in detail with examples.

If Statement

If statement is a powerful decision making statement and is called to control the flow of execution of statements. It is implemented in different forms

Simple If

```
General form:      if(expression)
                                          {
                     Statements ;
                              }
                     Statements;
```

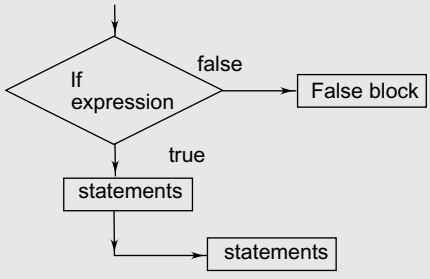


Example:

```
Main ()
    {
          Int a,b,c,d;
          float ratio;
          printf(enter the values");
          scanf("%d%d%d",&a,&b,&c);
          if(c-d!=0)
    {
          Ratio=(float(a+b)/float(c-d));
          Printf("ratio=%f",ratio);
    }
```

If Else Statement

```
General form:
            If(expression)
               {
                  True block
               }
            Else
                  {
                        False block
                  }
                  Statements;
```



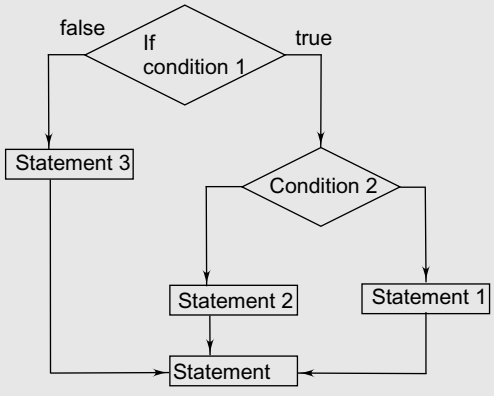
Example:

```
If(c-d!=0)
   {
        Ratio=float(a+b)/float(c-d);
        Printf("ratio=%f",ratio);
 }
        Else
        Printf("c-d is zero");
 }
```

Nested or if Else Statement

```
General form:

 If(condition 1)
   {
      If(condition 2)
      {
       Statement 1;
      }
        Else
           {
            Statement 2;
           }
             Else
               {
                Statement 3;
               }
                Statement x;
```

In nested if else series if decisions are involved. It has more than one if else statements.



Example:

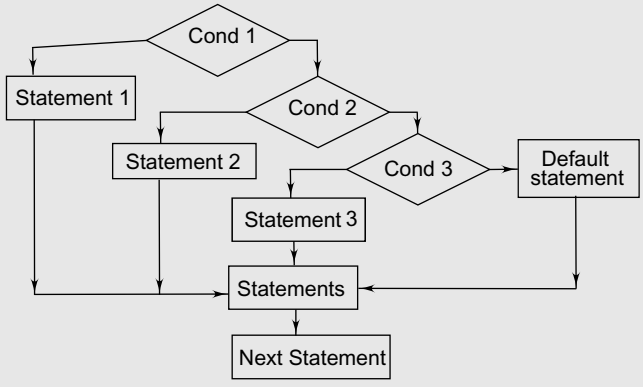
```
Main()
  {
       Float a,b,c;
       Printf("enter the values");
       Scanf("%F%f%f",&a,&b,&c);
       Printf("largest value is");
        If(a>b)
         {
           If(a>c)
```

```
            Printf("%f",a);
            Else
          Printf("%f",c);
        }
          Else
            {
                    If(c>b)
                     Printf("%f",c);
                          Else
                                Printf("%f",b);
            }
        }
```

Else if Lader

```
General form:
       If(condition 1)
          Statement 1;
        Else if(condition 2)
          Statement 2;
        Else if(condition 3)
          Statement 3;
        Else if(condition n)
          Statement n;
        Else
          Default statement;
          Statements;
```

Here, the conditions are evaluated from the top document cards.



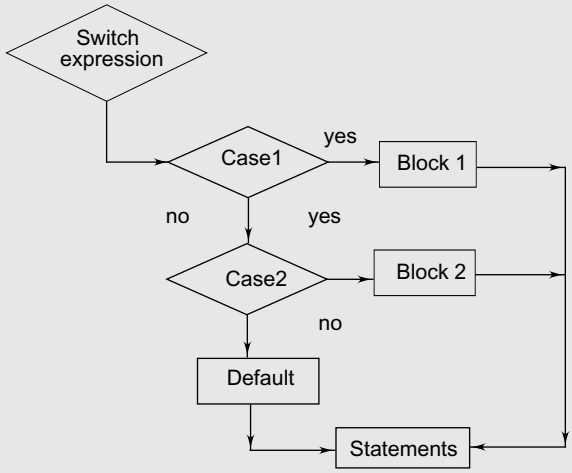
Example:

```
Main()
  {
   Int units, custno;
```

```
Float charges;
Printf("enter custno and units");
Scanf("%d%d",&custno,&units);
   If(units<=200)
      Charges=0.5 units;
   Else if(units<=400)
      Charges =100+0..6s(units-200);
   Else if(units<=600)
      Charges=390+units-600;
   Printf("custno %d", charges =%f", custno,charges);
        }
```

Switch and Break Statements

```
General form:
 Switch(expression)
    {
      Case 1:
                  Statements;
                  Break;
       Case 2 :
                 Statements;
                  Break;
       Default :
                  Statements:
                  Break;
    }
      Statements;
```



Example:

```
Switch(choice)
 {
          Case 1 ''+':
                   C=a+b;
                   Break;
          Case '-':
                   C=a-b;
                   Break;
          Default:
                   Printf("operation not possible");
 }
```

Conditional Operator Statement

```
General form:
  Conditional expression ? expression 1: expression 2
```

Example:

```
       Flag(x<0) ? 0:1;
            Salary= 4x+100     for x<40
                    300       for  x=40
                    4.5x+150  for  x>40
   The data can be written as
     Salary=(x!=40)?(x<40)?(4*x+100)L4.5*x+150):300;
```

Go to Statement

Go to requires a label in order to identify the place where the branch is to be made. A label is any valid variable name

Example:

```
Main ()
  {
    Double  x,y;
    Read:
             Scanf("%f",&x);
             If(x<0) goto read;
             Y=sqrt(x);
             Printf("%f%f",x,y);
             Go to read;
  }
```

2. (a) Write a brief note on auto and static storage classes?

Ans: Variables in C language are not only data type but are also of storage class that provides information about their location and visibility. The storage class decides the portion of the program within which the variables are recognised.

Example:

```
int m;
      main ()
      {
            Int I;
            Float balance;
       }
```

The variable *m* which has been declared before the main is called global variable. It can be used in all the functions in the program. It need not be declared in other functions. A global variable is also known as external variable. The variables *I* and *balance* are called local variables. They are visible only inside functions in which they are declared.

'C' provides a variety of storage class specifiers that can be used to declare explicitly the scope and lifetime of variables. The concept of scope and lifetime are important only in malfunctioning and multiple file programming.

Four storage class specifiers are **Auto**, **register**, **static**, and **extern**.

Auto: Local variable known only to the function in which if is declared, default is auto.

Declaration:

Auto int x;

Auto variables have garbage values unless they are initialized explicitly.

Static: Local variable which exists and retains its value even after the control is transferred to the calling function.

Declaration:

Static int x;

Static and extern variables are initialized automatically to zero.

(b) Write a short note on call by reference?

Ans: Functions are of four types

1. with argument with return
2. no argument with return
3. with argument no return
4. no argument no return

While using with arguments case, the arguments may be passed as direct values. Sometimes they are passed with their address. This type of passing address of variable is called **call by reference**.

Using the pointers in function, we can pass the address of variables in the functions.

Example:

```
Main()
  {
      Int x,y;
      X=100;
```

```
        Y=200;
         Printf("before exchange x:%d y: %d", x,y) ;
         Exchange(&x,&y);
         Printf("after exchange: x=%d y=%d",x,y);
}
         Exchange(a,b)
         Int *a,*b;
         {
               Int t;
               T=*a;
               *a=*b;
               *b=t;
         }
```

3. What is an array? What are the different types of arrays? Explain.

Ans: An array is a group of related data items that share a common name. All the elements should be of same data type.

Different types of arrays are:

1. one-dimensional arrays
2. two-dimensional arrays
3. multi-dimensional arrays

Let us discuss about them in detail.

One-dimensional arrays

A list of items can be given one variable name using only one subscript and such a variable is called a single subscripted variable or a one-dimensional array.

Accessing

Values can be initialized to arrays as three types

(i) direct :

```
N[0]=1
N[1]=2
N[2]=3
```

(ii) `n[3]={0,1,2};`

(iii) using for loop:

```
for(i=0;i<3;i++)
scanf("%d",&n[i]);
n[4]= n[0]+n[2];
n[6]= n[i]+3;
```

Declaration

General form is type variable name[size]

Example: float height[50];

Program

```
main()
 {
        Int I;
        Float x[10], value, total;
      Printf("enter 10 real no");
      For(i=0;i<10;i++)
      {
        Scanf("%f", &value);
        X[i]=value;
      }
        Total=0.0;
        For(i=0;i<10;i++)
        Printf(""x[%2d]=%st",i+1,x[i]);
        Printf("total=%2f",total);
}
```

Two-Dimensional Arrays

Initializing

They have elements in rows and columns.

Two-dimensional arrays may be initialized by following methods.

```
N[2] [3]={0,0,0,1,1,1,};
N[2][3]={{0,0,0},{1,1,1}};
        (OR)
N[2][3]={
            {0,0,0},
            {1,1,1},
          };
```

Declaration:

Type array name[row size][column size];

Program

```
/*To perform the matrix addition by using two-dimensional arrays */
Main ()
 {
 Int a[50][50], b[50][50],c[100][100],I,j,k;
  Printf("enter the elements");
```

```
    For(i=0;i<3;i++)
    For(j=0;j<3;j++)
    Scanf("%d",&a[i][j]);
    For(i=0;i<3;i++)
    For(j=0;j<3;j++)
    Scanf("%d",&b[i][j]);
    C[i][j]=0;
    For(i=0;i<3;i++)
    For(j=0;j<3;j++)
      {
        C[i][j]=a[i][j]+b[i][j];
         Printf("\n");
      }
    Printf("%d",c[i][j]);
}
```

Multi-Dimensional Arrays

Declaration

Type array name [s1] [s2]…..[sm];

Example: int survey[3][5][12];
Float table[5][4][3];

4. Define structure and write the general format for declaring and accessing members?

Ans: Structure can store data items in consequent memory locations. The advantage of structures is that the data items may be of different data types.

Declaring structures:

```
Struct student
    {
      Char name[20];
       Int stno;                                                    (1)
     };
```

Memory allocation is as follows:

```
Struct tagname
   {
     Data type n1;
     Datatype n2;
   };
```

In the case of (1) no memory allocation is not done, only the structure is defined for allocating.

```
Struct bokkbank
  {
        Char title[20];
        Char author[15];
        Int pages;
        Float price;
  };
Struct bokbankk bookk1,book2,bok3;
```

Tag name is optional.

Giving values to members:

The link between a member and a variable is established by using 'dot operator' or 'period operator'.

```
   Book1.pages=250;
  Book1.price=28.50;
```

Also initialized using scanf( ) statement

```
Scanf("%s",book1.title);
Scanf("%d",bok1.pages);
```

Structure initialization:

```
       Main ()
           {
              Static struct
           {
              Int weight;
              Float height;
                        }
             Student={60,180,75};
             -----------------
             -----------------
           }
```

Also can be initialized as

```
Main ()
  {
      Struct st_record
        {
            Int weight;
            Float height;
        };
      Static struct st_record st1={60,180,75};
      Static struct st_record st2={53,170,60};
      -------------
      -------------
```

```
      }
            (OR)

       Struct st_recrd
            {
                   Int weight;
                   Float height;
             }     st1={60,180,75};
    Main (0
       {
              Static struct st_recrd st2={53,170,60};
             ---------------
             ----------------
         }
```

Accessing members of structure:

Structure members can be accessed by writing variable name.member

Dot(.) operator links structure variable to the data member. It is the highest precedence operator and associatively is left to right.

Assigning values to structure:

One way is to assign values at the time of declarating structure variables.

```
Struct person p1={"ravi", "21", "male", "student"}
```

Another way is to assign values to each member separately as shown below.

```
Strcpy(p1.name, "ravi");
P1.age=21;
Strcpy(p1.gender, "male");
Strcpy(p1.occupation, "student");
```

Here is an example demonstrating the structure with initialized values.

```
/* structure with initialized values */

#include<stdio.h>
#include<conio.h>
Void main ( )
{
       Struct organization
         {
           bb Char name[20];
          Char designation[20];
   Int sal;
};
Struct organization emp1={"ramu", "secretary", "8000"};
Struct organization emp2={"raju", "manager", "18000"};
```

```
Printf("details of employee one =%c%c%d",emp1.name ,emp1.designation,
emp1.sal);
Printf("details of employee one =%c%c%d",emp2.name ,emp2.designation,
emp2.sal);
}
```

5. What is purpose of library function feof()? How is feof() utilized within a program that updates an unformatted data file. Explain.

Ans: Feof ( ) is a library function which represents end of file. It is an error handling function. It gives the end of data in a file. The file pointer moves by one character position for every operation of **getc** or **putc**. The **getc** will return an end of file marker **eof**, when end of the file has been reached. Therefore, the reading should be terminated when **eof** is encountered.

Program

```
#include<stdio.h>
Main()
     {
        FILE *f1;
        Char c;
        Printf("data input");
         F1=fpen("input file", "w");
        While(c=getchar()!=eof)
         Putc(,f1);
        Fclose(f1);
        Printf("data utput");
        F1=fpen("input",: "r");
         While(c=getc(f1)!=eof)
        Printf("%c",c);
         Fclose(f1);
}
```

The file input is reopened for reading. The program then reads its context character by character and displays it on the screen. Reading is terminated when **gets** encounters the end of file of mark. Like **scanf**, **fscanf** also returns the numbers of items that are successfully read. When the end of the file is reached, it returns the value **eof**.

6. Write a program to sort the elements whose worst and average case is o(n log n)?

Ans: The sorting technique which has worst and average case o(n log n) is merge sort.

The merge sort is the sorting technique used to merge two sorted lists. If sorted lists are not given, first lists are to be sorted individually using any sorting techniques.

Of all sorting techniques, quick sort is most efficient for sorting purpose.

Program

```
#include <stdio.h>
```

```
#include <conio.h>

#define MAX_ARY 10

void merge_sort(int x[], int end, int start);
       int main(void) {
       int ary[MAX_ARY];
       int j = 0;

       printf("\n\nEnter the elements to be sorted: \n");
       for(j=0;j<MAX_ARY;j++)
         scanf("%d",&ary[j]);

       /* array before mergesort */
       printf("Before    :");
       for(j = 0; j < MAX_ARY; j++)
         printf(" %d", ary[j]);

       printf("\n");
       merge_sort(ary, 0, MAX_ARY - 1);

       /* array after mergesort */
       printf("After Merge Sort :");
       for(j = 0; j < MAX_ARY; j++)
         printf(" %d", ary[j]);

       printf("\n");
       getch();
         }

/* Method to implement Merge Sort*/
void merge_sort(int x[], int end, int start) {

                                    int j = 0;
                                    const int size = start - end + 1;
                                    int mid  = 0;
                                    int mrg1 = 0;
                                    int mrg2 = 0;
                                    int executing[MAX_ARY];

                                    if(end == start)
                                    return;
                                    mid = (end + start) / 2;
```

```
                        merge_sort(x, end, mid);
                        merge_sort(x, mid + 1, start);

                        for(j = 0; j < size; j++)
                        executing[j] = x[end + j];

                        mrg1 = 0;
                        mrg2 = mid - end + 1;

                        for(j = 0; j < size; j++) {
                        if(mrg2 <= start - end)
                          if(mrg1 <= mid - end)
                          if(executing[mrg1] > executing[mrg2])
                          x[j + end] = executing[mrg2++];
                        else
                          x[j + end] = executing[mrg1++];
                        else
                          x[j + end] = executing[mrg2++];
                        else
                          x[j + end] = executing[mrg1++];
          }
}
```

If the given two lists are not sorted lists, sort the lists using quick sort.

Program

```
        #include<stdio.h>
        main()
        {
          int x[10],i,n;
          clrscr();
          printf("enter no of elements:");
          scanf("%d",&n);
          printf("enter %d elements:",n);
          for(i=1;i<=n;i++)
          scanf("%d",&x[i]);
          quicksort(x,1,n);
          printf("sorted elements are:");
          for(i=1;i<=n;i++)
          printf("%3d",x[i]);
          getch();
        }
          quicksort(int x[10],int first,int last)
```

```
{
  int pivot,i,j,t;
  if(first<last)
    {
      pivot=first;
      i=first;
      j=last;
      while(i<j)
      {
      while(x[i]<=x[pivot] && i<last)
      i++;
      while(x[j]>x[pivot])
      j--;
      if(i<j)
      {
      t=x[i];
      x[i]=x[j];
      x[j]=t;
      }
      }
 t=x[pivot];
 x[pivot]=x[j];
 x[j]=t;
 quicksort(x,first,j-1);
 quicksort(x,j+1,last);
}
 }
```

Merge Sort Process

A = {56,78};
B = {45,67,89} two sorted lists.
C = {45,56,67,78,89}

A	56,	78		
B	45,	67,	89	
C	45	56		

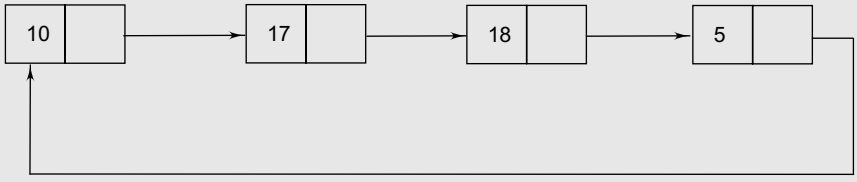
A	56,	78		
B	45,	67,	89	
C	45	56		
A	56	78		
B	45	67	89	
C	45	56	67	

A	56	78		
B	45	67	89	
C	45	56	67	78

C = {45,56,67,78,89} is the final sorted list using merge sort.

7. Give an algorithm/C program to reverse a singly linked circular list in place.

Ans: Circular linked list: The small change to the structure of a linear list such that the link field in the last node contains a pointer back to the first node rather than a NULL. Such a list is called a circular linked list.



Here is the routine which reverses circular list.

```
typedef struct node Node;

void reverse(Node** headRef)
{
 Node* result = NULL;
 Node* current = *headRef;
 Node* next;

 while (current != NULL)
 {
       next = current->next; // tricky: note the next node
       current->next = result; // move the node onto the result
       result = current;
       current = next;
       if (current == *headRef)
 {
       //met the beginning node...
       break;
    }
  }
  (*headRef)->next = result;
  *headRef = result;
}
```

Program

```
#include<conio.h>
#include<stdio.h>
struct node
{
int data;
struct node *next;
};
struct node *root;
void addnode()
{
int item;
struct node *newnode,*p;
printf("\n Enter the item");
scanf("%d", &item);
newnode=(struct node*) malloc(sizeof(struct node));
newnode->data=item;
newnode->next=NULL;
if(root==NULL)
{
root=newnode;
return;
}
____ _________      ________
p=root;
while(p->next!=NULL)
p=p->next;
p->next=newnode;
}
void display()
{
struct node*p=root;
while(p!=NULL)
{
printf("\n %d ",p->data);
p=p->next;
}
}
void reverse()
{
struct node *p,*oldnode=NULL,*newnode;
p=root;
while(p!=NULL)
```

```
{
newnode=p;
p=p->next;
if(oldnode==NULL)
{
oldnode=newnode;
oldnode->next=NULL;
}
else
{
newnode->next=oldnode;
oldnode=newnode;
}
}
root=oldnode;
}
void main()
{
int i;
clrscr();
for(i=1;i<=10;i++)
{
addnode();
}
printf("\nbefore reverse");
display();
printf("\n After reverse");
reverse();
display();
return 0;
}
```

8. Write an algorithm to perform deletion operation in a binary tree search?

Ans: There are four possible cases that we need to consider

(a) No node in the tree contains the specified data

(b) The node containing the data has no children

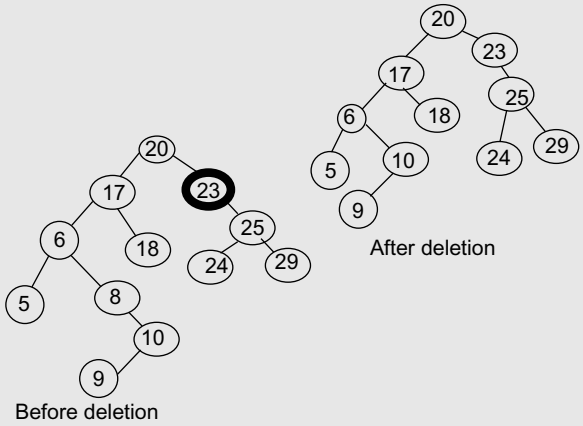
(c) The node containing data has exactly one child

(d) The node containing data has two children

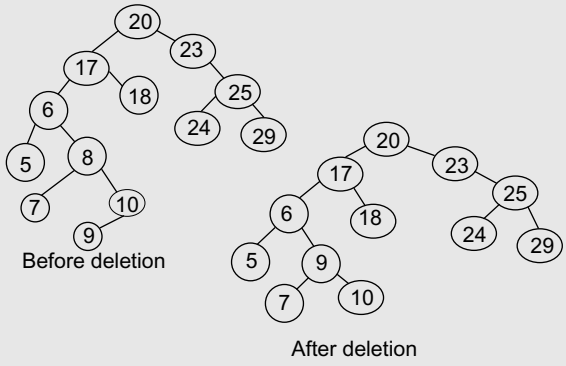
Case a: We merely need to print the message that data item is not present in the tree.

Case b: Since the node to be deleted has no children, the memory occupied by this should be freed and either the left link or right link of parent of this node should be set to NULL. Which of these to set to NULLL, depends upon whether the node being deleted is a left child or a right child of its parent.

Case c: Since the node to be deleted has one child, the solution is again rather simple. We have to adjust the pointer of the parent of the node to be deleted, such that after deletion it points to child of node being deleted.



Case d: Since the node to be deleted has two children, the solution is more complex. The whole logic of deleting a node with two children is to locate the in-order successor, copy its data and reduce the problem to a simple deletion of a node with one or zero child.



Algorithm to perform deletion operation in a binary search tree.
ALGORITHM:

Step 1: Start

Step 2: Define a structure btree

Step 3: Type define struct btree as node

Step 4: While(tree), begin

Step 5: Print MENU

Step 6: Write your choice

Step 7: If choice=1

Step 8: Enter your no of nodes

Step 9: Read nodes n

Step 10: for i=1 to n in steps of 1 do

Step 11: Print enter item

Step 12: Read item

Step 13: Root =call create (root, item).end for

Step 14: if choice=2

Step 15: Read element to be deleted

Step 16: Call delete(root, item) end for

Step 17: If choice=3

Step 18: Call preorder(root)

Step 19: Call postorder(root)

Step 20: Call inorder(root)

Step 21: Break, end of switch

Step 22: Stop

For insert function

Step 1: Start

Step 2: If t= null

Step 3: Allocate memory to temp

Step 4: Temp->data =item

Step 5: Temp-> lc=null

Step 6: Temp->rc=null

Step 7: return t to main and end

Step 8: If item< (l->data)

Step 9: T->lc=call insert(e->lc, t)

Step 10: T->rc=call insert(e->lc,t)

Step 11: Return t

Step 12: Stop

For ***DELETION*** function

Step 1: Start

Step 2: x=d

Step 3: while x!=null

Step 4: If x->data =t

Strep5: Break

Step 6: Parent =x

Step 7: if t<x->data

Step 8: t=t→lc

Step 9: t=l→rc

Step 10: If x→lc!=null &&x→rc!=null

Step 11: parent =x

Step 12: If parent==null

Step 13: parent→lc==null

Step 14: parent→rc==null

Step 15: If p->lc=x->rc

Step 16: If p->rc=x->rc

Step 17: While insert->lc=null

Step 18: Insert=insert->la

Step 19: x->data=insert->data

Step 20: x=insert

Step 21: Return d

Step 22: Stop

Processing to perform deletion operation in a binary search tree.

Binary tree before deletion
7 8 9 10 11 12 13 14 15
Delete element 10
Binary tree after deletion
7 8 9 11 12 13 14 15

C Programming and Data Structures
May/June 2008

SET- 4

1. (a) Define an Algorithm.

Ans: In solving a problem or to develop a program we should have two criteria.

1. algorithm
2. flow chart

Algorithm: An algorithm is a method of representing the step-by-step procedure for solving a problem. An algorithm is very useful for finding the right answer to problem or to a difficult problem by breaking the problem into simple cases.

Algorithm for in-order traversal (recursive):

Step 1: check if tree is empty by verifying root pointer R. Print tree empty if R=NULL
Step 2: if leftptr(R)!=NULL then call in-order (leftptr(R))
Step 3: print data(R).
Step 4: if rightptr(R)!= NULL then call in-order(rightptr(R))
Step 5: return.

(b) What is the use of flow chart?

Ans: Flow chart is a diagrammatic representation of an algorithm. It is built using different types of boxes and symbols. The operation to be performed is written in the box. All symbols are inter-connected and processed to indicate the flow of information and processing.

Uses of flow chart:

1. Flow chart helps us to understand the program easily.
2. As different boxes are used to specify the type of operation performed, it helps in easy understanding of complex programs.

(c) What are the different steps followed in program development ?

Ans: A 'C ' program has to pass through many phases for its successful execution and to achieve desired output.

The various steps involved in program development are as follows

1. Editing phase
2. Compilation phase
3. Linking phase
4. Execution phase

Editing Phase

In the editing phase, a C program is entered into a file through a text editor. Unix operating system provides a text editor which can be accessed through the Command 'Vi' Modern, compiler vendors provide Integrated Development Environment IDEs. The file is saved on the disk with an extension of 'C'. Corrections in the program at later stages are done through these editors. Once the program has been written, it requires to be translated into machine language.

Compilation Phase

This phase is carried out by a program called as compiler. Compiler translates the source code into the object code. The compilation phase cannot be proceeded successfully until and unless the source code is error-free. The compiler generates messages if it encounters syntactic errors in the source code. The error-free source code is translated into object code and stored in a separate file with an extension '.obj'.

Linking Phase

In this phase, the linker links all the files and functions with the object code under execution. For example, if a user uses a 'printf' function in his\ her program, the linker links the user program's apostrophe object code with the object code of the printf function. Now the object code is ready for next phase.

Execution Phase

In this phase, the executable object code is loaded into the memory and the program execution begins. We may encounter errors during the execution phase even though compilation phase is successful. The errors may be run-time errors and logical errors.

2. Write a program to find the sum of a given series by using function with argument and return value e= 2+3/1!-6/2!+9/3!-12/4!.....!

Ans:

Program:

```
#include<stdio.h>
#include<math.h>
long fact(int);
```

```
void main()
  {
        int x,i,n;
          float s=0,c;
             clrscr();
        printf("\n enter the value of x\t");
        scanf("%d",&x);
        /*perform the looping operation*/
          for(i=0,n=0;i<=n;i=i+2,n++)
        s=s+(pow(-1,n)*pow(x,i)/fact(i));
        printf("\n the result is %f",s);
         getch();
 }
/* calling sub program*/
long fact(int x)
{
long int y=1;
while(x!=0)
{
       y=y*x;
       x--;
}
 return y;
```

3. Write a program and explain the working of malloc() and calloc() functions.

Ans: Memory space is required by the variable in a program to calculate, and assign during execution. The process of allocating memory out runtime is known as dynamic memory allocation. There are some memory management functions in C that can be used for allocating and freeing memory during program execution.

Some memory allocation functions are

Malloc()	Allocates required size of bytes and returns a pointer to the first byte of allocated space.
Calloc()	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
Free()	Free previously allocated space.
Realloc	Modifies the size of previously allocated space.

Memory allocation using malloc() function:

Block of memory may be allocated using malloc() function. Malloc() function reserves a block of memory of specified size and returns a pointer of typed void.

Syntax:-

```
Ptr=(cast-type#)malloc(byte size);
 Ptr- pointer of cast type.
```

Malloc returns a pointer (of cast type) to an area of memory with the size of given bytes.

Example:- y=(int*)malloc(100*size of(int));

Malloc allocates a block of contiguous bytes. The allocation can fail if the space in the heap is not sufficient. If it fails, it returns null.

Example program that uses a table of integers whose size will be specified interactively at run time.

```
#include<stdio.h>
#define null 0
Main()
{
 int *p,*table;
 int size;
printf("enter size of table");
scanf("%d",&size);
printf("\n");
if(table=(int*)malloc(size*sizel of(int)==null)
{
printf("no space available");
exit(1);
printf("address of first byte is %u",table);
printf("input table values");
for(p=table;p<table+size;p++)
scanf("%d",&p);
for(p=table+size-1;p>=table;p--)
Printf("%d is stored address %u",*p,p);
 }
```

Memory allocation using calloc() function:-

Calloc() is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. Malloc allocates a single block of storage, each of them, being of the same size, and then sets all bytes to zero. The general form of calloc() is

```
Ptr=(cast_type*)calloc(n, elem_size);
```

The above statement allocates contiguous space for n blocks, each of size elem_size bytes. All bytes are initialized to zero and a pointer to first byte of the allocated region is returned. If there is not enough space a null pointer is returned. Calloc() allocates blocks of memory.

Calloc takes two arguments which are number of elements to allocated memory and size in bytes of a single variable.
Calloc initializes the allocated memory to zero.

```
#include<stdio.h>
include<stdio.h>
#define null 0
Main()
{
  Char *buffer;
If(buffer=(char*)malloc(10))==null)
{
printf("malloc faile");
exit(1);
}
Printf("buffer of size %deated \n",msize(buffer));
Strcpy(buffer,"hyderabad");
Printf("buffer contains%s",buffer);
If(buffer=(char*)realloc(buffer,15)=!null)
{
 Printf("reallocation failed");
Exit(1);
}
Printf("buffer size modified");
Printf("buffer still contains %s \n",buffer);
Strcpy("buffer", "secunderabad");
Printf("\n buffer now contains %s \n",buffer);
Free(buffer);
}
```

4 . Define structure and write the general format for declaring and accessing members.

Ans. Structure can store data items in consequent memory locations. The advantage of structures is that the data items may be of different data types.

Declaring structures:

```
Struct student
     {
         Char name[20];
          Int stno;     (1)
       };
```

```
Memory allocation is as follows:
        Struct tagname
           {
               Data type n1;
               Data type  n2;
            };
```

With (1) no memory allocation is not done only the structure is defined for allocating.

```
       Struct   bokkbank
              {
                     Char title[20];
                     Char author[15];
                     Int pages;
                     Float price;
                  };
```

Struct bokbankk bookk1,book2,bok3;

Tag name is optional.

Giving values to members:

The link between a member and a variable is established by using 'dot operator' or 'period operator'.

Book1.pages=250;

Book1.price=28.50;

Also initialized using scanf( ) statement

Scanf("%s",book1.title);

Scanf("%d",bok1.pages);

Structure initialization:

```
Main ()
     {
       Static struct
          {
                  Int weight;
                  Float height;
           }
      Student={60,180,75};
       -----------------
       -----------------
     }
```

Also can be initialized as

```
    Main ()
```

```
{
    Struct st_record
     {
         Int weight;
         Float height;
      };
   Static struct st_record st1={60,180,75};
   Static struct st_record st2={53,170,60};
    ------------
------------
 }
     (OR)

 Struct st_recrd
   {
        Int weight;
        Float height;
   } st1={60,180,75};
Main (0
  {
        Static struct st_recrd st2={53,170,60};
        ---------------
        ----------------
    }
```

Accessing members of structure:

Structure members can be accessed by writing variable name.member

Dot(.) operator links structure variable to the data member. It is the highest precedence operator and associatively is left to right.

Assigning values to structure:

One way is to assign values at the time of declaring structure variables.

```
Struct person p1={"ravi", "21", "male", "student"}
```

Another way is to assign values to each member separately as shown below.

```
Strcpy(p1.name, "ravi");
P1.age=21;
Strcpy(p1.gender, "male");
Strcpy(p1.occupation, "student");
```

Here is an example demonstrating the structure with initialized values.

```
/* structure with initialized values */
#include<stdio.h>
#include<conio.h>
Void main ( )
 {
```

```
Struct organization
  {
        bb    Char     name[20];
     Char    designation[20];
        Int   sal;
  };
Struct organization emp1={"ramu", "secretary", "8000"};
Struct organization emp2={"raju", "manager", "18000"};
Printf("details of employee one =%c%c%d",emp1.name ,emp1.designation, emp1.sal);
Printf("details of employee one =%c%c%d",emp2.name ,emp2.designation, emp2.sal);
 }
```

5. Describe various types of files with an example for each.

Ans: File: A file is a collection of records. Each record provides an information to the user. These files are arranged on the disk and can be accessed through file handling information provided by the standard 'C' library.

Basic operations on files

The files consists of large amount of data which can be read or modified depending upon the requirement. The basic operations that are performed on files are

1. Opening a file
2. Reading or writing a file
3. Closing a file

These operations are carried out with the help of a structure FILE (stdio.h file) and a file pointer.

There are two types of files. They are

1. Sequential file
2. Random access file

1. Sequential file:

In a sequential file system, records are stored in a serial order and retrieved in the same order. If a read operation is to be performed on the last record, then all the records before the last record need to be read because of which a lot of time is wasted.

2. Random access file:

In sequential access of file, data is read from or written sequentially i.e., from beginning of the file to end of the file in order. It is achieved using functions fprintf( ), fscanf( ), getc( ), putc( ) etc. In random access of a file, a particular part of a file can be accessed irrespective of other parts.

Random access of files is achieved with the help of functions fseek( ), ftell( )
and rewind ( ) which are available in I/O library. The functionality of each of these functions are given below.

(a) ftell( ):

n= ftell(fp);

Here 'n' gives the current position of the pointer from beginning. It gives the information regarding number of bytes to be read or written already.

(b) rewind( )

This function takes file pointer to the starting of the file, and resets it.

Syntax:

Rewind(fp);

This statement rewinds the file pointer fp to the beginning of the file.

(c) fseek( ):

fseek( ) is a file function that is used to move the file pointer to a desired location in the file. It is generally used when a particular part of the file is to be accessed. The main purpose of the file fseek( ) is to position the file pointer to any location on the stream.

Syntax:

Fseek(fileptr, offset, position);

'fileptr' is a file pointer

Offset:

It specifies the number or variable of type long to reposition the pointer forward and negative integer is used to move the pointer backward.

Position:

It specifies the current position.

The values that can be assigned to position are

Value	position
0	beginning of file
1	current position
2	end of file (EOF)

The fseek( ) returns zero, if all the operations are performed successful and returns -1 if an error occurs such as EOF is marked.

Example:

Program:

```
#include<stdio.h>
#include<conio.h>
 Main ( )
    {
       FILE *fp;
       Long int n;
```

```
Char c, fname[25];
Printf("enter the file name:\n");
Getc(fname);
Fp=fopen(fname,'w');
While(c=getchar( )!=EOF)
Putc(c,fp);
Fclose(fp);
Fp=fopen(fname,"r");
N=0;
While(!feof(fp))
  {
      Fseek(fp,n,0);
        Printf("position of %c is %d\n",
       getc(fp),ftell(fp));
             Fclose(fp);
  }
```

6. Write a program to explain selection sort. Which type of technique does it belong to?

Ans: **Selection sort**: In sorting technique, sorting is done by selecting the largest element in the unsorted list and placing it at the appropriate position (by swapping).

In each pass, the largest element in the list is placed at an appropriate position. If there are 'n' elements in the list.

→ in the first pass, all the elements are compared and largest element is placed at the last position.

→ in second pass, first n–1 elements are compared and process continues for a list of n–z elements, then for n–3 elements and so on until first element.
If there are 'n' elements, we require n-1 passes.

Example:

Consider a list of elements 12, 21, 50, 14, 2, 20.

	12	21	50	14	2	20
Pass 1:	12	21	20	14	2	50
Pass 2:	12	2	20	14	21	50
Pass 3:	12	2	14	20	21	50
Pass 4:	12	2	14	20	21	50
Pass 5:	2	12	14	20	21	50

In pass 1, the largest element is obtained by comparing from first element onwards. Here the largest element is 50. So, the positions of 50 and n^{th} element are interchanged as shown.
In pass 2, the first n-1 elements are compared and the largest element is obtained which is 21. So, the position of 21 and $(n–1)^{th}$ element are interchanged.

The process is repeated until all the elements are sorted.

Algorithm for selection sort:

Step 1: start.
Step 2: repeat through step 5 for i=n-1 to '0' do.
Step 3: [initializing]
Large=a[0]
Index=0
Step 4: repeat through step for j=1 to I do.
Step 5: [obtain the largest element in a pass]
If a[j] > large
Large=a[j]
Index=j;
Step 6: [placing the largest element in its proper position]
a[index] = a[i]
a[i] = large
end

C program for selection sort

Let us assume that an array named elements[] will hold the array to be sorted and maxsize is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature.

```
#include <stdlib.h>
#include <stdio.h>
#define MAXSIZE 500
void selection(int elements[], int maxsize);
int elements[MAXSIZE],maxsize;
int main()
{
int i;
printf("\nHow many elements you want to sort:");
scanf("%d",&maxsize);
printf("\nEnter the values one by one: ");
for (i = 0; i < maxsize; i++)
{
printf ("\nEnter element %i :",i);
scanf("%d",&elements[i]);
}
```

```
printf("\nArray before sorting:\n");
for (i = 0; i < maxsize; i++)
printf("[%i], ",elements[i]);
printf ("\n");
selection(elements, maxsize);
printf("\nArray after sorting:\n");
for (i = 0; i < maxsize; i++)
printf("[%i], ", elements[i]);
}
void selection(int elements[], int array_size)
{
int i, j, k;
_________    ________     __   _
int min, temp;
for (i = 0; i < maxsize-1; i++)
{
min = i;
for (j = i+1; j < maxsize; j++)
{
if (elements[j] < elements[min])
min = j;
}
temp = elements[i];
elements[i] = elements[min];
elements[min] = temp;
}
}
```

Let us now analyze the efficiency of the selection sort. You can easily understand from the algorithm that the first pass of the program does (maxsize - 1) comparisons. The next pass does (maxsize - 2) comparisons and so on. Thus, the total number of comparisons at the end of the sort would be :

(maxsize - 1) + (maxsize - 2) + … + 1 = maxsize * (maxsize - 1) / 2 = O(maxsize2).

7. What is the difference between a queue and a circular queue? Explain circular queue operations?

Queues are data structures that, like the stack, have restructions on where you can add and remove elements.

Ans: Circular queue is a queue in which elements are stored in circular manner. Circular queue is implemented using arrays.

Let us consider an array CQ that contains ‘k’ elements in which CQ[1] comes after CQ[k] in

the array. A linear queue can be transformed to the circular queue when last room comes just before the first room.

In circular queue, if an element is inserted when rear=k, then this element is assigned CQ[1] i.e. instead of incrementing the rear value to k+1, rear is reset to 2. If there is only a single element, the front=rear and if that element is deleted then front=null and rear=null, which indicate that queue is empty. It is very easy to perform insertion and deletion operation on linear queue. At the same time there are many drawbacks of implementing simple queues.

1. If one element is deleted, that empty memory location cannot be used again, whereas in circular queues it can be used.
2. It is not possible to insert an element even at the beginning of the queue if that location is empty. Because of these limitations, disk space is wasted.

In order to overcome these problem, circular queues are implemented.

The function queue_store and queue_retrieve would change in the case of a circular queue.

```
Void queue_store(char*p)
    {
        If((end+1==start)||(end+1-max&&start==0))
        {
            Printf("queue is full");
           Return;
         }
        Arr[end]=p;
        End++;
        If(end==max)
        End=0;
      }
     Char*queue_retrieve[void]
     {
       If(start==max)
       Start=0;
       If(start==end)
         {
             Printf("queue is empty");
         }
         Start++;
         Return arr[start-1];
        }
```

8. Explain tree traversal in detail.

Ans:

Tree traversal :

A tree can be traversed in three major ways. They are

a. in-order traversal

b. pre-order traversal
c. post-order traversal

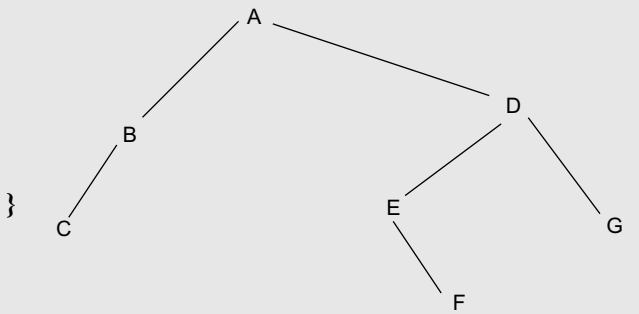
a. In-order traversal:

In-order traversal is given by following steps:
1. traverse the left sub-tree
2. process the root node
3. traverse the right sub-tree

Algorithm for in-order traversal (recursive):

Step 1: check if tree is empty by verifying root pointer R. Print tree empty if R=NULL
Step 2: if leftptr(R)!=NULL then call in-order (leftptr(R))
Step 3: print data(R).
Step 4: if rightptr(R)!= NULL then call in-order(rightptr(R))
Step 5: return.

Example:



So the in-order of the binary tree in the figure above is C-B-A-E-F-D-G.

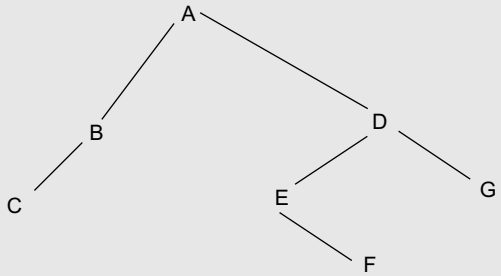
Pre-order traversal:

Pre-order traversal is defined as follows:
1. process the root node
2. traverse the left sub tree in pre-order.
3. traverse the right sub tree in pre- order

Algorithm for in-order traversal :

Step 1: check if tree is empty by verifying root pointer R
If R=NULLL, then print "tree empty"
Step 2: print data(R)
Step 3: if leftptr(R)!=NULL then call pre-order(leftptr(R))
Step 4: if rightptr(R)!=NULL call pre-order(rightptr(R))
Step 5: return

Example:



Pre-order traversal is A B C D E F G

Post-order traversal:

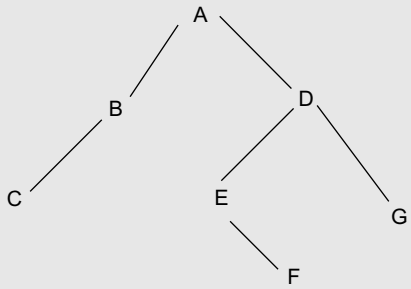
Post-order traversal is defined as follows:

1. traverse the left sub-tree
2. traverse the right sub-tree
3. process the root node.

Algorithm for post-order traversal :

Step 1: check if tree is empty by verifying root pointer R. Print tree empty if R=NULL, then print "tree empty"
Step 2: if leftptr(R)!=NULL then call post-order (leftptr(R))
Step 3: Print data(R).
Step 4: if rightptr(R)!= NULL then call post-order(rightptr(R))
Step 5: return.

Example:



Post-order traversal for the above tree is C-B-F-E-G-D-A

Bibliography

Barkakati, N., *Microsoft C Bile*, SAMS, 1990.
Barker. L., *C Tools for Scientists and Engineers*, McGraw-Hill, 1989.
Berry, R.E. and Meekings, B.A.E., *A Book on C*, Macmillan, 1987.
Hancock, L. and Krieger, M., *The C Primer*, McGraw-Hill, 1987.
Hunt, W.J., The *C Toolbox,* Addison-Wesley, 1985.
Hunter, B.H., *Understanding* C, Sybex, 1985.
Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*, Prentice-Hall, 1977.
Kochan, S.G., *Programming in C*, Hyden, 1983.
Miller, L.H. and Quilici, E.A., *C Programming Language: An Applied Perspective*, John Wiley & Sons, 1987.
Purdum, J.J., *C Programming Guide*, Que Corporation, 1985.
Radcliffe, R.A., *Encyclopaedia C*, Sybex, 1990.
Schildt, H., *C Made Easy*, Osborne McGraw-Hill, 1987.
Schildt, H., *Advanced C*, Osborne McGraw-Hill, 1988
Tim Grady, M., *Turbo C! Programming Principles and Practices*, McGraw-Hill, 1989.
WSI Staff, *C User's Handbook*, Addison-Wesely, 1984.
Wortman, L.A., and Sidebottom, T.O., *The C Programming Tutor*, Prentice-Hall, 1984.

Review Questions

UNIT-I

(1) Write a program to illustrate bitwise operators.
(2) Write a program to print all the alphabets and their equivalent ASCII values.
(3) Write a program to calculate the sum of squares of the first *n* natural numbers using while loop.
(4) Write a program to illustrate short-hand operators used in C.
(5) Write a program to print the multiplication table in the following format.

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

(6) Write a program to calculate the factorial of a given number.
(7) Write a program to calculate sum of squares of cubes of the first *n* natural numbers.
(8) Write a program to reverse the given number.
(9) Write a program to calculate m^n-value using do-while loop.
(10) Write a program to check whether the given number is a palindrome or not.
(11) Write a program to check whether the given number is an Amstrong number or not.
(12) Write a program to check whether the given number is a perfect number or not.
(13) Write a program to print the following format.

```
        *
      * *
    * * *
  * * * *
* * * * *
```

(14) Write a program to calculate a lucky number.
(15) Write a program to calculate the result of the following series accurate up to 7^{th} digit.
$x+x^3/3!+x^5/5!+\ldots\ldots\ldots\ldots\ldots$

(16) An electric power distribution company charges its domestic consumers as follows.

Consumption Units	*Rate of Charge*
0–200	Rs 0.50 per unit
201–400	Rs 100 plus Rs 0.65 per unit excess 200
401–600	Rs 230 plus Rs 0.80 per unit excess of 400

Write a C program that reads the customer number and power consumed and prints the amount to be paid by the customer.

(17) Program to find the roots of the quadratic equation using bisection method.

UNIT-II

(18) Write a program to print the elements of an array in reverse order.
(19) Write a program to add all the elements of a two-dimensional array.
(20) Write a program to find the transpose of a given matrix.
(21) Write a program to find the smallest and largest element in a two-dimensional array.
(22) Write a program to illustrate call by value.
(23) Write a function to calculate the factorial of a given number.
(24) Write a function to sort the elements of an array.
(25) Write a recursive function to find the roots of the quadratic expression using bisection method.
(26) Write a program to copy a string to another without using string handling functions.
(27) Write a program to check whether a given string is a palindrome or not.
(28) Write a function to find the largest element in an array.
(29) Write a recursive function power (base, exponent) that when invoked returns base exponent.
(30) Write a program to sort the characters in a given string.
(31) Write a function to convert all the uppercase letters to lowercase and lowercase letters to uppercase in a given string.
(32) Write a program to undefine an already defined macro.
(33) Write a program to illustrate conditional compilation using #if, #end if, #else.

UNIT-III

(34) Write a program to calculate length of the string using pointers.
(35) Write a function to swap two variables using pointers.
(36) Write a program to illustrate pointer arithmetic.
(37) Write a function to calculate the sum of two numbers using pointers to functions.
(38) Write a program to perform matrix multiplication using pointers.
(39) Write a program to find the largest in an array using pointers.
(40) Write a program to arrange the given numbers in ascending order.
(41) Write a C program using pointer for string comparison.

(42) Write a program to reverse the string using pointers.
(43) Write a program to perform sorting using command line arguments.
(44) The names of employees of an organization are stored in three arrays, namely, first name, second name, and last name. Write a program using pointers to concatenate the three parts into one string to be called name.
(45) Write a program to merge two sorted arrays so that the resultant array is in sorted order using pointers.

UNIT-IV

(46) Write a C program to compute the monthly pay of 100 employees using each employee's name and basic pay. The DA is computed as 52% of the basic pay. Gross salary (Basic pay+DA).Print the employee's name and gross salary.
(47) Write a program to calculate and print the studentwise total for 50 students and 3 subjects. The structure should contain 3 subjects and the total.
48) Write a program to calculate and print the studentwise total for 50 students and 3 subjects using pointers. The structure should contain 3 subjects.
(49) Write a program to store the information of vehicles using bit fields to store the status information. Assume the vehicle object consists of type, fuel and model member fields. Assume appropriate number of bits for each field.
(50) Write a program for illustration of user-defined data types using typedef.
(51) Write a program for illustration of nested structures.
(52) Write a program to add or delete a record of a particular employee based on his code. The structure should contain name, designation, code, salary. Program should also provide the flexibility to update any employee's record.
(53) Define a structure that represents a complex number (contains two floating point members, called real and imaginary). Write functions to add, subtract, multiply and divide two complex numbers.
(54) A company markets hardware items. Create a structure "hwitem" that stores the title of the item, its price, an array of three floats so that it can record the sale in rupees of a particular item for the last three months, category of the item and its original equipment manufacturer. Write a short program that provides a facility to read information of N number of items, append new items, and display all records.
(55) Define a structure to represent a date. Use your structures that accept two different dates in the format mmdd of the same year. Also, write a C program to display the month names of both dates.
(56) Consider a structure master to include information like name, code, pay, experience and write a program to delete and display the information contained in the master variables for a given code.
(57) Write a program to use structure within union. Display the contents of the structure elements.

UNIT - V

(58) Write a C program to read a text file and to count number of characters, number of words and number of sentences and write in an output file.

(59) Write a C program to read the text file containing some paragraph. Use fseek() and read the text after skipping *n* characters from the beginning of the file.

(60) Write a C program to read the text file containing some paragraph. Print the first *n* characters from the beginning of the file.

(61) Write a program to print the output of the following format in an OUTPUT file.

Number	*Square*	*Cube*
2	4	8
3	9	27

(62) Write a C program to read data from a file named INPUT containing numbers. Write all even numbers to a file called EVEN and odd numbers to a file called ODD.

(63) Write a program to print the *n* characters from m^{th} position in the file.

(64) Write a program to print the current position value of the file pointer.

(65) Write a program to check whether end of file has been reached or not using feof() function.

(66) Write a program to check whether any data errors are present and print the relevant information using ferror() function.

(67) Write a program to detect an error while opening a file that does not exist.

(68) Write a C program to open a pre-existing file and add information at the end of the file. Display the contents of the file before and after appending.

(69) Write a program to copy one file to another.

(70) Write a program to read a text file and convert the file contents in capital (uppercase) and write the contents in an output file.

(71) Candidates have to score 90 or above in an IQ test to be considered eligible for taking further tests. All candidates who do not clear the IQ test are sent rejection letters and others are sent call letters for further tests. The selected list and other conversations have to be written to file.

UNIT - VI

(72) Write a non-recursive simulation of towers of Hanoi.

(73) Write a recursive function for the towers of Hanoi.

(74) Write a program to convert prefix to postfix using stacks.

(75) Write a program to convert postfix expression to prefix expression.

(76) Write a program to evaluate postfix expression.

(77) Program to evaluate prefix expression.

(78) Write a program to implement dequeues.

(79) Write a program to implement priority queues.

(80) Write a program to check whether a given string is a palindrome or not using stacks.

UNIT-VII

(81) Write a program in 'C' to form a list consisting of the intersection of the elements of two lists.
(82) Write a routine to reverse elements of a doubly linked list by traversing the list only once.
(83) Write a program to count the number of non-leaf nodes in a binary tree.
(84) Write a program to count the number of leaf nodes in a binary tree.
(85) Write a program to implement a circular using double-linked list.
(86) Write a program to perform polynomial addition.
(87) Write a program to perform polynomial multiplication.
(88) Write a function to reverse the single linked list.
(89) Write non-recursive functions for binary tree traversals.
(90) Write a program to implement binary search tree using arrays.
(91) Write a C program to exchange two nodes of a singly linked list.
(92) Write a program to implement breadth first search.
(93) Write a program to implement depth first search.
(94) Represent a singly linked list using an array. Write routines to insert and delete elements for this representation.
(95) Write a routine SPLIT() to split a singly linked list into two lists so that all elements in odd position are in one list and those in even position are in another list.
(96) Represent a doubly linked list using an array. Write routines to insert and delete elements for this representation.
(97) Write a function to merge doubly linked lists and arrange the numbers in ascending order.

UNIT-VIII

(98) Write a program to implement merge sort.
(99) Derive the efficiency of merge sort.
(100) Derive the efficiency of binary search.