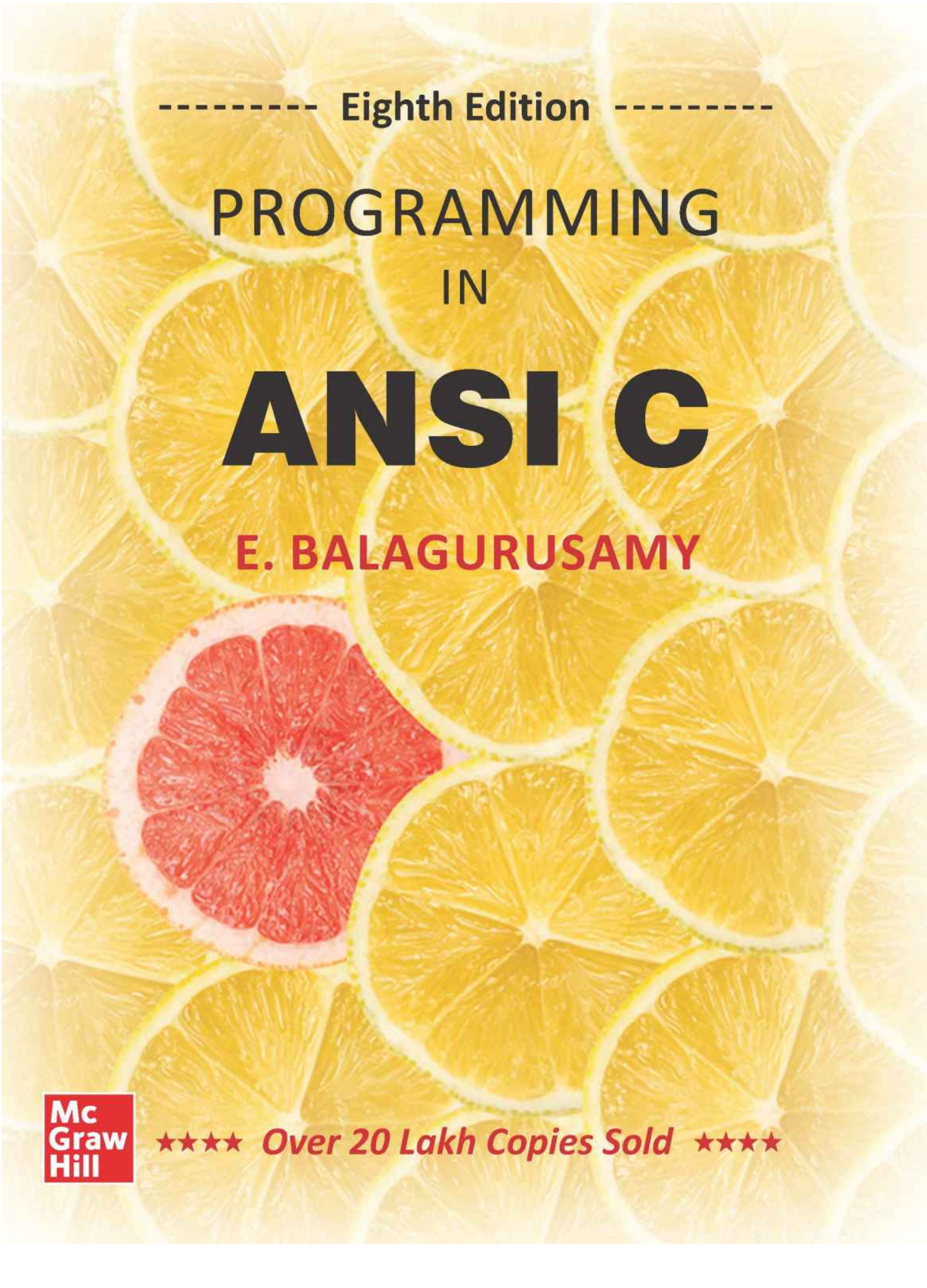
--------- Eighth Edition ----------

PROGRAMMING IN ANSI C

E. BALAGURUSAMY

Mc Graw Hill

★★★★ *Over 20 Lakh Copies Sold* ★★★★



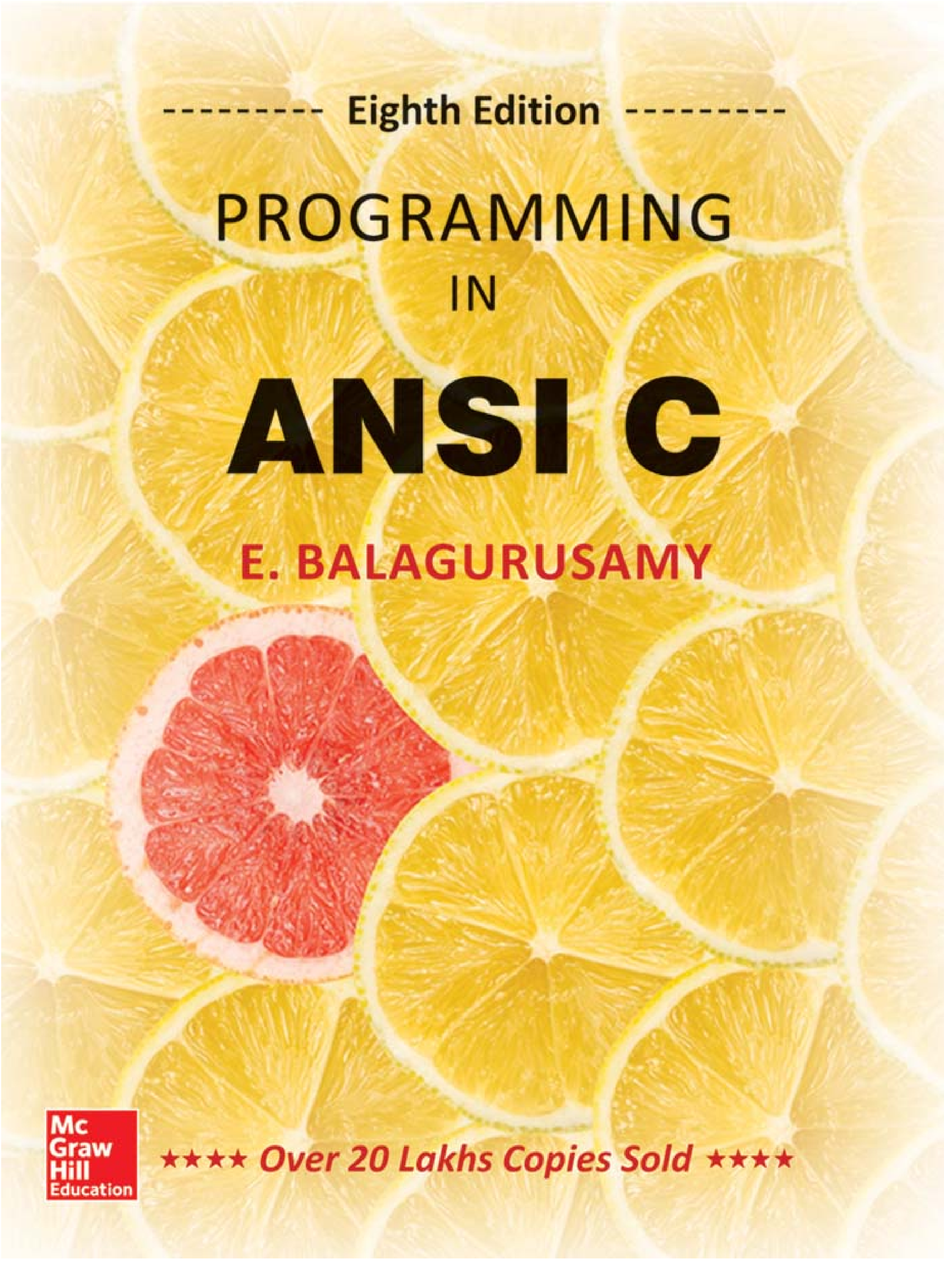
--------- Eighth Edition ---------

PROGRAMMING IN ANSI C

E. BALAGURUSAMY

Mc Graw Hill

★★★★ *Over 20 Lakh Copies Sold* ★★★★


--------- Eighth Edition ---------
PROGRAMMING
IN
ANSI C
E. BALAGURUSAMY
Mc Graw Hill Education
★★★★ Over 20 Lakhs Copies Sold ★★★★

PROGRAMMING IN ANSI C

Eighth Edition

About the Author

E Balagurusamy is presently the Chairman of EBG Foundation, Coimbatore. In the past he has also held the positions of member, Union Public Service Commission, New Delhi and Vice-Chancellor, Anna University, Chennai. He is a teacher, trainer and consultant in the fields of Information Technology and Management. He holds an ME (Hons) in Electrical Engineering and PhD in Systems Engineering from the Indian Institute of Technology, Roorkee. His areas of interest include Object-Oriented Software Engineering, E-Governance: 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:

- *Fundamentals of Computers*
- *Computing Fundamentals and C Programming 2/e*
- *Programming in C#, 4/e*
- *Programming with Java, 5/e*
- *Object-Oriented Programming with C++, 7/e*
- *Programming in BASIC, 3/e*
- *Numerical Methods*
- *Reliability Engineering*

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

PROGRAMMING IN ANSI C

Eighth Edition

E Balagurusamy

Chairman
EBG Foundation
Coimbatore

McGraw Hill Education (India) Private Limited

CHENNAI

McGraw Hill Education Offices

Chennai 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

McGraw Hill Education (India) Private Limited

Published by McGraw Hill Education (India) Private Limited
444/1, Sri Ekambara Naicker Industrial Estate, Alapakkam, Porur, Chennai - 600 116

Programming in ANSI C, 8/e

Print Edition:
ISBN-13: 978-93-5316-513-0
ISBN-10: 93-5316-513-X

E-Book Edition:
ISBN-13: 978-93-5316-514-7
ISBN-10: 93-5316-514-8

1 2 3 4 5 6 7 8 9 D101417 23 22 21 20 19

Printed and bound in India.

Managing Director: *Lalit Singh*

Senior Portfolio Manager: *Hemant K Jha*

Sr. Manager—Content Development and Production Services: *Shalini Jha*
Sr. Content Developer: *Vaishali Thapliyal*
Content Developer: *Shweta Pant*

Production Head—Higher Education and Professional: *Satinder S Baveja*

Senior Manager—Production: *Piyaray Pandita*

General Manager—Production: *Rajender P Ghansela*
Manager—Production: *Reji Kumar*

Typeset at The Composers, 260, C.A. Apt., Paschim Vihar, New Delhi 110 063 and printed and bound in India at

Cover Printer:

Cover Designer: The Computerminds

Cover Image Source: Shutterstock

Visit us at: www.mheducation.co.in

Write to us at: info.india@mheducation.com

CIN: U80302TN2010PTC111532

Toll Free Number: 1800 103 5875

Preface

C is a powerful, flexible, portable and elegantly structured programming language. Because C combines the features of high-level language with the elements of an assembler, it is suitable for both systems and applications programming. It is undoubtedly the most widely used general-purpose language today in operating systems, and embedded system development. Its influence is evident in almost all modern programming languages. Since its standardization in 1989, C has undergone a series of changes and improvements in order to enhance the usefulness of the language. The version that incorporates the new features is now referred to as C11.

New to this Edition

Distinguished as one of the bestsellers in the market, 'Programming in ANSI C' is taking forward another step in the pursuit of perfection. The title keeps up with its predecessors by balancing between theory and practical approach of the subject. In order to build a firm foundation, a new chapter 'Introduction to Computing' has been introduced. The appendix 'Projects' has been revamped with two new and exciting projects, that will help students to hone their programming skills and bring them to practical usefulness. Existing chapters have been further enriched with additional write-ups, tips and a new 'Test Your Skills' section around the content. Two new sections, Multiple Choice Questions and Interview Questions, have been introduced in each chapter, with the intent to give more practice to students and prepare them for different interviews and competitive exams. The section on Programming Exercises has also been enriched with some new questions.

Organization of the Book

Programming in ANSI C in its eighth edition starts with two introductory chapters. Chapter 1, 'Introduction to Computing', introduces students to different components of a computer and discusses problem solving aspects of computing. Chapter 2, 'Overview of C', discusses the basic structure of C programs and their execution. Chapter 3 discusses how to declare the constants, variables, and data types. Operators and expressions are presented in Chapter 4. Chapter 5 deals with the managing of Input and Output Operations. Chapter 6 talks about Branching. The concept of decision making and looping is explained in Chapter 7. Arrays, Character Arrays, and Strings have been discussed in Chapters 8 and 9. User-defined functions, structures and unions are covered in Chapters 10 and 11, while Chapter 12 talks about Pointers. Chapter 13 briefs about file management in C. Dynamic Memory Allocation and Linked Lists are presented in Chapter 14. Preprocessor is explained in Chapter 15. Guidelines helpful in developing a C Program are listed in Chapter 16.

Salient Features of the Book

- A comprehensive book covering the fundamentals as well as latest features of C programming.
- Each chapter follows an outcome-based learning approach as per 'Bloom's Taxonomy'.
- Strong pedagogical features, that include:
  - ♦ Worked out problems
  - ♦ Tips and important notes
  - ♦ 24 detailed case studies
  - ♦ Multiple choice questions
  - ♦ Programming and debugging questions
  - ♦ Interview questions
- Two new and exciting projects on:
  - ♦ Electricity Board Management System
  - ♦ Making Web Services in C
- An engaging appendix on 'Graphic Programming Using C'

Digital Supplements

The digital supplement can be accessed at the given link (http://www.mhhe.com/balagurusamy/ansic8e). It contains the following components:

- Projects
- e-Case studies
- Additional information and questions

Acknowledgements

This book is my sincere attempt to make a footprint on the immensely vast and infinite sands of knowledge. I would request the readers to utilise this book to the maximum extent.

I owe a special thanks to the entire team of McGraw Hill Education India.

E Balagurusamy

Publisher's Note

McGraw Hill Education (India) invites suggestions and comments from you, all of which can be sent to*info.india@mheducation.com* (kindly mention the title and author name in the subject line).

Piracy-related issues may also be reported.

Contents

Chapter 1

Introduction to Computing

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

- LO 1.1 Know the structure of a computer and have an idea about its functioning
- LO 1.2 Learn to develop programming logic through algorithms and flowcharts

INTRODUCTION

Computing is an operation that involves processing of data by making use of a computer. To be able to use a computer and perform computation, we first need to be aware of the different components that are present in it and understand how these components may be used to serve our purposes. In this chapter, we will briefly discuss the different hardware components of a computer. We will also discuss methods to develop programs using which we can achieve our objective to solve problems and perform various computational activities.

COMPONENTS OF A COMPUTER

LO 1.1
Know the structure of a computer and have an idea about its functioning

A computer is an electronic device which may be used to perform various computations involving arithmetic and logical operations. Figure 1.1 shows the major components of a computer.

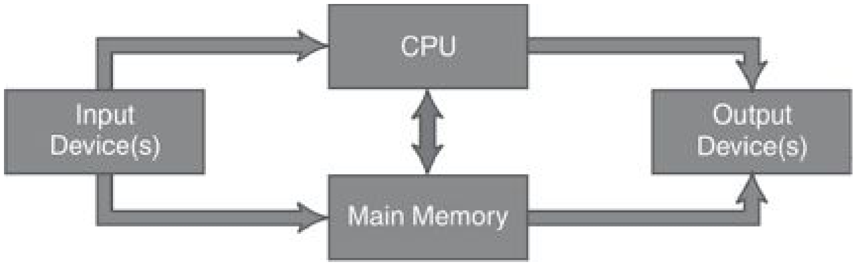


Fig. 1.1 Components of a computer

As seen in the figure, the major components in a computer are the central processing unit (CPU), input device(s), output devices(s) and the memory. These components interact with each other using buses, as shown in the figure.

Central Processing Unit

The CPU is the unit in which all processing activities take place. The CPU is a classic example of a very large scale integrated (VLSI) electronic circuit that consists of the arithmetic and logic unit (ALU), the control unit (CU) and some registers. Figure 1.2 shows the layout of a CPU. The ALU contains circuits using which various arithmetic and logical operations may be executed. These circuits include adders (for addition), subtractors (for subtraction), comparators (for comparison of operands), etc. The CU, also known as the brain of a computer, is the unit which is responsible for generating control signals. These control signals control the working of the various components within a computer. The registers present inside the CPU may be considered to be a set of high-speed storage devices meant to serve as working memory for the ALU. Some of these registers are assigned the task of holding special data or instructions. Such registers are known as special purpose registers, whereas others are known as general purpose registers. The units present inside the CPU interact with each other through an internal bus. Signals generated by the CU also control the activities of the various components within the CPU. Apart from the internal bus, three other buses also interface the CPU. The control bus carries control signals generated by the CU to the memory and all the peripherals. The address bus carries an address to locate a word in the memory. Finally, the data bus is used to carry data and information to and from between the components of the computer. In addition to the units mentioned so far, modern day CPUs are also equipped with high-speed, on-chip memory units known as the cache memory.

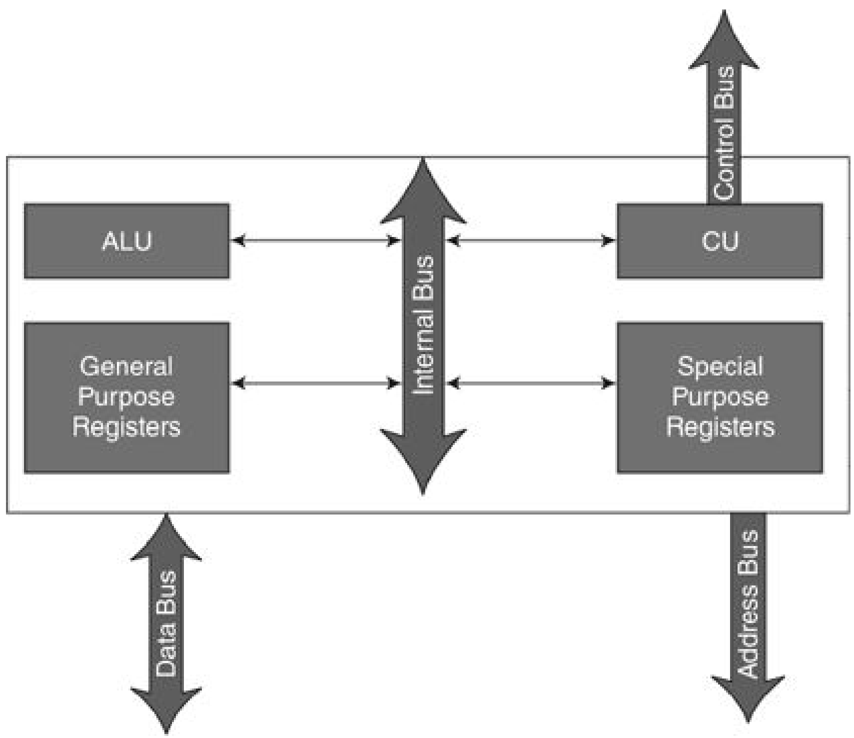


Fig. 1.2 Layout of a model CPU

Input Devices

Input devices are the ones through which users are allowed to input data and other instructions to the computer. The most popular choices for input devices are the keyboard and the mouse. Other examples include scanners, readers, etc. Most input devices are electro-mechanical in nature which accept the inputs from the user through mechanical means, such as depressing of keys in a keyboard to enter some data or moving the mouse around on a surface to locate a position. Information acquired through such mechanical means is digitized (converted into electronic signals) and transmitted to the CPU/memory.

Output Devices

Just as input devices are used to convey messages to the computer, output devices are used by the computer to convey messages back to the users. The most common output device that almost all computers have is the monitor. The monitor is used to display the results of computations, messages, etc. Other examples of output devices are printers, speakers, plotters, etc.

The input and the output devices are sometimes known as peripheral devices. These auxiliary devices may be connected to a computer and used. They are auxiliary because their existence within a computer is not mandatory unlike the CPU or the memory.

Memory

Memory/storage unit is the component of a computer that serves data storage purposes. These units are of varying access speeds and volatility. Access speed refers to the time taken to store/retrieve data from the memory. Volatility refers to the duration of time or the conditions under which data is stored in the memory. Based on these characteristic features, the computer’s memory may be categorized into different types. Figure 1.3 shows the classification of the computer’s memory.

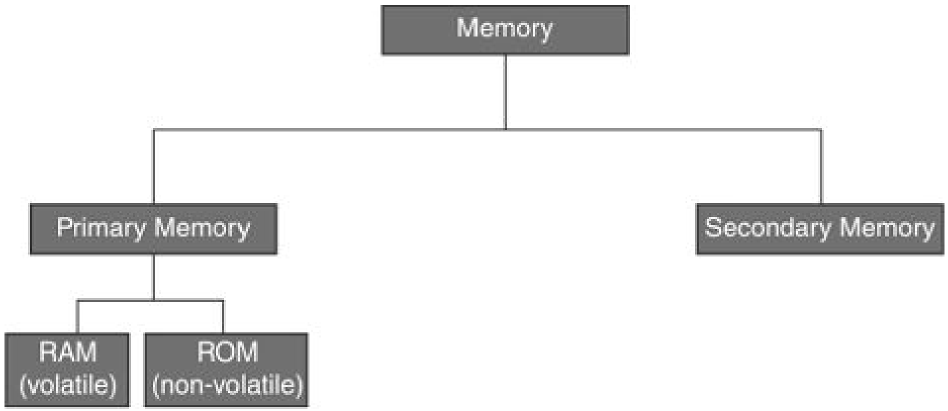


Fig. 1.3 Classification of the computer's memory

As shown in the figure, the memory is broadly classified into primary memory and secondary memory. The primary memory, also known as the main memory, is so called because the CPU can directly interact with it through the data bus and the address bus. The main memory is of two major types, the random access memory (RAM) and the read only memory (ROM). The RAM is a volatile memory that stores instructions and data related to the program currently being executed. The RAM is volatile because it would lose all the information stored in it as soon as the power to the computer is switched off. Unlike the RAM which allows both the read and write operations, the ROM is a read-only, non-volatile memory. The ROM does not permit any write operation and is non-volatile because the information stored in it would be retained even after the computer has been switched off. Apart from other important routines, the ROM houses the boot strap loader. The boot strap loader is a system software that is responsible for loading the memory resident portion of the operating system from the secondary memory into the RAM once the computer has been switched on and the power on self tests (POST) have been conducted. The secondary memory, also known as auxiliary memory, is a large depository of non-volatile storage space that can store information for as long as we want. In this respect it is worth mentioning that the secondary memory is slower compared to the access time of the primary memory. The hard disk, flash drives, etc. are some examples of secondary memory.

Concept of Hardware and Software

The hardware of a computer consists of various electronic, electro-mechanical and electro-magnetic devices along with the interfacing buses connecting them. In other words, the hardware of the computer consists of the bare machinery. Software, on the other hand, consists of the set of instructions, commands, programs, etc. that can make the bare machinery operate and fulfill our objectives. If the hardware is the body, then the software is the soul of a system. For a computer to operate, both the hardware and the software are indispensable.

Types of Software

Depending upon the type of work a software is required to perform, it can be classified into two categories, system software and application software.

System Software

System softwares are the ones that closely interact with the hardware of a computer. These softwares depend upon the architecture of the underlying computer. They are responsible for controlling the resources of the system and perform other activities related to the system. The different kinds of system softwares are operating system, loader, linker and translators.

Operating System

Operating system is a system software which acts as an interface between the user and the bare hardware. The operating systems currently in use are Linux, Windows 10, UNIX, macOS High Sierra, etc. The major functions of an operating system are:

- Process and processor (CPU) management
- Memory management
- Device management
- File management

To put it simply, the operating system manages all the resources of a computer. It also interprets all the commands that it receives from the user to the underlying hardware. Today's operating systems come with a smart graphic user interface (GUI) that is easy to operate. They also offer several utilities such as calculators, text editors, scanning softwares, drawing tools and image editing tools.

Translators

Translators cover a class of system softwares which translate a program written in one programming language into another. Compiler, assembler, interpreter, etc. are few of the most used translators. The source programming language is either a high-level programming language or assembly language. The target language in almost all cases is machine language which is essentially in binary (1's and 0's – high voltage and low voltage), a language understood by the computer. Translators are system softwares because they need a thorough understanding of the architectural configuration of the underlying hardware. This knowledge is necessary because the target code is machine dependent. The machine instruction set supported by a particular architecture may not be supported by another.

Compilers are system softwares that translate a program written in high level language (HLL) into machine language. C, C++, Java, etc. are examples of HLLs. Thus, compilers have to be aware of the syntactic nuances of the HLL and the instruction set, addressing modes, register configuration, etc. of the machine in which the target program is supposed to be executed. The syntactic specification of a programming language is given to the compiler using the grammar of the language. Thus, a compiler designed for the C programming language would not be able to process a program written in the Java language. This is because the C compiler has been provided with the grammar rules of the C language and would consider statements written using Java as incorrect. The output of a compiler is in machine language and uses the instructions supported by the CPU of the computer. Thus, a compiler designed to generate machine code for a computer housing an Intel processor would not be able to generate machine code for a computer containing a Motorola processor. For any compiler it is essential to specify both the source language as well as the target machine in which the output program will be executed. The machine code generated by the compiler is known as object code.

Assemblers translate a program written in assembly language to machine language. Assembly language consists of instructions supported by the underlying processor. The difference between assembly and machine language is that assembly language uses mnemonics such as ADD, SUB, etc., whereas machine language uses the corresponding binary operation codes for each of these mnemonics.

Linker

The linker links together the object codes belonging to all the sub-routines/functions of a program and forms an executable code. This executable code may be stored in the secondary memory and can be loaded into the main memory as and when required for execution.

Loader

The loader is a system software that loads a program to be executed from the secondary memory into a designated area of the main memory. Modern day operating systems have a loader incorporated within them. Absolute loaders, relocatable loaders, direct linking loaders are few forms of loader. A special loader known as the boot strap loader deserves a special mention in this respect. As mentioned earlier, the boot strap loader loads the memory resident portion of the operating system into the main memory immediately after the computer is switched on. The operating system then takes control of the computer.

Application Software

Application softwares are the ones that are meant to perform a specific activity and aid users in various ways. Word processing softwares, image editing, spread sheet packages, database softwares and accounting softwares are all examples of application softwares. Each of these has been designed to serve a special purpose.

Art of Programming through Algorithms and Flowcharts

LO 1.2
Learn to develop programming logic through algorithms and flowcharts

We have seen in the previous section how the software is invisible but indispensable part of a computer that enables the users to interact with the computer and makes it work as per one's specification and requirements to achieve a computation objective. A software can be considered as a set of programs which work in a cohesive manner to perform a task. A program is a set of instructions which when executed in sequence performs a task.

Let us consider the example of a program which when executed would enable the user to enter two numbers, add them and display the result. It is clear that this program would contain three sets of instructions; the first set would enable users to input the data, the second set to add the numbers and the third set to display the result. When writing the program, the programmer would have planned these three sets and used statements within it to implement each of them. The example considered here is an extremely simple one.

However, more often than not, the problems which a computer is supposed to solve would be more complex than this. Such complex problems would invariably require complex solutions. These solutions involve a lot of complex arithmetic operations and logical decision making. The need of the hour here is a tool which would enable us to represent the logic required in the program and make it possible to transform the logic to an equivalent set of instructions which when executed by the computer would solve the problem. Algorithms and flowcharts are two such tools. These tools are independent of the programming language as well as the target machine. Thus, the logic represented using one of these tools may be used to write programs in any language of the programmer's choice. In fact, logic should always be independent of programming languages. Irrespective of the tool and language chosen to write a program, programming logic constructs are of three important types (apart from input/output and sub-routine call constructs):

- Imperative statements (action)
- Conditional statements (decision making)
- Iterative statements (repetition/Loops)

The tools which we are talking about should be capable of representing each of these three kinds of constructs because a program comprises statements which fall into one of these three categories.

It is advisable to always use tools such as algorithms and flowcharts to first plan and then write a program. After all it has been rightly said that well planned is half done.

Algorithm

An algorithm is a *finite* set of *unambiguous* instructions which, when executed, performs a task *correctly* . There are three characteristic features in this description of an algorithm. They are:

- The number of steps required to perform the task should be finite.
- Each of the instruction in the algorithm should be unambiguous in nature; meaning on the execution of each such instruction, the outcome should be definite and predictable.
- Finally, the algorithm should solve the problem correctly. In other words, on the execution of the algorithm, the objective should be fulfilled.

There are no defined rules to write an algorithm. However, most authors prefer to use English-like statements to represent the action that needs to be performed at each step. We will consider different examples to demonstrate each of the three logic constructs.

EXAMPLE 1 **(Use of imperative constructs)**

Problem – Develop an algorithm to find the average of three numbers taken as input from the user.

Algorithm AVERAGE

```
Step 0  START
Step 1  INPUT first number into variable A
Step 2  INPUT second number into variable B
Step 3  INPUT third number into variable C
Step 4  COMPUTE SUM - A + B + C
Step 5  COMPUTE AVG - SUM / 3
Step 6  DISPLAY AVG
Step 7  END
```

Steps 1, 2 and 3 are statements that will enable the input of three different numbers. Step 6 is used to display the result. Steps 4 and 5 are of particular interest to us as these are the two imperative statements which instruct/command to perform the computation. Thus, imperative statements are the ones which command the processor to perform a task.

As mentioned earlier, an algorithm has no set rules for writing. This same algorithm could very well be written by not using the word COMPUTE in steps 4 and 5. In a similar manner, step 1 could have been simply written as 'INPUT A'. Use of words such as COMPUTE in steps 4 and 5, or the phrase, 'first number into variable' in step 1 serves as an annotation, which increases the readability and understandability of the algorithm.

EXAMPLE 2 **(Conditional construct)**

Problem – Develop an algorithm to divide one number by another and find the quotient.

Algorithm DIVISION

```
Step 0  START
Step 1  INPUT first number into variable A
Step 2  INPUT second number into variable B
Step 3  IF B !- 0 THEN
                  Q - A/B
                  DISPLAY Q
        END IF
Step 4  END
```

Dividing one number by another is perhaps one of the simplest instructions that can be given to the CPU for processing. However, in dividing two numbers one should always be careful to check whether the divisor is 0. A computer has no way to store an infinitely large number. So, it does not permit a division by 0. An attempt to do so forcefully causes the computer to generate a runtime ‘Divide Overflow ’ error and the processing is immediately aborted. The algorithm developed here checks to see whether the divisor B is 0. Division is performed only if B is not 0 and the result is displayed. Step 3 is the statement where a decision is taken whether the divisor is 0. We use the word ‘IF ’ to do so. The condition ‘B != 0 ’ is evaluated to check whether B is 0. If B is not 0, then the condition is evaluated to be true. In this case, the two statements
‘Q = A / B ’ and ‘DISPLAY Q ’ would be executed in sequence. Once done, the control would move on to step 4 which causes the algorithm to halt. In case B is 0, the condition would evaluate to be false. Then the control of the algorithm would directly move to step 4 without attempting the division operation. The symbol, ‘!= ’ corresponds to ‘not equal to ’. Once again this is not a standard notation for an algorithm. Some authors use ‘<> ’ instead to represent ‘not equal to ’. When transforming into a program, ‘not equal to ’ would be replaced by the corresponding logical operator supported by the programming language. The ‘END IF ’ in step 3 indicates the end of the conditional statement construct.

EXAMPLE 3 (Conditional construct)

Problem – Develop an algorithm to find the maximum of two numbers input by the user.

```
Algorithm MAXIMUM
Step 0  START
Step 1  INPUT first number into variable A
Step 2  INPUT second number into variable B
Step 3  IF A > B THEN
                 MAX - A
        ELSE
                 MAX - B
        END IF
Step 4  DISPLAY MAX
Step 5  END
```

This example, like the previous one, also demonstrates a conditional statement construct. However, depending upon whether the outcome of the condition ‘ A > B ’ is true or false two different tasks would be performed. The use of the word ‘ ELSE ’ makes this algorithm slightly different from the previous one. However, decision-making is involved in both of them. The values of A and B are compared to determine whether A is greater than B (A > B). If A is greater than B, then it implies that A is the maximum of the two (MAX = A) , else it implies that B is the maximum of the two (MAX = B) .

EXAMPLE 4 **(Iteration construct)**

Problem – Develop an algorithm to find the sum of the first N natural numbers where N would be input by the user.

```
Algorithm SUM-N
Step 0  START
Step 1  INPUT N
Step 2  I - 1
Step 3  SUM - 0
Step 4  REPEAT Steps (a) – (b) WHILE I <-N
               (a) SUM - SUM + I
               (b) I - I + 1
        END WHILE
Step 5  DISPLAY SUM
Step 6  END
```

The objective of this algorithm is to calculate 1 + 2 + … + N. We use the variable SUM to store the result. The variable I is initialized to 1 and the variable SUM to 0. The algorithm works by first adding 1 into SUM , followed by 2, and then by 3 and so on. The process continues till N is added into SUM . The values 1, 2, 3, …, N are generated by using the variable I. If I is initially 1, it is then incremented to become 2, followed by 3, …, finally N. The following are the sequence of steps that would be executed one after another:

```
SUM - SUM + 1
SUM - SUM + 2
SUM - SUM + 3
.
.
.
SUM - SUM + N
```

Instead of writing out N addition statements (note that the value of N is not known at development time) one after another, we use the following form:

```
SUM - SUM + I, for all I - 1 to N
```

We monotonically increment the value of I and repeat the execution of the statement SUM = SUM + I until the last value of I, viz. N, has been added into SUM . Incrementing I next would cause it to hold N + 1. This is when the condition at Step 4 ‘ I <= N ’ fails and the control of the algorithm moves to step 5. The execution of the statements ‘ SUM = SUM + I ’ and ‘ I = I + 1 ’ would be repeated while ‘ I<=N ’. The control of the algorithm remains within the loop (iteration) till the condition ‘ I <= N ’ evaluates to true. In writing this algorithm, a strong assumption has been made, that is, the user always inputs a positive whole number for N.

Iterative constructs are thus used to simulate repetition. They are used when one or more statements need to be executed repeatedly. Consider the situation where we want to display the word ‘ HELLO ’ 100 times. We can write DISPLAY ‘ HELLO ’ 100 times as follows:

```
 DISPLAY 'HELLO'   (1st. time)
 DISPLAY 'HELLO'   (2nd. time)

DISPLAY 'HELLO'   (3rd. time)
.
.
.
DISPLAY 'HELLO'   (100th. time)
```

Here we need to repeat just one statement 100 times. Similarly, if we want to repeat the execution of an entire statement block 10000 times or even more, imagine the length of such an algorithm. Writing codes in such a manner may become a bit cumbersome. Instead we can put the statement in an iterative construct and iterate (repeat) it 100 times. A variable such as I would be used to keep a track of the number of times the iteration has taken place. The modified construct would be as follows:

```
REPEAT steps (a)-(b) while I <- 100
      (a) DISPLAY 'HELLO'
      (b) I - I + 1
END WHILE
```

Like before, I would have to be initialized to 1.

Pseudocode

Pseudocodes are a means to represent an algorithm in a coded form. The specifications used in a pseudocode are syntactically not stringent as in a program code. However, they represent the steps of an algorithm in a coded form which very closely resembles program code. Given a pseudocode, translation of the same into program code is relatively easy. A pseudocode may be considered to be an intermediate code between an algorithm and program code. The pseudocode for the algorithm in Example 4 is as follows:

```
Pseudocode SUMN
{
      print 'Enter a value for the number N: '
      scan N
      I - 1
      SUM - 0
      while I <- N
      {
              SUM - SUM + I
              I - I +1
      }
      print 'The sum is: ', SUM
}
```

It should be noted that a pseudocode is slightly more cryptic compared to an algorithm and is closer in form to a C program code. The C code for the pseudocode shown here would be as follows:

```
#include <stdio.h>
main()
{
      int I, SUM, N;
      printf("\nEnter a value for the number N:");
      scanf("%d", &N);
```

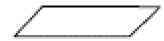
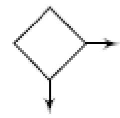
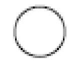
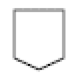
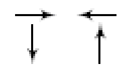
```
		while(I <= N)
		{
			SUM = SUM + I;
			I = I + 1;
		}
		printf("\The sum is: %d", SUM);
	}
```

A quick comparison between the pseudocode and the corresponding program using the C language shows that function names such as ‘ printf ’ and ‘ scanf ’ have replaced pseudocodes such as ‘ print ’ and ‘ scan ’. ‘ print ’ and ‘ scan ’ are codes for representing display and input operations. They are pseudo (false) because these codes would not be understood by the C compiler or the machine. Nevertheless, they are still in an encoded form. It is worth noting that the statements in the C program are syntactically bound. However, such restrictions are not to be found in a pseudocode.

Flowchart

Just like the algorithm, the flowchart is also a tool to represent logic. As the name indicates, a flowchart helps us graphically visualize the flow of control within the sequence of statements. At times it becomes easier for us to understand logic better when we have a picture in front of us. A flowchart proves to be really handy at such times. Unlike the algorithm, the flowchart uses some standard notation to graphically represent logic. Table 1.1 shows the graphical representation of some common constructs.

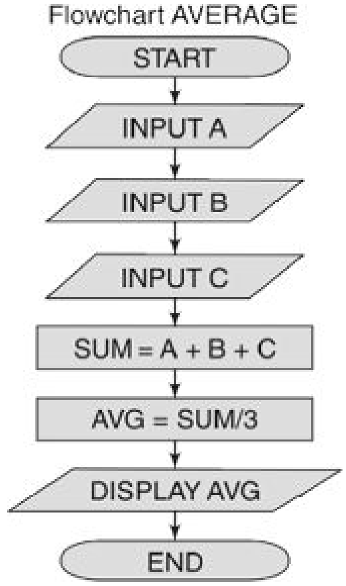
Table 1.1 Notations used in a flowchart

Name	*Description*	*Symbol*
Terminal symbol	This symbol is used to mark the beginning and end of a flowchart.	
Input/output box	This box is used to represent input operations or output operations.	
Process box	This box is used to represent processing operations. More specifically, this symbol is used to represent the imperative logic construct.	
Decision box	This diamond-shaped symbol is used to represent conditional statement constructs. Each such box has two exit points which correspond to the evaluation of the condition to true and false.	
Connector (intra-page)	This symbol is used to transfer the flow of control from one point to another within a page. A connector is usually labeled to match with its counterpart.	
Off-page connector (inter-page)	This symbol is used to transfer the flow of control from one point on a page to another point on a different page. This notation is useful when the flowchart spans more than one page.	
Arrowed lines	These are used to show the sequence of flow from one box to another.	

The symbols shown here are some basic ones that are used in flowcharts. Apart from these, there also exist other symbols to represent more constructs such as sub-routine call, reading from database and printing.

Let us now understand the use of these symbols to draw flowcharts. For ease of understanding, we consider the same set of examples which we have considered for understanding algorithms.

Flowchart AVERAGE



EXAMPLE 5 (Use of imperative constructs)

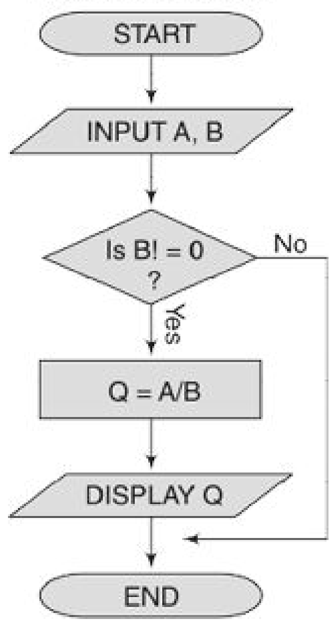
Problem – Develop an algorithm to find the average of three numbers taken as input from the user.

Algorithm AVERAGE

Step 0 START
Step 1 INPUT first number into variable A
Step 2 INPUT second number into variable B
Step 3 INPUT third number into variable C
Step 4 COMPUTE SUM - A + B + C
Step 5 COMPUTE AVG - SUM/3
Step 6 DISPLAY AVG
Step 7 END

The steps in the algorithm shown in the left match with the steps of the flowchart shown in the right.

Flowchart DIVISION



EXAMPLE 6 (Conditional construct)

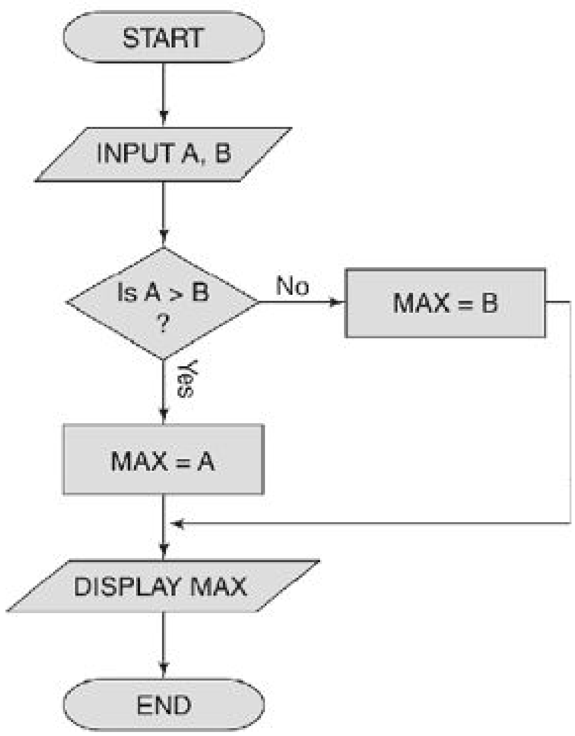
Problem – Develop an algorithm to divide one number by another and find the quotient.

Algorithm DIVISION

```
Step 0  START
Step 1  INPUT first number into variable A
Step 2  INPUT second number into variable B
Step 3  IF B !- 0 THEN
                Q - A / B
                DISPLAY Q
        END IF
Step 4  END
```

In the flowchart shown on the right, notice the merging of two input operations into one input box. Also notice the use of the decision box. The calculation of the quotient and the display of the same takes place along the ‘ Yes ’ path, i.e. when the condition ‘ B != 0 ’ evaluates to true. In case the condition evaluates to false, the control directly moves to the terminal box labeled ‘ END ’.

Flowchart MAXIMUM



EXAMPLE 7 (Conditional construct)

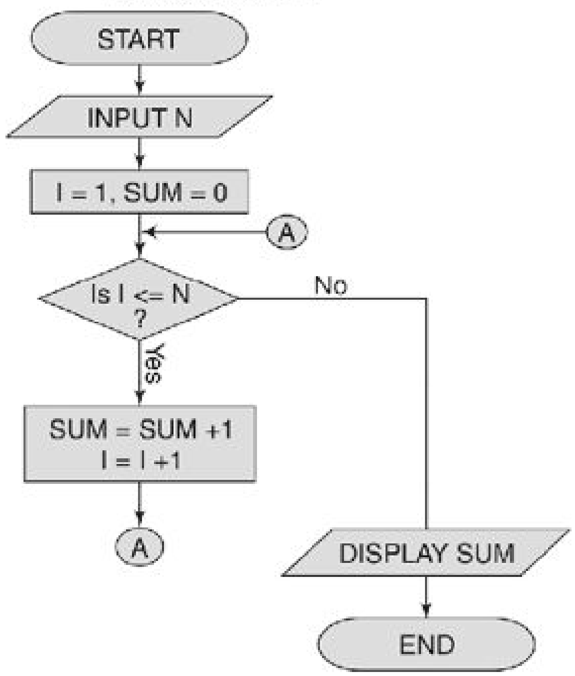
Problem – Develop an algorithm to find the maximum of two numbers input by the user.

Algorithm MAXIMUM

```
Step 0   START
Step 1   INPUT first number into variable A
Step 2   INPUT second number into variable B
Step 3   IF A > B THEN
                  MAX - A
         ELSE
                  MAX - B
         END IF
Step 4   DISPLAY MAX
Step 5   END
```

The flowchart shown on the right computes the maximum of two numbers A and B. Do note the separate actions that take place along the Yes and the No paths. Also note how the display of the MAX takes place irrespective of whether A or B is the maximum. The Yes and the No paths merge before the display of the MAX , and then finally the flowchart terminates.

Flowchart SUM-N



EXAMPLE 8 (Iteration construct)

Problem – Develop an algorithm to find the sum of the first N natural numbers where N would be input by the user.

Algorithm SUM-N

```
Step 0  START
Step 1  INPUT N
Step 2  I - 1
Step 3  SUM - 0
Step 4  REPEAT Steps (a) – (b) WHILE I <-N
              (a) SUM - SUM + I
              (b) I - I + 1
        END WHILE
Step 5  DISPLAY SUM
Step 6  END
```

Note the use of a connector labeled, ‘A’ in this flowchart.

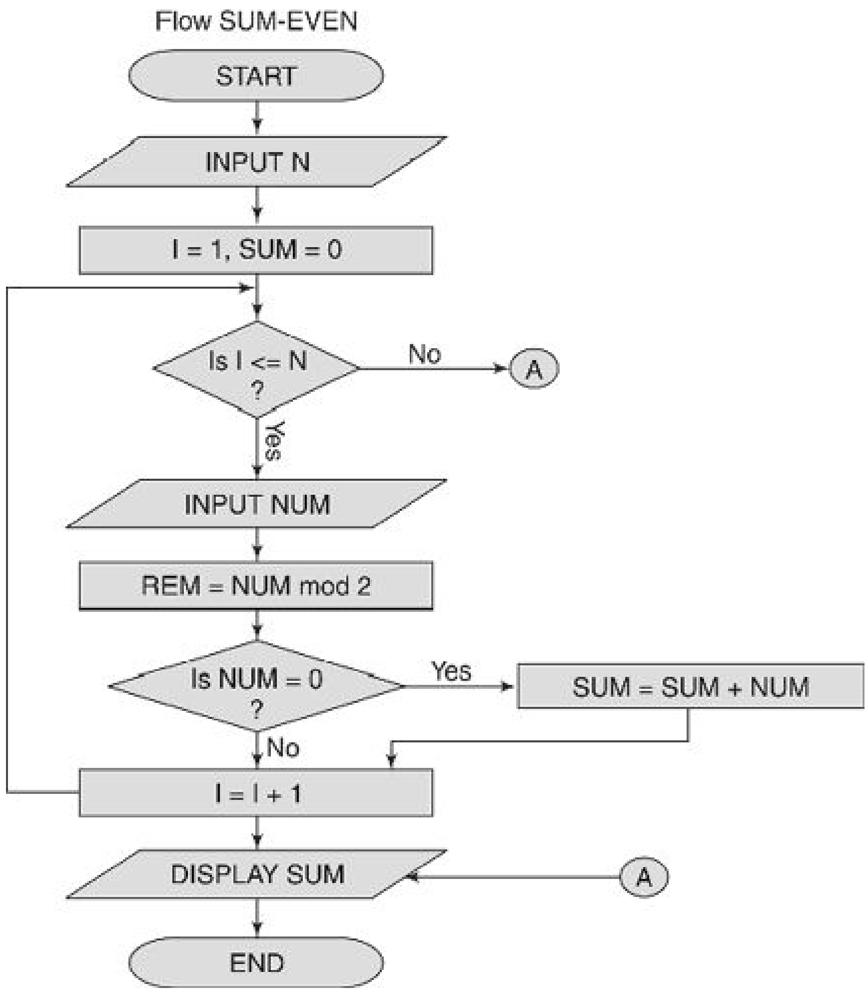
Let us now consider an example which combines both an iterative as well as a conditional statement construct. For this example we will first write an algorithm, then draw an algorithm and finally transform the logic into a C program.

EXAMPLE 9

Problem – Write an algorithm, draw a flowchart, and finally write a C program for finding the sum of only the even numbers out of ‘N ’ numbers input by the user.

Algorithm SUM-EVEN

```
Step 0  START
Step 1  INPUT N
Step 2  I - 1
Step 3  SUM - 0
Step 4  REPEAT steps (a)-(d) WHILE I <- N
              (a) INPUT NUM
              (b) REM - NUM mod 2
              (c) IF REM -- 0
                      SUM - SUM + NUM
                  END IF
                          (d) I - I + 1
        END WHILE
Step 5  DISPLAY SUM
Step 6  END
```



Let us now attempt to write the C code corresponding to the above algorithm/flowchart. Readers may skip this transformation part without any loss of continuity if they have difficulty understanding C syntactic specifications. The transformation into a C program can be read after Chapter 2.

Program

```
/* Program SUM-EVEN */
#include <stdio.h>
main() {
        int n, i, sum, num, rem;

        printf("\nEnter the number of elements : ");
        scanf("%d", &n);                              /*Step 1*/
        i = 1;                                        /*Step 2*/
        sum = 0;                                      /* Step 3*/

        while(i <=n) {                                /*Step 4 while loop begins*/
                printf("\nEnter data number %d : ", i);
                scanf("%d", &num);                    /*(a) of loop*/
                rem = num % 2;                        /*(b) of loop (calculation of
                                                               remainder when num is
                                                               divided by 2)*/

                if (rem == 0)                         /*(c) of loop (num is even)*/
                        sum = sum + num;              /*code if condition is true*/
                i = i + 1;                            /*(d) of loop*/
        }                                             /*Step 4 while loop ends
        printf("\nThe sum is %d", sum);               /*Step 5*/
} /*end of main*/
```

On executing the above code after compilation and linking, the following will be the output for n = 3.

Output

Enter the number of elements : 3

Enter data number 1 : 40

Enter data number 2 : 57

Enter data number 3 : 78

The sum is 118

While writing the algorithm and the program it has been assumed that the count of the number of elements input by the user is greater or equal to 1. The logic could have been made more robust by checking whether the number input by the user in step 1 is greater or equal to 1 or not. An appropriate prompt may be displayed in case an invalid number has been input.

KEY CONCEPTS

- **HARDWARE:** The major Hardware components of a computer are the CPU, primary memory and input and output Devices. **[LO 1.1]**
- **SOFTWARE:** Two types of softwares are system and application software. **[LO 1.1]**
- **ALGORITHM:** An algorithm uses English-like statements to show the flow of logic in a step-wise manner. **[LO 1.2]**
- **FLOWCHART:** Flowchart is a graphical tool to represent logic flow. **[LO 1.2]**

ALWAYS REMEMBER

- The **central processing unit** is a VLSI circuit. **[LO 1.1]**
- The **control unit** generates control signals and sends them to the different components of a computer. **[LO 1.1]**
- The **arithmetic logic unit** has many circuits inside it that are capable of performing various arithmetic and logic operations. **[LO 1.1]**
- **Input devices** are used by the user to feed data and other information to the computer. **[LO 1.1]**
- **Output devices** are used by the computer to display an output or give the result of an operation to the user. **[LO 1.1]**
- **Memory** is of two types – the **primary** and the **secondary** memory. **[LO 1.1]**
- The **CPU** has direct connections to the **primary** memory. **[LO 1.1]**
- The **RAM** is a volatile memory. **[LO 1.1]**
- The **ROM** is a non-volatile memory. **[LO 1.1]**
- The secondary memory serves as a data reservoir. **[LO 1.1]**
- **System softwares** are those which interact with the hardware of the computer. **[LO 1.1]**
- The **operating system** acts as an interface between the user and the hardware. **[LO 1.1]**

- It is not possible to execute a program written in a high level language or assembly language directly without first translating it. **[LO 1.1]**
- A **high level language** uses a **compiler** for translation. **[LO 1.1]**
- A compiler is always typical of a programming language. **[LO 1.1]**
- Before writing a program, the **algorithm** and/or the **flowchart** for the same should be developed. **[LO 1.2]**
- Algorithms and flowcharts should be independent of programming languages. **[LO 1.2]**
- The three main programming constructs are **imperative, conditional and iterative** statements. **[LO 1.2]**
- Algorithms do not have a fixed style of writing. **[LO 1.2]**
- Algorithms should be finite and correct and the statements in it should be unambiguous. **[LO 1.2]**
- Flowcharts give us a visual depiction of logic flow. **[LO 1.2]**

REVIEW QUESTIONS

1.1 Write an algorithm and draw a flowchart to find the area of a circle of a given radius. **[LO 1.2 E]**

1.2 Write an algorithm and draw a flowchart to find the roots of a quadratic equation. **[LO 1.2 M]**

1.3 Write an algorithm and draw a flowchart to compute the simple interest on principal P, rate R% and duration T years. **[LO 1.2 E]**

1.4 Write an algorithm and draw a flowchart to compute the amount accrued if a sum of P is invested at a rate of R% for a duration of T years, where the interest is compounded every year. **[LO 1.2 E]**

1.5 Write an algorithm and draw a flowchart to find the sum of the digits of a 3-digit number. **[LO 1.2 M]**

1.6 Write an algorithm and draw a flowchart to find out whether a year is a leap year. **[LO 1.2 E]**

1.7 Write an algorithm and draw a flowchart to assign the grades of a student as per the following rules: **[LO 1.2 M]**

Marks	*Grade*
>= 90	O
>= 80 and < 90	E
>= 70 and < 80	A
>= 60 and < 70	B
>= 50 and < 60	C
>= 40 and < 50	D
< 40	F

1.8 Write an algorithm and draw a flowchart to find out whether three points (x1, y1), (x2, y2) and (x3, y3) lie on a straight line. **[LO 1.2 M]**

1.9 Write an algorithm and draw a flowchart to find out the age of a person correct to number of day(s), month(s) and year(s). **[LO 1.2 M]**

1.10 Write an algorithm and draw a flowchart to count the number of digits present in a number. **[LO 1.2 M]**

1.11 Write an algorithm and draw a flowchart to find the sum of the following series:

$$2 + 4 + 6 + \ldots + N \text{ terms}$$ **[LO 1.2 E]**

1.12 Write an algorithm and draw a flowchart to find the sum of the following series: $1! + 2 + 3! + \ldots + N!$ **[LO 1.2 M]**

1.13 Write an algorithm and draw a flowchart to find the sum of the following series: $1\times2 + 2\times3 + 3\times4 + \ldots + N$ terms **[LO 1.2 M]**

1.14 Write an algorithm and draw a flowchart to find out whether a number is an Armstrong number. **[LO 1.2 E]**

1.15 Write an algorithm and draw a flowchart to find the sum of all prime numbers between 1 and 200. **[LO 1.2 H]**

1.16 Write an algorithm and draw a flowchart to display in words the digits of a number. **[LO 1.2 M]**

1.17 Write an algorithm and draw a flowchart to find out the sum of the following series:

$1^2 + 3^2 + 5^2 + \ldots +$ N terms **[LO 1.2 E]**

E for Easy, **M** for Medium and **H** for High

Chapter **2**

Overview of C

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

- **LO 2.1** Produce an overview of C programming language
- **LO 2.2** Exemplify the elementary C concepts through sample programs
- **LO 2.3** Illustrate the use of user-defined functions and math functions through sample programs
- **LO 2.4** Describe the basic structure of C program
- **LO 2.5** Recognize the programming style of C language
- **LO 2.6** Describe how a C program is compiled and executed

HISTORY OF C

'C' seems a strange name for a programming language. But this strange sounding language is one of the most popular computer languages today because it is a structured, high-level, machine independent language. It allows software developers to develop programs without worrying about the hardware platforms where they will be implemented.

The root of all modern languages is ALGOL, introduced in the early 1960s. ALGOL was the first computer language to use a block structure. Although it never became popular in USA, it was widely used in Europe. ALGOL gave the concept of structured programming to the computer science community. Computer scientists like Corrado Bohm, Guiseppe Jacopini and Edsger Dijkstra popularized this concept during 1960s. Subsequently, several languages were announced.

In 1967, Martin Richards developed a language called BCPL (Basic Combined Programming Language) primarily for writing system software. In 1970, Ken Thompson created a language using many features of BCPL and called it simply B. B was used to create early versions of UNIX operating system at Bell Laboratories. Both BCPL and B were "typeless" system programming languages.

C was evolved from ALGOL, BCPL and B by Dennis Ritchie at the Bell Laboratories in 1972. C uses many concepts from these languages and added the concept of data types and other powerful features. 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. UNIX is one of the most popular network operating systems in use today and the heart of the Internet data superhighway.

For many years, C was used mainly in academic environments, but eventually with the release of many 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 variety of operating system and hardware platforms.

During 1970s, C had evolved into what is now known as " *traditional C* ". The language became more popular after publication of the book ' *The C Programming Language'* by Brian Kerningham and Dennis Ritchie in 1978. The book was so popular that the language came to be known as "K&R C" among the programming community. The rapid growth of C led to the development of different versions of the language that were similar but often incompatible. This posed a serious problem for system developers.

To assure that the C language remains standard, in 1983, American National Standards Institute (ANSI) appointed a technical committee to define a standard for C. The committee approved a version of C in December 1989 which is now known as ANSI C. It was then approved by the International Standards Organization (ISO) in 1990. This version of C is also referred to as C89.

During 1990's, C++, a language entirely based on C, underwent a number of improvements and changes and became an ANSI/ISO approved language in November 1977. C++ added several new features to C to make it not only a true object-oriented language but also a more versatile language.

During the same period, Sun Microsystems of USA created a new language **Java** modelled on C and C++.

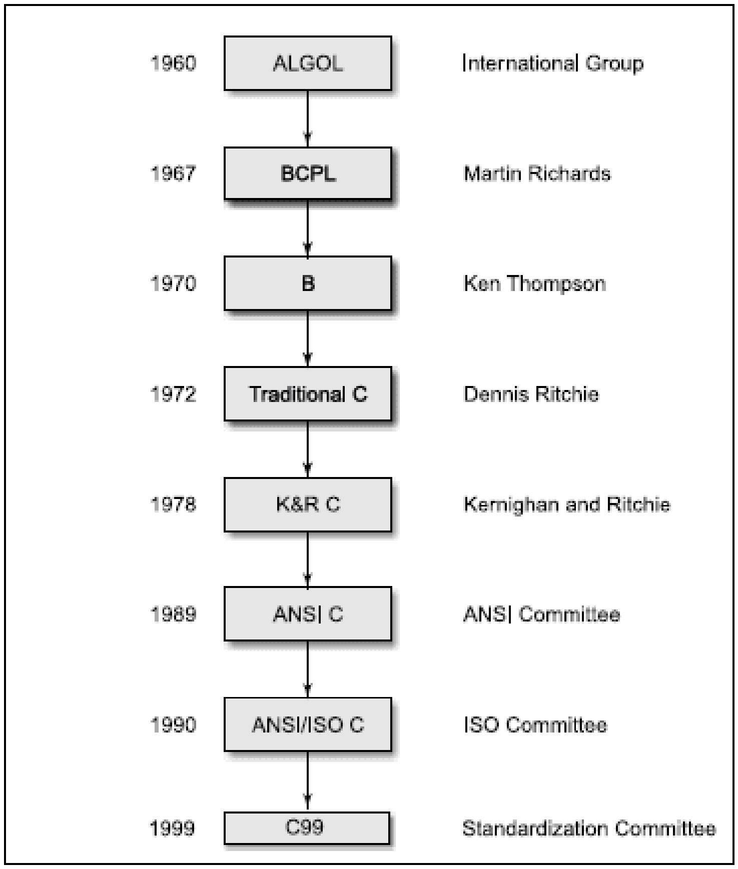


Fig. 2.1 History of ANSI C

All popular computer languages are dynamic in nature. They continue to improve their power and scope by incorporating new features and C is no exception. Although C++ and Java were evolved out of C, the standardization committee of C felt that a few features of C++/Java, if added to C, would enhance the usefulness of the language. The result was the 1999 standard for C. This version is usually referred to as C99. The history and development of C is illustrated in Fig. 2.1 .

Although C99 is an improved version, still many commonly available compilers do not support all of the new features incorporated in C99. We, therefore, discuss all the new features added by C99 in an appendix separately so that the readers who are interested can quickly refer to the new material and use them wherever possible.

IMPORTANCE OF C

LO 2.1
Produce an overview of
C programming language

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 in ANSI C 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 C library. With the availability of a large number of functions, the programming task becomes simple.

Before discussing specific features of C, we shall look at some sample C programs, and analyze and understand how they wok.

Sample Program 1: Printing a Message

Consider a very simple program given in Fig. 2.2 .

> **LO 2.2**
> Exemplify the elementary C concepts through sample programs

```
#include <stdio.h>

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

Fig. 2.2 A program to print one line of text

This program when executed will produce the following output:

I see, I remember

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

Although comments can appear anywhere, they cannot be nested in C. That means, we cannot have comments inside comments. Once the compiler finds an opening token, it ignores everything until it finds a closing token. The comment line

```
/* = = = =/* = = = = */ = = = = */
```

is not valid and therefore results in an error.

Since comments do not affect the execution speed and the size of a compiled program, we should use them liberally in our programs. 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. 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 func tion. This 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. 2.2 . 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 character 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 !

Note *Some authors recommend the inclusion of the statement.*

#include <stdio.h>

at the beginning of all programs that use any input/output library functions. However, this is not necessary for the functions ***printf*** and ***scanf*** which have been defined as a part of the C language. See Chapter 5 for more on input and output functions.

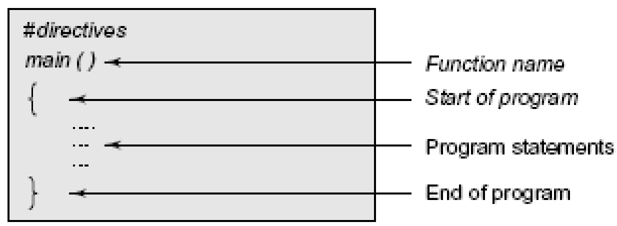


Fig. 2.3 Format of simple C programs

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 example that printed **I see, I remember** is one of the simplest programs. Figure 2.3 highlights the general format of such simple programs. All C programs need a **main** function.

The ***main*** Function

The main is a part of every C program. C permits different forms of main statement. Following forms are allowed.

- main()
- int main()
- void main()
- main(void)

- void main(void)
- int main(void)

The empty pair of parentheses indicates that the function has no arguments. This may be explicitly indicated by using the keyword **void** inside the parentheses. We may also specify the keyword **int** or **void** before the word **main** . The keyword **void** means that the function does not return any information to the operating system and **int** means that the function returns an integer value to the operating system. When **int** is specified, the last statement in the program must be “return 0”. For the sake of simplicity, we use the first form in our 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. 2.4 .

```
/* Programm 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 */
}                                            /*      line13 */
```

Fig. 2.4 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**) and **amount** is a floating (**float**) point number. Declaration statements must appear at the beginning of the functions as shown in Fig. 2.4 . 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. 2 f* 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.

SAMPLE PROGRAM 3: INTEREST CALCULATION

The program in Fig. 2.5 calculates the value of money at the end of each year of investment, assuming an interest rate of 11 percent and prints the year, and the corresponding amount, in two columns. The output is shown in Fig. 2.6 for a period of 10 years with an initial investment of 5000.00. The program uses the following formula:

Value at the end of year = Value at start of year (1 + interest rate)

In the program, the variable **value** represents the value of money at the end of the year while **amount** represents the value of money at the start of the year. The statement

amount = value ;

makes the value at the end of the *current* year as the value at start of the *next* year.

```
/*———— INVESTMENT PROBLEM ————*/
#define PERIOD 10
#define PRINCIPAL   5000.00
/*———— MAIN PROGRAM BEGINS ————*/
main()
{ /*———— DECLARATION STATEMENTS ————*/
```

```
     int year;
     float amount, value, inrate;
/*———————— ASSIGNMENT STATEMENTS ————————*/
     amount - PRINCIPAL;
     inrate - 0.11;
     year - 0;
/*———————— COMPUTATION STATEMENTS ————————*/
/*—————— COMPUTATION USING While LOOP ——————*/
     while(year <- PERIOD)
     {  printf("%2d  %8.2f\n",year, amount);
          value - amount + inrate * amount;
          year - year + 1;
          amount - value;
   }
/*—————————— while LOOP ENDS —————————*/
}
/*—————————— PROGRAM ENDS —————————*/
```

Fig. 2.5 Program for investment problem

Let us consider the new features introduced in this program. The second and third lines begin with **#define** directives. A **#define** directive defines value to a *symbolic constant* for use in the program. Whenever a symbolic name is encountered, the compiler substitutes the value associated with the name automatically. To change the value, we have to simply change the definition. In this example, we have defined two symbolic constants **PERIOD** and **PRINCIPAL** and assigned values 10 and 5000.00 respectively. These values remain constant throughout the execution of the program.

```
0       5000.00
1       5550.00
2       6160.50
3       6838.15
4       7590.35
5       8425.29
6       9352.07
7      10380.00
8      11522.69
9      12790.00
10     14197.11
```

Fig. 2.6 Output of the investment program

The ***#define*** Directive

A **#define** is a preprocessor compiler directive and not a statement. Therefore **#define** lines should not end with a semicolon. Symbolic constants are generally written in uppercase so that they are easily distinguished from lowercase variable names. **#define** directives are usually placed at the beginning before the **main()** function. Symbolic constants are not declared in declaration section. Preprocessor directives are discussed in Chapter 15.

We must note that the defined constants are not variables. We may not change their values within the program by using an assignment statement. For example, the statement

```
PRINCIPAL = 10000.00;
```

is illegal.

The declaration section declares **year** as integer and **amount, value** and **inrate** as floating point numbers. Note all the floating-point variables are declared in one statement. They can also be declared as

```
float amount;
float value;
float inrate;
```

When two or more variables are declared in one statement, they are separated by a comma.

All computations and printing are accomplished in a **while** loop. **while** is a mechanism for evaluating repeatedly a statement or a group of statements. In this case as long as the value of **year** is less than or equal to the value of **PERIOD** , the four statements that follow **while** are executed. Note that these four statements are grouped by braces. We exit the loop when **year** becomes greater than **PERIOD** . The concept and types of loops are discussed in Chapter 7.

C supports the basic four arithmetic operators (–, +, *, /) along with several others. They are discussed in Chapter 4.

TEST YOUR SKILLS

1. Write a program that will print your mailing address in the following form: **[M]**

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

SAMPLE PROGRAM 4: USE OF SUBROUTINES

LO 2.3
Illustrate the use of user-defined functions and math functions through sample programs

So far, we have used only **printf** function that has been provided for us by the C system. The program shown in Fig. 2.7 uses a user-defined function. A function defined by the user is equivalent to a subroutine in FORTRAN or subprogram in BASIC.

Figure 2.7 presents a very simple program that uses a **mul ( )** function. The program will print the following output.

Multiplication of 5 and 10 is 50

```
/*———— PROGRAM USING FUNCTION ————*/
int mul (int a, int b); /*—— DECLARATION ———*/
/*———— MAIN PROGRAM BEGINS —————*/
     main ()
     {
          int a, b, c;
          a = 5;
          b = 10;
          c = mul (a,b);
```

```
          printf ("multiplication of %d and %d is %d",a,b,c);
     }
/* ———————      MAIN PROGRAM ENDS
                MUL() FUNCTION STARTS ————————*/
          int mul (int x, int y)
          int p;
          {
                p = x*y;
                return(p);
          }
/* ————————— MUL () FUNCTION ENDS ————————*/
```

Fig. 2.7 A program using a user-defined function

The **mul ( )** function multiplies the values of **x** and **y** and the result is returned to the **main ( )** function when it is called in the statement

c = mul (a, b);

The **mul ( )** has two *arguments* **x** and **y** that are declared as integers. The values of **a** and **b** are passed on to **x** and **y** respectively when the function **mul ( )** is called. User-defined functions are considered in detail in Chapter 10.

SAMPLE PROGRAM 5: USE OF MATH FUNCTIONS

We often use standard mathematical functions such as cos, sin, exp, etc. We shall see now the use of a mathematical function in a program. The standard mathematical functions are defined and kept as a part of C **math library** . If we want to use any of these mathematical functions, we must add an **#include** instruction in the program. Like **#define,** it is also a compiler directive that instructs the compiler to link the specified mathematical functions from the library. The instruction is of the form

#include <math.h>

math.h is the filename containing the required function. Figure 2.8 illustrates the use of cosine function. The program calculates cosine values for angles 0, 10, 20…………180 and prints out the results with headings.

```
/*——— PROGRAM USING COSINE FUNCTION ——— */
#include <math.h>
#define     PI 3.1416
#define     MAX  180
main ( )
{
     int angle;
     float x,y;
     angle - 0;
     printf("  Angle     Cos(angle)\n\n");
     while(angle <- MAX)
```

```
        {
            x = (PI/MAX)*angle;
            y = cos(x);
            printf("%15d %13.4f\n", angle, y);
            angle = angle + 10;
        }
    }
Output
            Angle           Cos(angle)
              0               1.0000
             10               0.9848
             20               0.9397
             30               0.8660
             40               0.7660
             50               0.6428
             60               0.5000
             70               0.3420
             80               0.1736
             90              -0.0000
            100              -0.1737
            110              -0.3420
            120              -0.5000
            130              -0.6428
            140              -0.7660
            150              -0.8660
            160              -0.9397
            170              -0.9848
            180              -1.0000
```

Fig. 2.8 Program using a math function

Another **#include** instruction that is often required is

#include <stdio.h>

stdio.h refers to the *standard* I/O header file containing standard input and output functions

The *#include* Directive

As mentioned earlier, *C* programs are divided into modules or functions. Some functions are written by users, like us, and many others are stored in the *C* library. Library functions are grouped category-wise and stored in different files known as *header files* . If we want to access the functions stored in the library, it is necessary to tell the compiler about the files to be accessed.

This is achieved by using the preprocessor directive **#include** as follows:

#include<filename>

filename is the name of the library file that contains the required function definition. Preprocessor directives are placed at the beginning of a program.

> **TEST YOUR SKILLS**
>
> Write a C program to input two numbers: a and b. Next, compute the value of a raised to the power of b. **[LO 2.3 E]**

A list of library functions and header files containing them are given in Appendix III.

BASIC STRUCTURE OF C PROGRAMS

> **LO 2.4**
> Describe the basic structure of 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. 2.9 .

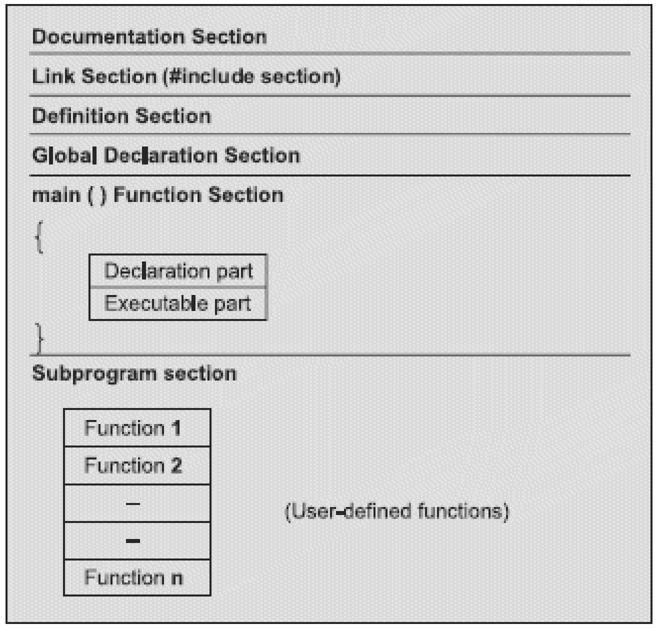


Fig. 2.9 An overview of a C program

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(;).

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.

Programming Style

LO 2.5
Recognize the Programming style of C language

Unlike some other programming languages (COBOL, FORTRAN, etc.,) C is a *free-form* _language. That is, the C compiler does not care, where on the line we begin typing. While this may be a licence for bad programming, we should try to use this fact to our advantage in developing readable programs. Although several alternative styles are possible, we should select one style and use it with total consistency.

First of all, we must develop the habit of writing programs in lowercase letters. C program statements are written in lowercase letters. Uppercase letters are used only for symbolic constants.

Braces, group program statements together and mark the beginning and the end of functions. A proper indentation of braces and statements would make a program easier to read and debug. Note how the braces are aligned and the statements are indented in the program of Fig. 2.5 .

Since C is a free-form language, we can group statements together on one line. The statements

```
a = b;
x = y + 1;
z = a + x;
```

can be written on one line as

```
a = b; x = y+1; z = a+x;
```

The program

```
main( )
{
        printf("hello C");
}
```

may be written in one line like

```
main( ) {printf("Hello C")};
```

However, this style make the program more difficult to understand and should not be used. In this book, each statement is written on a separate line.

The generous use of comments inside a program cannot be overemphasized. Judiciously inserted comments not only increase the readability but also help to understand the program logic. This is very important for debugging and testing the program.

EXECUTING A 'C' PROGRAM

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; and
4. Executing the program.

Figure 2.10 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.

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

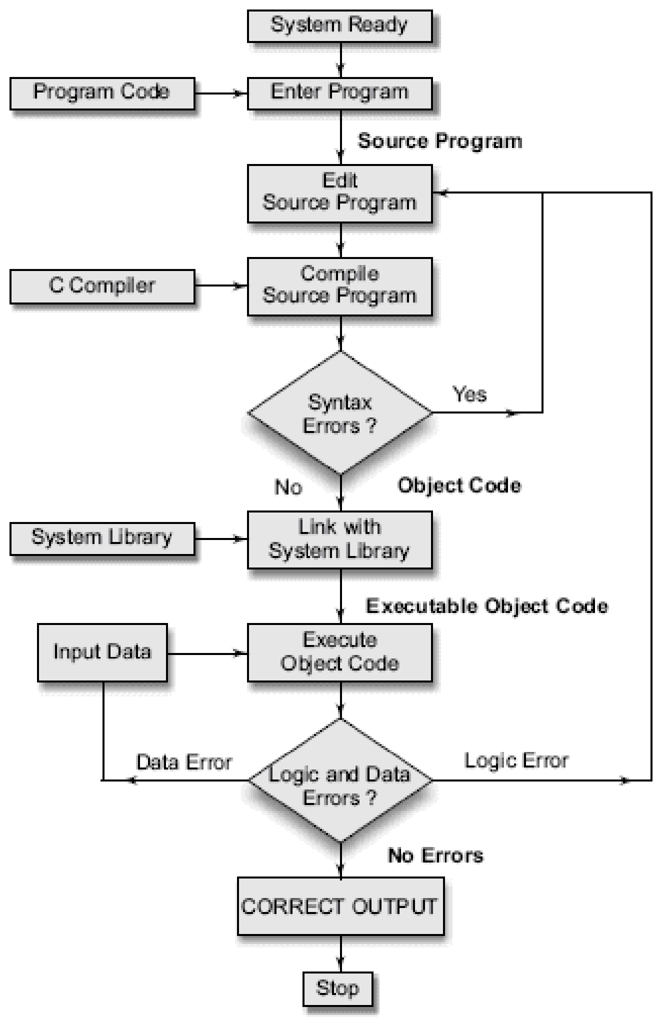


Fig. 2.10 Process of compiling and runnig a C program

Source code is the file containing the C source program. This program is input to a compiler. The compiler is a system software that translates the source program to **object code** . The object code contains an equivalent set of machine instructions corresponding to the source program which is understood by the computer. A given C program may make use of several library functions. Thus, the object code of the program would have to be linked to the object codes of the library functions. The linker is a system software that links together all the object codes to generate the final **executable code** . This executable code is submitted to the operating system for execution whenever required. The generation of the executable code is shown in Fig. 2.11 .

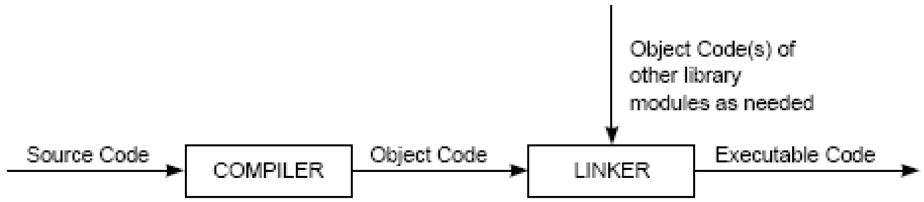


Fig. 2.11 Generation of executable code from source code

In this respect, it is worth noting that the compiler is specific to a programming language, and the object code is dependent upon the platform on which it is meant to run.

UNIX System

Creating the Program

LO 2.6
Describe how a C program is compiled and executed

Once we load the UNIX operating system into the memory, the computer is ready to receive 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 then 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 source 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 any 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 produce 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 overwritten 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 files 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. 2.12 .

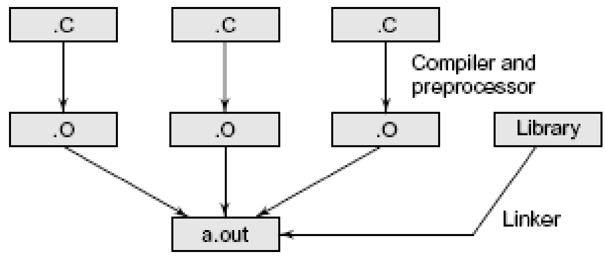


Fig. 2.12 Compilation of multiple files

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

will compile the source files *mod1.c* and *mod2.c* into objects 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 files 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 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.

WINDOWS SYSTEM

Unlike in the Linux system, the Windows system does not come with an in-built C compiler. There are several IDEs. An IDE is an integrated development environment which provides programmers the ways and means to write, save, compile, edit, link and run programs. These IDEs have to be separately installed on a Windows system to be used. Typical examples are Borland C/C++, Microsoft Visual Studio, Dev-C++, Turbo C/C++, etc. Most of these IDEs have an extremely friendly graphic user interface. Thus, most of the work can be done by the click of a mouse. Some of these IDEs also work in the DOS environment. The IDE would then have to be invoked by opening the DOS box/command prompt. Once in the IDE environments, the tools offered by it would help us perform the required activities of compiling, linking and executing.

KEY CONCEPTS

- **#define :** Is a preprocessor compiler directive. **[LO 2.2]**
- **printf :** Is a predefined standard C function that writes the output to the stdout (standard output) stream. **[LO 2.2]**
- **scanf :** Is a predefined standard C function that reads formatted input from stdin (standard input) stream. **[LO 2.2]**

- **PROGRAM :** Is a sequence of instructions written to perform a specific task in the computer. **[LO 2.4]**

ALWAYS REMEMBER

- C is a structured, high-level, machine independent language. **[LO 2.1]**
- ANSI C and C99 are the standardized versions of C language. **[LO 2.1]**
- C combines the capabilities of assembly language with the features of a high level language. **[LO 2.1]**
- C is robust, portable and structured programming language. **[LO 2.1]**
- Every C program requires a **main()** function (Use of more than one **main()** is illegal). The place **main** is where the program execution begins. **[LO 2.2]**
- The execution of a function begins at the opening brace of the function and ends at the corresponding closing brace. **[LO 2.2]**
- C programs are written in lowercase letters. However, uppercase letters are used for symbolic names and output strings. **[LO 2.2]**
- 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. **[LO 2.2]**
- Every program statement in a C language must end with a semicolon. **[LO 2.2]**
- All variables must be declared for their types before they are used in the program. **[LO 2.2]**
- 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. **[LO 2.2]**
- 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. **[LO 2.2]**

- The sign # of compiler directives must appear in the first column of the line. **[LO 2.2]**
- 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. **[LO 2.3]**
- The structure of a C program comprises of various sections including Documentation, Link, Definition, Global Declaration, main () function and Sub program section. **[LO 2.4]**
- C is a free-form language and therefore a proper form of indentation of various sections would improve legibility of the program. **[LO 2.5]**
- The execution of a C program involves a series of steps including: creating the program, compiling the program, linking the program with functions from C library and executing the program. **[LO 2.6]**
- The command used for running a C program in UNIX system is *a.out* . **[LO 2.6]**
- The command used for running a C program in MS-DOS system is *file.exe* where file is the name of the program that has already been compiled. **[LO 2.6]**
- When braces are used to group statements, make sure that the opening brace has a corresponding closing brace. **[LO 2.6]**

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

2.1 Which of the following characterize the typical features of C language? **[LO 2.1 M]**

(a) High level

(b) Machine independent

(c) Structured

(d) All of the above

2.2 A typical C program comprises the following sections:

- A - Documentation
- B - Definition
- C - Link
- D - main ()
- E - Subprogram section
- F - Global declaration

Identify the correct sequence of the C program sections: **[LO 2.4 H]**

2.3 Which of the following types of errors are detected during program compilation? **[LO 2.6 M]**

(a) Syntax

(b) Logical

(c) Data

(d) All of the above

2.4 Consider the following C statements:

A	a-b; x -1; z-a+x;
B	a-b;x-1;z-a+x;

Which of the above programming styles are correct in C? **[LO 2.5 M]**

(a) A is correct, B is incorrect

(b) A is incorrect, B is correct

(c) Both A and B are correct

(d) Both A and B are incorrect

2.5 C language comprises ______ number of keywords. **[LO 2.1 E]**

(a) 32

(b) 16

(c) 64

(d) 128

REVIEW QUESTIONS

2.1 State whether the following statements are *true* or *false* .

(a) Every line in a C program should end with a semicolon. **[LO 2.2 E]**

(b) The closing brace of the **main( )** in a program is the logical end of the program. **[LO 2.2 E]**

(c) Comments cause the computer to print the text enclosed between /* and */ when executed. **[LO 2.2 E]**

(d) Every C program ends with an END word. **[LO 2.2 M]**

(e) A **printf** statement can generate only one line of output. **[LO 2.2 M]**

(f) The purpose of the header file such as **stdio.h** is to store the source code of a program. **[LO 2.3 M]**

(g) A line in a program may have more than one statement. **[LO 2.5 M]**

(h) Syntax errors will be detected by the compiler. **[LO 2.6 M]**

(i) In C language lowercase letters are significant. **[LO 2.2 H]**

(j) **main( )** is where the program begins its execution. **[LO 2.2 H]**

(k) Every C program must have at least one user-defined function. **[LO 2.3 E]**

(l) Declaration section contains instructions to the computer. **[LO 2.4 E]**

(m) Only one function may be named **main( )** **[LO 2.2 M]**

(n) Comments serve as internal documentation for programmers. **[LO 2.2 E]**

(o) In C, we can have comments inside comments. **[LO 2.2 M]**

(p) Use of comments reduces the speed of execution of a program. **[LO 2.2 H]**

(q) A comment can be inserted in the middle of a statement. **[LO 2.2 H]**

2.2 Fill in the blanks with appropriate words in each of the following statements.

(a) Every program statement in a C program must end with a ___________. **[LO 2.2 E]**

(b) The _____________ Function is used to display the output on the screen. **[LO 2.2 E]**

(c) The _____________ header file contains mathematical functions. **[LO 2.3 M]**

(d) The escape sequence character _____________ causes the cursor to move to the next line on the screen. **[LO 2.2 H]**

(e) C programs are written in lower case letters whereas upper case letters are mainly used to define ___________ **[LO 2.2 M]**

(f) C language offers several built-in __________ that can be used in a program by including the relevant header files. **[LO 2.1 E]**

(g) _____________ indicates the starting point of a C program. **[LO 2.2 M]**

2.3 Remove the semicolon at the end of the **printf** statement in the program of Fig. 2.2 and execute it. What is the output? **[LO 2.2 M]**

2.4 In the Sample Program 2, delete line-5 and execute the program. How helpful is the error message? **[LO 2.2 H]**

2.5 Modify the Sample Program 3 to display the following output: **[LO 2.2 M]**

Year	Amount
1	5500.00
2	6160.00
—	___________
—	___________
10	14197.11

2.6 Why and when do we use the **#define** directive? **[LO 2.2 E]**

2.7 Why and when do we use the **#include** directive? **[LO 2.3 M]**

2.8 What does **void main(void)** mean? **[LO 2.2 H]**

2.9 Distinguish between the following pairs:

(a) main( ) and void main(void) **[LO 2.2 H]**

(b) int main( ) and void main( ) **[LO 2.2 M]**

2.10 Why do we need to use comments in programs? **[LO 2.2 E]**

2.11 Why is the look of a program is important? **[LO 2.5 E]**

2.12 Where are blank spaces permitted in a C program? **[LO 2.5 sM]**

2.13 Describe the structure of a C program. **[LO 2.4 M]**

2.14 Describe the process of creating and executing a C program under UNIX system. **[LO 2.6 M]**

2.15 How do we implement multiple source program files? **[LO 2.6 H]**

DEBUGGING EXERCISES

2.1 Find errors, if any, in the following program: **[LO 2.2 E]**

```
/* A simple program
int main( )
{
     /* Does nothing */
}
```

2.2 Find errors, if any, in the following program: **[LO 2.2 M]**

```
#include (stdio.h)
void main(void)
{
     print("Hello C");
}
```

2.3 Find errors, if any, in the following program: **[LO 2.3 H]**

```
Include <math.h>
main { }
(
     FLOAT X;
     X - 2.5;
     Y - exp(x);
     Print(x,y);
)
```

PROGRAMMING EXERCISES

2.1 Write a program to display the equation of a line in the form

$$ax + by = c$$

for a = 5, b = 8 and c = 18. **[LO 2.2 E]**

2.2 Write a C program that uses an in-built function to draw a 3D bar. **[LO 2.3 E]**

2.3 Write a program to output the following multiplication table: **[LO 2.2 M]**

```
5 × 1 = 5
5 × 2 = 10
5 × 3 = 15
•     •
•     •
5 × 10 = 50
```

2.4 Given the values of three variables a, b and c, write a program to compute and display the value of x, where

$$x = \frac{a}{b - c}$$

Execute your program for the following values:

(a) a = 250, b = 85, c = 25 **[LO 2.2 M]**

(b) a = 300, b = 70, c = 70 **[LO 2.2 M]**

Comment on the output in each case.

2.5 Write a C program that reads the value of distance travelled by a car and the time taken for the same. Next, compute the speed at which the car travelled. **[LO 2.2 M]**

2.6 Write a C program to print the current system date. **[LO 2.3 M]**

2.7 Given two integers 20 and 10, write a program that uses a function add( ) to add these two numbers and sub( ) to find the difference of these two numbers and then display the sum and difference in the following form: **[LO 2.3 M]**

```
20 + 10 = 30
20 – 10 = 10
```

2.8 Modify the above program to provide border lines to the address. **[LO 2.2 H]**

2.9 Write a program using one print statement to print the pattern of asterisks as shown below: **[LO 2.2 H]**

```
*
*   *
*   *   *
*   *   *   *
```

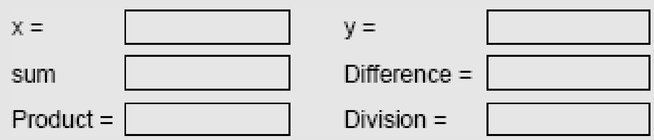
2.10 Write a program that will print the following figure using suitable characters. **[LO 2.2 H]**

2.11 Area of a triangle is given by the formula

$$A = \sqrt{S(S-a)(S-b)(S-c)}$$

Where a, b and c are sides of the triangle and 2S = a + b + c. Write a program to compute the area of the triangle given the values of a, b and c. **[LO 2.2 H]**

2.12 Write a program to display the following simple arithmetic calculator **[LO 2.2 H]**



2.13 Distance between two points (x_1, y_1) and (x_2, y_2) is governed by the formula

$$D^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

Write a program to compute D given the coordinates of the points. **[LO 2.3 H]**

2.14 A point on the circumference of a circle whose center is (o, o) is (4,5). Write a program to compute perimeter and area of the circle. (Hint: use the formula given in the Ex. 2.11) **[LO 2.3 H]**

2.15 The line joining the points (2,2) and (5,6) which lie on the circumference of a circle is the diameter of the circle. Write a program to compute the area of the circle. **[LO 2.3 H]**

INTERVIEW QUESTIONS

2.1 Name some of the key features of C programming language. **[LO 2.1 E]**

2.2 Which symbol is used to indicate the end of most C programming statements? **[LO 2.2 E]**

2.3 Do you consider C as a high-level programming language? **[LO 2.1 M]**

2.4 Which punctuation symbols are used to represent code blocks in C? **[LO 2.2 M]**

2.5 Which function is the starting point in a C program? **[LO 2.2 M]**

2.6 Which return value of main() function indicates successful completion of the program? **[LO 2.2 H]**

E for Easy, **M** for Medium and **H** for High

Chapter 3

Constants, Variables and Data Types

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

- LO 3.1 Know the C character set and keywords
- LO 3.2 Describe constants and variables
- LO 3.3 Identify the various C data types
- LO 3.4 Discuss how variables are used in a program
- LO 3.5 Explain how constants are used in a program

INTRODUCTION

A pro gramming 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.

CHARACTER SET

LO 3.1
Know the C character set and keywords

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 entire character set is given in Table 3.1.

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

Table 3.1 C Character Set

Letters		*Digits*
Uppercase A.....Z		All decimal digits 0 9
Lowercase a.....z		
	Special Characters	
, comma		& ampersand
. period		^ caret
; semicolon		* asterisk
: colon		– minus sign
? question mark		+ plus sign
‘ apostrophe		< opening angle bracket
“ quotation mark		(or less than sign)
! exclamation mark		> closing angle bracket
| vertical bar		(or greater than sign)
/ slash		(left parenthesis
\ backslash		) right parenthesis
~ tilde		[left bracket
_ under score		] right bracket
$ dollar sign		{ left brace
% percent sign		} right brace
		# number sign
	White Spaces	
	Blank space	
	Horizontal tab	
	Carriage return	
	New line	
	Form feed	

Trigraph Characters

Table 3.2 ANSI C Trigraph Sequences

Trigraph sequence	Translation
??=	# number sign
??(	[left bracket
??)	] right bracket
??<	{ left brace
??>	} right brace
??!	| vetical bar
??/	\ back slash
??/	^ caret
??-	~ tilde

Many non-English keyboards do not support all the characters mentioned in Table 3.1. ANSI C introduces the concept of “trigraph” sequences to provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters (two question marks followed by another character) as shown in Table 3. 2.

For example, if a keyboard does not support square brackets, we can still use them in a program using the trigraphs ??(and ??).

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.

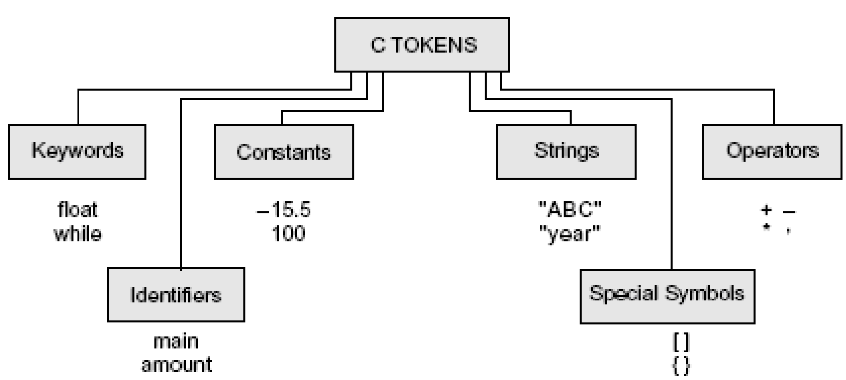


Fig. 3.1 C tokens and examples

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 of ANSI C are listed in Table 3.3. All keywords must be written in lowercase. Some compilers may use additional keywords that must be identified from the C manual.

Note *C99 adds some more keywords. See the Appendix "C99 Features".*

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.

Table 3.3 ANSI C Keyword

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

Rules for Identifiers

1. First character must be an alphabet (or underscore).
2. Must consist of only letters, digits or underscore.
3. Only first 31 characters are significant.
4. Cannot use a keyword.
5. Must not contain white space

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 .

LO 3.2
Describe constants and variables

Integer Constants

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

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

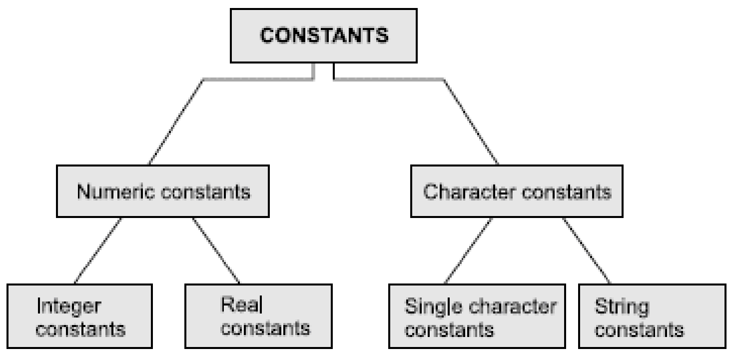


Fig. 3.2 Basic types of C constants

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

15 750 20,000 $1000

are illegal numbers.

Note *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 integer are:

037 0 0435 0551

A sequence of digits preceded by 0x or 0X is considered as *hexadecimal* integer. They may also include alphabets A through F or a through f. The letter 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. Examples:

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

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

WORKED-OUT PROBLEM 3.1 E

Representation of integer constants on a 16-bit computer.

The program in Fig. 3.3 illustrates the use of integer constants on a 16-bit machine. 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
            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 3777
```

Fig. 3.3 Representation of integer constants on 16-bit machine

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 for 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 3.4.

Table 3.4 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. Example of character constants is as follows:

‘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

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 represents an integer value, it is also possible to perform arithmetic operations on character constants. They are discussed in Chapter 9.

String Constants

Table 3.5 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

A string constant is a sequence of characters 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 value 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 3.5. Note that each one of them represents one character, although they consist of two characters. These characters combinations are known as *escape sequences* .

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 Worked Out Problem 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. (In C99, at least 63 characters are significant.)
3. Uppercase and lowercase are significant. That is, the variable **Total** is not the same as **total** or **TOTAL** .
4. It 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 3.6.

Table 3.6 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	
int_type	Valid	Keyword may be part of a name

DATA TYPES

C language is rich in its *data types.* Storage representations and machine instructions to handle constants differ from machine to machine. The variety

LO 3.3
Identify the various C data types

of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.

ANSI C supports three classes of data types:

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

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.

All C compilers support five fundamental data types, namely integer **(int),** character (**char**), floating point (**float),** double-precision floating point (**double)** and **void.** 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 3.7. We discuss briefly each one of them in this section.

Note *C99 adds three more data types, namely* ***_Bool****,* ***_Complex****, and* ***_Imaginary****. See the Appendix "C99 Features".*

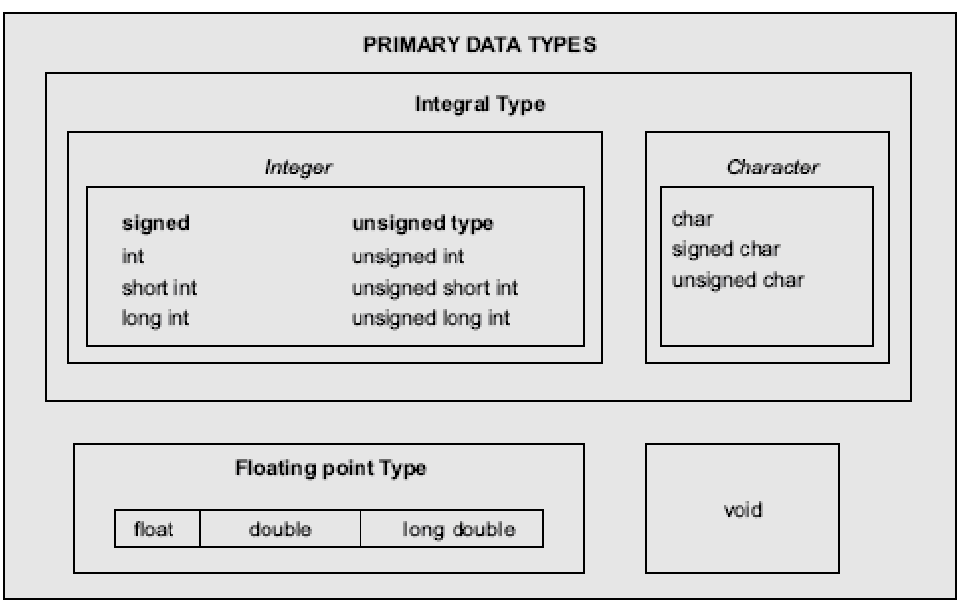


Fig. 3.4 Primary data types in C

Integer Types

Table 3.7 Size and Range of Basic Data Types on 16-bit Machines

Data type	*Range of values*
char	–128 to 127
int	–32,768 to 32,767
float	3.4e–38 to 3.4e+e38
double	1.7e–308 to 1.7e+308

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.

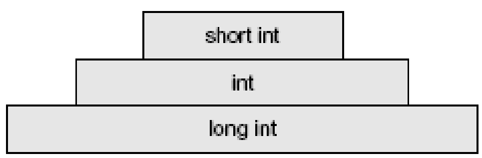


Fig. 3.5 Integer types

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. ANSI C defines these types so that they can be organized from the smallest to the largest, as shown in Fig. 3.5 . 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 from 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 3.8 shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

Note *C99 allows **long long** integer types. See the Appendix "C99 Features".*

Table 3.8 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

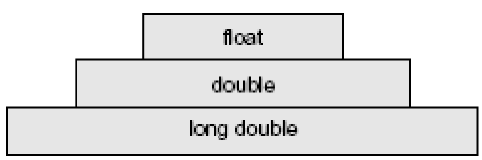


Fig. 3.6 Floating-point types

Floating point (or real) numbers are stored in 32 bits (on all 16 bit 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. Floating point numbers are represented within the computer's memory using the IEEE standard. The relationship among floating types is illustrated Fig. 3.6 .

Void Types

The **void** type has no values. This is usually used to specify the type of functions. The type of a function is said to be **void** when it does not return any value to the calling function. It can also play the role of a generic type, meaning that it can represent any of the other standard types.

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 and 255, **signed chars** have values from –128 to 127.

TEST YOUR SKILLS

Write a C program that makes use of different data types to input and print student's name, age and average marks. **[M]**

DECLARATION OF VARIABLES

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

LO 3.4
Discuss how variables are used in a program

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

Declaration of variables must be done at the beginning of a statement block before they are used in a 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 3.9 shows various data types and their keyword equivalents.

Table 3.9 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. 3.7 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. 3.7 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 variable as unsigned, then we must do so using both the terms like **unsigned char** .

Default Values of Constants

Integer constants, by default, represent **int** type data. We can override this default by specifying unsigned or long after the number (by appending U or L) as shown below:

Literal	Type	Value
+111	int	111
–222	int	–222
45678U	unsigned int	45,678
–56789L	long int	–56,789
987654UL	unsigned long int	9,87,654

Similarly, floating point constants, by default represent **double** type data. If we want the resulting data type to be **float** or **long double** , we must append the letter f or F to the number for **float** and letter l or L for **long double** as shown below:

Literal	Type	Value
0.	double	0.0
.0	double	0.0
12.0	double	12.0
1.234	double	1.234
–1.2f	float	–1.2
1.23456789L	long double	1.23456789

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 inclared as **int** variable and name1[50] and name2[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 assigned 0, 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 3.10.

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.

Table 3.10 Storage Classes and Their Meaning

Storage class	*Meaning*
auto	Local variable known 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 variables known to all functions in the file and to functions in other files which have been linked to this file.
register	Local variable which is requested to be stored in a CPU register subject to availability by the programmer.

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 wll contain garbage.

WORKED-OUT PROBLEM 3.2

E

The program in Fig. 3.8 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. 3.9 .

```
Program
            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. 3.8 Use of scanf function for interactive computing

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. 3.8 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.

WORKED-OUT PROBLEM 3.3

Program in Fig. 3.9 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 %.12lf 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 up to six decimal places.

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.

```
Program
          main()
          {
          /*..........DECLARATIONS............................*/
               float    x, p ;
               double     y, q ;
               unsigned   k ;
          /*..........DECLARATIONS 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) ;
               printf("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 = 9.876543210000
               y = 9.876543
               k = 54321 p = 1.00000 q = 1.000000000000
```

Fig. 3.9 Examples of assignments

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 and 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'. The value is assigned to the variable **number** .

WORKED-OUT PROBLEM 3.4 M

Worked-Out Problem 3 discussed in Chapter 2 can be converted into a more flexible interactive program using **scanf** as shown in Fig. 3.10 .

In this case, computer requests the user to input the values of the amount to be invested, interest 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, up to 'period' and produces output as shown in Fig. 3.10 .

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. 3.10 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 no detect such errors.

TEST YOUR SKILLS

Write a program that requests two float type numbers from the user and then divides the first number by the second and display the result along with the numbers. **[M]**

DEFINING SYMBOLIC CONSTANTS

LO 3.5
Explain how constants are used in a program

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 mark-sheets 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. These are

1. problem in modification of the program and
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 lowercase letters
#define PRICE$ 100	Invalid	$ symbol is not permitted in name

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;

The volatile type qualifier is particularly relevant for multi-threaded programs and programs involving parallel processes.

> **Note** *C99 adds another qualifier called* restrict*. See the Appendix "C99 Features".*

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 h old. 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.

KEY CONCEPTS

- **IDENTIFIERS:** Refer to the names of variables, functions and arrays. **[LO 3.1]**
- **CONSTANTS:** Refer to fixed values that do not change during the execution of a program. **[LO 3.2]**
- **STRING CONSTANT:** Is a sequence of characters enclosed in double quotes where characters could be letters, numbers, special characters or blank space. **[LO 3.2]**
- **VARIABLE:** Is a data name that may be used to store a data value. **[LO 3.2]**
- **STORAGE CLASS:** Provides information related to the location and visibility of a variable. **[LO 3.4]**
- **scanf :** Is a predefined standard C function that reads formatted input from stdin (standard input) stream. **[LO 3.4]**

ALWAYS REMEMBER

- Do not use the underscore as the first character of identifiers (or variable names) because many of the identifiers in the system library start with underscore. **[LO 3.1]**
- Use only 31 or less characters for identifiers. This helps ensure portability of programs. **[LO 3.1]**
- Do not use keywords or any system library names for identifiers. **[LO 3.1]**
- Use meaningful and intelligent variable names. **[LO 3.2]**
- Do not create variable names that differ only by one or two letters. **[LO 3.2]**
- Integer constants, by default, assume **int** types. To make the numbers **long** or **unsigned** , we must append the letters L and U to them. **[LO 3.2]**
- Floating point constants default to **double** . To make them to denote **float** or **long double** , we must append the letters F or L to the numbers. **[LO 3.2]**

- Use single quote for character constants and double quotes for string constants. **[LO 3.2]**
- A character is stored as an integer. It is therefore possible to perform arithmetic operations on characters. **[LO 3.2]**
- Do not use lowercase l for long as it is usually confused with the number 1. **[LO 3.3]**
- C does not provide any warning or indication of overflow. It simply gives incorrect results. Care should be exercised in defining correct data type. **[LO 3.3]**
- Each variable used must be declared for its type at the beginning of the program or function. **[LO 3.4]**
- All variables must be initialized before they are used in the program. **[LO 3.4]**
- Do not combine declarations with executable statements. **[LO 3.4]**
- Do not use semicolon at the end of **#define** directive. **[LO 3.4]**
- The character # should be in the first column. **[LO 3.4]**
- Do not give any space between # and **define.** **[LO 3.4]**
- A variable defined before the main function is available to all the functions in the program. **[LO 3.4]**
- A variable defined inside a function is local to that function and not available to other functions. **[LO 3.4]**
- A variable can be made constant either by using the preprocessor command **#define** at the beginning of the program or by declaring it with the qualifier **const** at the time of initialization. **[LO 3.5]**

BRIEF CASES

1. Calculation of Average of Numbers[LO 3.4, 3.5 M]

A program to calculate the average of a set of N numbers is given in Fig. 3.11 .

```
Program

            #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 = 3.880
```

Fig. 3.11 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[LO 3.5 M]

The program presented in Fig. 3.12 converts the given temperature in fahrenheit to celsius using the following conversion formula:

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

Program

```
#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. 3.12 Temperature conversion—fahrenheit-celsius

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. An 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- c olumn output as shown.

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

3.1 Which of the following escape sequences moves the cursor position to the new line? **[LO 3.2 E]**

(a) \n

(b) \r

(c) \t

(d) \v

3.2 Which of the following is not a C language keyword? **[LO 3.1 M]**

(a) Volatile

(b) enum

(c) unsigned

(d) go

3.3 Which of the following is the correct way of specifying long signed integer data type? **[LO 3.3 M]**

(a) signed long int

(b) long int

(c) unsigned long int

(d) Both 1 and 2

3.4 Which of the following is not a C storage class? **[LO 3.4 M]**

(a) auto

(b) static

(c) register

(d) volatile

3.5 Which of the following is the correct way of defining a symbolic constant? **[LO 3.5 M]**

(a) #define MAX_MARKS 100

(b) #define MAX_MARKS 100

(c) #define 100 MAX_MARKS

(d) #define 100 MAX_MARKS

3.6 Which of the following statement is true about character constants? **[LO 3.2 H]**

(a) The statement printf("%c", '97'); will print the letter a.

(b) Each character constant represents an integer value.

(c) It is possible to perform arithmetic operations on character constants.

(d) All of the above.

REVIEW QUESTIONS

3.1 State whether the following statements are *true* or *false* .

(a) All variables must be given a type when they are declared. **[LO 3.2 E]**

(b) ANSI C treats the variables **name** and **Name** to be same. **[LO 3.2 E]**

(c) Character constants are coded using double quotes. **[LO 3.2 E]**

(d) The keyword **void** is a data type in C. **[LO 3.3 E]**

(e) Character constants are enclosed in single quotes while string constants are enclosed in double quotes. **[LO 3.2 M]**

(f) Initialization is the process of assigning a value to a variable at the time of declaration. **[LO 3.4 E]**

(g) The **scanf** function can be used to read only one value at a time. **[LO 3.4 E]**

(h) C allows its keywords to be also used as identifiers. **[LO 3.1 M]**

(i) Auto variables are by default initialized to 0 as soon as they are declared. **[LO 3.4 H]**

(j) Floating point constants, by default, denote **float** type values. **[LO 3.2 M]**

(k) Like variables, constants have a type. **[LO 3.2 M]**

(l) All **static** variables are automatically initialized to zero. **[LO 3.4 M]**

3.2 Fill in the blanks with appropriate words.

(a) A variable can be made constant by declaring it with the qualifier ________ at the time of initialization. **[LO 3.5 E]**

(b) _______ is the largest value that an unsigned short int type variable can store. **[LO 3.3 M]**

(c) A global variable is also known as ________ variable. **[LO 3.4 M]**

(d) The keyword _____ can be used to create a data type identifier. **[LO 3.4 H]**

(e) If the number 987612347 is to be sued as an unsigned long integer then it must be appended by _______. **[LO 3.3 E]**

(f) __________ and ___________ are used to define user-defined data types in C. **[LO 3.3 M]**

(g) The largest positive integer value can be stored in __________ type variable. **[LO 3.3 H]**

3.3 What are trigraph characters? How are they useful? **[LO 3.1 E]**

3.4 Describe the four basic data types. How could we extend the range of values they represent? **[LO 3.3 E]**

3.5 What is an unsigned integer constant? What is the significance of declaring a constant unsigned? **[LO 3.4 M]**

3.6 Describe the characteristics and purpose of escape sequence characters. **[LO 3.2 M]**

3.7 What is a variable and what is meant by the "value" of a variable? **[LO 3.2 E]**

3.8 How do variables and symbolic names differ? **[LO 3.5 E]**

3.9 State the differences between the declaration of a variable and the definition of a symbolic name. **[LO 3.5 M]**

3.10 What are the qualifiers that an **int** can have at a time? **[LO 3.3 E]**

3.11 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? **[LO 3.4 M]**

3.12 What are enumeration variables? How are they declared? What is the advantage of using them in a program? **[LO 3.4 E]**

3.13 Describe the purpose of the qualifiers **const** and **volatile** . **[LO 3.5 M]**

3.14 When dealing with very small or very large numbers, what steps would you take to improve the accuracy of the calculations? **[LO 3.4 H]**

3.15 Which of the following are invalid constants and why? **[LO 3.2 M]**

0.0001	5 × 1.5	99999
+100	75.45 E-2	"15.75"
–45.6	–1.79 e + 4	0.00001234

3.16 Which of the following are invalid variable names and why? **[LO 3.2 M]**

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

3.17 What would be the value of x after execution of the following statements? **[LO 3.4 H]**

```
int x, y = 10;
char z = 'a';
x = y + z;
```

3.18 Explain the following with examples:

(a) Enumerated types **[LO 3.4 E]**

(b) Type def **[LO 3.4 E]**

3.19 Distinguish between the following:

(a) Global and local variables **[LO 3.4 E]**

(b) Initialization and assignment of variables **[LO 3.4 E]**

(c) Automated and static variables **[LO 3.4 M]**

DEBUGGING EXERCISES

3.1 Find errors, if any, in the following declaration statements. **[LO 3.4 M]**

```
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;
```

3.2 Identify syntax errors in the following program. After corrections, what output would you expect when you execute it? **[LO 3.4 H]**

```
#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)
     }
```

PROGRAMMING EXERCISES

3.1 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 +....+ 1/n

The value of n should be given interactively through the terminal. **[LO 3.4 H]**

3.2 The price of one kg of rice is Rs. 16.75 and one kg of sugar is Rs. 15. Write a program to get these values from the user and display the prices as follows: **[LO 3.4 M]**

```
*** LIST OF ITEMS ***
Item     Price
Rice     Rs 16.75
Sugar    Rs 15.00
```

3.3 Write program to count and print the number of negative and positive numbers in a given set of numbers. Test your program with a suitable set of numbers. Use **scanf** to read the numbers. Reading should be terminated when the value 0 is encountered. **[LO 3.4 M]**

3.4 Write a program to do the following: **[LO 3.2 M]**

(a) Declare x and y as integer variables and z as a short integer variable.

(b) Assign two 6 digit numbers to x and y

(c) Assign the sum of x and y to z

(d) Output the values of x, y and z

Comment on the output.

3.5 Write a program to illustrate the use of **typedef** declaration in a program. **[LO 3.4 M]**

3.6 Write a program to illustrate the use of symbolic constants in a real-life application. **[LO 3.5 H]**

3.7 Write a C program to input an integer and print its table. **[LO 3.4 E]**

3.8 Write a C program to print the square of a number. **[LO 3.4 E]**

3.9 Write a C program to input an integer between 0 and 128 and print its ASCII character. **[LO 3.2 M]**

3.10 Write a C program to input the value of days and convert it into years, weeks and days. **[LO 3.4 M]**

3.11 Write a C program to input the distance travelled by a car and the fuel consumed. Next, compute the mileage of the car. **[LO 3.4 M]**

3.12 Write a C program to input the amount value and break it into the smallest possible Indian currency notes. **[LO 3.4 M]**

INTERVIEW QUESTIONS

3.1 Is it possible to declare an identifier that starts with an underscore? **[LO 3.1 M]**

3.2 Is it possible to declare an identifier that ends with an underscore? **[LO 3.1 M]**

3.3 What is the return type of printf function? **[LO 3.4 H]**

3.4 What is the difference between declaring and defining a variable? **[LO 3.2 H]**

3.5 What is long long int? **[LO 3.3 H]**

3.6 What is the return type of scanf function? **[LO 3.4 H]**

E for Easy, **M** for Medium and **H** for High

Chapter 4

Operators and Expressions

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

LO 4.1 Know the various built-in operators of C language

LO 4.2 Identify bitwise and special operators

LO 4.3 Determine how arithmetic expressions are evaluated

LO 4.4 Explain type conversions in expressions

LO 4.5 Discuss how operator precedence and associativity rules are applied

INTRODUCTION

C supports a rich set of built-in 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 or 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

An expression is a sequence of operands and operators that reduces to a single value. For example,

10 + 15

is an expression whose value is 25. The value can be any type other than ***void*** .

Arithmetic Operators

LO 4.1
Know the various built-in operators of C language

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 operation produces the remainder of an integer division. Examples of use 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:

> **TIP**
> Always use integer operands with the % operator.

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 trunction 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). That is

–14 % 3 = –2

–14 % –3 = –2

14 % –3 = 2

WORKED-OUT PROBLEM 4.1

E

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

```
Program
                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

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.

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

whereas

15/10 = 1

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

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.

Relational Operator Complements

Among the six relational operators, each one is a complement of another operator.

> is complement of < =

< is complement of > =

= = is complement of ! =

We can simplify an expression involving the *not* and the *less than* operators using the complements as shown below:

Actual one	*Simplified one*
!(x < y)	x >= y
!(x > y)	x <= y
!(x != y)	x == y
!(x <= y)	x > y
!(x >= y)	x < y
!(x == y)	x ! = y

TIP

It is advisable not to use floating point operands with relational operators.

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.

Note *Relative precedence of the relational and logical operators is as follows:*

```
Highest     !
            > >=  < <=
            == !=
            &&
Lowest      ||
```

It is important to remember this when we use these operators in compound expressions.

It is important to remember this when we use these operators in compound expressions.

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.

WORKED-OUT PROBLEM 4.2 E

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
            #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 *=

INCREMENT AND DECREMENT OPERATORS

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

++ and – –

The operator ++ adds 1 to the operand, while – – subtracts 1. Both are unary operators and takes 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 and 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.

Let us take up another example. Consider the following segment of code:

```
a = 4;
b = a++ + a++;
```

After the execution of the above segment, ‘a’ would hold the value 6 and ‘b’ would hold the value 8. This happens because the increment operation takes place only after the evaluation of the addition operation. The values of the two operands for the addition operation is 4 and 4, respectively, thus the sum is 8 which is stored in ‘b’. Once the addition has been completed, two consecutive post increments would take place, and as a result, the final value of ‘a’ would be 6.

Let us now consider the following segment of code:

```
a = 4;
b = ++a + ++a;
```

After the execution of the above segment, ‘a’ would hold the value 6 and ‘b’ would hold the value ‘12’. This is because the segment uses the pre-increment operator. Before the addition operation can take place, ‘a’ is incremented twice, as a result of which it becomes 12. Hence, when the addition finally takes place, the sum yields 12 which is assigned to ‘b’.

Rules for ++ and – – Operators

- Increment and decrement operators are unary operators and they require variable as their operands.
- When postfix + + (or – –) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.

> **TIP**
> It is better not to club the increment and decrement operators with too many other operators as far as possible.

- When prefix + +(or – –) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associatively of + + and – – operators are the same as those of unary + and unary –.

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;
```

Test Your Skills

Write a C program to input a character and determine whether it is a vowel or consonant. **[M]**

Bitwise Operators

LO 4.2
Identify bitwise and special 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 4.5 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

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 15.

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.

WORKED-OUT PROBLEM 4.3

M

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 is 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
            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

ARITHMETIC EXPRESSIONS

> **LO 4.3**
> Determine how arithmetic expressions are evaluated

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 4.6. Remember that C does not have an operator for exponentiation.

Table 4.6 Expressions

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

EVALUATION OF EXPRESSIONS

Expressions are evaluated using an assignment statement of the form:

```
variable = expression;
```

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.

WORKED-OUT PROBLEM 4.4 E

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.

```
Program
            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

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
.

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.

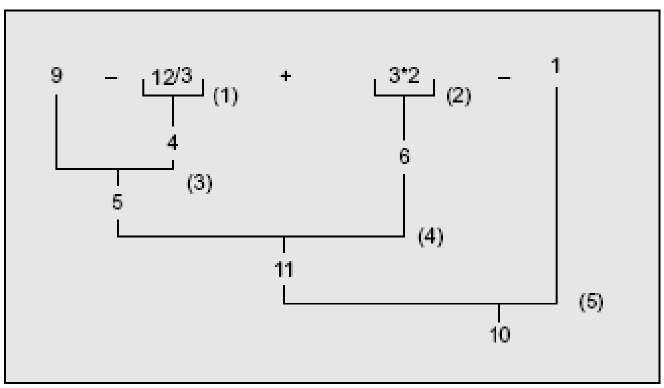


Fig. 4.5 Illustration of hierarchy of operations

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 remains the same as 5 (i.e ., equal to the number of arithmetic operators).

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 parenthesis. 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.

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.

WORKED-OUT PROBLEM 4.5

M

Write a C program for the following expression: a=5<=8 && 6!=5.

```
#include <stdio.h>
#include <conio.h>
   void main()
   {
   int a;
   a = 5<=8 && 6!=5;
   printf("%d", a);
   getch();
   }
Output
   1
```

Fig. 4.6 Program for the expression: a = 5 < = 8 && 6! = 5

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 of underflow.

WORKED-OUT PROBLEM 4.6

M

Output of the program in Fig. 4.7 shows round-off errors that can occur in computation of floating point numbers.

Program

```
/*———— 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. 4.7 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.

TEST YOUR SKILLS

Write a C program to find the roots of a quadratic equation. **[M]**

TYPE CONVERSIONS IN EXPRESSIONS

Implicit Type Conversion

LO 4.4
Explain type conversions in expressions

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without loosing any significance. This automatic conversion is known as *implicit type conversion.*

During evaluation it adheres to very strict rules of type conversion. 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. 4.8 .

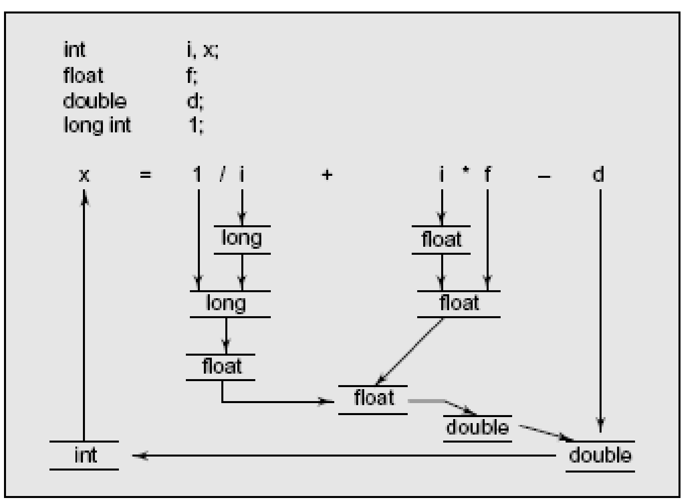


Fig. 4.8 Process of implicit type conversion

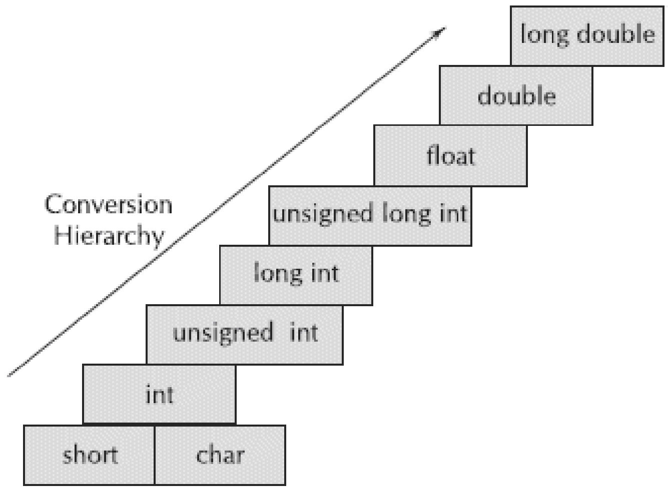
Given below is the sequence of rules that are applied while evaluating expressions.

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 be 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.**

Conversion Hierarchy

Note that, C uses the rule that, in all expressions except assignments, any implicit type conversions are made from a lower size type to a higher size type as shown below:



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.

Explicit Conversion

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 *explicit conversion* or *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.

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.

TIP

Instead of relying upon implicit type conversions, explicit type casting may be used to improve the understandability of the program.

WORKED-OUT PROBLEM 4.7

H

Figure 4.9 shows a program using a cast to evaluate the equation

$$\text{sum} = \sum_{i=1}^{n} (1/i)$$

```
Program
            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) ;
            }
        }
```

Fig. 4.9 Use of a cast

OPERATOR PRECEDENCE AND ASSOCIATIVITY

LO 4.5
Discuss how operator precedence and associativity rules are applied

As mentioned earlier 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 these 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 provides 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 been discussed.

Table 4.8 Summary of C Operators

Operator	*Description*	*Associativity*	*Rank*
( ) [ ]	Function call Aray element reference	Left to right	1
+ – ++ – – ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical negation Ones complement Pointer reference (indirection) Address Size of an object Type cast (conversion)	Right to left	2
* / %	Multiplication Division Modulus	Left to right	3
+ –	Addition Subtraction	Left to right	4
<< >>	Left shift Right shift	Left to right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to right	6
== |=	Equality Inequality	Left to right	7
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
= * = /= %= += –= &= ^= |= <<= >>=	Assignment operators	Right to left	14
,	Comma operator	Left to right	15

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.

Rules of Precedence and Associativity

- Precedence rules decides the order in which different operators are applied
- Associativity rule decides the order in which multiple occurrences of the same level operator are applied

Mathematical Functions

Mathematical functions such as cos, sqrt, log, etc. are frequently used in analysis of real-life problems. Most of the C compilers support these basic math functions. However, there are systems that have a more comprehensive math library and one should consult the reference manual to find out which functions are available. Table 4.9 lists some standard math functions.

Table 4.9 Math functions

Function	*Meaning*
Trigonometric	
acos(x)	Arc cosine of x
asin(x)	Arc sine of x
atan(x)	Arc tangent of x
atan 2(x,y)	Arc tangent of x/y
cos(x)	Cosine of x
sin(x)	Sine of x
tan(x)	Tangent of x
Hyperbolic	
cosh(x)	Hyperbolic cosine of x
sinh(x)	Hyperbolic sine of x
tanh(x)	Hyperbolic tangent of x
Other functions	
ceil(x)	x rounded up to the nearest integer
exp(x)	e to the x power (e^x)
fabs(x)	Absolute value of x.
floor(x)	x rounded down to the nearest integer
fmod(x,y)	Remainder of x/y
log(x)	Natural log of x, x > 0
log10(x)	Base 10 log of x, x > 0
pow(x,y)	x to the power y (x^y)
sqrt(x)	Square root of x, x > = 0

Note:

1. **x** and **y** should be declared as **double.**
2. In trigonometric and hyperbolic functions, **x** and **y** are in radians.
3. All the functions return a **double** .
4. C99 has added **float** and **long double** versions of these functions.
5. C99 has added many more mathematical functions.
6. See the Appendix "C99 Features" for details.

As pointed out earlier in Chapter 2, to use any of these functions in a program, we should include the line:

 # include <math.h>

in the beginning of the program.

KEY CONCEPTS

- **OPERATOR:** Is a symbol that is used to perform mathematical or logical operation on data and variables. **[LO 4.1]**
- **EXPRESSION:** Is a combination of operands and operators that reduce to a single value. **[LO 4.1]**
- **INTEGER EXPRESSION:** Is an arithmetic expression involving integer operands. **[LO 4.1]**
- **REAL ARITHMETIC:** Is an arithmetic expression involving real operands. **[LO 4.1]**
- **RELATIONAL OPERATORS:** Are used to compare two operands and result in either one (true) or zero (false). **[LO 4.1]**
- **LOGICAL OPERATORS:** Are used to combine two or more relational expressions. **[LO 4.1]**
- **ASSIGNMENT OPERATORS:** Are used to assign an expression value to a variable. **[LO 4.1]**
- **BITWISE OPERATORS:** Are special operators that are used to manipulate data at bit level. **[LO 4.2]**
- **ARITHMETIC OPERATION:** Refers to the evaluation of arithmetic expression as per operator precedence rules. **[LO 4.3]**

ALWAYS REMEMBER

- Use *decrement* and *increment* operators carefully. Understand the difference between **postfix** and **prefix** operations before using them. **[LO 4.1]**
- Do not use *increment* or *decrement* operators with any expression other than a *variable identifier* . **[LO 4.1]**
- It is illegal to apply modules operator % with anything other than integers. **[LO 4.1]**
- The result of an expression is converted to the type of the variable on the left of the assignment before assigning the value to it. Be careful about the loss of information during the conversion. **[LO 4.1]**
- It is an error if any space appears between the two symbols of the operators ==, !=, <= and >=. **[LO 4.1]**

- It is an error if the two symbols of the operators !=, <= and >= are reversed. **[LO 4.1]**
- Use spaces on either side of binary operator to improve the readability of the code. **[LO 4.1]**
- Do not use increment and decrement operators to floating point variables. **[LO 4.1]**
- Do not confuse the equality operator == with the assignment operator =. **[LO 4.1]**
- Use sizeof operator to determine the length of arrays and structures where their sizes are not already known . **[LO 4.2]**
- Be aware of side effects produced by some expressions. **[LO 4.3]**
- Avoid any attempt to divide by zero. It is normally undefined. It will either result in a fatal error or in incorrect results. **[LO 4.3]**
- Do not forget a semicolon at the end of an statement. **[LO 4.3]**
- Do not use a variable in an expression before it has been assigned a value. **[LO 4.3]**
- Integer division always truncates the decimal part of the result. Use it carefully. Use casting where necessary. **[LO 4.4]**
- Add parentheses wherever you feel they would help to make the evaluation order clear. **[LO 4.5]**
- Understand clearly the precedence of operators in an expression. Use parentheses, if necessary. **[LO 4.5]**
- Associativity is applied when more than one operator of the same precedence are used in an expression. Understand which operators associate from right to left and which associate from left to right. **[LO 4.5]**

BRIEF CASES

1. Salesman's Salary [LO 4.3 M]

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. 4.10 .

```
Program
            #define BASE_SALAR    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. 4.10 Program of salesman's salary

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 rate

2. Solution of the Quadratic Equation[LO 4.5 H]

An equation of the form

$$ax^{2} + bx + c = 0$$

is known as the *quadratic equation* . The values of x that satisfy the equation are known as the *roots* of the equation. A quadratic equation has two roots which are given by the following two formulae:

$$\text{root1} = \frac{-b + \text{sqrt}(b^2 - 4ac)}{2a}$$

$$\text{root 2} = \frac{-b - \text{sqrt}(b^2 - 4ac)}{2a}$$

A program to evaluate these roots is given in Fig. 4.11 . The program requests the user to input the values of **a** , **b** and **c** and outputs **root 1** and **root 2** .

Program

```
#include <math.h>
main()
{
     float a, b, c, discriminant,
               root1, root2;
     printf("Input values of a, b, and c\n");
     scanf("%f %f %f", &a, &b, &c);
     discriminant = b*b - 4*a*c ;
     if(discriminant < 0)
          printf("\n\nROOTS ARE IMAGINARY\n");
     else
     {
          root1 = (-b + sqrt(discriminant))/(2.0*a);
          root2 = (-b - sqrt(discriminant))/(2.0*a);
          printf("\n\nRoot1 = %5.2f\n\nRoot2 = %5.2f\n",
                      root1,root2 );
     }
}
```

Output

```
Input values of a, b, and c
2 4 -16
Root1 = 2.00
Root2 = -4.00
Input values of a, b, and c
1 2 3
ROOTS ARE IMAGINARY
```

Fig. 4.11 Solution of a quadratic equation

The term (b^2 –4ac) is called the *discriminant* . If the discriminant is less than zero, its square roots cannot be evaluated. In such cases, the roots are said to be imaginary numbers and the program outputs an appropriate message.

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

4.1 Which of the following is not an arithmetic operator? **[LO 4.1 M]**

(a) %

(b) /

(c) *

(d) All of the above are arithmetic operators

4.2 Which of the following is an assignment operator? **[LO 4.1 M]**

(a) =

(b) %=

(c) /=

(d) All of the above

4.3 Which of the following statements is correct about increment and decrement operators? **[LO 4.1 H]**

(a) A prefix increment operator first adds 1 to the operand and then the result is assigned to the variable on the left.

(b) A postfix decrement operator first assigns the value to the variable on the left and then decrements the operand.

(c) The precedence and associatively of ++ and — operators are the same as those of unary + and unary –.

(d) All of the above are correct

4.4 Which of the following operators has the lowest level of precedence **[LO 4.5 M]**

(a) %

(b) &

(c) &&

(d) ?=

4.5 Which of the following operators is used to determine the size of the operand? **[LO 4.2 E]**

(a) auto

(b) size

(c) sizeof

(d) &

REVIEW QUESTIONS

4.1 State whether the following statements are *true* or *false* .

(a) The expression *!(x<=y)* is same as the expression *x>y* . **[LO 4.1 E]**

(b) A unary expression consists of only one operand with no operators. **[LO 4.1 E]**

(c) All arithmetic operators have the same level of precedence. **[LO 4.3 E]**

(d) An expression statement is terminated with a period. **[LO 4.3 E]**

(e) The operators <=, >= and != all enjoy the same level of priority. **[LO 4.5 E]**

(f) The modulus operator % can be used only with integers. **[LO 4.1 M]**

(g) In C, if a data item is zero, it is considered false. **[LO 4.1 M]**

(h) During the evaluation of mixed expressions, an implicit cast is generated automatically. **[LO 4.4 M]**

(i) An explicit cast can be used to change the expression. **[LO 4.4 M]**

(j) Associativity is used to decide which of several different expressions is evaluated first. **[LO 4.5 M]**

(k) Parentheses can be used to change the order of evaluation expressions. **[LO 4.5 M]**

(l) During modulo division, the sign of the result is positive, if both the operands are of the same sign. **[LO 4.1 H]**

4.2 Fill in the blanks with appropriate words.

(a) The expression containing all the integer operands is called________ expression. **[LO 4.1 E]**

(b) C supports as many as _______relational operators. **[LO 4.1 E]**

(c) The ___________operator returns the number of bytes the operand occupies. **[LO 4.2 E]**

(d) _________is used to determine the order in which different operators in an expression are evaluated. **[LO 4.5 E]**

(e) An expression that combines two or more relational expressions is termed as __________expression. **[LO 4.1 M]**

(f) The use of ________ on a variable can change its type in the memory. **[LO 4.4 M]**

(g) The order of evaluation can be changed by using ______ in an expression. **[LO 4.5 M]**

(h) The operator _________cannot be used with real operands. **[LO 4.1 H]**

(i) __________ and ___________ operators have the highest and lowest levels of precedence, respectively. **[LO 4.5 M]**

(j) <= is a __________ operator while && is a ___________ operator. **[LO 4.1 E]**

(k) A division operation involving integer operands truncates the resultant value. This situation can be avoided by using ___________. **[LO 4.4 M]**

(l) The result of a relational expression is either _____ or ______. **[LO 4.1 E]**

4.3 Given the statement **[LO 4.3 M]**

int a = 10, b = 20, c;

determine whether each of the following statements are true or false.

(a) The statement a = + 10, is valid.

(b) The expression a + 4/6 * 6/2 evaluates to 11.

(c) The expression b + 3/2 * 2/3 evaluates to 20.

(d) The statement a + = b; gives the values 30 to a and 20 to b.

(e) The statement ++a++; gives the value 12 to a

(f) The statement a = 1/b; assigns the value 0.5 to a

4.4 Declared **a** as *int* and **b** as *float* **,** state whether the following statements are true or false. **[LO 4.3 H]**

(a) The statement a = 1/3 + 1/3 + 1/3; assigns the value 1 to a.

(b) The statement b = 1.0/3.0 + 1.0/3.0 + 1.0/3.0; assigns a value 1.0 to b.

(c) The statement b = 1.0/3.0 * 3.0 gives a value 1.0 to b.

(d) The statement b = 1.0/3.0 + 2.0/3.0 assigns a value 1.0 to b.

(e) The statement a = 15/10.0 + 3/2; assigns a value 3 to a.

4.5 Which of the following expressions are true? **[LO 4.3 M]**

(a) !(5 + 5 >=10)

(b) 5 + 5 = = 10 || 1 + 3 = = 5

(c) 5 > 10 || 10 < 20 && 3 < 5

(d) 10 ! = 15 && !(10<20) || 15 > 30

4.6 Which of the following arithmetic expressions are valid ? If valid, give the value of the expression; otherwise give reason. **[LO 4.5 M]**

(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

4.7 Write C assignment statements to evaluate the following equations: **[LO 4.5 H]**

[redacted]

4.8 Identify unnecessary parentheses in the following arithmetic expressions. **[LO 4.5 H]**

(a) ((x – (y/5)+z)%8) + 25

(b) ((x – y) * p)+q

(c) (m*n) + (– x/y)

(d) x/(3*y)

4.9 Determine the value of each of the following logical expressions if a = 5, b = 10 and c = – 6 **[LO 4.5 M]**

(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

4.10 What is the output of the following program? **[LO 4.1 M]**

```
main ( )
{
char x;
int y;
x = 100;
y = 125;
printf ("%c\n", x) ;
printf ("%c\n", y) ;
printf ("%d\n", x) ;
}
```

4.11 Find the output of the following program? **[LO 4.1 M]**

```
main ( )
{
```

```
int x = 100;
printf("%d/n", 10 + x++);
printf("%d/n", 10 + ++x);
}
```

4.12 What is printed by the following program? **[LO 4.5 M]**

```
main
{
int x = 5, y = 10, z = 10 ;
x = y == z;
printf("%d",x ) ;
}
```

4.13 What is the output of the following program? **[LO 4.1 E]**

```
main ( )
{
int x = 100, y = 200;
printf ("%d", (x > y)? x : y);
}
```

4.14 What is the output of the following program? **[LO 4.4 M]**

```
main ( )
{
unsigned x = 1 ;
signed char y = -1 ;
if(x > y)
printf(" x > y");
else
printf("x<= y") ;
}
```

Did you expect this output? Explain.

4.15 What is the output of the following program? Explain the output. **[LO 4.1 M]**

```
main ( )
{
int x = 10 ;
if(x = 20) printf("TRUE") ;
else printf("FALSE") ;
```

```
}
```

4.16 What is printed when the following is executed? **[LO 4.5 M]**

```
for (m = 0; m <3; ++m)
printf("%d/n", (m%2) ? m: m+2);
```

4.17 What is the output of the following segment when executed? **[LO 4.1 E]**

```
int m = - 14, n = 3;
printf("%d\n", m/n * 10) ;
n = -n;
printf("%dn", m/n * 10);
```

DEBUGGING EXERCISES

4.1 What is the error, if any, in the following segment? **[LO 4.4 E]**

```
int x = 10 ;
float y = 4.25 ;
x = y%x ;
```

4.2 What is the error in each of the following statements? **[LO 4.1 M]**

(a) if (m == 1 & n ! = 0)
printf("OK");
(b) if (x = < 5)
printf ("Jump");

4.3 Find errors, if any, in the following assignment statements and rectify them. **[LO 4.5 M]**

(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

PROGRAMMING EXERCISES

4.1 Write a program that reads a floating-point number and then displays the right-most digit of the integral part of the number.

4.2 Modify the above program to display the two right-[LO 4.1 M] most digits of the integral part of the number. [LO 4.1 M]

4.3 Given an integer number, write a program that displays the number as follows: [LO 4.3 H]

First line : all digits

Second line : all except first digit

Third line : all except first two digits

…….

Last line : The last digit

For example, the number 5678 will be displayed as:

5 6 7 8

6 7 8

7 8

8

4.4 The straight-line method of computing the yearly depreciation of the value of an item is given by

[redacted]

Write a program to determine the salvage value of an item when the purchase price, years of service, and the annual depreciation are given. [LO 4.3 E]

4.5 Write a program that will read a real number from the keyboard and print the following output in one line: [LO 4.4 M]

[redacted]

4.6 The total distance travelled by a vehicle in t seconds is given by

distance = $ut + (a\,t^2)/2$

Where u is the initial velocity (metres per second), a is the acceleration (metres 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. [LO 4.5 M]

4.7 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

$$\text{TBO} = \sqrt{\frac{2 \times \text{setup costs}}{\text{demand rate} \times \text{holding cost per unit time}}}$$

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). **[LO 4.5 H]**

4.8 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. **[LO 4.5 H]**

4.9 Write a program to read a four digit integer and print the sum of its digits. **[LO 4.1 E]**

Hint: Use / and % operators.

4.10 Given three values, write a program to read three values from keyboard and print out the largest of them without using **if** statement. **[LO 4.2 H]**

4.11 Write a program to read two integer values m and n and to decide and print whether m is a multiple of n. **[LO 4.1 M]**

4.12 Write a program to read three values using **scanf** statement and print the following results: **[LO 4.3 M]**

(a) Sum of the values

(b) Average of the three values

(c) Largest of the three

(d) Smallest of the three

4.13 The cost of one type of mobile service is Rs. 250 plus Rs. 1.25 for each call made over and above 100 calls. Write a program to read customer codes and calls made and print the bill for each customer. **[LO 4.5 H]**

4.14 Write a program to print a table of **sin** and **cos** functions for the interval from 0 to 180 degrees in increments of 15 a shown here. **[LO 4.1 E]**

x (degrees)	sin (x)	cos (x)
0	......	......
15	......	......
...		
...		
180	......	......

4.15 Write a program to compute the values of square-roots and squares of the numbers 0 to 100 in steps 10 and print the output in a tabular form as shown below. **[LO 4.5 M]**

Number	Square-root	Square
0	0	0
100	10	10000

4.16 Write a program to illustrate the use of cast operator in a real life situation. **[LO 4.4 H]**

4.17 Write a C program to shift the given data by two bits to the left. **[LO 4.1 M]**

4.18 Write a C program to compute the value of the expression x=a-b/3+c*2-1. **[LO 4.3 E]**

4.19 Write a C program to input a date value and determine whether the entered day, month, and year values are valid. **[LO 4.3 H]**

4.20 Write a C program to input the sides of a triangle and determine whether the triangle is isoceles or not. **[LO 4.3 M]**

4.21 Write a C program that reads two numbers and performs their division. If the division is not possible, then an error messgage, 'Division not possible' is displayed. **[LO 4.3 M]**

4.22 Input the value of 4 variables a, b, c and d and compute the resultant value of following expressions: **[LO 4.3 M]**

(a + b) * (c / d)

(a + b) * c / d

a + (b * c) / d

INTERVIEW QUESTIONS

4.1 What will be the value of x and y after following statements are executed?

```
x=5;
y = x++ + ++x + x++;
```

Explain the logic while calculating the answer. **[LO 4.3 H]**

4.2 Evaluate the result for the following logical expression, !(1 && !(0 || 1)) **[LO 4.1 H]**

4.3 What is l value? **[LO 4.3 M]**

4.4 What is the difference between | and || operator? **[LO 4.2 M]**

4.5 Write the code for swapping two variable values without using a third variable. **[LO 4.3 H]**

4.6 What would be the value of b after below code is executed in C?

```
int a=5, b;
b=++a + ++a;
```

[LO 4.3 H]

E for Easy, **M** for Medium and H for High

Chapter 5

Managing Input and Output Operations

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

- **LO 5.1** Describe how a character is read
- **LO 5.2** Express how a character is written
- **LO 5.3** Explain formatted input
- **LO 5.4** Discuss formatted output

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 keyboard. 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.

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.

READING A CHARACTER

LO 5.1
Describe how a character is read

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 **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** . 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.

WORKED-OUT PROBLEM 5.1

The program in Fig. 5.1 shows the use of **getchar** 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 or y, it outputs the message

My name is BUSY BEE

otherwise, outputs

You are good for nothing

Note *There is one line space between the input text and output message.*

Program

```
#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\nMy name is BUSY BEE\n");
          else
          printf("\n\nYou 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 to read a character from keyboard

The **getchar** may be called successively to read the characters contained in a line of text. For example,
the following program segment reads characters from keyboard one after another until the 'Return' key is pressed.

```
---——
---——
char character;
character - ' ';
while(character !- '\n')
{
     character = getchar();
}
---——
---——
```

Warning

The **getchar()** accepts any character keyed in. This includes RETURN and TAB. This means 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()** in a loop interactively. A dummy **getchar()** may be used to 'eat' the unwanted newline character. We can also use the **fflush** function to flush out the unwanted characters.

Note *We shall be using decision statements like **if**, **if...else** and **while** extensively in this chapter. They are discussed in detail in Chapters 6 and 7.*

WORKED-OUT PROBLEM 5.2 M

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 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** .

Program:

```
#include <stdio.h>
#include <ctype.h>
main()
{
   char character;
   printf("Press any key\n");
   character = getchar();
   if (isalpha(character) > 0)/* Test for letter */
      printf("The character is a letter.");
   else
      if (isdigit (character) > 0)/* Test for digit */
         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

C supports many other similar functions, which are given in Table 5.1. The prototypes of these character functions are contained in the file **ctype.h** and therefore the statement

#include <ctype.h>

must be included in the program.

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 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?

WRITING A CHARACTER

Like **getchar** , **putchar** is used for writing characters one at a time to the terminal. It takes the form as shown below:

LO 5.2
Express how a character is written

```
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.

WORKED-OUT PROBLEM 5.3

M

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 lowercase alphabet; otherwise takes the value FALSE. The function **toupper** converts the lowercase argument into an uppercase alphabet while the function **tolower** does the reverse.

```
Program
            #include <stdio.h>
            #include <ctype.h>
            main()
            {
               char alphabet;
               printf("Enter an alphabet");
               putchar('\n'); /* move to next line */
               alphabet = getchar();
               if (islower(alphabet))
               putchar(toupper(alphabet));/* Reverse and display */
            else
               putchar(tolower(alphabet)); /* Reverse and display */
            }
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 cast

FORMATTED INPUT

Formatted input refers to an input data that has been arranged in a particular format. For example, consider the following data:

LO 5.3
Explain formatted input

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. (**scanf** means *scan* formatted.)

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 *arg* 1, *arg2, ..., argn* specify the address of locations where the data is stored. Control string and arguments are separated by commas.

Control string (also known as *format 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:

```
%wd
```

The percentage 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 (unread 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 width 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. Punctuation 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 be terminated 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 and **h** to read short integers.

Note *We have provided white space between the field specifications. These spaces are not necessary with the numeric input, but it is a good practice to include them.*

WORKED-OUT PROBLEM 5.4

E

Various input formatting options for reading integers are experimented in the program shown in Fig. 5.4.

Program

```
main()
{
   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 %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 two integers\n");
   scanf("%d %d", &a,&x);
   printf("%d %d \n\n",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

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 that 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 been assigned to the first variable in the immediately following **scanf** statement.

NOTE: It is legal to use a non-whitespace character between field specifications. However, the **scanf** expects a matching character in the given location. For example,

scanf("%d-%d", &a, &b);

accepts input like

123-456

to assign 123 to a and 456 to b.

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

> **TIP**
> The data type of a variable should be double checked for size to avoid overflow errors.

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.

WORKED-OUT PROBLEM 5.5 **E**

Reading of real numbers (in both decimal point and exponential notation) is illustrated in Fig. 5.5.

```
Program
            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("\n\np = %.12lf\np = %.12e", 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 = 4.142857142857
              q = 1.856789012346e+001
```

Fig. 5.5 Reading of real numbers

Inputting 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.

WORKED-OUT PROBLEM 5.6 H

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** .

Program

```
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("%d %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 123456789012r
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

Some versions of **scanf** support the following conversion specifications for strings:

%[characters]

%[^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.

WORKED-OUT PROBLEM 5.7 H

The program in Fig. 5.7 illustrates the function of %[ ] specification.

```
Program-A
          main()
          {
            char address[80];
            printf("Enter address\n");
            scanf("%[a-z]", address);
            printf("%-80s\n\n", address);
          }
Output
            Enter address
            new delhi 110002
            new delhi
Program-B
          main()
          {
            char address[80];
            printf("Enter address\n");
            scanf("%[^\n]", address);
            printf("%-80s", address);
          }
Output
            Enter address
            New Delhi 110 002
            New Delhi 110 002
```

Fig. 5.7 Illustration of conversion specification%[] for strings

Reading Blank Spaces

We have earlier 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 ***s* canf** to read strings with spaces. Remember that the lowercase and uppercase letters are distinct. See Fig. 5.7 .

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.

Note *A space before the %c specification in the format string is necessary to skip the white space before p.*

Detection of Errors in Input

When a **scanf** function completes reading its list, it returns the value of number of items that are successfully 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

Thus,

printf("%d", scanf("%d %f %s", &a, &b, name));

will print 3 if

20 150.25 motor

is entered.

In case,

20 motor 150.25

is entered, then the printf statement will print 1.

WORKED-OUT PROBLEM 5.8 H

The program presented in Fig. 5.8 illustrates the testing for correctness of reading of data by **scanf** function.

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 *The character '2' is assigned to the character variable c.*

```
Program
        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 - 2
```

Fig. 5.8 Detection of errors in scanf input

Commonly used **scanf** format codes are given in Table 5.2

Table 5.2 Commonly used 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

Note *C99 adds some more format codes. See the Appendix "C99 Features".*

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 a '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 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.

Rules for scanf

- Each variable to be read must have a filed specification.
- For each field specification, there must be a variable address of proper type.
- Any non-whitespace character used in the format string must have a matching character in the user input.
- The scanf reads until:
  - A whitespace character is found in a numberic specification, or
  - The maximum number of characters have been read or
  - An error is detected, or
  - The end of file is reached

TIP

Variables should be assigned meaningful names. This will not only enhance the readability but also reduce the chances of the programmer using a wrong format string for a data type during input/output operations.

Formatted Output

LO 5.4
Discuss 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 following 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 formatted **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 one line 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("%-06d", 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. The minus (–) and zero (0) are known as *flags.*

Long integers may be printed by specifying **ld** in the place of **d** in the format specification. Similarly, we may use **hd** for printing short integers.

WORKED-OUT PROBLEM 5.9

M

The program in Fig. 5.9 illustrates the output of integer numbers under various formats.

```
Program
            main()
            {
               int m - 12345;
               long n - 987654;
               printf("%d\n",m);
               printf("%10d\n",m);
               printf("%010d\n",m);
               printf("%-10d\n",m);
               printf("%10ld\n",n);
               printf("%10ld\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.

$$\mathbf{w \geq p+7}$$

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* are also possible by using flags 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

printf(“%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);
```

WORKED-OUT PROBLEM 5.10

All the options of printing a real number are illustrated in Fig. 5.10.

```
Program
            main()
            {
               float y = 98.7654;
               printf("%7.4f\n", y);
               printf("%f\n", y);
               printf("%7.2f\n", y);
               printf("%-7.2f\n", y);
               printf("%07.2f\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

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 variety of specifications in printing a string "NEW DELHI 110001", containing 16 characters (including banks).

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				

WORKED-OUT PROBLEM 5.11 M

Printing of characters and strings is illustrated in Fig. 5.11.

```
Program
            main()
            {
              char x = 'A';
              char name[20] = "ANIL KUMAR GUPTA";
              printf("OUTPUT OF CHARACTERS\n\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

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, the output results will be incorrect.

Table 5.3 Commonly used 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
%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 Ox

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 Commonly used ***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.

Note *C99 adds some more format codes. See the Appendix " C99 Features".*

TEST YOUR SKILLS

1. Write a C program to enter the student's grades and print the average of grades. [E]
2. Write a C program to input any two angle values of a triangle and deduce its third angle value. [E]

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 are 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");
```

KEY CONCEPTS

- **getchar :** Reads one character from standard input. **[LO 5.1]**

- **putchar :** Reads one character from standard input. **[LO 5.2]**
- **FORMATTED INPUT:** Reads one character from standard input. **[LO 5.3]**
- **CONTROL STRING:** Is a combination of format specifications, escape sequences and characters that are to be printed on the screen. **[LO 5.4]**
- **FORMATTED OUTPUT:** Refers to the output data that has been arranged in a specific format to enhance readability. **[LO 5.4]**

ALWAYS REMEMBER

- While using **getchar** , care should be exercised to clear any unwanted characters in the input stream. **[LO 5.1]**
- Do not forget to include < **stdio.h** > headerfiles when using functions from standard input/output library. **[LO 5.1]**
- Do not forget to include < **ctype.h** > header file when using functions from character handling library. **[LO 5.2]**
- Provide proper field specifications for every variable to be read or printed. **[LO 5.3]**
- Enclose format control strings in double quotes. **[LO 5.3]**
- Do not forget to use address operator & for basic type variables in the input list of **scanf** . **[LO 5.3]**
- Do not specify any precision in input field specifications. **[LO 5.3]**
- Do not provide any white-space at the end of format string of a **scanf** statement. **[LO 5.3]**
- Do not use commas in the format string of a **scanf** statement. **[LO 5.3]**
- Use double quotes for character string constants. **[LO 5.4]**
- Use single quotes for single character constants. **[LO 5.4]**
- Provide sufficient field width to handle a value to be printed. **[LO 5.4]**
- Be aware of the situations where output may be imprecise due to formatting. **[LO 5.4]**

- Do not forget to close the format string in the **scanf** or **printf** statement with double quotes. **[LO 5.4]**
- Using an incorrect conversion code for data type being read or written will result in runtime error. **[LO 5.4]**
- Do not forget the comma after the format string in **scanf** and **printf** statements. **[LO 5.4]**
- Not separating read and write arguments is an error. **[LO 5.4]**
- Using an address operator & with a variable in the **printf** statement will result in runtime error. **[LO 5.4]**

BRIEF CASES

1. Inventory Report **[LO 5.3, 5.4 M]**

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
I019	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.

Program

```
#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:I019 321 215.50
Enter code, quantity, and rate:M315 89 725.00
           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
```

Fig. 5.12 Program for inventory report

2. Reliability Graph[LO 5.4 H]

Problem : The reliability of an electronic component is given by

$$\text{reliability (r)} = e^{-\lambda t}$$

where *l* 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 operating times, from 0 to 3000 hours. The failure rate *l* (lambda) is 0.001.

Problem

```
#include <math.h>
#define LAMBDA 0.001
main()
{
  double t;
  float r;
  int i, R;
  for (i-1; i<-27; ++i)
  {
    printf("--");
  }
  printf("\n");
  for (t-0; t<-3000; t+-150)
  {
    r - exp(-LAMBDA*t);
    R - (int)(50*r+0.5);
    printf(" |");
    for (i-1; i<-R; ++i)
  {
    printf("*");
  }
    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.

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

5.1 String constants are enclosed within ________ symbols. **[LO 5.4 M]**
(a) Single quote
(b) Double quote
(c) []
(d) None of the above

5.2 Which of the following is used to separate specifiers in a format string within a scanf statement? **[LO 5.3 E]**
(a) Single white space
(b) Comma
(c) Double quotes
(d) Semi-colon

5.3 Which of the following functions can be used to print a single character as output? **[LO 5.4 M]**
(a) scanf()
(b) printf()

(c) purchar()

(d) Both 2 and 3

5.4 Which of the following statements are true about scanf() function? **[LO 5.3 H]**

(a) The reading will be terminated when scanf encounters a 'mismatch' of data or a character that is not valid for the value is being read.

(b) The scanf() function returns the value of number of items that are successfully read.

(c) Any unread data items in a line are considered as part of the data input line to the next scanf () function call.

(d) All of the above are true

5.5 Which of the following statements is false about print() function? **[LO 5.4 H]**

(a) The format specifications in a printf() statement should match the variables in number, order and type.

(b) If the width of a value being printed by the print() function is greater than the width specified in the format specification, then the value gets printed in full, overriding the minimum specification.

(c) Using an address operator & with a variable in the printf statement will result in runtime error.

(d) All of the above are true

5.6 Which of the following format specifiers can be used to read floating type values? **[LO 5.3 M]**

(a) %e

(b) %f

(c) %g

(d) All of the above

REVIEW QUESTIONS

5.1 State whether the following statements are *true* or *false* .

(a) The purpose of the header file <studio.h> is to store the programs created by the users.

(b);The C standard function that receives a single character [LO 5.1 E] from the keyboard is **getchar** .

(c)The input list in a **scanf** statement can contain one or more [LO 5.1 E] variables. **[LO 5.3 E]**

(d)The **scanf** function cannot be used to read a single character from the keyboard. **[LO 5.3 E]**

(e)The **getchar** cannot be used to read a line of text from the keyboard. **[LO 5.1 M]**

(f) Variables form a legal element of the format control string of a **printf** statement. **[LO 5.4 M]**

(g)The format specification %+ –8d prints an integer left-justified in a field width of 8 with a plus sign, if the number is positive. **[LO 5.4 M]**

(h)If the field width of a format specifier is larger than the actual width of the value, the value is printed right-justified in the field. **[LO 5.4 M]**

(i)The format specification %5s will print only the first 5 characters of a given string to be printed. **[LO 5.4 M]**

(j)When an input stream contains more data items than the number of specifications in a **scanf** statement, the unused items will be used by the next **scanf** call in the program.

(k) Format specifiers for output convert internal **[LO 5.3 H]** representations for data to readable characters.

(l)The print list in a **printf** statement can contain function **[LO 5.4 H]** calls. **[LO 5.4 H]**

5.2 Fill in the blanks in the following statements.

(a)The _________ specification is used to read or write a short integer. **[LO 5.3 E]**

(b)For reading a double type value, we must use the specification _________ . **[LO 5.3 E]**

(c) For using character functions, we must include the header file _________ in the program. **[LO 5.4 E]**

(d) To print the data left-justified, we must use _________ in the field specification. **[LO 5.4 E]**

(e) The conversion specifier _________ is used to print integers in hexadecimal form. **[LO 5.4 M]**

(f) The specification _________ is used to read a data from input list and discard it without assigning it to many variable. **[LO 5.3 M]**

(g) The specification %[ ] is used for reading strings that contain _________ . **[LO 5.3 M]**

(h) By default, the real numbers are printed with a precision of _________ decimal places. **[LO 5.4 M]**

(i) The specifier _________ prints floating-point values in the scientific notation. **[LO 5.4 M]**

(j) The specification _________ may be used in **scanf** to terminate reading at the encounter of a particular character. **[LO 5.3 H]**

(k) __________ and ___________ format specifiers are used to print a single character and string respectively. **[LO 5.4 E]**

(l) Long integers are printed by specifying ___________ in the place of d in the format specification. Likewise, short integers are printed using ___________. **[LO 5.4 M]**

5.3 Distinguish between the following pairs:

(a) ***getchar*** and ***scanf*** functions. **[LO 5.1 E]**

(b) %s and %c specifications for reading. **[LO 5.3 E]**

(c) %g and %f specification for printing. **[LO 5.4 E]**

(d) %s and %[ ] specifications for reading. **[LO 5.3 M]**

(e) %f and %e specifications for printing. **[LO 5.4 M]**

5.4 Write **scanf** statements to read the following data lists: **[LO 5.3 M]**

(a) 78 B 45

(b) 123 1.23 45A

(c) 15-10-2002

(d) 10 TRUE 20

5.5 State the outputs produced by the following **printf** statements. **[LO 5.4 H]**

(a) printf (“%d%c%f”, 10, ‘x’, 1.23);

(b) printf (“%2d %c %4.2f”, 1234,, ‘x’, 1.23);

(c) printf (“%d\t%4.2f”, 1234, 456);

(d) printf (“\”%08.2f\””, 123.4);

(e) printf (“%d%d %d”, 10, 20);

For questions 5.6 to 5.10 assume that the following declarations have been made in the program:

```
int year, count;
float amount, price;
char code, city[10];
double root;
```

5.6 What will be the values stored in the variables **year** and **code** when the data **[LO 5.3 H]**

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);

5.7 The variables **count, price** , and **city** have the following values: **[LO 5.4 H]**

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);

5.8 State what (if anything) is wrong with each of the following output statements: **[LO 5.4 M]**

(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);

5.9 In response to the input statement **[LO 5.3 M]**

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** ?

5.10 How can we use the **getchar** ( ) function to read multicharacter strings? **[LO 5.1 H]**

5.11 How can we use the **putchar** ( ) function to output multicharacter strings? **[LO 5.2 H]**

5.12 What is the purpose of **scan f** ( ) function? **[LO 5.3 E]**

5.13 Describe the purpose of commonly used conversion characters in a **scanf** ( ) function. **[LO 5.3 E]**

5.14 What happens when an input data item contains **[LO 5.3 H]**

(a) more characters than the specified field width and

(b) fewer characters than the specified field width?

5.15 What is the purpose of **print** ( ) function? **[LO 5.4 E]**

5.16 Describe the purpose of commonly used conversion characters in a **printf** ( ) function. **[LO 5.4 E]**

5.17 How does a control string in a **printf** ( ) function differ from the control string in a **scanf** ( ) function?

5.18 What happens if an output data item contains **[LO 5.4 H]** **[LO 5.4 M]**

(a) more characters than the specified field width and

(b) fewer characters than the specified field width?

5.19 How are the unrecognized characters within the control string are interpreted in **[LO 5.4 H]**

(a) **scanf** function; and

(b) **printf** unction?

DEBUGGING EXERCISES

5.1 State errors, if any, in the following input statements. **[LO 5.3 M]**

(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);

PROGRAMMING EXERCISES

5.1 Given the string "WORDPROCESSING", write a program to read the string from the terminal and display the same in the following formats: **[LO 5.4 M]**

(a) WORD PROCESSING

(b) WORD

PROCESSING

(c) W.P.

5.2 Write a program to read the values of x and y and print the results of the following expressions in one line: **[LO 5.3 M]**

(a) $\frac{x+y}{x-y}$

(b) $\frac{x+y}{2}$

(c) (x+y)(x–y)

5.3 Write a program to read the following numbers, round them off to the nearest integers and print out the results in integer form: **[LO 5.4 H]**

35.7 50.21 – 23.73 – 46.45

5.4 Write a program that reads 4 floating point values in the range, 0.0 to 20.0, and prints a horizontal bar chart to represent these values using the character * as the fill character. For the purpose of the chart, the values may be rounded off to the nearest integer. For example, the value 4.36 should be represented as follows.

[LO 5.3, 5.4 H]

[redacted]

Note that the actual values are shown at the end of each bar.

5.5 Write an interactive program to demonstrate the process of multiplication. The program should ask the user to enter two two-digit integers and print the product of integers as shown below.

[LO 5.4 H]

[redacted]

5.6 Write a program to read three integers from the keyboard using one **scanf** statement and output them on one line using:

[LO 5.4 H]

(a) three **printf** statements,

(b) only one **printf** with conversion specifiers, and

(c) only one **printf** without conversion specifiers.

5.7 Write a program that prints the value 10.45678 in exponential format with the following specifications: **[LO 5.4 M]**

(a) correct to two decimal places;

(b) correct to four decimal places; and

(c) correct to eight decimal places.

5.8 Write a program to print the value 345.6789 in fixed-point format with the following specifications:

[LO 5.4 M]

(a) correct to two decimal places;

(b) correct to five decimal places; and

(c) correct to zero decimal places.

5.9 Write a program to read the name ANIL KUMAR GUPTA in three parts using the **scanf** statement and to display the same in the following format using the **printf** statement. **5.3, 5.4 H] [LO**

(a) ANIL K. GUPTA

(b) A.K. GUPTA

(c) GUPTA A.K.

5.10 Write a program to read and display the following table of data. **[LO 5.3, 5.4 H]**

[redacted]

The name and code must be left-justified and price must be right-justified.

5.11 Write a C program to print inventory stock report using some sample data. The report should show item code, quantity and inventory location as formatted output. **[LO 5.4 H]**

5.12 Write a C program to display the Pascal's triangle. **[LO 5.2 H]**

5.13 Write a C program to input a currency value in Dollars and display its equivalent Euro and INR amounts. You may use current exchange rate for conversion purpose. **[LO 5.2 E]**

5.14 Write a C program to display the following pattern. **[LO 5.2 H]**

[redacted]

5.15 Write a C program to input an investment amount and compute its fixed deposit cumulative return after 10 years at arate of interest of 8.75%. **[LO 5.4 E]**

INTERVIEW QUESTIONS

5.1 How will you print the % character on the output screen? **[LO 5.4 M]**

5.2 What will be the output of following code?

```
#define char int
void main()
{
```

```
char i;
printf("%d",sizeof(i));
}
```

[LO 5.4 H]

5.3 Will the following statement compile successfully?

```
printf("%d",printf("Hello"));
```

[LO 5.4 H]

5.4 Is it possible to run a program successfully without including header file? For instance, will the following code run successfully?

```
void main()
{
printf("Hello World\n");
getch();
}
```

[LO 5.4 H]

5.5 What will be the output of the following printf statement?

```
printf("%d");
```

[LO 5.4 H]

5.6 What will be the output of the following code?

```
#define rate 10
void main()
{
#define rate 20
printf("rate = %d",rate);
}
```

[LO 5.4 H]

E for Easy, **M** for Medium and **H** for High

Chapter 6

Decision Making and Branching

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

- LO 6.1 Discuss decision making with if statement
- LO 6.2 Describe if...else statement
- LO 6.3 Explain switch statement
- LO 6.4 Know conditional operator
- LO 6.5 Illustrate how goto statement is used for unconditional branching

INTRODUCTION

W e 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 by supporting the following statements:

1. **if** statement
2. **switch** statement
3. Conditional operator statement
4. **goto** statement

These statements are popularly known as *decision-making statements* . Since these statements 'control' the flow of execution, they are also known as *control statements* .

We have already used some of these statements in the earlier examples. Here, we shall discuss their features, capabilities and applications in more detail.

DECISION MAKING WITH IF STATEMENT

LO 6.1
Discuss 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)

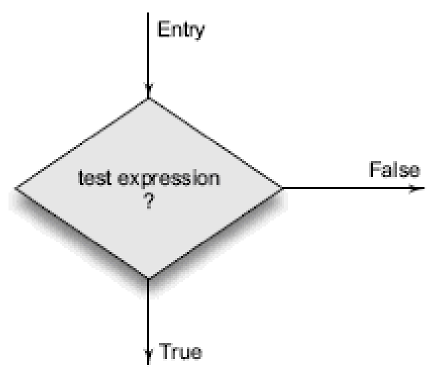


Fig. 6.1 Two-way branching

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression (relation or condition) is 'true' (or 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 .

Some examples of decision making, using **if** statements 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. The different forms are:

1. Simple **if** statement
2. **if.....else** statement
3. Nested **if....else** statement
4. **else if** ladder.

We shall discuss each one of them in the next few section.

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 .

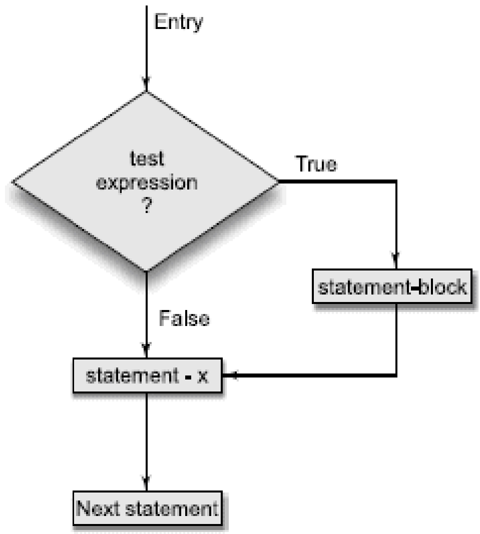


Fig. 6.2 Flowchart of simple **if** control

Consider the following segment of a program that is written for processing of marks obtained in an entrance examination.

```
.........
.........
if (category -- SPORTS)
{
     marks - 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 adde.

WORKED-OUT PROBLEM 6.1

E

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

```
Program
            main()
            {
              int a, b, c, d;
              float ratio;

              printf("Enter four integer values\n");
              scanf("%d %d %d %d", &a, &b, &c, &d);

              if (c-d != 0) /* Execute statement block */
              {
                  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 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.

TIP

It is better to avoid float operands in conjunction with the equal/not equal condition checking operators.

Note the use of **float** conversion in the statement evaluating the **ratio** . This is necessary to avoid truncation due to integer division. Remember, the output of the first run –3.181818 is printed correct to six decimal places. The answer contains a round off error. If we wish to have higher accuracy, we must use **double** or **long double** data type.

The simple **if** is often used for counting purposes. The Worked-Out Problem 6.2 illustrates this.

WORKED-OUT PROBLEM 6.2 M

The program in Fig. 6.4 counts the number of boys whose weight is less than 50 kg 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 of **weight** is less than 50, then the following statement is executed, which in 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

```
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 kg\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 kg
            and height > 170 cm =3
```

Fig. 6.4 Use of **if** for counting

Applying De Morgan's Rule

While designing decision statements, we often come across a situation where the logical NOT operator is applied to a compound logical expression, like !(x&&y||!z). However, a positive logic is always easy to read and comprehend than a negative logic. In such cases, we may apply what is known as

De Morgan's rule to make the total expression positive. This rule is as follows:

"Remove the parentheses by applying the NOT operator to every logical expression component, while complementing the relational operators"

That is,

x becomes !x

!x becomes x

&& becomes ||

|| becomes &&

Examples:

!(x && y || !z) becomes !x || !y && z

!(x < = 0 || !condition) becomes x >0&& condition

THE IF.....ELSE STATEMENT

The **if...else** statement is an extension of the simple **if** statement. The general form is

LO 6.2
Describe if...else statement

```
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** statements 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 the statement-x.

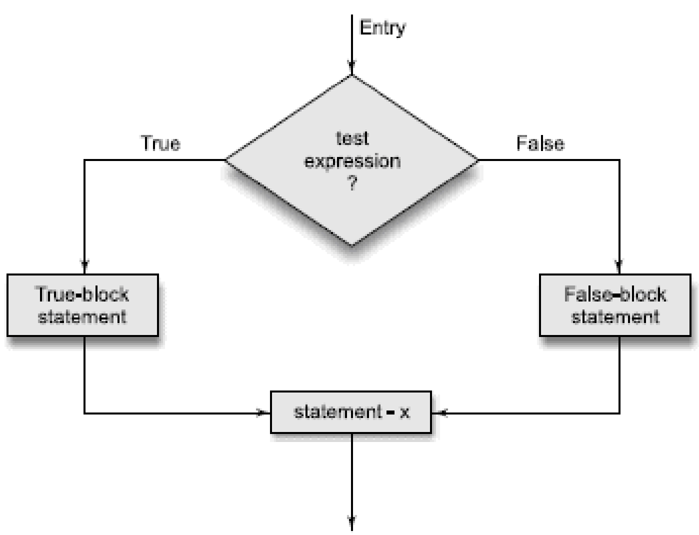


Fig. 6.5 Flowchart of **if......else** control

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 student 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 the 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");
..........
.........
```

WORKED-OUT PROBLEM 6.3

H

A program to evaluate the power series.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^2}{3!} + \cdots + \frac{x^n}{n!}, \; 0 < x < 1$$

is given in Fig. 6.6. It uses **if......else** to test the accuracy.

The power series contains the recurrence relationship of the type

$$T_n = T_{n-1} \left(\frac{x}{n}\right) \text{ for } n > 1$$

$$T_1 = x \text{ for } n = 1$$

$$T_0 = 1$$

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

$$e^x = T_0 + T_1 + T_2 + ... + T_n = \text{sum}$$

Program

```
#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 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.

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 shown below:

```
if      (test condition-1)
{
    if  (test condition-2);
    {
        statement -1;
    }
    else
    {
        statement -2;
    }
}
else
{
    statement -3;
}
statement -x;
```

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.

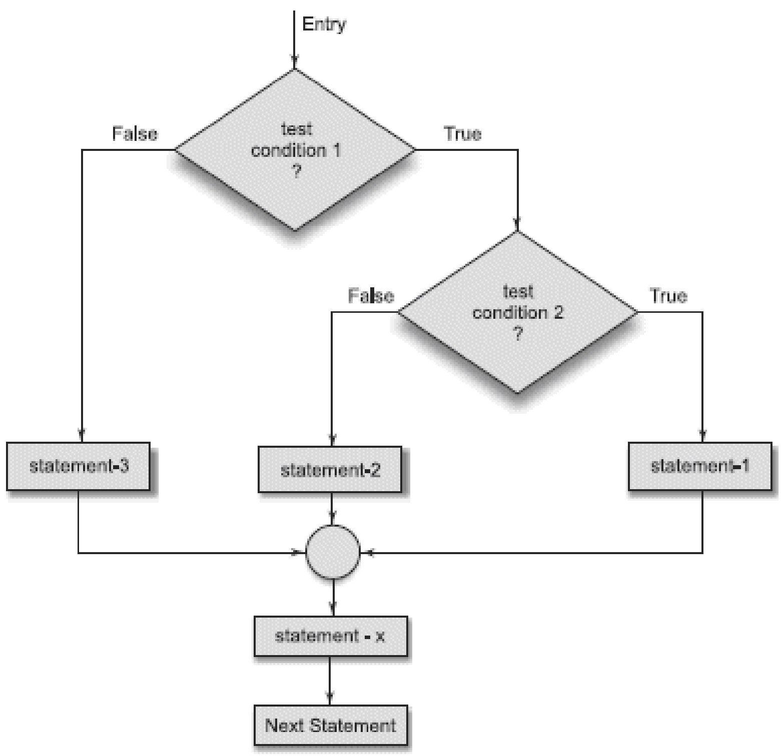


Fig. 6.7 Flow chart of nested **if…else** statements

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 balance, 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 (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** .

> **TIP**
> Braces should be used to clearly define statement blocks under if/else parts to prevent logical errors.

WORKED-OUT PROBLEM 6.4 H

The program in Fig. 6.8 selects and prints the largest of the three numbers using nested **if....else** statements.

Program

```
#include <stdio.h>
main()
{
        int A, B, C;
        printf("Enter three values\n");
        scanf("%d %d %d", &A, &B, &C);
        printf("\nLargest value is ");
        if (A>B)
        {
                if (A>C)
                        printf("%d\n", A);
                else
                        printf("%d\n", C);
        }
        else
        {
                if (C>B)
                        printf("%d\n", C);
                else
                        printf("%d\n", B);
        }
}
```

Output

```
Enter three values
23445 20379 30843
Largest value is 30843
```

Fig. 6.8 Selecting the largest of three numbers

Dangling Else Problem

One of the classic problems encountered when we start using nested **if….else** statements is the dangling else. This occurs when a matching **else** is not available for an **if** . The answer to this problem is very simple. Always match an **else** to the most recent unmatched **if** in the current block. In some cases, it is possible that the false condition is not required. In such situations, **else** statement may be omitted

> **TIP**
> For nested if-else statements it always benefits by remembering that an else is always paired with the nearest unmatched if.

"**else** is always paired with the most recent unpaired **if** "

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:

```
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;
     statement-x;
```

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. Fig. 6.9 shows the logic of execution of **else if** ladder statements.

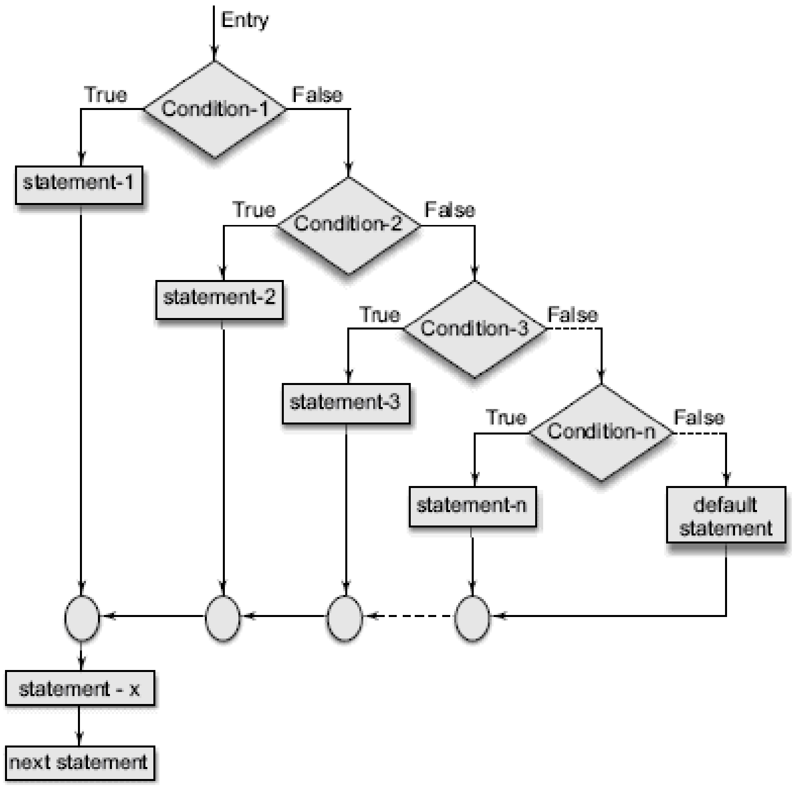


Fig. 6.9 Flow chart of **else..if** ladder

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 (H)
60 to 79	First Division (1)
50 to 59	Second Division (2)
40 to 49	Third Division (3)
0 to 39	Fail (F)

This grading can be done using the **else if** ladder as follows:

```
if (marks > 79)
     grade = 'H';
else if (marks > 59)
     grade = '1';
else if (marks > 49)
     grade = '2';
else if (marks > 39)
     grade = '3';
else
     grade = 'F';
printf ("%c\n", grade);
```

Consider another example given below:

```
——
——
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 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 is left to the programmer. However, 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. In the above two examples, RED, GREEN, WHITE and YELLOW are assumed to be symbolic constants.

WORKED-OUT PROBLEM 6.5 H

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 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.

```
Program
            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

Rules for Indentation

When using control structures, a statement often controls many other statements that follow it. In such situations it is a good practice to use *indentation* to show that the indented statements are dependent on the preceding controlling statement. Some guidelines that could be followed while using indentation are listed below:

- Indent statements that are dependent on the previous statements; provide at least three spaces of indentation.
- Align vertically else clause with their matching if clause.

- Use braces on separate lines to identify a block of statements.
- Indent the statements in the block by at least three spaces to the right of the braces.
- Align the opening and closing braces.
- Use appropriate comments to signify the beginning and end of blocks.
- Indent the nested statements as per the above rules.
- Code only one clause or statement on each line.

TEST YOUR SKILLS

Write a C program to input the cost price and selling price of an item and display the profit. [E]

THE SWITCH STATEMENT

LO 6.3
Explain switch statement

We have seen that when one of the many alternatives is to be selected, we can use an **if** statement 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 below:

```
switch (expression)
{
   case value-1:
               block-1
               break;
   case value-2:
               block-2
               break;
   ......
   ......
   default:
               default-block
               break;
}
statement-x;
```

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 successfully 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.

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** . (ANSI C permits the use of as many as 257 case labels).

The selection process of **switch** statement is illustrated in the flow chart shown in Fig. 6.11 .

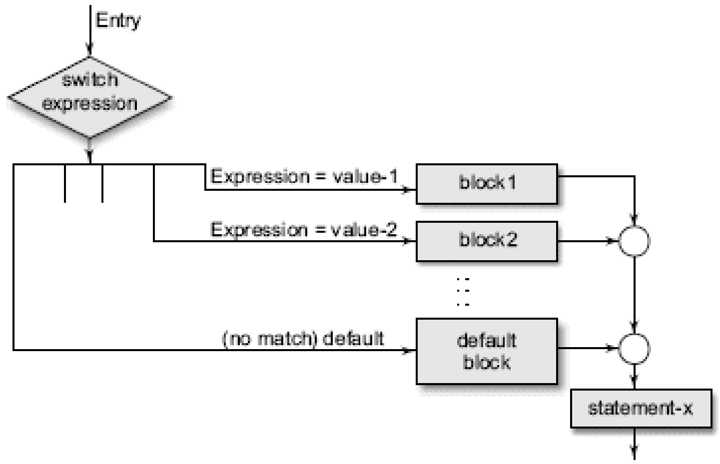


Fig. 6.11 Selection process of the **switch** statement

The **switch** statement can be used to grade the students as discussed in the last section, as illustrated below:

```
——
——
index = marks/10
switch (index)
{
        case 10:
        case 9:
        case 8:
               grade = 'H';
               break;
        case 7:
        case 6:
               grade = '1';
               break;
        case 5:
               grade = '2';
               break;
        case 4:
               grade = '3';
               break;
        default:
               grade = 'F';
               break;
        }
        printf("%c\n", grade);
        ——
        ——
```

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 = 'H';

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** statement. ANSI C permits 15 levels of nesting.

Rules for Switch Statement

> **TIP**
> Always use character or integer variables with switch.

- The **switch** expression must be an integer type.
- Case labels must be constants or constant expressions.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with colon.
- The **break** statement transfers the control out of the **switch** statement.
- The **break** statement is optional. That is, two or more case labels may belong to the same statements.
- The **default** label is optional. If present, it will be executed when the expression does not find a matching case label.
- There can be at most one **default** label.
- The **default** may be placed anywhere but usually placed at the end.
- It is permitted to nest **switch** statements.

WORKED-OUT PROBLEM 6.6 E

Write a complete C program that reads a value in the range of 1 to 12 and print the name of that month and the next month. Print error for any other input value.

```
Program
            #include<stdio.h>
            #include<conio.h>
            #include<stdlib.h>
            void main()
            {
              char month[12][20] = {"January","February","March","April","May","June",
              "July","August","September","October","November","December"};
              int i;

              printf("Enter the month value: ");
              scanf("%d",&i);

              if(i<1 || i>12)
              {
                printf("Incorrect value!!\nPress any key to terminate the program...");
                getch();
                exit(0);
              }

              if(i!=12)
                printf("%s followed by %s",month[i-1],month[i]);
              else
                printf("%s followed by %s",month[i-1],month[0]);

            getch();
            }
Output
            Enter the month value: 6
            June followed by July
```

Fig. 6.12 Program to read and print name of months in the range of 1 and 12

TEST YOUR SKILLS

Write a C program that makes use of the switch-case statement to compute the number of occurrences of the various numeric digits in the given number. **[M]**

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:

LO 6.4
Know conditional operator

conditional expression ? expression1 : expression2

The *conditional expression* is evaluated first. If the result is non-zero, *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 £ 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. However, the readability is poor. It is better to use **if** statements when more than a single nesting of conditional operator is required.

WORKED-OUT PROBLEM 6.7

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. 6.13 .

Program

```
#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 = (loan1>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. 6.13 Illustration of the conditional operator

The program uses the following variables:

loan3 - present loan amount requested

loan2 - previous loan amount pending

loan1 - previous to previous loan pending

sum23 - sum of loan2 and loan3

sancloan - loan sanctioned

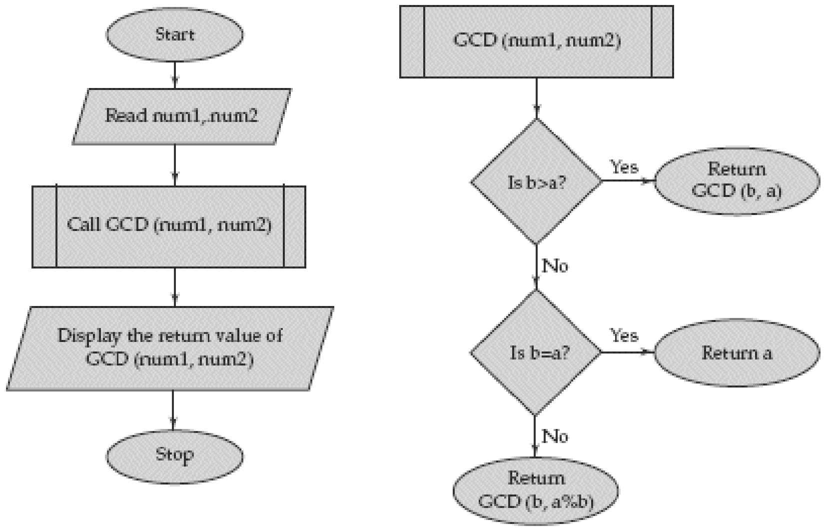
The rules for sanctioning new loan are:

1. loan1 should be zero.
2. loan2 + loan3 should not be more than MAXLOAN.

Note the use of **long int** type to declare variables.

WORKED-OUT PROBLEM 6.8 M

Write a program to determine the Greatest Common Divisor (GCD) of two numbers.


Start
Read num1,.num2
Call GCD (num1, num2)
Display the return value of GCD (num1, num2)
Stop
GCD (num1, num2)
Is b>a?
Yes
Return GCD (b, a)
No
Is b=a?
Yes
Return a
No
Return GCD (b, a%b)

Algorithm

```
Step 1 – Start
Step 2 – Accept the two numbers whose GCD is to be found (num1, num2)
Step 3 – Call function GCD(num1,num2)
Step 4 – Display the value returned by the function call GCD(num1, num2)
Step 5 – Stop
GCD(a,b)
Step 1 – Start
Step 2 – If b > a goto Step 3 else goto Step 4
Step 3 – Return the result of the function call GCD(b,a) to the calling function
Step 4 – If b - 0 goto Step 5 else goto Step 6
Step 5 – Return the value a to the calling function
Step 6 – Return the result of the function call GCD(b,a mod b) to the calling function
```

Program

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

int GCD(int m, int n);
void main()
     {
          int num1,num2;
          clrscr();
          printf("Enter the two numbers whose GCD is to be found: ");
          scanf("%d %d",&num1,&num2);

          printf("\nGCD of %d and %d is %d\n",num1,num2,GCD(num1,num2));
          getch();
     }

     int GCD(int a, int b)
     {
          if(b>a)
             return GCD(b,a);
          if(b--0)
             return a;
          else
             return GCD(b,a%b);
     }
```

Output

```
Enter the two numbers whose GCD is to be found: 18 12
GCD of 18 and 12 is 6
```

Fig. 6.14 Program to determine GCD of two numbers

The Program in Fig. 6.14 makes use of the user-defined recursive function GCD(). Declaring, defining and calling of functions will be discussed in detail in a later chapter.

Some Guidelines for Writing Multiway Selection Statements

Complex multiway selection statements require special attention. The readers should be able to understand the logic easily. Given below are some guidelines that would help improve readability and facilitate maintenance.

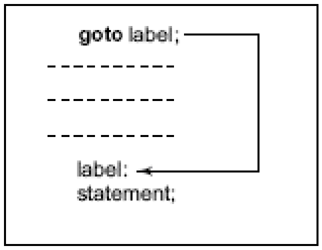
- Avoid compound negative statements. Use positive statements wherever possible.
- Keep logical expressions simple. We can achieve this using nested if statements, if necessary (KISS - Keep It Simple and Short).
- Try to code the normal/anticipated condition first.
- Use the most probable condition first. This will eliminate unnecessary tests, thus improving the efficiency of the program.
- The choice between the nested if and switch statements is a matter of individual's preference.
  A good rule of thumb is to use the switch when alter-native paths are three to ten.
- Use proper indentations (See Rules for Indentation).
- Have the habit of using default clause in switch statements.
- Group the case labels that have similar actions.

THE GOTO STATEMENT

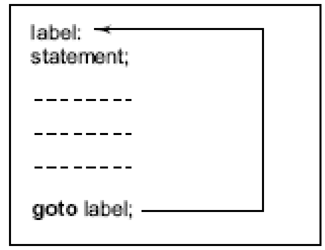
LO 6.5
Illustrate how goto statement is used for unconditional branching

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.

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 *label:* 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.

> **TIP**
> goto statements should be avoided as far as possible. Use of goto makes the program unstructured.

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. Worked-Out Problem 6.9 illustrates how such infinite loops can be eliminated.

WORKED-OUT PROBLEM 6.9 E

Program presented in Fig. 6.15 illustrates the use of the **goto** statement. The program evaluates the 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

```
#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("Value = %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
      Value -3 is negative
      75.000000       8.660254
      11.250000       3.354102
      End of computation
```

Fig. 6.15 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:

```
----
----
while (----)
{
  for (----)
  {
  ----
  ----
  if (----)goto end_of_program;
  ----
  }                                 Jumping
----                                out of
----                                loops
}
end_of_program:
```

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.

KEY CONCEPTS

- **DECISION-MAKING STATEMENTS:** Are the statements that control the flow of execution in a program. **[LO 6.1]**
- switch **STATEMENT:** Is a multi-way decision making statement that chooses the statement block to be executed by matching the given value with a list of case values. **[LO 6.3]**
- **CONDITIONAL OPERATOR:** Is a two-way decision making statement that returns one of the two expression values based on the result of the conditional expression. **[LO 6.4]**
- goto **STATEMENT:** Is used for unconditional branching. It transfers the flow of control to the place where matching label is found. **[LO 6.5]**
- **INFINITE LOOP:** Is a condition where a set of instructions is repeatedly executed. **[LO 6.5]**

ALWAYS REMEMBER

- Be aware of any side effects in the control expression such as if(x++). **[LO 6.1]**
- Check the use of =operator in place of the equal operator = =. **[LO 6.1]**
- Do not give any spaces between the two symbols of relational operators = =, !=, >= and <=. **[LO 6.1]**
- Writing !=, >= and <= operators like =!, => and =< is an error. **[LO 6.1]**
- Remember to use two ampersands (&&) and two bars (| | |) for logical operators. Use of single operators will result in logical errors. **[LO 6.1]**
- Do not forget to place parentheses for the if expression. **[LO 6.1]**
- It is an error to place a semicolon after the if expression. **[LO 6.1]**
- Do not use the equal operator to compare two floating-point values. They are seldom exactly equal. **[LO 6.1]**
- Avoid using operands that have side effects in a logical binary expression such as (x – –&&++y). The second operand may not be evaluated at all. **[LO 6.1]**

- Be aware of dangling **else** statements. **[LO 6.2]**
- Use braces to encapsulate the statements in **if** and **else** clauses of an if…. else statement. **[LO 6.2]**
- Do not forget to use a break statement when the cases in a switch statement are exclusive. **[LO 6.3]**
- Although it is optional, it is a good programming practice to use the default clause in a switch statement. **[LO 6.3]**
- It is an error to use a variable as the value in a case label of a switch statement. (Only integral constants are allowed.) **[LO 6.3]**
- Do not use the same constant in two case labels in a switch statement. **[LO 6.3]**
- Try to use simple logical expressions. **[LO 6.4]**
- Be careful while placing a goto label in a program as it may lead to an infinite loop condition. **[LO 6.5]**

BRIEF CASES

1. Range of Numbers[LO 6.1, 6.2 M]

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. 6.16 .

```
Program
            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        : %f\nAverage : %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. 6.16 Calculation of range of values

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.

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;

Note that this program can be written without using **goto** statements. Try

.

2. Pay-Bill Calculations [LO 6.2, 6.5 M]

Problem **:** A manufacturing company has classified its executives into four levels for the benefit of certain perks. The levels and corresponding perks are shown below:

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. 6.17 . Note that the last statement should be an executable statement. That is, the label **stop:** cannot be the last line.

Program

```
#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. 6.17 Pay-bill calculations

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

6.1 Nesting is allowed for which of the following statements? **[LO 6.3 H]**

(a) If

(b) If else

(c) Switch

(d) All of the above

6.2 What will be the output of the following if-else statement? **[LO 6.2 H]**

```
if(x=5)
printf("Condition is true\n");
else
printf("Condition is false\n");
```

(a) String 'Condition is true' is printed.

(b) x is assigned the value 5 and string 'Condition is true' is printed.

(c) String 'Condition is false' is printed.

(d) None of the above

6.3 For the following if-else construct, **[LO 6.2 M]**

```
if(a>=5 && a<=10)
printf("True");
else
printf("False");
```

which of the following statements is correct?

(a) True if 5>=a>=10, false otherwise

(b) True if 5<=a<=10, false otherwise

(c) True if 5<=a>=10, false otherwise

(d) None of the above

6.4 For the following if-else construct, **[LO 6.2 M]**

```
if(x%3==0 && x%7==0)
printf("True");
else
printf("False");
```

which of the following statements is correct?

(a) True if the given positive integer x is a multiple of 3 and 7; false otherwise

(b) True if the given positive integer x is a multiple of 3 or 7; false otherwise

(c) False if the given positive integer x is a multiple of 3 and 7; true otherwise

(d) False if the given positive integer x is a multiple of 3 or 7; true otherwise

6.5 What will be the output of the following program? **[LO 6.2 H]**

```
main()
{
x=5;
if(x==5);
printf("Hello World");
else
printf("I am in else block");
}
```

(a) String 'Hello World' is printed.

(b) String 'I am in else block' is printed.

(c) Nothing is printed.

(d) Compiler gives syntax error and the program is not executed.

REVIEW QUESTIONS

6.1 State whether the following are *true* or *false* :

(a) A **switch** expression can be of any type. **[LO 6.3 E]**

(b) A program stops its execution when a **break** statement is encountered. **[LO 6.3 E]**

(c) Each case label can have only one statement. **[LO 6.3 E]**

(d) The **default** case is required in the **switch** statement. **[LO 6.3 E]**

(e) When **if** statements are nested, the last **else** gets associated with the nearest **if** without an **else** . **[LO 6.2 M]**

(f) One **if** can have more than one **else** clause. **[LO 6.2 M]**

(g) Each expression in the **else if** must test the same variable. **[LO 6.2 M]**

(h) A **switch** statement can always be replaced by a series of **if..else** statements. **[LO 6.3 M]**

(i) Any expression can be used for the **if** expression. **[LO 6.1 H]**

(j) The predicate !((x >= 10)¦(y = = 5)) is equivalent to (x < 10) && (y !=5). **[LO 6.1 H]**

(k) It is mandatory to include an else block while using an if statement. **[LO 6.2 H]**

(l) The default block can be placed at the beginning of the switch case construct. **[LO 6.3 H]**

6.2 Fill in the blanks in the following statements.

(a) The ________ operator is true only when both the operands are true. **[LO 6.1 E]**

(b) Multiway selection can be accomplished using an **else if** statement or the ___________ statement. **[LO 6.3 E]**

(c) The _______ statement when executed in a **switch** statement causes immediate exit from the structure. **[LO 6.3 E]**

(d) The expression ! (x ! = y) can be replaced by the expression _________. **[LO 6.1 M]**

(e) The ternary conditional expression using the operator ?: could be easily coded using _______statement. **[LO 6.4 M]**

(f) The _______________ operator is used to build two-way decision-making statement that returns one of the two values based on the result of the expression. **[LO 6.4 E]**

(g) The _________ statement is used to branch unconditionally from one point to another in a C program. **[LO 6.5 E]**

6.3 The following is a segment of a program: **[LO 6.1 M]**

```
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.

6.4 Rewrite each of the following without using compound relations:

(a) if (grade <= 59 && grade >= 50) **[LO 6.1 E]**

```
second = second + 1;
```

(b) if (number > 100 || number < 0) **[LO 6.2 M]**

```
printf(" Out of range");
else
sum = sum + number;
```

(c) if ((M1 > 60 && M2 > 60) || T > 200) **[LO 6.2 M]**

```
printf(" Admitted\n");
else
printf(" Not admitted\n");
```

6.5 Assuming x = 10, state whether the following logical expressions are true or false. **[LO 6.1 H]**

(a) x = = 10 && x > 10 && !x

(b) x = = 10 || x > 10 && ! x

(c) x = = 10 && x > 10 || ! x

(d) x = = 10 || x > 10 || !x

6.6 Find errors, if any, in the following switch related statements. Assume that the variables x and y are of int type and x = 1 and y = 2 **[LO 6.3 H]**

(a) switch (y);

(b) case 10;

(c) switch (x + y)

(d) switch (x) {case 2: y = x + y; break};

6.7 Simplify the following compound logical expressions **[LO 6.1 M]**

(a) !(x <=10)

(b) !(x = = 10) ||! ((y = = 5) || (z < 0))

(c) ! ((x +y = = z) && !(z > 5)

(d) !((x <=5) && (y = = 10) & & (z < 5))

6.8 Assuming that x = 5, y = 0, and z = 1 initially, what will be their values after executing the following code segments? **6 .2 M] [LO**

```
(a) if (x && y)
        x = 10;
    else
        y = 10;
(b) if (x || y || z)
        y = 10;
    else
        z = 0;
(c) if (x)
      if (y)
        z = 10;
    else
        z = 0;
(d) if (x = = 0 || x & & y)
      if (!y)

    z = 0;
else
    y = 1;
```

6.9 Assuming that x = 2, y = 1 and z = 0 initially, what will be their values after executing the following code segments? **[LO 6.3 M]**

```
(a)  switch (x)
          {
               case 2:
                    x = 1;
                    y = x + 1;
               case 1:
                    x = 0;
                    break;
          default:
                    x = 1;
                    y = 0;
          }
(b)  switch (y)
          {
          case 0:
               x = 0;
               y = 0;
          case 2:
               x = 2;
               z = 2;
          default:
               x = 1;
               y = 2;
     }
```

6.10 What is the output of the following program? **[LO 6.2 M]**

```
main (    )
{
          int m - 5 ;
          if (m < 3) printf("%d" , m+1) ;
          else if(m < 5) printf("%d", m+2);
          else if(m < 7) printf("%d", m+3);
          else printf("%d", m+4);
}
```

6.11 What is the output of the following program? **[LO 6.2 M]**

```
main ( )
{
          int m - 1;
          if ( m--1)
          {
               printf ( " Delhi " ) ;
               if (m -- 2)
               printf( "Chennai" ) ;
               else
               printf("Bangalore") ;
          }
          else;
          printf(" END");
}
```

6.12 What is the output of the following program? **[LO 6.4 M]**

```
main( )
{
          int m ;
          for (m = 1; m<5; m++)
               printf(%d\n", (m%2) ? m : m*2);
}
```

6.13 What is the output of the following program? **[LO 6.5 M]**

```
main( )
{
          int m, n, p ;
          for ( m = 0; m < 3; m++ )
          for (n = 0; n<3; n++ )
          for ( p = 0; p < 3;; p++ )
          if ( m + n + p == 2 )
          goto print;

          print :
          printf("%d, %d, %d", m, n, p);
}
```

6.14 What will be the value of x when the following segment is executed? **[LO 6.4 E]**

int x = 10, y = 15;

x = (x<y)? (y+x) : (y-x) ;

6.15 What will be the output when the following segment is executed? **[LO 6.2 M]**

```
          int x = 0;
          if (x >= 0)
          if ( x > 0 )
          printf("Number is positive");
else
printf("Number is negative");
```

6.16 What will be the output when the following segment is executed? **[LO 6 .3 M]**

```
char ch = 'a' ;
switch (ch)

{
          case 'a' :
          printf( "A" ) ;
          case'b':
          Printf ("B") ;
          default :
          printf(" C ") ;
}
```

6.17 What will be the output of the following segment when executed? **[LO 6.2 E]**

```
int x = 10, y = 20;
if( (x<y) || (x+5) > 10 )
printf("%d", x);
else
printf("%d", y);
```

6.18 What will be output of the following segment when executed? **[LO 6.2 M]**

```
int a = 10, b = 5;
if (a > b)
{
        if(b > 5)
        printf("%d", b);
}
else
        printf("%d", a);
```

DEBUGGING EXERCISES

6.1 Find errors, if any, in each of the following segments:

(a) if (x + y = z && y > 0) **[LO 6.1 E]**

printf(" ");

(b) if (p < 0) || (q < 0) **[LO 6.1 E]**

printf (" sign is negative");

(c) if (code > 1); **[LO 6.2 M]**

a = b + c

else

a = 0

6.2 Find the error, if any, in the following statements: **[LO 6.1 H]**

(a) if (x > = 10) then

printf ("\n") ;

(b) if x > = 10

printf ("OK") ;

(c) if (x = 10)

printf ("Good") ;

(d) if (x = < 10)

printf ("Welcome") ;

PROGRAMMING EXERCISES

6.1 Write a program to determine whether a given number is 'odd' or 'even' and print the message

NUMBER IS EVEN

or

NUMBER IS ODD

(a) without using **else** option **[LO 6.1 E]** , and (b) with **else** option. **[LO 6.2 E]**

6.2 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. **[LO 6.1 M]**

6.3 A set of two linear equations with two unknowns x1 and x2 is given below: **[LO 6.2 H]**

$ax_1 + bx_2 = m$

$cx_1 + dx_2 = n$

The set has a unique solution

$$x1 = \frac{md - bn}{ad - cb}$$

$$x2 = \frac{na - mc}{ad - cb}$$

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.

6.4 Given a list of marks ranging from 0 to 100, write a program to compute and print the number of students: **[LO 6.2 M]**

(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.

6.5 Admission to a professional course is subject to the following conditions: **[LO 6.2 M]**

(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.

6.6 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. **[LO 6.2 H]**

Number	*0.0*	*0.1*	*0.2*	*. . . . . . .*	*0.9*
0.0					
1.0					
2.0					
3.0			x		y
9.0					

6.7 Shown below is a Floyd's triangle. **[LO 6.2 H]**

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
```

6.8 A cloth showroom has announced the following seasonal discounts on purchase of items: **[LO 6.1, 6.3 H]**

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.

6.9 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, **[LO 6.1 M]**

(b) **else if** statements, and **[LO 6.2 M]**

(c) conditional operator ? : **[LO 6.4 M]**

6.10 Write a program to compute the real roots of a quadratic equation

$$ax^2 + bx + c = 0$$

[LO 6.2 H]

The roots are given by the equations

$$x_1 = -b + \frac{\sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = -b - \sqrt{\frac{b^2 - 4ac}{2a}}$$

The program should request for the values of the constants a, b and c and print the values of x_1 and x_2. Use the following rules:

(a) No solution, if both a and b are zero

(b) There is only one root, if a = 0 (x = –c/b)

(c) There are no real roots, if $b^2 - 4ac$ is negative

(d) Otherwise, there are two real roots

Test your program with appropriate data so that all logical paths are working as per your design. Incorporate appropriate output messages.

6.11 Write a program to read three integer values from the keyboard and displays the output stating that they are the sides of right-angled triangle. **[LO 6.2 M]**

6.12 An electricity board charges the following rates for the use of electricity: **[LO 6.2 H]**

For the first 200 units: 80 P per unit

For the next 100 units: 90 P per unit

Beyond 300 units: Rs 1.00 per unit

All users are charged a minimum of Rs. 100 as meter charge. If the total amount is more than Rs. 400, then an additional surcharge of 15% of total amount is charged.

Write a program to read the names of users and number of units consumed and print out the charges with names.

6.13 Write a program to compute and display the sum of all integers that are divisible by 6 but not divisible by 4 and lie between 0 and 100. The program should also count and display the number of such values. **[LO 6.2 M]**

6.14 Write an interactive program that could read a positive integer number and decide whether the number is a prime number and display the output accordingly. **[LO 6.2 M]**

Modify the program to count all the prime numbers that lie between 100 and 200.

NOTE : A prime number is a positive integer that is divisible only by 1 or by itself.

6.15 Write a program to read a double-type value x that represents angle in radians and a character-type variable T that represents the type of trigonometric function and display the value of

(a) sin(x), if s or S is assigned to T,

(b) cos (x), if c or C is assigned to T, and

(c) tan (x), if t or T is assigned to T

using (i) **if......else** statement **[LO 6.2 M]** , and (ii) **switch** statement. **[LO 6.3 M]**

6.16 Write a C program to input the numeric week day value (starting from Monday as 1) and display the corresponding name of the week day. **[LO 6.3 M]**

6.17 Write a C program to input two numbers a and b and display whether

(a) a is greater than b

(b) b is greater than b

(c) or, a and b are equal **[LO 6.2 E]**

6.18 Write a C program to input the total percentage of marks of a student and display its Division using below rules:

(a) Greater than or equal to 80 percent - "First Division"

(b) Greater than or equal to 60 percent and less than 80 per cent - "Second Division"

(c) Less than 60 percent - "Third Division" **[LO 6.2 M]**

6.19 Using Switch case block, input the name of the month from the user and display the corresponding number of days in that month. **[LO 6.3 M]**

6.20 Write a C program that inputs a string and counts the number of capital and small alphabets in that string. **[LO 6.1 M]**

INTERVIEW QUESTIONS

6.1 Till how many levels can nested blocks be created in C? **[LO 6.3 E]**

6.2 What happens if the conditional expression is missing in an if statement? **[LO 6.1 E]**

6.3 What is the purpose of goto statement? **[LO 6.5 E]**

6.4 What will be the output of following code? **[LO 6.2 H]**

```
void main()
```

```
{
int a=5;
if(a<0);
printf("a is negative");
else
printf("a is positive");
}
```

6.5 What could be the if expressions that always return true and false values? **[LO 6.1 E]**

6.6 What types of values are permitted to be used with a switch statement? **[LO 6.3 M]**

E for Easy, **M** for Medium and **H** for High

Chapter **7**

Decision Making and Looping

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

LO 7.1 Discuss while statement
LO 7.2 Explain do statement
LO 7.3 Describe for statement
LO 7.4 Illustrate how jumps are applied in loops

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:

```
-------------
-------------
sum = 0;
n = 1;
loop:
sum = sum + n*n;
if (n == 10)
    goto print;
else
{
    n = n+1;
    goto loop;
}
print:
-------------
-------------
```

Loop

n = 10,
end of loop

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 increased or decreased 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 st atements 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 flow charts 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. The entry-controlled and exit-controlled loops are also known as *pre-test* and *post-test* loops respectively.

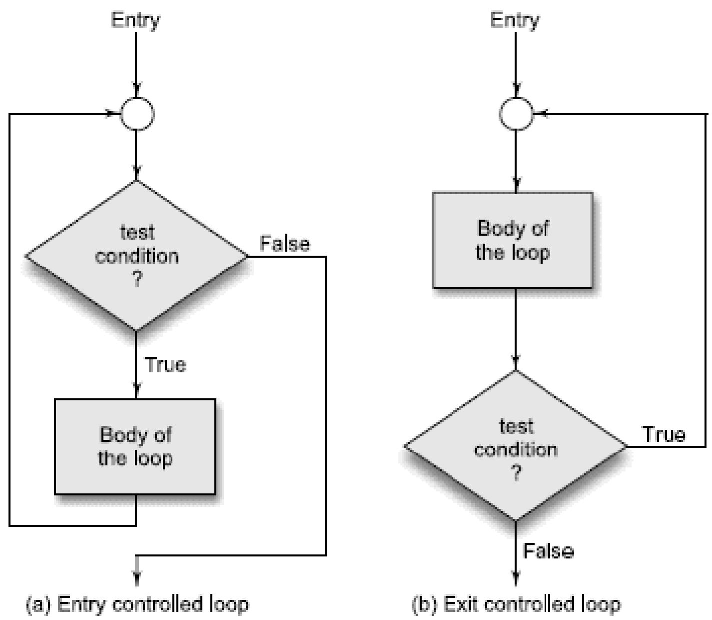


Fig. 7.1 Loop control structures

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 condition variable.
2. Execution of the statements in the loop.
3. Test for a specified value of the condition variable for execution of the loop.
4. Incrementing or updating the condition variable.

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 *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.

Sentinel Loops

Based on the nature of control variable and the kind of value assigned to it for testing the control expression, the loops may be classified into following two general categories:

1. Counter-controlled loops
2. Sentinel-controlled loops

When we know in advance exactly how many times the loop will be executed, we use a *counter-controlled loop* . We use a control variable known *as counter* . The counter must be initialized, tested and updated properly for the desired loop operations. The number of times we want to execute the loop may be a constant or a variable that is assigned a value. A counter-controlled loop is sometimes called *definite repetition loop* .

In a *sentinel-controlled loop* , a special value called a *sentinel* value is used to change the loop control expression from true to false. For example, when reading data we may indicate the "end of data" by a special value, like –1 and 999. The control variable is called **sentinel** variable. A sentinel-controlled loop is often called *indefinite repetition loop* because the number of repetitions is not known before the loop begins executing.

The while Statement

LO 7.1
Discuss while statement

The simplest 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 Section 7.1 as follows:

```
              --------
              sum = 0;
              n = 1;                    /* Initialization  */
        ┌───► while(n <= 10)            /* Testing */
        │     {
loop    │          sum = sum + n * n;
        │          n = n+1;             /* Incrementing */
        └───► }
              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. This is a typical example of counter-controlled loops. The variable n is called ***counter*** or ***control variable*** .

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;. This is a typical example of sentinel-controlled loops. The character constant 'y' is called *sentinel* value and the variable **character** is the condition variable, which often referred to as the *sentinel variable* .

WORKED-OUT PROBLEM 7.1 M

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 exists the loop.

Program

```
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;            /* Initialisation */
  /* LOOP BEGINs */
  while ( count <= n)   /* Testing */
  {
    y = y*x;
    count++;            /* Incrementing */
  }
  /* END OF LOOP */
  printf("\nx = %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

Test Your Skills

The factorial of an integer m is the product of consecutive integers from 1 to m. That is,
factorial m = m! = m x (m–1) x ...... x 1.
Write a program that computes and prints a table of factorials for any given m. **[M]**

The do Statement

LO 7.2
Explain 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.

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
        {
loop         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, and assigned to the *sentinel* variable **number** .

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 in lies between 0 and 100.

Consider another example:

```
        -------
        I = 1;                           /* Initializing */
        sum = 0;
        do
   ┌──> {
   │      sum = sum + I;
loop│     I = I+2;                       /* Incrementing */
   │    }
   └──> while(sum < 40 || I < 10);      /* Testing */
        printf("%d %d\n", I, sum);
        -------
```

The loop will be executed as long as one of the two relations is true.

WORKED-OUT PROBLEM 7.2

M

A program to print the multiplication table from 1 x 1 to 12 x 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.

Program:

```
#define COLMAX 10
#define ROWMAX 12
main()
{
     int row,column, y;
     row = 1;
     printf("     MULTIPLICATION TABLE     \n");
     printf("-----------------------\n");
     do /*......OUTER LOOP BEGINS........*/
     {
          column = 1;
          do /*.......INNER LOOP BEGINS.......*/
          {
            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**

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.

TIP

Always remember that while is entry-controlled, whereas do-while is exit-controlled.

THE for STATEMENT

Simple 'for' Loops

LO 7.3
Describe for statement

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 or decrement)
{
     body of the 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 or i > 0, 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 or decremented* using an assignment statement, such as i = i+1 or 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.

Note *C99 enhances the **for** loop by allowing declaration of variables in the initialization permits portion. See the Appendix "C99 Features".*

Consider the following segment of a program:

```
        for ( x = 0 ; x <= 9 ; x = x+1)
loop   {
          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 Section 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;

i s 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/decrementing* , 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);

WORKED-OUT PROBLEM 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.

Program

```
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

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** .

WORKED-OUT PROBLEM 7.4

E

The program in Fig. 7.5 shows how to write a C program to print nth Fibonacci number.

```
Program
  # include <stdio.h>

  void main()
  {
            int num 1-0, num2-1, n, i, fib;

            printf("\n\nEnter the value of n: ");
            scanf ("%d", &n);
            for (i - 1; i <- n-2; i++)
            {
                      fib-num1 + num2;
                      num1-num2;
                      num2-fib;
          }
          printf("\nnth fibonacci number (for n - %d) - %d, n,fib);
  }
```

Fig. 7.5 Program to print nth fibonacci number

WORKED-OUT PROBLEM 7.5

H

The program in Fig. 7.6 shows how to write a C program to print all the prime numbers between 1 and n, where 'n' is the value supplied by the user.

```
Program
  # include <stdio.h>

  void main()
  {
```

```
        int prime(int);
        int n.i;
        int temp:
        printf("Enter the value of n: ");
        scanf ("%d", &n);
        printf("Prime numbers between 1 and %d are:\n".n);
        for (i-2; i<-n;i++)
        {
                temp-prime(i); /*Call function prime*/
                if(temp---99)
                        continue;
                else
                        printf("%d\t", i);
        }
}/*end of main()*/

int prime (int num)
{
        int j:
        for (j-2;j<num; j++)
                if(num%j--0)
                        return (-99);
        if (j--num)
                return(num);
}

Output
  Enter the value of n: 20
  Prime numbers between 1 and 20 are:
  2    3    5    7    11    13    17    19
```

Fig. 7.6 Program to print all prime numbers between 1 and n

In the program shown in Fig. 7.6 , a user-defined function named, *prime* () has been defined and used. The function checks to see whether a given number is prime or not. User-defined functions have been considered in details in a later chapter.

Additional Features of for Loop

T he **for** loop in C has several capabilities that are not found in other loop 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,

for (p=1, n=0; n<17; ++n)

Note 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", i, sum);
}
```

The loop uses a compound test condition with the counter variable **i** and sentinel 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 an *‘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 simply 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 follows:

```
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 a *null statement.*

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:

```
----------
----------
for (i = 1; i < 10; ++i)
{
    ----------
    ----------
    for ( j = 1; j != 5; ++j)
    {                                   Inner   Outer
     ---------                          loop    loop
     ---------
    }
     ---------
     ---------
}
----------
----------
```

The nesting may continue up to any desired level. The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each **for** statement. (ANSI C allows up to 15 levels of nesting. However, some compilers permit more).

The program to print the multiplication table discussed in Program 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);
     }
     printf("\n");
}
-------------
```

The outer loop controls the rows while the inner loop controls the columns.

WORKED-OUT PROBLEM 7.6 **M**

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. 7.7.

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 which are as follows:

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.

Program

```
#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. 7.7 Illustration of **nested for loops**

WORKED-OUT PROBLEM 7.7

H

The program in Fig. 7.8 shows how to write a program to display a pyramid.

Program

```
#include <stdio.h>
#include <conio.h>
void main()
  {
    int num,i,y,x=40;
    clrscr();
    printf("\nEnter a number for \ngenerating the
    pyramid:\n");
    scanf("%d",&num);
    for(y=0;y<=num;y++)
    {
        gotoxy(x,y+1);
        for(i=0-y;i<=y;i++)
        printf("%3d",abs(i));
        x=x-3;
    }
    getch();
    }
```

Output

```
Enter a number for
generating the pyramid:
7
```

```
              0
            1 0 1
          2 1 0 1 2
        3 2 1 0 1 2 3
      4 3 2 1 0 1 2 3 4
    5 4 3 2 1 0 1 2 3 4 5
  6 5 4 3 2 1 0 1 2 3 4 5 6
7 6 5 4 3 2 1 0 1 2 3 4 5 6 7
```

Fig. 7.8 Program to build a pyramid

Selecting a Loop

Given a problem, the programmer's first concern is to decide the type of loop structure to be used. To choose one of the three loop supported by C, we may use the following strategy:

- Analyse the problem and see whether it required a pre-test or post-test loop.
- If it requires a post-test loop, then we can use only one loop, **do while** .
- If it requires a pre-test loop, then we have two choices: **for** and **while** .
- Decide whether the loop termination requires counter-based control or sentinel-based control.
- Use **for** loop if the counter-based control is necessary.
- Use **while** loop if the sentinel-based control is required.
- Note that both the counter-controlled and sentinel-controlled loops can be implemented by all the three control structures.

TEST YOUR SKILLS

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.) **[M]**

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 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.

LO 7.4
Illustrate how jumps are applied in loops

Jumping Out of a Loop

An early exiti 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 Figs 7.9 and 7.10.

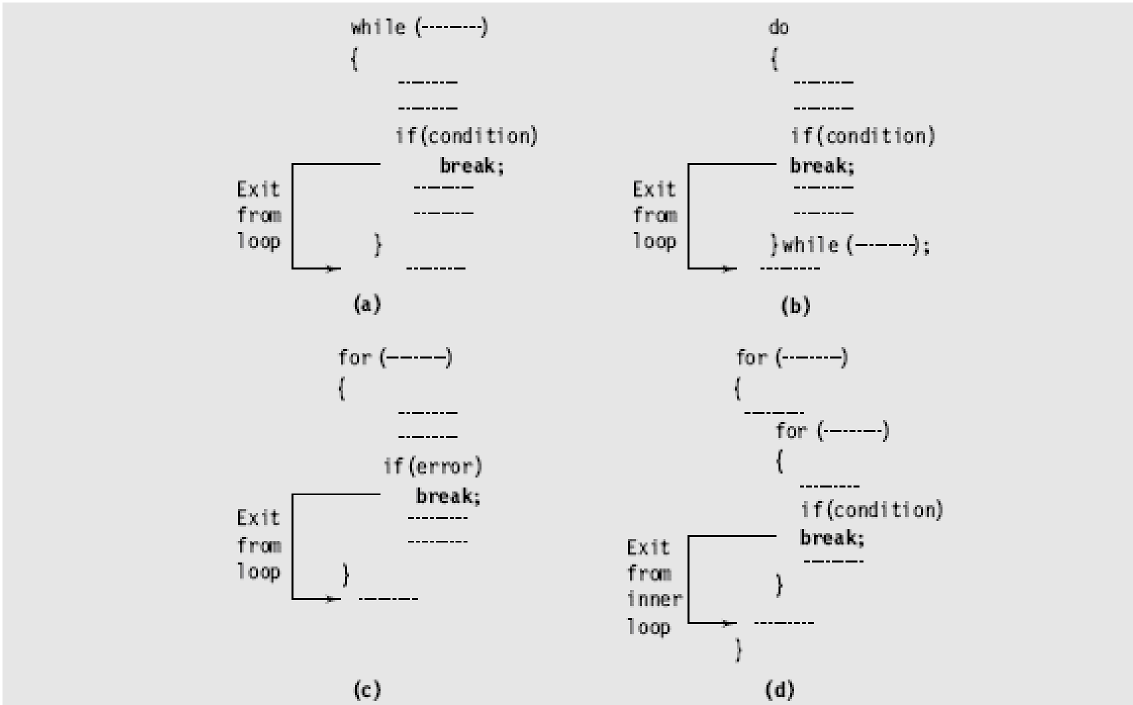


Fig. 7.9 Exiting a loop with **break** statement

When a **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.

The **goto** statement transfers the control of a program from one point to another. However, the statement should be used with due diligence, as such control transfers makes the program un-structured.

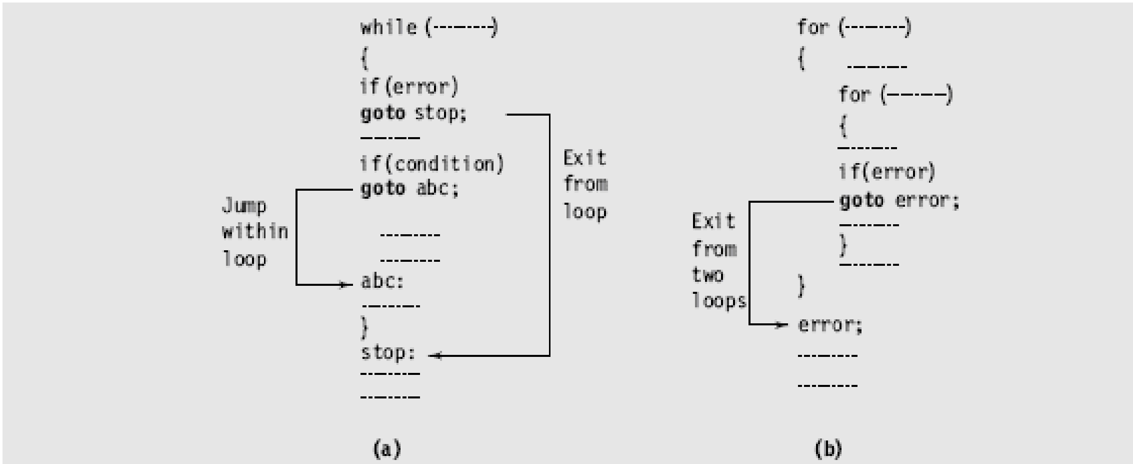


Fig. 7.10 Jumping within and exiting from the loops with **goto** statement

WORKED-OUT PROBLEM 7.8 E

The program in Fig. 7.11 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.

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. 7.11 Use of **break** in a program

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.

WORKED-OUT PROBLEM 7.9

H

A program to evaluate the series.

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \ldots.. + x^n$$

for $-1 < x < 1$ with 0.01 per cent accuracy is given in Fig. 7.12. 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 x^n reaches the desired accuracy. The value of n that decides the number of loop 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

```
#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:
      ;        /* Null Statement */
}
```

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. 7.12 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.

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 following 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!

Skipping a Part of a Loop

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. 7.13 . 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.

```
while (test-condition)
{
---------
if (---------)
    continue;
---------
---------
}
(a)
```

```
do
{
     ---------
     if (---------)
        continue;
    ---------
    ---------
} while (test-condition);
(b)
```

```
for (initialization; test condition; increment)
{
    ---------
    if (---------)
            continue;
    ---------
    ---------
}
            (c)
```

Fig. 7.13 Bypassing and continuing i loops

WORKED-OUT PROBLEM 7.10

M

The program in Fig. 7.14 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 typed 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:
          #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\n Square 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. 7.14 Use of **continue** statement

Avoiding goto

As mentioned earlier, it is a good practice to avoid using **goto** . 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 **goto** jumps shown in Fig. 7.15 would cause problems and therefore must be avoided.

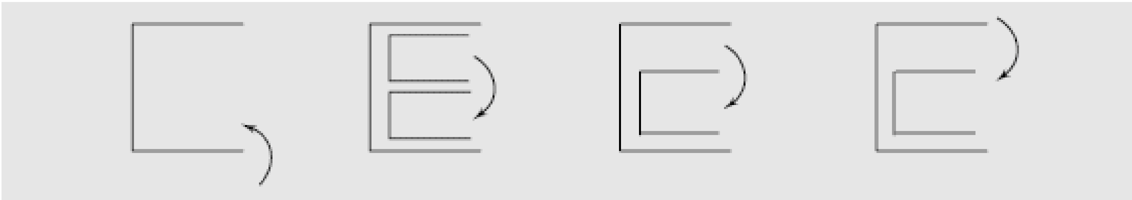


Fig. 7.15 goto jumps to be ovoided

Jumping out of the Program

We have just seen that we can jump out of a loop using either the **break** statement or **goto** statement. In a similar way, we can jump out of a program by using the library function **exit( ).** In case, due to some reason, we wish to break out of a program and return to the operating system, we can use the **exit( )** function, as shown below:

```
........
........
if (test-condition) exit(0) ;
........
........
```

The **exit( )** function takes an integer value as its argument. Normally *zero* is used to indicate normal termination and a *nonzero* value to indicate termination due to some error or abnormal condition. The use of **exit( )** function requires the inclusion of the header file **<stdlib.h>.**

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 relational operators.

if (expression ==0)

is equivalent to

if(!expression)

Similarly,

if (expression! = 0)

is equivalent to

if (expression)

For example,

if (m%5 == 0 && n%5 == 0) is same as **if (!(m%5)&&!(n%5))**

KEY CONCEPTS

- **CONTROL STATEMENT:** Tests certain conditions and directs the repeated execution of the body of the loop. **[LO 7.1]**
- **PROGRAM LOOP:** Executes a sequence of statements repeatedly until some conditions for termination of the loop are satisfied. **[LO 7.1]**
- while **STATEMENT:** Is an entry-controlled loop that evaluates the test-condition first and then executes the body of the loop if the condition is true.**[LO 7.1]**
- do **STATEMENT:** Executes the body of the loop first and then evaluates the test-condition in the subsequent while statement. **[LO 7.2]**
- break **STATEMENT:** Terminates the loop and takes the program control to the statement immediately following the loop. **[LO 7.4]**
- **continue STATEMENT:** Skips the remaining part of the loop and takes the program control to the next loop iteration. **[LO 7.4]**

ALWAYS REMEMBER

- It is a common error to use wrong relational operator in test expressions. Ensure that the loop is evaluated exactly the required number of times. **[LO 7.1]**
- Avoid a common error using = in place of = = operator. **[LO 7.1]**

- Do not compare floating-point values for equality. **[LO 7.1]**
- When performing an operation on a variable repeatedly in the body of a loop, make sure that the variable is initialized properly before entering the loop. **[LO 7.1]**
- Indent the statements in the body of loops properly to enhance readability and understandability. **[LO 7.1]**
- Use of blank spaces before and after the loops and terminating remarks are highly recommended. **[LO 7.1]**
- Do not forget to place the semicolon at the end of **do ….while** statement. **[LO 7.2]**
- Do not forget to place the *increment* statement in the body of a **while** or **do…while** loop. **[LO 7.2]**
- Avoid using **while** and **for** statements for implementing exit-controlled (post-test) loops. Use **do…while** statement. Similarly, do not use **do…while** for pre-test loops. **[LO 7.2]**
- Placing a semicolon after the control expression in a **while** or **for** statement is not a syntax error but it is most likely a logic error. **[LO 7.3]**
- Using commas rather than semicolon in the header of a **for** statement is an error. **[LO 7.3]**
- Do not change the control variable in both the **for** statement and the body of the loop. It is a logic error. **[LO 7.3]**
- Although it is legally allowed to place the initialization, testing, and increment sections outside the header of a **for** statement, avoid them as far as possible. **[LO 7.3]**
- Although it is permissible to use arithmetic expressions in initialization and increment section, be aware of round off and truncation errors during their evaluation. **[LO 7.3]**
- Although statements preceding a **for** and statements in the body can be placed in the **for** header, avoid doing so as it makes the program more difficult to read. **[LO 7.3]**
- The use of **break** and **continue** statements in any of the loops is considered unstructured programming. Try to eliminate the use of

these jump statements, as far as possible. **[LO 7.4]**

- Avoid the use of **goto** anywhere in the program. **[LO 7.4]**
- Use the function **exit()** only when breaking out of a program is necessary. **[LO 7.4]**

BRIEF CASES

1. Table of Binomial Coefficients [LO 7.1, 7.2 M]

Problem: Binomial coefficients are used in the study of binomial distributions and reliability of multicomponent redundant systems. It is given by

$$B(m,x) = \binom{m}{x} = \frac{m!}{x!(m-x)!},\ m \geq x$$

A table of binomial coefficients is required to determine the binomial coefficients for any set of m and x.

Problem Analysis **:** The binomial coefficient can be recursively calculated as follows:

$$B(m, 0) = 1$$

$$B(m, x) = B(m,x-1)\left[\frac{m-x+1}{x}\right],\ x = 1,2,3,\ldots,m$$

Further,

$$B(0, 0) = 1$$

That is, the binomial coefficient is one when either x is zero or m is zero. The program in Fig. 7.16 prints the table of binomial coefficients for m = 10. The program employs one **do** loop and one **while** loop.

Program

```
#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;
          }
          printf("\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. 7.16 Program to print **binomial coefficient** table

2. Histogram [LO 7.3 M]

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. 7.17 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

```
#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. 7.17 Program to draw a **histogram**

3. Minimum Cost **[LO 7.3, 7.4 M]**

Problem: The cost of operation of a unit consists of two components C1 and C2 which can be expressed as functions of a parameter p as follows:

$$C1 = 30 - 8p$$

$$C2 = 10 + p^2$$

The parameter p ranges from 0 to 10. Determine the value of p with an accuracy of + 0.1 where the cost of operation would be minimum.

Problem Analysis:

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. 7.18 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
            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 A p = 4.0
```

Fig. 7.18 Program of minimum cost problem

4. Plotting of Two Functions [LO 7.3, 7.4 H]

***Problem* :** We have two functions of the type

$$y1 = \exp(-ax)$$

$$y2 = \exp(-ax^2/2)$$

Plot the graphs of these functions for x varying from 0 to 5.0.

Problem Analysis : Initially when x = 0, y1 = y2 =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. 7.19 plots these two functions simultaneously. (0 for y1, * for y2, and # for the common point).

Program

```
#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; i <= 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
                                  Y ------->
  0 --------------------------------------------------------------------------
    |                                                                      #
    |                                                              0 --- *
    |                                                         0------ *
    |                                                    0-------- *
    |                                               0------ *
    |                                          0------ *
    |                                       0---- *
    |                                     0- *
    |                                   #
    |                              * -0
 X  |                         *--- 0
    |                      *----- 0
    |                   * ------ 0
    |                *--------0
    |             *-------- 0
    |           *-------0
    |         * --------0
    |  *-------0
    |  *------0
    |  *-----0
    |
```

Fig. 7.19 Plotting of two functions

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

7.1 Which of the following lopping statements end with a semi-colon? **[LO 7.2 E]**

(a) while

(b) do while

(c) for

(d) None of these

7.2 Which of the following control structures are used in a structured programming approach? **[LO 7.4 E]**

(a) Sequence

(b) Selection

(c) Repetition

(d) All of these

7.3 A typical repetition control structure comprises which of the following parts? **[LO 7.1 E]**

(a) Initialisation of condition variable

(b) Execution block

(c) Test condition and counter update instruction

(d) All of these

7.4 Which of the following statements can be used to immediately exit from the program? **[LO 7.4 M]**

(a) exit()

(b) break

(c) goto

(d) Both 1 and 3

7.5 Which of the following statements skip the execution of the remaining part of the loop? **[LO 7.4 H]**

(a) break

(b) continue

(c) both 1 and 2

(d) None of these

REVIEW QUESTIONS

7.1 State whether the following statements are *true* or *false* .

(a) In a pretest loop, if the body is executed **n** times, the test expression is executed **n** + 1 times. **[LO 7.1 M]**

(b) The number of times a control variable is updated always equals the number of loop iterations. **[LO 7.1 H]**

(c) The **do…while** statement first executes the loop body and then evaluate the loop control expression. **[LO 7.2 E]**

(d) An exit-controlled loop is executed a minimum of one time. **[LO 7.2 E]**

(e) The three loop expressions used in a **for** loop header must be separated by commas. **[LO 7.3 E]**

(f) **while** loops can be used to replace **for** loops without any change in the body of the loop. **[LO 7.3 M]**

(g) Both the pretest loops include initialization within the statement. **[LO 7.3 M]**

(h) In a **for** loop expression, the starting value of the control variable must be less than its ending value. **[LO 7.3 M]**

(i) The initialization, test condition and increment parts may be missing in a **for** statement. **[LO 7.3 H]**

(j) The use of **continue** statement is considered as unstructured programming. **[LO 7.4 M]**

7.2 Fill in the blanks in the following statements.

(a) The sentinel-controlled loop is also known as _______ loop. **[LO 7.1 E]**

(b) In a counter-controlled loop, variable known as _____ is used to count the loop operations. **[LO 7.1 E]**

(c) In an exit-controlled loop, if the body is executed n times, the test condition is evaluated ________times. **[LO 7.2 M]**

(d) A **for** loop with the no test condition is known as ______ loop. **[LO 7.3 M]**

(e) The _________statement is used to skip a part of the statements in a loop. **[LO 7.4 E]**

(f) The recurring sequence set up by a for loop can be exactly replicated by using a __________ loop instead. **[LO 7.3 M]**

(g) The ________ statement is used to exit the loop immediately and continue with the statement immediately following the loop. **[LO 7.4 M]**

7.3 Can we change the value of the control variable in **for** statements? If yes, explain its consequences. **[LO 7.3 E]**

7.4 What is a null statement? Explain a typical use of it. **[LO 7.1 M]**

7.5 Use of **goto** should be avoided. Explain a typical example where we find the application of **goto** becomes necessary. **[LO 7.4 H]**

7.6 How would you decide the use of one of the three loops in C for a given problem? **[LO 7.3 H]**

7.7 How can we use **for** loops when the number of iterations are not known? **[LO 7.3 M]**

7.8 Explain the operation of each of the following **for** loops. **[LO 7.3 M]**

```
(a) for ( n = 1; n != 10; n += 2)
sum = sum + n;
(b) for (n = 5; n <= m; n -=1)
sum = sum + n;
(c) for (n = 1; n <= 5;)
sum = sum + n;
(d) for ( n = 1; ; n = n + 1)
sum = sum + n;
(e) for (n = 1; n < 5; n ++)
n = n -1
```

7.9 What would be the output of each of the following code segments?

(a) count = 5; **[LO 7.1 M]**

```
while (count -- > 0)
printf(count);
```

(b) count = 5; **[LO 7.1 M]**

while (-- count > 0)

printf(count);

(c) count = 5; **[LO 7.2 E]**

do printf(count);

while (count > 0);

(d) for (m = 10; m > 7, m -=2) **[LO 7.3 M]**

printf(m);

7.10 Compare, in terms of their functions, the following pairs of statements:

(a) while and do...while **[LO 7.2 E]**

(b) while and for **[LO 7.3 E]**

(c) break and continue **[LO 7.4 E]**

(d) break and goto **[LO 7.4 M]**

(e) continue and goto **[LO 7.4 M]**

7.11 Analyse each of the program segments that follow and determine how many times the body of each loop will be executed.

(a) [LO 7.1 E]

```
x = 5;
y = 50;
while (x <= y)
{
  x = y/x;
  -----
  -----
}
```

(b) [LO 7.1 M]

```
int m = 10;
int n = 7;
while ( m % n >= 0)
{
  ---
  m = m + 1;
  n = n + 2;
  ---
}
```

(c) [LO 7.2 E]

```
m = 1;
do
{
  -----
  -----
  m = m+2;
}
while (m < 10);
```

(d) [LO 7.3 M]

```
int i;
for (i = 0; i <= 5; i = i+2/3)
{
  -----
  -----
  -----
}
```

7.12 Write a **for** statement to print each of the following sequences of integers: **[LO 7.3 H]**

(a) 1, 2, 4, 8, 16, 32

(b) 1, 3, 9, 27, 81, 243

(c) – 4, –2, 0, 2, 4

(d) –10, –12, –14, –18, –26, – 42

7.13 Change the following **for** loops to **while** loops: **[LO 7.3 M]**

(a) for (m = 1; m < 10; m = m + 1)

printf(m);

(b) for (; scanf("%d", & m) != -1;)

printf(m);

7.14 Change the **for** loops in Exercise 7.13 to **do** loops. **[LO 7.3 M]**

7.15 What is the output of following code? **[LO 7.1 M]**

```
int m = 100, n = 0;
while ( n == 0 )
{
            if ( m < 10 )
                break;
             m = m-10;
```

7.16 What is the output of the following code? **[LO 7.2 E]**

```
int m = 0 ;
do
{
     if (m > 10 )
            continue ;
  m = m + 10 ;
} while ( m < 50 ) ;
printf("%d", m);
```

7.17 What is the output of the following code? **[LO 7.2 E]**

```
int n = 0, m = 1 ;
do
{
     printf(m) ;
     m++ ;
}
while (m <= n) ;
```

7.18 What is the output of the following code? **[LO 7.3 M]**

```
int n = 0, m ;
for (m = 1; m <= n + 1 ; m++ )
     printf(m);
```

7.19 When do we use the following statement? **[LO 7.3 H]**

```
for (; ; )
```

DEBUGGING EXERCISES

7.1 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);                                    [LO 7.1 M]
    {
            count = 1;
            sum = sum + x;
            count = count + 1;
    }
(b) name = 0;                                               [LO 7.2 M]
    do { name = name + 1;
    printf("My name is John\n");}
    while (name = 1)
(c) do;                                                     [LO 7.3 E]
    total = total + value;
    scanf("%f", &value);
    while (value != 999);
(d) for (x = 1, x > 10; x = x + 1)                          [LO 7.3 E]
    {
                -----
                _____
                _____
    }
(e) m = 1;                                                  [LO 7.3 E]
    n = 0;
    for ( ; m+n < 10; ++n);
    printf("Hello\n");
    m = m+10
(f) for (p = 10; p > 0;)                                    [LO 7.3 M]
    p = p - 1;
    printf("%f", p);
```

PROGRAMMING EXERCISES

7.1 Given a number, write a program using **while** loop to reverse the digits of the number. For example, the number **[LO 7.1 H]**

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.)

7.2 Write a program to compute the sum of the digits of a given integer number. **[LO 7.1 M]**

7.3 The numbers in the sequence **[LO 7.2 H]**

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.)

7.4 Write a program to evaluate the following investment equation **[LO 7.3 M]**

$V = P(1+r)^{n}$

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. This 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.)

7.5 Write programs to print the following outputs using **for** loops. **[LO 7.3 H]**

```
(a) 1              (b) * * * * * *
    2 2                    * * * *
    3 3 3                    * * *
    4 4 4 4                    * *
    5 5 5 5 5                    *
```

7.6 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. **7.3, 7.4 M]** **[LO**

7.7 Rewrite the program of case study 7.4 (plotting of two curves) using **else...if** constructs instead of **continue** statements. **[LO 7.9 H]**

7.8 Write a program to print a table of values of the function **[LO 7.3 M]** 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					

7.9 Write a program using **for** and **if** statement to display the capital letter S in a grid of 15 rows and 18 columns as shown below. **[LO 7.3 H]**

```
* * * * * * * * * * * * * * * * * * * * * * * *
* * - - - - - - - - - - - - - - - - - - - - - - * *
* * * * * * * * * - - - - - - - - - - - - - - * *
* * * *
* * * *
* * * *
* * * * * - - - - - - - - - - - - - - - - * * * *
- - - - - - - - - - - - - - - - - - - - - - * * * *
- - - - - - - - - - - - - - - - - - - - - * * * *
- - - - - - - - - - - - - - - - - - - - * * * *
- - - - - - - - - - - - - - - - - - - - * * * *
- - - - - - - - - - - - - - - - - - - - * * * *
* * * * - - - - - - - - — - - - - - - - * * * *
* * * - - - - - - - - - - - - - - - - - * * * *
* * - - - - - - - - - - - - - - - - - - - * * * *
```

7.10 Write a program to compute the value of Euler's number e, that is used as the base of natural logarithms. Use the following formula. **[LO 7.3 H]**

e = 1 + 1/1! + 1 /2! + 1 /3! + ... + 1 /n!

Use a suitable loop construct. The loop must terminate when the difference between two successive values of e is less than 0.00001.

7.11 Write programs to evaluate the following functions to 0.0001% accuracy. **[LO 7.3 H]**

(a) $\sin x = x - x^3/3! + x^5/5! - x^7/7! + \ldots\ldots$

(b) $\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + \ldots\ldots$

(c) $SUM = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \ldots\ldots$

7.12 The present value (popularly known as book value) of an item is given by the relationship.

$$P = c\,(1-d)^n$$ **[LO 7.3 H]**

where c = original cost
d = rate of depreciation (per year)
n = number of years
p = present value after y years.

If P is considered the scrap value at the end of useful life of the item, write a program to compute the useful life in years given the original cost, depreciation rate, and the scrap value.

The program should request the user to input the data interactively.

7.13 Write a program to print a square of size 5 by using the character **S** as shown below: **[LO 7.3 M]**

```
(a) S  S  S  S  S          (b) S  S  S  S  S
    S  S  S  S  S              S           S
    S  S  S  S  S              S           S
    S  S  S  S  S              S           S
    S  S  S  S  S              S  S  S  S  S
```

7.14 Write a program to graph the function **[LO 7.3 M]**

y = sin (x)

in the interval 0 to 180 degrees in steps of 15 degrees. Use the concepts discussed in the Case Study 4 in Chapter 7.

7.15 Modify the program of Exercise 7.16 to print the character O instead of S at the center of the square as shown below.

[LO 7.3 M]

```
S S S S S
S S S S S
S S O S S
S S S S S
S S S S S
```

7.16 Given a set of 10 two-digit integers containing both positive and negative values, write a program using **for** loop to compute the sum of all positive values and print the sum and the number of values added. The program should use **scanf** to read the values and terminate when the sum exceeds 999. Do not use **goto** statement. **[LO 7.3 H]**

7.17 Write a C program to display a coloured line. **[LO 7.1 M]**

7.18 Write a C program to display the following pattern. **[LO 7.3 M]**

```
    1
   A B
  2 3 4
 C D E F
5 6 7 8 9
```

7.19 Write a C program to display the following pattern: **[LO 7.3 H]**

```
    1
  1 2 1
1 2 3 2 1
  1 2 1
    1
```

7.20 Write a C program to display the following pattern: **[LO 7.3 H]**

```
1
0 1
1 0 1
0 1 0 1
```

INTERVIEW QUESTIONS

7.1 What is the value of variable a after the following code is executed? **[LO 7.3 M]**

```
int a;
for(a=0;a<10;a++)
{
}
```

7.2 What is the final value of x after the following code is executed, **[LO 7.3 M]**

```
for(x=9; x>=0; x--)
{
}
```

7.3 What is the minimum number of times a do-while loop is guaranteed to execute? **[LO 7.2 E]**

7.4 What will be the value of x and y after following statements are executed?

```
int x = 10,y = 20;
x = x++ + ++y;
y = ++x + y++;
```

Explain the logic while calculating the answer. **[LO 7.1 H]**

7.5 What will be the output of the following code:

```
i=0;
while(i++<5)
{
if(i==0)
printf("Zero\n");
else
printf("Hello World\n");
}
```

[LO 7.1 H]

7.6 What is the purpose of exit statement? **[LO 7.4 E]**

7.7 Is it possible to print integer values from 1 to 100 without using any looping construct? **[LO 7.1 H]**

7.8 Can you give examples of pre-test and post-test loops? **[LO 7.1 E]**

E for Easy, **M** for Medium and **H** for High

Chapter 8

Array

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

- **LO 8.1** Define the concept of arrays
- **LO 8.2** Determine how one-dimensional array is declared and initialized
- **LO 8.3** Know the concept of two-dimensional arrays
- **LO 8.4** Discuss how two-dimensional array is declared and initialized
- **LO 8.5** Describe multi-dimensional arrays
- **LO 8.6** Explain dynamic arrays

INTRODUCTION

So far we have used only the fundamental data types, namely **char, int, float, double** and variations of **int** and **double** . Although these types are very useful, they are constrained by the fact that a variable of these types can store only one value at any given time. Therefore, they can be used only to handle limited amounts of data. In many applications, however, we need to handle a large volume of data in terms of reading, processing and printing. To process such large amounts of data, we need a powerful data type that would facilitate efficient storing, accessing and manipulation of data items. C supports a derived data type known as *array* that can be used for such applications.

An array is a *fixed-size* sequenced collection of elements of the same data type. It is simply a grouping of like-type data. In its simplest form, an array can be used to represent a list of numbers, or a list of names. Some examples where the concept of an array can be used:

- List of temperatures recorded every hour in a day, or a month, or a year.

- List of employees in an organization.
- List of products and their cost sold by a store.
- Test scores of a class of students.
- List of customers and their telephone numbers.
- Table of daily rainfall data.

and so on.

Since an array provides a convenient structure for representing data, it is classified as one of the *data structures* in C. Other data structures include structures, lists, queues and trees. A complete discussion of all data structures is beyond the scope of this text. However, we shall consider structures in Chapter 10 and lists in Chapter 13.

As we mentioned earlier, an array is a sequenced collection of related data items that share a common name. For instance, we can use an array name *salary* to represent a *set of salaries* of a group of employees in an organization. We can refer to the individual salaries by writing a number called *index* or *subscript* in brackets after the array name. For example,

salary [10]

represents the salary of 10 th employee. While the complete set of values is referred to as an array, individual values are called *elements* .

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, we can use a loop construct, discussed earlier, with the subscript as the control variable to read the entire array, perform calculations, and print out the results.

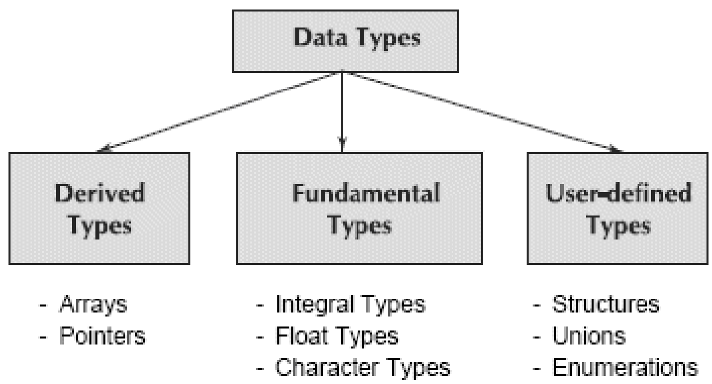
We can use arrays to represent not only simple lists of values but also tables of data in two, three or more dimensions. In this chapter, we introduce the concept of an array and discuss how to use it to create and apply the following types of arrays.

- One-dimensional arrays
- Two-dimensional arrays

- Multidimensional arrays

Data Structures

C supports a rich set of derived and user-defined data types in addition to a variety of fundamental types as shown below:



Arrays and structures are referred to as *structured data types* because they can be used to represent data values that have a structure of some sort. Structured data types provide an organizational scheme that shows the relationships among the individual elements and facilitate efficient data manipulations. In programming parlance, such data types are known as *data structures*.

In addition to arrays and structures, C supports creation and manipulation of the following data structures:

- Linked Lists
- Stacks
- Queues
- Trees
- Graphs

ONE-DIMENSIONAL ARRAYS

LO 8.1
Define the concept of arrays

A list of items may be given one variable name, and the individual elements may be accessed by the specification of their relative positions with respect to the start of the list. Such relative positions are referred to as the index or subscript of the element. Such a list in which the elements are accessible by the variable name assigned to the list and its subscript is known as a *one-dimensional array* . For instance, we use the equation

$$A = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

to calculate the average of n values of x . The subscripted variable x_i refers to the i^{th} element of x . In *C* single-subscripted variable x_i can be expressed as x]. Notice that the subscript of the first element, that is, x [0] is 0. For[i 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 **number** as follows:

int number[5];

and the computer reserves five storage locations as shown below:

	number [0]
	number [1]
	number [2]
	number [3]
	number [4]

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 subscripts 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.*

TIP

In C array subscripts begin at 0.

DECLARATION OF ONE-DIMENSIONAL ARRAYS

Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in memory. The general form of array declaration is

LO 8.2
Determine how one-dimensional array is declared and initialized

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 size should be either a numeric constant or a symbolic constant.

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’
‘\0’

When the compiler sees a character string, it terminates it with an additional null character. Thus, the element **name[10]** holds the null character ‘\0’. *When declaring character arrays, we must allow one extra element space for the null terminator .*

WORKED-OUT PROBLEM 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 x1,x2,....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.

```
Program
            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

Note *C99 permits arrays whose size can be specified at run time. See Appendix "C99 Features".*

INITIALIZATION OF ONE-DIMENSIONAL ARRAYS

After an array is declared, its elements must be initialized. Otherwise, they will contain "garbage". An array can be initialized at either of the following stages:

- At compile time
- At run time

Compile Time Initialization

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:

```
type array-name[size] = { list of values };
```

The values in the list are separated by commas. For example, the statement

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,

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.

The *size* may be omitted. In such cases, the compiler allocates enough space for all initialized elements. For example, the statement

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

char name[ ] = {'J','o', 'h', 'n', '\0'};

declares the **name** to be an array of five characters, initialized with the string "John" ending with the null character. Alternatively, we can assign the string literal directly as under:

char name [ ] = "John";

(Character arrays and strings are discussed in detail in Chapter 8.)

Compile time initialization may be partial. That is, the number of initializers may be less than the declared size. In such cases, the remaining elements are inilialized to *zero* , if the array type is numeric and *NULL* if the type is char. For example,

int number [5] = {10, 20};

will initialize the first two elements to 10 and 20 respectively, and the remaining elements to 0. Similarly, the declaration.

char city [5] = {'B'};

will initialize the first element to 'B' and the remaining four to NULL. It is a good idea, however, to declare the size explicitly, as it allows the compiler to do some error checking.

Remember, however, if we have more initializers than the declared size, the compiler will produce an error. That is, the statement

int number [3] = {10, 20, 30, 40};

will not work. It is illegal in C.

Run Time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. For example, consider the following segment of a C program.

```
--------
--------
for (i = 0; i < 100; i = i+1)
{
   if   i < 50
        sum[i] = 0.0;          /* assignment statement */
   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 at run time.

We can also use a read function such as **scanf** to initialize an array. For example, the statements

```
int x [3];
scanf("%d%d%d", &x[0], &[1], &x[2]);
```

will initialize array elements with the values entered through the keyboard , at runtime.

WORKED-OUT PROBLEM 8.2

M

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 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.

Program

```
#define     MAXVAL     50
#define     COUNTER    11
main()
{
     float       value[MAXVAL];
     int            i, low, high;
     int   group[COUNTER] = {0,0,0,0,0,0,0,0,0,0,0};
```

```
    /* . . . . . . . .READING AND COUNTING . . . . . .*/
    for( i - 0 ; i < MAXVAL ; i++ )
    {
    /*. . . . . . . .READING OF VALUES . . . . . . . . */
    scanf("%f", &value[i]) ;
    /*. . . . . .COUNTING FREQUENCY OF GROUPS. . . . . */
      ++ group[ (int) ( value[i]) / 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 65 56 76 67 (Input data)
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**

Note that we have used an initialization statement.

int group [COUNTER] = {0,0,0,0,0,0,0,0,0,0,0};

which can be replaced by

int group [COUNTER] = {0};

This will initialize all the elements to zero.

WORKED-OUT PROBLEM 8.3

H

The program shown in Fig. 8.3 shows the algorithm, flowchart and the complete C program to find the two's compliment of a binary number.

Problem Analysis

The 2's complement of a binary number is defined as the 1's complement of the number plus 1. Consider the following examples:

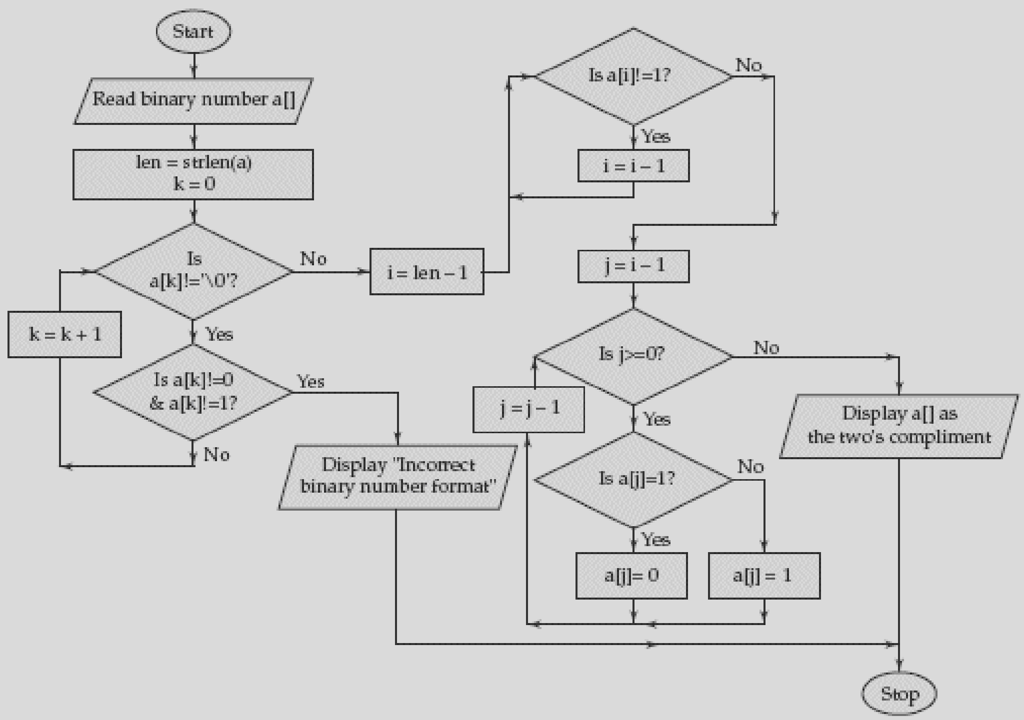
Number 1's Complement 2's Complement

Number	1's Complement	2's Complement
(11010101)	(00101010)	(00101011)
(11111111)	(00000000)	(00000001)
(10011000)	(01100111)	(01101000)
(00000000)	(11111111)	(00000000)

```
Algorithm
Step 1 – Start
Step 2 – Read a binary number string (a[])
Step 3 – Calculate the length of string str (len)
Step 4 – Initialize the looping counter k=0
Step 5 – Repeat Step 6 while a[k] != '\0'
Step 6 –        If a[k]!= 0 AND a[k]!= 1
                        Display error "Incorrect binary number format" and terminate the
                                                                            program
                else
                        k = k + 1
                End If
        End While
Step 7 – Initialize the looping counter i = len - 1
Step 8 – Repeat Step 9 while a[i]!='1'
Step 9 –        i = i – 1
        End While
Step 10 – Initialize the looping counter j = i - 1
Step 11 – Repeat Step 14-15 while j >= 0
Step 14 –       If a[j]=1
                        a[j]='0'
                else
```

```
                    a[j]='1'
               End  If
Step 15 –      j = j - 1
          End While
Step 16 – Display a[] as the two's compliment
Step 17 – Stop
```

Flowchart



Program

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
void main()
{
        char a[16];
        int i,j,k,len;

        printf("Enter a binary number: ");
        gets(a);
        len=strlen(a);
```

```
                    for(k=0;a[k]!='\0'; k++)
                    {
                            if (a[k]!='0' && a[k]!='1')
                            {
                                    printf("\nIncorrect binary number format...the
                                                                    program will quit");
                                    exit(0);
                            }
                    }
                    for(i=len-1;a[i]!='1'; i--) ;
                    for(j=i-1;j>=0;j--)
                    {
                            if(a[j]=='1')
                                    a[j]='0';
                            else
                                    a[j]='1';
                    }
                    printf("\n2's compliment = %s",a);
            }
Output
   Enter a binary number: 01011001001
   2's compliment = 10100110111
```

Fig. 8.3 Algorithm, flowchart and C program to find two's compliment of a binary number

TEST YOUR SKILLS

Write a C program to input an array of 10 integers and find the maximum and minimum number in the array. Also, calculate the average of all values in the array. **[E]**

TWO-DIMENSIONAL ARRAYS

> **LO 8.3**
> Know the concept of two-dimensional arrays

So far we have discussed the array variables that can store a list of values. There could be situations where a table of values will have to be stored. Consider the following data table, which shows the value of sales of three items by four sales girls:

	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.

In mathematics, we represent a particular value in a matrix by using two subscripts such as $\mathbf{v}_{\mathbf{ij}}$. Here **v** denotes the entire matrix and $\mathbf{v}_{\mathbf{ij}}$ refers to the value in the i th row and j th 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. 8.4 . 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.

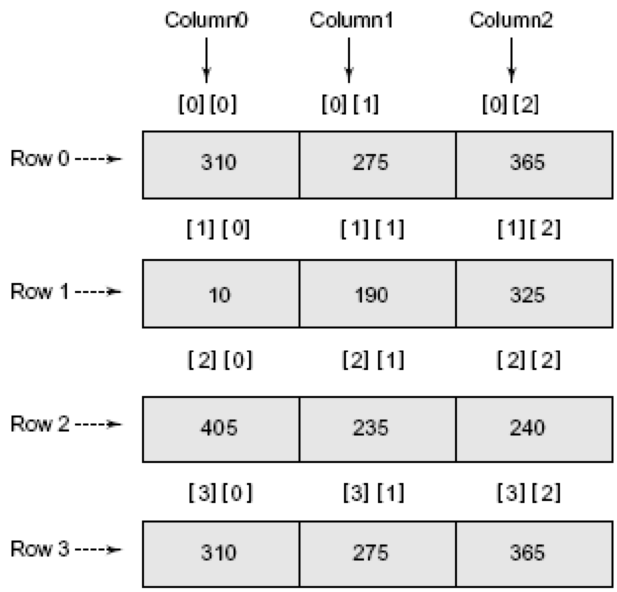


Fig. 8.4 Representation of a **two-dimensional** array in memory

WORKED-OUT PROBLEM 8.4

L

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

```
Program
        #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 ; i < 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

WORKED-OUT PROBLEM 8.5

H

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. 8.6 . 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:

(a) Total sales by m^{th} girl = $\sum_{j=0}^{2}$ value [m][j] (girl_total[m])

(b) Total value of n^{th} item = $\sum_{i=0}^{3}$ value [i][n] (item_total[n])

(c) Grand total = $\sum_{i=0}^{3} \sum_{j=0}^{2}$ value[i][j]

= $\sum_{i=0}^{3}$ girl_total[i]

= $\sum_{j=0}^{2}$ item_total[j]

Program

```
#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. 8.6 Illustration of two-dimensional arrays

INITIALIZING TWO-DIMENSIONAL ARRAYS

A 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,

LO 8.4
Discuss how two-dimensional array is declared and initialized

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

int table[2][3] = {{0,0,0}, {1,1,1}};

by surrounding the elements of the each row by braces.

We can also initialize a two-dimensional array in the form of a matrix as shown below:

```
int table [ ] [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.

When the array is completely initialized with all values, explicitly, we need not specify the size of the first dimension. That is, the statement

```
int table [ ] [3] = {
                    { 0, 0, 0},
                    { 1, 1, 1}
                  };
```

is permitted.

If the values are missing in an initializer, they are automatically set to zero. For instance, the statement

```
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.

```
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. The following statement will also achieve the same result:

```
int m [3] [5] = { 0, 0};
```

WORKED-OUT PROBLEM 8.6

M

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 coded 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. 8.7 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.

Program

```
main()
{
   int i, j, car;
   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

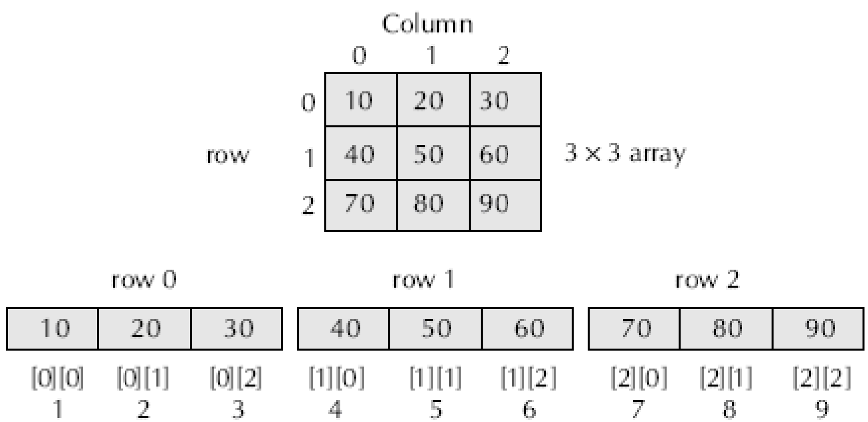
```
For each person, enter the city code
followed by the car code.
Enter the letter X to indicate end.
M 1  C  2  B  1  D  3  M  2  B  4
C 1  D  3  M  4  B  2  D  1  C  3
D 4  D  4  M  1  M  1  B  3  B  3
C 1  C  1  C  2  M  4  M  4  C  2
D 1  C  2  B  3  M  1  B  1  C  2
D 3  M  4  C  1  D  2  M  3  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. 8.7 Program to tabulate a survey data

> **TIP**
> Added care should be taken to select an appropriate array size.

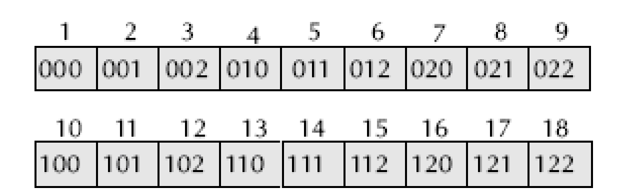
The subscripts in the definition of a two-dimensional array represent rows and columns. This format maps the way that data elements are laid out in the memory. The elements of all arrays are stored contiguously in increasing memory locations, essentially in a single list. If we consider the memory as a row of bytes, with the lowest address on the left and the highest address on the right, a simple array will be stored in memory with the first element at the left end and the last element at the right end. Similarly, a two-dimensional array is stored “row-wise, starting from the first row and ending with the last row, treating each row like a simple array. This is illustrated below.



Memory Layout

F or a multi-dimensional array, the order of storage is that the first element stored has 0 in all its subscripts, the second has all of its subscripts 0 except the far right which has a value of 1 and so on.

The elements of a 2 x 3 x 3 array will be stored as und er



The far right subscript increments first and the other subscripts increment in order from right to left. The sequence numbers 1, 2,……, 18 represents the location of that element in the list.

WORKED-OUT PROBLEM 8.7

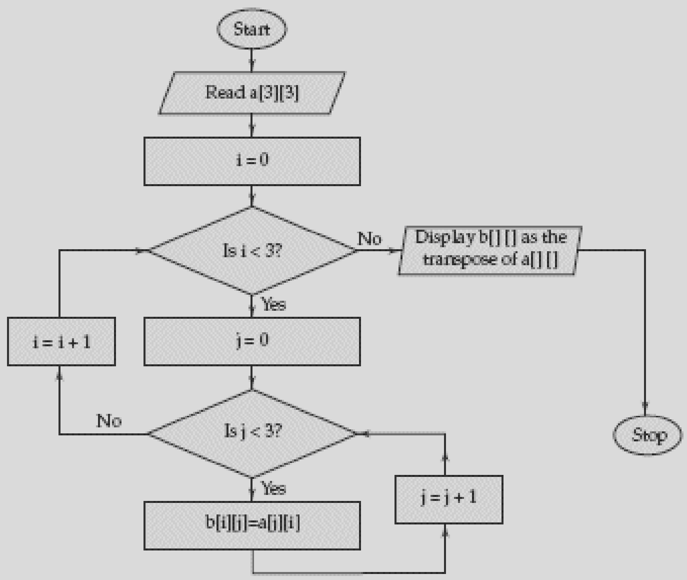
E

The program in Fig. 8.8 shows how to find the transpose of a matrix.

Algorithm

```
Step 1 – Start
Step 2 – Read a 3 X 3 matrix (a[3][3])
Step 3 – Initialize the looping counter i = 0
Step 4 – Repeat Steps 5-9 while i<3
Step 5 – Initialize the looping counter j = 0
Step 6 – Repeat Steps 7-8 while j<3
Step 7 – b[i][j]=a[j][i]
Step 8 – j = j + 1
Step 9 – i = i + 1
Step 10 – Display b[][] as the transpose of the matrix a[][]
Step 11 – Stop
```

Flowchart



Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
int i,j,a[3][3],b[3][3];
clrscr();
printf("Enter a 3 X 3 matrix:\n");
for(i=0;i<3;i++)
{
  for(j=0;j<3;j++)
```

```
        {
        printf("a[%d][%d] = ",i,j);
        scanf("%d",&a[i][j]);
        }
    }
    printf("\nThe entered matrix
    is: \n");
    for(i=0;i<3;i++)
    {
        printf("\n");
        for(j=0;j<3;j++)
        {
        printf("%d\t",a[i][j]);
        }
    }
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        b[i][j]=a[j][i];
    }
    printf("\n\nThe transpose of the matrix is: \n");

    for(i=0;i<3;i++)
    {
        printf("\n");
        for(j=0;j<3;j++)
        {
        printf("%d\t",b[i][j]);
        }
        }
        getch();
        }
```

Output

```
    Enter a 3 X 3 matrix:
    a[0][0] = 1
    a[0][1] = 2
    a[0][2] = 3
    a[1][0] = 4
    a[1][1] = 5
    a[1][2] = 6
    a[2][0] = 7
```

```
a[2][1] = 8
a[2][2] = 9
The entered matrix is:
   1    2    3
   4    5    6
   7    8    9
The transpose of the matrix is:
   1    4    7
   2    5    8
   3    6    9
```

Fig. 8.8 Program to find transpose of a matrix

WORKED-OUT PROBLEM 8.8

M

The program in Fig. 8.9 shows how to multiply the elements of two N × N matrices.

Program

```
#include<stdio.h>
void main()
{
int a1[10][10],a2[10][10],c[10][10],i,j,k,a,b;
printf("Enter the size of the square matrix\n");
scanf ("%d", &a);
b=a;
printf("You have to enter the matrix elements in row-wise fashion\n");
for(i=0;i<a;i++)
{
for(j=0;j<b;j++)
{
printf("\nEnter the next element in the 1st matrix=");
scanf("%d",&a1[i][j]);
}
}
for(i=0;i<a;i++)
{
for(j=0;j<b;j++)
{
printf("\n\nEnter the next element in the 2nd matrix=");
scanf("%d",&a2[i][j]);
}
```

```
 }
printf("\n\nEntered matrices are\n");
for(i=0;i<a;i++)
{
printf("\n");
for(j=0;j<b;j++)
printf(" %d ",a1[i][j]);
}
printf("\n");
for(i=0;i<a;i++)
{
printf("\n");
for(j=0;j<b;j++)
printf(" %d ",a2[i][j]);
}
printf("\n\nProduct of the two matrices is\n");
for(i=0;i<a;i++)
for(j=0;j<b;j++)
{
c[i][j]=0;
for(k=0;k<a;k++)
c[i][j]=c[i][j]+a1[i][k]*a2[k][j];
}
for(i=0;i<a;i++)
{
```

```
        {
        printf("\n");
        for(j-0;j<b;j++)
        printf(" %d ",c[i][j]);
        }
        }
    Output
            Enter the size of the square matrix
            2
            You have to enter the matrix elements in row-wise fashion
            Enter the next element in the 1st matrix-1
            Enter the next element in the 1st matrix-0
            Enter the next element in the 1st matrix-2
            Enter the next element in the 1st matrix-3
            Enter the next element in the 2nd matrix-4
            Enter the next element in the 2nd matrix-5
            Enter the next element in the 2nd matrix-0
```

```
            Enter the next element in the 2nd matrix-2
            Entered matrices are
               1  0
               2  3
               4  5
               0  2
            Product of the two matrices is
               4  5
               8  16
```

Fig. 8.9 Program for N × N matrix multiplication

TEST YOUR SKILLS

Write a C program to compute the sum of two 3 × 3 matrices. **[M]**

MULTI-DIMENSIONAL ARRAYS

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multi-dimensional array is

LO 8.5
Describe multi-dimensional arrays

```
type array_name[s1][s2][s3]....[sm];
```

where s_i is the size of the ith 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.

If the first index denotes year, the second city and the third month, then the element **survey[2][3][10]** 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				

ANSI C does not specify any limit for array dimension. However, most compilers permit seven to ten dimensions. Some allow even more.

Dynamic Arrays

A So far, we created arrays at compile time. An array created at compile time by specifying size in the source code has a fixed size and cannot be modified at run time. The process of allocating memory at compile time is known as *static memory allocation* and the arrays that receive static memory allocation are called *static arrays* . This approach works fine as long as we know exactly what our data requirements are.

LO 8.6
Explain dynamic arrays

Consider a situation where we want to use an array that can vary greatly in size. We must guess what will be the largest size ever needed and create the array accordingly. A difficult task in fact! Modern languages like C do not have this limitation. In C it is possible to allocate memory to arrays at run time. This feature is known as *dynamic memory allocation* and the arrays created at run time are called *dynamic* arrays. This effectively postpones the array definition to run time.

Dynamic arrays are created using what are known as *pointer variables* and *memory management functions* **malloc** , **calloc** and **realloc** . These functions are included in the header file **<stdlib.h>.** The concept of dynamic arrays is used in creating and manipulating data structures such as linked lists, stacks and queues. We discuss in detail pointers and pointer variables in Chapter 11 and creating and managing linked lists in Chapter 14.

More About Arrays

What we have discussed in this chapter are the basic concepts of arrays and their applications to a limited extent. There are some more important aspects of application of arrays. They include:

- using printers for accessing arrays;
- passing arrays as function parameters;
- arrays as members of structures;
- using structure type data as array elements;

- arrays as dynamic data structures; and
- manipulating character arrays and strings.

These aspects of arrays are covered later in the following chapters:

Chapter 9 : Strings

Chapter 10 : Functions

Chapter 11 : Structures

Chapter 12 : Pointers

Chapter 14 : Linked Lists

KEY CONCEPTS

- **ARRAY:** Is a fixed-size seq uenced collection part of elements of the same data type. **[LO 8.1]**
- **ONE-DIMENSIONAL ARRAY:** Is a list of items that has one variable name and one subscript to access the items. **[LO 8.1]**
- **STRUCTURED DATA TYPES:** Represent data values that have a structure of some sort. For example, arrays, structures, etc. **[LO 8.1]**
- **SEARCHING:** Is the process of finding the location of the specified element in the list. **[LO 8.2]**
- **SORTING:** Is the process of rearranging elements in the list as per ascending or descending order. **[LO 8.2]**
- **TWO-DIMENSIONAL ARRAY:** Is an array of arrays that has two subscripts for accessing its values. It is used to represent table or matrix data. **[LO 8.3]**
- **MULTI-DIMENSIONAL ARRAY:** Is an array with more than one dimension. Examples of multi-dimensional arrays are two-dimensional array, three-dimensional array and so on. **[LO 8.5]**
- **DYNAMIC ARRAYS:** Are the arrays declared using dynamic memory allocation technique. **[LO 8.6]**
- **DYNAMIC MEMORY ALLOCATION:** Is the process of allocating memory at run time. **[LO 8.6]**

- **STATIC ARRAYS:** Are the arrays declared using static memory allocation technique. **[LO 8.6]**
- **STATIC MEMORY ALLOCATION:** Is the process of allocating memory at compile time. **[LO 8.6]**

ALWAYS REMEMBER

- We need to specify three things, namely, name, type and size, when we declare an array. **[LO 8.1]**
- Use of invalid subscript is one of the common errors. An incorrect or invalid index may cause unexpected results. **[LO 8.1]**
- Always remember that subscripts begin at 0 (not 1) and end at size –1. **[LO 8.2]**
- Defining the size of an array as a symbolic constant makes a program more scalable. **[LO 8.2]**
- Be aware of the difference between the "kth element" and the "element k". The kth element has a subscript k-1, whereas the element k has a subscript of k itself. **[LO 8.2]**
- Do not forget to initialize the elements; otherwise they will contain "garbage". **[LO 8.2]**
- Supplying more initializers in the initializer list is a compile time error. **[LO 8.2]**
- When using expressions for subscripts, make sure that their results do not go outside the permissible range of 0 to size –1. Referring to an element outside the array bounds is an error. **[LO 8.2]**
- When using control structures for looping through an array, use proper relational expressions to eliminate "off-by-one" errors. For example, for an array of size 5, the following **for** statements are wrong:

```
for (i = 1; i < =5; i+ +)
for (i = 0; i < =5; i+ +)
for (i = 0; i = =5; i+ +)
for (i = 0; i < 4; i+ +)
```

[LO 8.2]

- Referring a two-dimensional array element like x[i, j] instead of x[i][j] is a compile time error. **[LO 8.3]**
- Leaving out the subscript reference operator [ ] in an assignment operation is compile time error. **[LO 8.3]**
- When initializing character arrays, provide enough space for the terminating null character. **[LO 8.4]**
- Make sure that the subscript variables have been properly initialized before they are used. **[LO 8.4]**
- During initialization of multi-dimensional arrays, it is an error to omit the array size for any dimension other than the first. **[LO 8.5]**
- While using static arrays, choose the array size in such a way that the memory space is efficiently utilized and there is no overflow condition. **[LO 8.6]**

BRIEF CASES

1. Median of a List of Numbers[LO 8.2, M]

When all the items in a list are arranged in an 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:

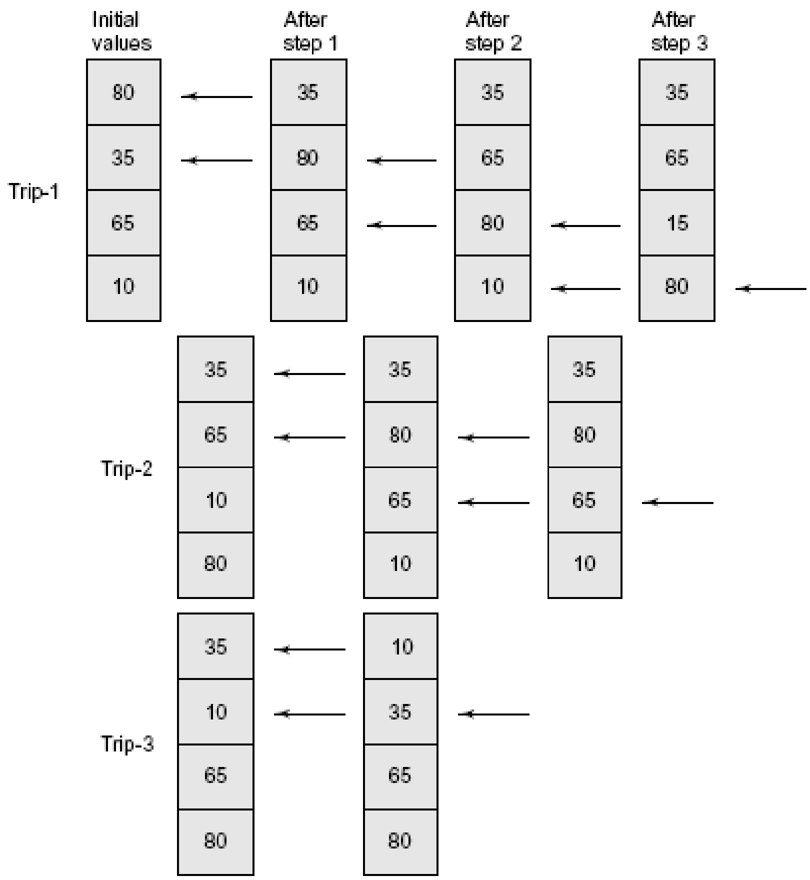
1. Read the items into an array while keeping a count of the items.
2. Sort the items in increasing order.
3. Compute median.

The program and sample output are shown in Fig. 8.10 . 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 algorithm 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

```
#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. 8.10 Program to sort a list of numbers and to determine median

2. Calculation of Standard Deviation **[LO 8.2, M]**

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. 8.11 .

Program

```
#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);
}
```

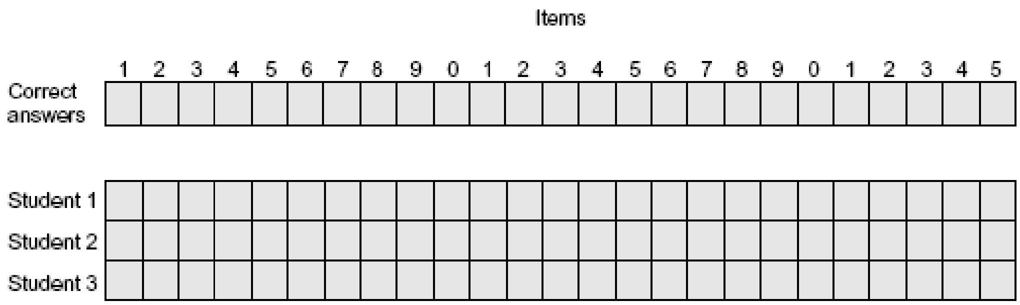
Output

```
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. 8.11 Program to calculate standard deviation

3. Evaluating a Test **[LO 8.2, H]**

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:



The 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. 8.12 . 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

```
#define STUDENTS 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; i < 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 printing 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
        abcddcbaabcdabcdddddddddd
        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. 8.12 Program to evaluate responses to a multiple-choice test

4. Production and Sales Analysis[LO 8.3, 8.4, H]

A company manufactures five categories of products and the number of items manufactured and sold are recorded product-wise 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. 8.13 . 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 in week i

$$= \sum_{J=1}^{5} \text{Svalue[i][j]}$$

Mproduct[j] = Value of jth type product manufactured during the month

$$= \sum_{i=1}^{4} \text{Mvalue[i][j]}$$

Sproduct[j] = Value of jth type product sold during the month

$$= \sum_{i=1}^{4} \text{Svalue[i][j]}$$

Mtotal = Total value of all the products manufactured during the month

$$= \sum_{i=1}^{4} \text{Mweek[i]} = \sum_{j=1}^{5} \text{Mproduct[j]}$$

Stotal = Total value of all the products sold during the month

$$= \sum_{i=1}^{4} \text{Sweek[i]} = \sum_{j=1}^{5} \text{Sproduct[j]}$$

Program

```
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[i] = 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(" GOOD  BYE\n\n");
            break;
        }
            switch(number)
        {  /* Beginning of switch */
            /* VALUE  MATRICES */
                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;
    /* WEEKLY  ANALYSIS */
        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;
    /* PRODUCT  WISE  ANALYSIS */
        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;
    /* GRAND  TOTALS */
        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   SALE
  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
GOOD  BYE
Exit from the program
```

Fig. 8.13 Program for production and sales analysis

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

8.1 How many integer values can the array N[5][5] store? **[LO 8.3 E]**

(a) 10

(b) 25

(c) 16

(d) 24

8.2 What index value should be used to access the last element of the integer array N[10]? **[LO 8.1 E]**

(a) 10

(b) 9

(c) 11

(d) None of the above

8.3 The process of creating a fixed-sized array by allocating memory space at compile time is called ____________. **[LO 8.6 E]**

(a) Static memory allocation

(b) Dynamic memory allocation

(c) Run time memory allocation

(d) None of the above

8.4 An array N[3] = {1, 2, 3} has been declared and initialized in a C program. What will be the output of the following statement:

printf("%d",N[3]); **[LO 8.2 H]**

(a) 3

(b) 2

(c) 4

(d) Garbage value

8.5 It is required to store the names of five students. Which of the following array declaration statements can be used for this? **[LO 8.3 M]**

(a) char student [4][30];

(b) char student [30][4];

(c) char student [5][30];

(d) char student [30][5];

8.6 An array ARR[5][20] has been declared. What will be the output of the following statement?

printf("%d",sizeof(ARR)); **[LO 8.4 M]**

(a) 100

(b) 200

(c) 400

(d) 800

REVIEW QUESTIONS

8.1 State whether the following statements are *true* or *false* .

(a) An array can store infinite data of similar type. **[LO 8.1 E]**

(b) In declaring an array, the array size can be a constant or variable or an expression. **[LO 8.2 E]**

(c) The declaration int x[2] = {1,2,3}; is illegal. **[LO 8.2 E]**

(d) When an array is declared, C automatically initializes its elements to zero. **[LO 8.2, 8.5 M]**

(e) An expression that evaluates to an integral value may be used as a subscript. **[LO 8.1, 8.2 M]**

(f) In C, by default, the first subscript is zero. **[LO 8.1, 8.2 M]**

(g) When initializing a multidimensional array, not specifying all its dimensions is an error. **[LO 8.1, 8.2 M]**

(h) When we use expressions as a subscript, its result should be always greater than zero. **[LO 8.1, 8.2 M]**

(i) In C, we can use a maximum of 4 dimensions for an array. **[LO 8.1, 8.2 M]**

(j) Accessing an array outside its range is a compile time error. **[LO 8.2 H]**

(k) A **char** type variable cannot be used as a subscript in an array. **[LO 8.2 H]**

(l) An unsigned long int type can be used as a subscript in an array. **[LO 8.2 H]**

8.2 Fill in the blanks in the following statements.

(a) The variable used as a subscript in an array is popularly known as __________ variable. **[LO 8.1 E]**

(b) An array that uses more than two subscript is referred to as ______ array. **[LO 8.2 E]**

(c) _______ is the process of arranging the elements of an array in order. **[LO 8.3 E]**

(d) An array can be initialized either at compile time or at ________. **[LO 8.6 M]**

(e) An array created using **malloc** function at run time is referred to as ______ array. **[LO 8.6 M]**

(f) If we try to access an array element without initializing the array, then it returns ________ value. **[LO 8.2 M]**

(g) To store a string in a character array, we must provide space for one extra element, which is _______. **[LO 8.2 E]**

8.3 Write a **for** loop statement that initializes all the diagonal elements of an array to one and others to zero as shown below. Assume 5 rows and 5 columns. **[LO 8.4 M]**

1	0	0	0	0	.....	0
0	1	0	0	0	.....	0
0	0	1	0	0	.....	0
. . . . .	. . . . .	. . . . .	. . . . .	. . . . .		. . . . .
0	0	0	0	0	.....	1

8.4 We want to declare a two-dimensional integer type array called **matrix** for 3 rows and 5 columns. Which of the following declarations are correct? **[LO 8.4 M]**

(a) int maxtrix [3],[5];

(b) int matrix [5] [3];

(c) int matrix [1+2] [2+3];

(d) int matrix [3,5];

(e) int matrix [3] [5];

8.5 Which of the following initialization statements are correct? **[LO 8.2 H]**

(a) char str1[4] = “GOOD”;

(b) char str2[ ] = “C”;

(c) char str3[5] = “Moon”;

(d) char str4[ ] = {‘S’, ‘U’, ‘N’};

(e) char str5[10] = “Sun”;

8.6 What is a data structure? Why is an array called a data structure? **[LO 8.1 E]**

8.7 What is a dynamic array? How is it created? Give a typical example of use of a dynamic array. **[LO 8.6 M]**

8.8 What happens when an array with a specified size is assigned **[LO 8.2 H]**

(a) with values fewer than the specified size; and

(b) with values more than the specified size.

8.9 Discuss how initial values can be assigned to a multidimensional array. **[LO 8.5 E]**

DEBUGGING EXERCISES

8.1 Identify errors, if any, in each of the following array declaration statements, assuming that ROW and COLUMN are declared as symbolic constants:

(a) int score (100); **[LO 8.2, 8.4 E]**

(b) float values [10,15]; **[LO 8.2, 8.4 E]**

(c) char name[15]; **[LO 8.2, 8.4 E]**

(d) float average[ROW],[COLUMN]; **[LO 8.2, 8.4 M]**

(e) double salary [i + ROW] **[LO 8.2, 8.4 M]**

(f) long int number [ROW] **[LO 8.2, 8.4 M]**

(g) int sum[ ]; **[LO 8.2, 8.4 H]**

(h) int array x[COLUMN]; **[LO 8.2, 8.4 H]**

8.2 Identify errors, if any, in each of the following initialization statements.

(a) int number[ ] = {0,0,0,0,0}; **[LO 8.2, 8.4 M]**

(b) float item[3][2] = {0,1,2,3,4,5}; **[LO 8.2, 8.4 M]**

(c) char word[ ] = {‘A’,‘R’, ‘R’, ‘A’, ‘Y’}; **[LO 8.2, 8.4 M]**

(d) int m[2,4] = {(0,0,0,0)(1,1,1,1)}; **[LO 8.2, 8.4 M]**

(e) float result[10] = 0; **[LO 8.2 H]**

8.3 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<4; i++)

scanf("%f", B[i]); **[LO 8.2 E]**

(b) for (i=1; i<=5; i++)

for(j=1; j<=4; j++)

A[i][j] = 0; **[LO 8.2, 8.4 M]**

(c) for (i=0; i<=4; i++)

B[i] = B[i]+i; **[LO 8.2, 8.4 M]**

(d) for (i=4; i>=0; i – –)

for (j=0; j<4; j++)

A[i][j] = B[j] + 1.0; **[LO 8.2, 8.4 M]**

8.4 What is the error in the following program? **[LO 8.2 M]**

```
main ( )
{
        int x ;
        float y [ ] ;
        ......
  }
```

8.5 What is the output of the following program? **[LO 8.2 M]**

```
main ( )
{
          int m [ ] = { 1,2,3,4,5 }
          int x, y = 0;
          for (x = 0; x < 5; x++ )
                  y = y + m [ x ];
          printf("%d", y) ;
}
```

8.6 What is the output of the following program? **[LO 8.2 M]**

```
main ( )
{
          chart string [ ] = "HELLO WORLD" ;
          int m;
          for (m = 0; string [m] != '\0'; m++ )
                if ( (m%2) == 0)
                      printf("%c", string [m] );
}
```

PROGRAMMING EXERCISES

8.1 Write a program for fitting a straight line through a set of points (x_i , y_i), i = 1,....,n. **[LO 8.4 H]**

The straight line equation is

y = mx + c

and the values of m and c are given y

$$m = \frac{n\Sigma\left(x_i y_i\right) - \left(\Sigma x_1\right)\left(\Sigma y_i\right)}{n\left(\Sigma x_i^2\right) - \left(\Sigma x_i\right)^2}$$

$$c = \frac{1}{n}\left(\Sigma y_i - m\,\Sigma x_i\right)$$

All summations are from 1 to n.

8.2 The daily maximum temperatures recorded in 10 cities during the month of January (for all 31 days) have been tabulated as follows: **[LO 8.4 M]**

Day \ City	1	2	3 - - - - - - - - - - -	10
1			- - - - - - - - - - -	
2				
3				
–				
–				
–				
–				
31				

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.

8.3 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. **[LO 8.2 M]**

8.4 The following set of numbers is popularly known as Pascal's triangle. **[LO 8.5 H]**

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.

8.5 The annual examination results of 100 students are tabulated as follows: **[LO 8.2 M]**

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.

8.6 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. **[LO 8.2 H]**

8.7 Two matrices that have the same number of rows and columns can be multiplied to produce a third matrix. Consider the following two matrices. **[LO 8.4 M]**

$$A = \begin{bmatrix} a_{11}\ a_{12}\ldots\ldots a_{1n} \\ a_{12}\ a_{22}\ldots\ldots a_{2n} \\ . \qquad\qquad . \\ . \qquad\qquad . \\ . \qquad\qquad . \\ a_{n1}\ldots\ldots\ a_{nn} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11}\ b_{12}\ldots\ldots b_{1n} \\ b_{12}\ b_{22}\ldots\ldots b_{2n} \\ . \qquad\qquad . \\ . \qquad\qquad . \\ . \qquad\qquad . \\ b_{n1}\ldots\ldots\ b_{nn} \end{bmatrix}$$

The product of **A** and **B** is a third matrix C of size n¥ n where each element of C is given by the following equation.

$$C_{ij} \sum_{k=1}^{n} = a_{ik}b_{kj}$$

Write a program that will read the values of elements of A and B and produce the product matrix **C.**

8.8 Write a program that fills a five-by-five matrix as follows: **[LO 8.4 M]**

- Upper left triangle with +1s
- Lower right triangle with –1s
- Right to left diagonal with zeros

Display the contents of the matrix using not more than two **printf** statements

8.9 Selection sort is based on the following idea:

Selecting the largest array element and swapping it with the last array element leaves an unsorted list whose size is 1 less than the size of the original list. If we repeat this step again on the unsorted list we will have an ordered list of size 2 and an unordered list size n–2 . When we repeat this until the size of the unsorted list becomes one, the result will be a sorted list.

Write a program to implement this algorithm. **[LO 8.2 M]**

8.10 Develop a program to implement the binary search algorithm. This technique compares the search key value with the value of the element that is midway in a “sorted” list. Then;

(a) If they match, the search is over.

(b) If the search key value is less than the middle value, then the first half of the list contains the key value.

(c) If the search key value is greater than the middle value, then the second half contains the key value.

Repeat this “divide-and-conquer” strategy until we have a match. If the list is reduced to one non-matching element, then the list does not contain the key value.

Use the sorted list created in Exercise 8.9 or use any other sorted list. **[LO 8.2 H]**

8.11 Write a program that will compute the length of a given character string. **[LO 8.2 E]**

8.12 Write a program that will count the number occurrences of a specified character in a given line of text. Test your program. **[LO 8.2 E]**

8.13 Write a program to read a matrix of size m ¥ n and print its transpose. **[LO 8.2 E]**

8.14 Every book published by international publishers should carry an International Standard Book Number (ISBN). It is a 10 character 4 part number as shown below.

0-07-041183-2

The first part denotes the region, the second represents publisher, the third identifies the book and the fourth is the check digit. The check digit is computed as follows:

Sum = (1 × first digit) + (2 × second digit) + (3 × third digit) + - - - - + (9 × ninth digit).

Check digit is the remainder when sum is divided by 11. Write a program that reads a given ISBN number and checks whether it represents a valid ISBN. **[LO 8.2 M]**

8.15 Write a program to read two matrices A and B and print the following: **[LO 8.4 L]**

(a) A + B; and

(b) A – B.

8.16 Write a C program to compute the sum of elements of two one-dimensional arrays and store the corresponding result in another array. **[LO 8.2 M]**

8.17 Write a C program to identify and count all the odd numbers in an array and calculate their sum. **[LO 8.2 M]**

8.18 Write a C program to input an amount in Indian currency and deduce the minimum currency denomination for that amount. **[LO 8.2 H]**

8.19 Write a C program to input a Roman number and compute its decimal equivalent. **[LO 8.2 M]**

8.20 Write a C program to input two NXN matrices and deduce their sum and product. **[LO 8.4 H]**

INTERVIEW QUESTIONS

8.1 What is the difference between null array and empty array? **[LO 8.2 H]**

8.2 What is the difference between malloc and calloc functions? **[LO 8.6 M]**

8.3 Which function can be used to convert a string to an integer and vice versa? **[LO 8.2 M]**

E for Easy, **M** for Medium and **H** for High

Chapter 9

Character Arrays and Strings

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

LO 9.1 Discuss how string variables are declared and initialized

LO 9.2 Explain how strings are read from terminal

LO 9.3 Describe how strings are written to screen

LO 9.4 Illustrate how strings are manipulated

INTRODUCTION

A string is a sequence of characters that is treated as a single data item. 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 string constant. Example:

"Man is obviously made to think."

If we want to include a double quote in the string to be printed, 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 include the following:

- 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 examine library functions that implement them.

DECLARING AND INITIALIZING STRING VARIABLES

C does not support strings as a data type. However, it allows us to represent strings as character arrays. In C, therefore, a string variable is any valid C variable name and is always declared as an array of characters. The general form of declaration of a string variable is:

LO 9.1
Discuss how string variables are declared and initialized

```
char string_name[ size ];
```

The *size* determines the number of characters in the string_name. Some examples are as follows:

```
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.

Like numeric arrays, character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms:

```
char city [9] = “NEW YORK“;
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

```
char string [ ] = {‘G’,‘O’,‘O’,‘D’,‘\0’};
```

defines the array **string** as a five element array.

We can also declare the size much larger than the string size in the initializer. That is, the statement.

```
char str[10] = “GOOD”;
```

is permitted. In this case, the computer creates a character array of size 10, places the value “GOOD” in it, terminates with the null character, and initializes all other elements to NULL. The storage will look like

G	O	O	D	\0	\0	\0	\0	\0	\0

However, the following declaration is illegal.

```
char str2[3] = “GOOD”;
```

This will result in a compile time error. Also note that we cannot separate the initialization from declaration. That is,

```
char str3[5];
str3 = “GOOD”;
```

is not allowed. Similarly,

```
char s1[4] = “abc”;
char s2[4];
s2 = s1; /* Error */
```

is not allowed. An array name cannot be used as the left operand of an assignment operator.

Terminating Null Character

You must be wondering, “why do we need a terminating null character?” As we know, a string is not a data type in C, but it is considered a data structure stored in an array. The string is a variable-length structure and is stored in a fixed-length array. The array size is not always the size of the string and most often it is much larger than the string stored in it. Therefore, the last element of the array need not represent the end of the string. We need some way to determine the end of the string data and the null character serves as the “end-of-string” marker.

TIP
A string is represented using a character array and is always terminated with the null character ('\0') having an ASCII of 0. The size of the character array storing a string should be at least one more than the length of the string (to store the null character).

TEST YOUR SKILLS
Write a C program to determine whether the input character is an alphabet or a digit. [E]

READING STRINGS FROM TERMINAL

Using scanf Function

LO 9.2
Explain how strings are read from terminal

The familiar input function **scanf** can be used with **%s** format specification to read in a string of characters. Example:

```
char address[10]
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 reading of string.

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. Note that unlike previous **scanf** calls, in the case of character arrays, the ampersand (&) is not required before the variable name.

The **address** array is created in the memory as shown bel ow:

N	E	W	\0	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Note that the unused locations are filled with garbage.

If we want to read the entire line “NEW YORK”, then we may use two character arrays of appropriate sizes. That is ,

```
char adr1[5], adr2[5];
scanf(“%s %s”, adr1, adr2);
```

with the line of text

NEW YORK

will assign the string “NEW” to **adr1** and “YORK” to **adr2** .

WORKED-OUT PROBLEM 9.1 **E**

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
            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** function

We can also specify the field width using the form %ws in the **scanf** statement for reading a specified number of characters from the input string. Example:

scanf("%ws", name);

Here, the two following things may happen:

1. The width **w** is equal to or greater than the number of characters typed in. The entire string will be stored in the string variable.
2. The width **w** is less than the number of characters in the string. The excess characters will be truncated and left unread.

Consider the following statements:

char name[10];

scanf("%5s", name);

The input string RAM will be stored as:

R	A	M	\0	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

The input string KRISHNA will be stored as:

K	R	I	S	H	\0	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Reading a Line of Text

We have seen just now that **scanf** with %**s** or %**ws** can read only strings without whitespaces. That is, they cannot be used for reading a text containing more than one word. However, C supports a format specification known as the *edit set conversion code* %[. .] that can be used to read a line containing a variety of characters, including whitespaces. Recall that we have used this conversion code in Chapter 5. For example, the program segment

```
char line [80];
scanf(”%[^\n]”, line);
printf(“%s”, line);
```

will read a line of input from the keyboard and display the same on the screen. We would very rarely use this method, as C supports an intrinsic string function to do this job. This is discussed in the next section.

Using getchar and gets

We have discussed in Chapter 5 as to how to read a single character from the terminal, using the **getchar** . We can use it 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. The **getchar** call takes the following form:

```
char ch;
ch = getchar( );
```

Note that the ***getchar*** *has no parameters.*

WORKED-OUT PROBLEM 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 (up to a maximum of 80 characters) into the string **line** using **getchar** . 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.

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.

Program

```
#include <stdio.h>
main( )
{
     char line[81], character;
     int c;
     c = 0;
     printf("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 Centre for Expert Systems, Hyderabad.
National Centre for Expert Systems, Hyderabad.
```

Fig. 9.2 Program to **read a line** of text from terminal

Another and more convenient method of reading a string of text containing whitespaces is to use the library function **gets** available in the *<stdio.h>* header file. This is a simple function with one string parameter and called as under:

gets (str);

str is a string variable declared properly. It reads characters into **str** from the keyboard until a new-line character is encountered and then appends a null character to the string. Unlike **scanf,** it does not skip whitespaces. For example the code segment

```
char line [80];
gets (line);
printf ("%s", line);
```

reads a line of text from the keyboard and displays it on the screen.

The last two statements may be combined as follows:

```
printf("%s", gets(line));
```

(*Be careful not to input more character that can be stored in the string variable used. Since C does not check array-bounds, it may cause problems.*)

> **TIP**
> Extra caution should be taken when using the gets function. It should be checked whether enough memory space has been allocated for the character array or not.

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.

WORKED-OUT PROBLEM 9.3 **M**

Write a program to copy one string into 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.**

Program

```
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

WORKED-OUT PROBLEM 9.4 **H**

The program in Fig. 9.4 shows how to write a program to find the number of vowels and consonants in a text string. Elucidate the program and flowchart for the program.

Algorithm

Step 1 – Start

Step 2 – Read a text string (str)

Step 3 – Set vow = 0, cons = 0, i = 0

Step 4 – Repeat steps 5-6 while (str[i]!='\0')

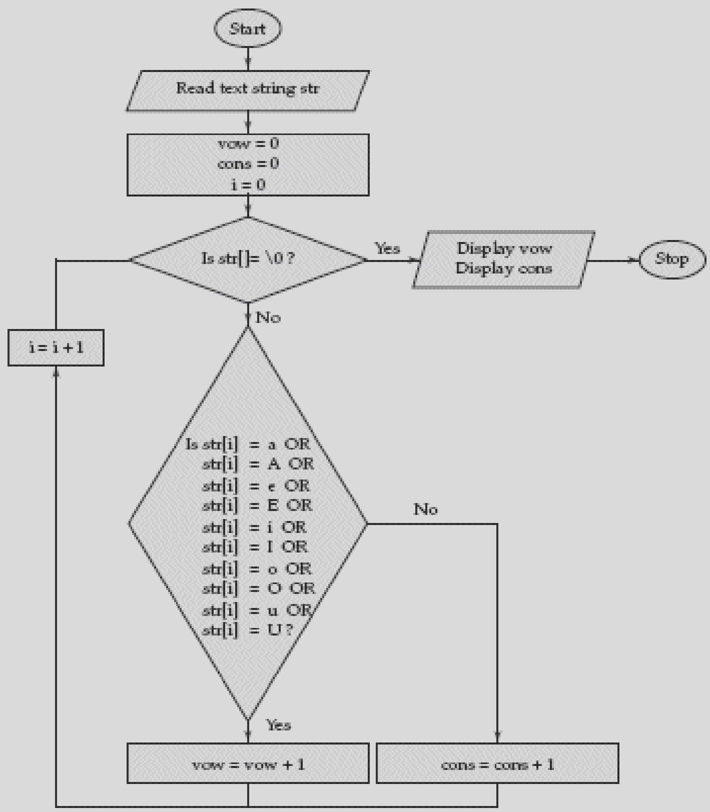
Step 5 – IF str[i] = 'a' OR str[i] = 'A' OR str[i] = 'e' OR str[i] = 'E' OR str[i] = 'i'
OR str[i] = 'I' OR str[i] = 'o' OR str[i] = 'O' OR str[i] = 'u' OR str[i] = 'U'
Increment the vowels counter by 1 (vow=vow+1)
Else Increment the consonants counter by 1 (cons=cons+1)
End If

Step 6 – i = i + 1

Step 7 – Display the number of vowels and consonants (vow, cons)

Step 8 – Stop

Flowchart



Program

```
#include <stdio.h>
#include <string.h>
void main(){
      char str[30];
      int vow=0,cons=0,i=0;
      printf("Enter a string: ");
      gets(str);
      while(str[i] != '\0')
      {
         if(str[i]== a' || str[i]=='A' || str[i]=='e' || str[i]=='E' || str[i]=='i'
```

```
        || str[i]=='I' || str[i]=='o' || str[i]=='O' || str[i]=='u' || str[i]=='U')
            vow++;
        else
            cons++;
        i++;
    }
    printf("\nNumber of Vowels = %d",vow);
    printf("\nNumber of Consonants = %d",cons);
}
Output
  Enter a string: Chennai
  Number of vowels = 3
  Number of consonants = 4
```

Fig. 9.4 Program to find the number of vowel and consonants in a text string

WRITING STRINGS TO SCREEN

Using printf Function

LO 9.3
Describe how strings are written 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. The Program 9.5 illustrates the effect of various '%s' specifications.

WORKED-OUT PROBLEM 9.5

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. 9.5 . 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.
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.
5. The specification % **.** ns prints the first n characters of the string.

Program

```
main()
{
   char country[15] - "United Kingdom";
   printf("\n\n");
   printf("*123456789012345*\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

```
*123456789012345*
-----
United Kingdom
United Kingdom
        United
United
Uni
United Kingdom
-----
```

Fig. 9.5 Writing strings using %s format

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. P rogram 9.5 illustrates this.

WORKED-OUT PROBLEM 9.6

Write a program using **for loop** to print the following output:

```
C
CP
CPr
CPro
.....
.....
CProgramming
CProgramming
.....
.....
CPro
CPr
CP
C
```

The outputs of the program in Fig. 9.6 , for variable specifications **%12.*s, %.*s,** and **%*.1s** are shown in Fig. 9.7 , which further illustrates the variable field width and the precision specifications.

Program

```
main()
{
  int c, d;
  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. 9.6 Illustration of variable field specifications by printing sequences of characters

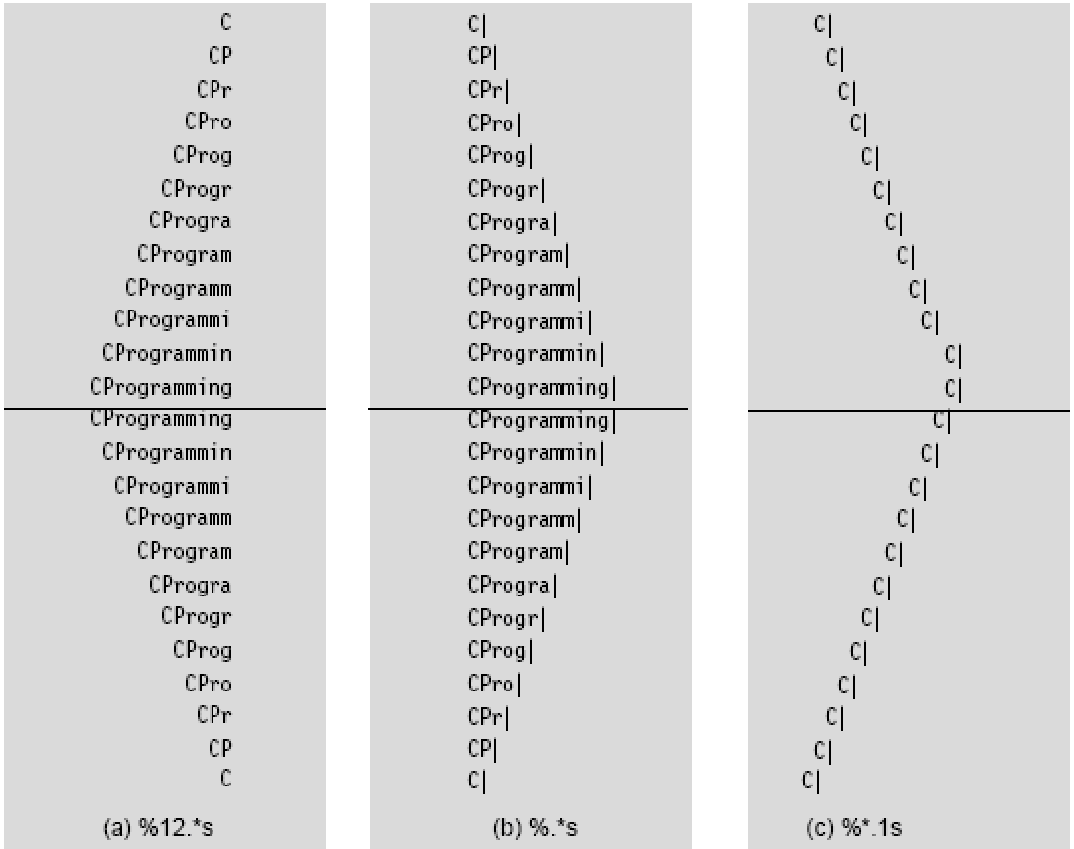


Fig. 9.7 Further illustrations of variable specifications

Using putchar and puts Functions

Like **getchar** , C supports another character handling function **putchar** to output the values of character variables. It takes the following form:

```
char ch = 'A';
putchar (ch);
```

The function **putchar** requires one parameter. This statement is equivalent to

printf("%c", ch);

We have used **putchar** function in Chapter 5 to write characters to the screen. We can use this function repeatedly to output a string of characters stored in an array using a loop. Example:

```
char name[6] = "PARIS"
for (i=0, i<5; i++)
```

```
putchar(name[i];
putchar('\n');
```

Another and more convenient way of printing string values is to use the function **puts** declared in the header file <*stdio.h*> . This is a one parameter function and invoked as under

```
puts ( str );
```

where **str** is a string variable containing a string value. This prints the value of the string variable **str** and then moves the cursor to the beginning of the next line on the screen. For example, the program segment

```
char line [80];
gets (line);
puts (line);
```

reads a line of text from the keyboard and displays it on the screen. Note that the syntax is very simple compared to using the **scanf** and **printf** statements.

ARITHMETIC OPERATIONS ON CHARACTERS

LO 9.4
Illustrate how strings are manipulated

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);
```

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);
```

n umber 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.** String conversion functions are stored in the header file <std.lib.h>.

WORKED-OUT PROBLEM 9.7 E

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. 9.8 . 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.

Program

```
main()
{
   char c;
   printf("\n\n");
   for( c = 65 ; c <= 122 ; c = c + 1 )
   {
      if( c > 90 && c < 97 )
         continue;
      printf("|%4d - %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. 9.8 Printing of the alphabet set in decimal and character form

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 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* . Program 9.9 illustrates the concatenation of three strings.

WORKED-OUT PROBLEM 9.8

M

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. 9.9 . 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
            main()
            {
              int i, j, k ;
              char first_name[10] = {"VISWANATH"} ;
              char second_name[10] = {"PRATAP"} ;
              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. 9.9 Concatenation of strings

Comparison of Two Strings

Once again, C does not permit the comparison of two strings directly. That is, the statements such as

if(name1 == 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");
```

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(string1, 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:

	0	1	2	3	4	5	6	7	8	9	0	1
Part1 =	V	E	R	Y		\0						

	0	1	2	3	4	5	6
Part2 =	G	O	O	D	\0		

	0	1	2	3	4	5	6
Part3 =	B	A	D	\0			

Execution of the statement

```
strcat(part1, part2);
```

will result in:

	0	1	2	3	4	5	6	7	8	9	0	1	2
Part1 =	V	E	R	Y		G	O	O	D	\0			

	0	1	2	3	4	5	6
Part2 =	G	O	O	D	\0		

while the statement `strcat(part1, part3);`

will result in:

	0	1	2	3	4	5	6	7	8	9	0	1	2
Part1 =	V	E	R	Y		B	A	D	\0				

	0	1	2	3	4	5	6
Part3 =	B	A	D	\0			

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 following 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. It takes the form

```
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.

WORKED-OUT PROBLEM 9.9 M

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. 9.10 . 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".

Program

```
#include <string.h>
main()
{  char s1[20], s2[20], s3[20];
   int x, l1, l2, l3;
   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 */
   l1 - strlen(s1);
   l2 - strlen(s2);
   l3 - strlen(s3);
/* output */
   printf("\ns1 - %s\t length - %d characters\n", s1, l1);
   printf("s2 - %s\t length - %d characters\n", s2, l2);
   printf("s3 - %s\t length - %d characters\n", s3, l3);
}
```

Output

```
Enter two string constants
? New York
Strings are not equal
s1 - NewYork    length - 7 characters
s2 - York       length - 4 characters
s3 - NewYork    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. 9.10 Illustration of string handling functions

WORKED-OUT PROBLEM 9.10

M

The program in Fig. 9.11 shows how to write a C program that reads a string and prints if it is a palindrome or not.

```
Program
#include <stdio.h>
#include <string.h>
void main()
{
       char chk='t', str[30];
       intlen, left, right;
       printf("\nEnter a string:");
       scanf("%s", str);
       len=strlen(str);
       left=0;
       right=len-1;
       while(left < right &&chk=='t')
       {
               if(!(str[left] == str[right]))
                          chk='f';
               left++;
               right--;
       }
       if(chk=='t')
              printf("\nThe string %s is a palindrome",str);
       else
              printf("\nThe string %s is not a palindrome",str);
       }
Output
Enter a string: nitin
The string nitin is a palindrome
```

Fig. 9.11 Program to check if a string is palindrome or not

Other String Functions

The header file **<string.h>** contains many more string manipulation functions. They might be useful in certain situations.

strncpy

In addition to the function **strcpy** that copies one string to another, we have another function **strncpy** that copies only the left-most n characters of the source string to the target string variable. This is a three-parameter function and is invoked as follows:

strncpy(s1, s2, 5);

This statement copies the first 5 characters of the source string **s2** into the target string **s1.** Since the first 5 characters may not include the terminating null character, we have to place it explicitly in the 6th position of **s2** as shown below:

s1[6] ='\0';

Now, the string **s1** contains a proper string.

strncmp

A variation of the function **strcmp** is the function **strncmp** . This function has three parameters as illustrated in the function call below:

strncmp (s1, s2, n);

this compares the left-most n characters of **s1** to **s2** and returns.

(a) 0 if they are equal;

(b) negative number, if s1 sub-string is less than s2; and

(c) positive number, otherwise.

strncat

This is another concatenation function that takes three parameters as shown below:

strncat (s1, s2, n);

This call will concatenate the left-most n characters of **s2** to the end of **s1.** Example:

S1 :	B	A	L	A	\0							

S2 :	G	U	R	U	S	A	M	Y	\0

After `strncat` (s1, s2, 4); execution:

S1 :	B	A	L	A	G	U	R	U	\0

strstr

It is a two-parameter function that can be used to locate a sub-string in a string. This takes the following forms:

strstr (s1, s2);

strstr (s1, “ABC”);

The function **strstr** searches the string **s1** to see whether the string **s2** is contained in **s1** . If yes, the function returns the position of the first occurrence of the sub-string. Otherwise, it returns a NULL pointer. Example:

```
if (strstr (s1, s2) == NULL)
     printf("substring is not found");
else
     printf("s2 is a substring of s1");
```

We also have functions to determine the existence of a character in a string. The function call

strchr(s1, ‘m’);

will locate the first occurrence of the character ‘m’ and the call

strrchr(s1, ‘m’);

will locate the last occurrence of the character ‘m’ in the string **s1** .

Warning

- When allocating space for a string during declaration, remember to count the terminating null character.
- When creating an array to hold a copy of a string variable of unknown size, we can compute the size required using the expression

  strlen (stringname) +1.
- When copying or concatenating one string to another, we must ensure that the target (destination) string has enough space to hold the incoming characters. Remember that no error message will be available even if this condition is not satisfied. The copying may overwrite the memory and the program may fail in an unpredictable way.
- When we use **strncpy** to copy a specific number of characters from a source string, we must ensure to append the null character to the target string, in case the number of characters is less than or equal to the source string.

TABLE OF STRINGS

We often use lists of character strings, such as a list of the names of students in a class, list of the 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:

```
char city[ ] [ ]
     {
        "Chandigarh",
        "Madras",
        "Ahmedabad",
        "Hyderabad",
        "Bombay"
     } ;
```

To access the name of the ith city in the list, we write

city[i-1]

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.

WORKED-OUT PROBLEM 9.11 **H**

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. 9.12 . It employs the method of bubble sorting described in Case Study 1 in the previous chapter.

```
Program
            #include <string.h>
            #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. 9.12 Sorting of strings in alphabetical order

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. For example, we can use **gets** function to read a line of text containing a series of words. We may also use **puts** function in place of **scanf** for output.

TEST YOUR SKILLS

Write a C program that uses strstr() function to determine whether one string is a sub-string of another string. [E]

OTHER FEATURES OF STRINGS

Other aspects of strings we have not discussed in this chapter include the following:

- Manipulating strings using pointers.
- Using string as function parameters.
- Declaring and defining strings as members of structures.

These topics will be dealt with later when we discuss functions, structures, and pointers.

KEY CONCEPTS

- **STRING:** Is a sequence of characters that is considered as a single data item. **[LO 9.1]**
- **strcat :** Concatenates two strings. [LO 9.4]
- **strcmp :** Compares two strings and determines whether they are equal or not. **[LO 9.4]**
- **strcpy :** Copies one string into another. **[LO 9.4]**
- **strstr :** Determines whether one string is a subset of another. **[LO 9.4]**

ALWAYS REMEMBER

- Character constants are enclosed in single quotes and string constants are enclosed in double quotes. **[LO 9.1]**
- Allocate sufficient space in a character array to hold the null character at the end. **[LO 9.1]**
- Avoid processing single characters as strings. **[LO 9.1]**
- It is a compile time error to assign a string to a character variable.

- The header file <stdio.h> is required when using standard **[LO 9.1]** I/O functions. **[LO 9.1]**
- The header file <stdlib.h> is required when using general utility functions. **[LO 9.1]**
- Using the address operator **&** with a **string** variable in the **scanf** function call is an error. **[LO 9.2]**
- Use %s format for printing strings or character arrays terminated by null character. **[LO 9.3]**
- Using a string variable name on the left of the assignment operator is illegal. **[LO 9.4]**
- When accessing individual characters in a string variable, it is logical error to access outside the array bounds. **[LO 9.4]**
- Strings cannot be manipulated with operators. Use string functions. **[LO 9.4]**
- Do not use string functions on an array **char** type that is not terminated with the null character. **[LO 9.4]**
- Do not forget to append the null character to the target string when the number of characters copied is less than or equal to the source string. **[LO 9.4]**
- Be aware the return values when using the functions **strcmp** and **strncmp** for comparing strings. **[LO 9.4]**
- When using string functions for copying and concatenating strings, make sure that the target string has enough space to store the resulting string. Otherwise memory overwriting may occur. **[LO 9.4]**
- The header file <ctype.h> is required when using character handling functions. **[LO 9.4]**
- The header file <string.h> is required when using string manipulation functions. **[LO 9.4]**

BRIEF CASES

1. Counting Words in a Text[LO 9.1, 9.2 M]

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. 9.13 . 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 test

```
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.

Program

```
#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. 9.13 Counting of characters, words and lines in a text

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.

2. Processing of a Customer List [LO 9.1, 9.2, 9.3, 9.4 M]

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 9.14 shows a program to achieve this.

Program

```
#include <string.h>
#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 Wilhelm 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. 9.14 Program to alphabetize a customer list

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

9.1 Which of the following values can be returned by the strcmp() function? **[LO 9.4 M]**

(a) Positive value

(b) Negative value

(c) Zero

(d) All of these

9.2 Which of the following functions is used to concatenate only specific number of characters from one string to another? **[LO 9.4 E]**

(a) strcat()

(b) strstr()

(c) strncat()

(d) None of these

9.3 Which of the following functions is used to search a substring in another string? **[LO 9.4 E]**

(a) searchstr()

(b) strstr()

(c) substr()

(d) None of these

9.4 Which of the following arrays can be used to store the names of 10 employees working in an organization? **[LO 9.4 M]**

(a) employee[10][80]

(b) employee[80][10]

(c) employee[][10][80]

(d) None of these

9.5 Which of the following is the correct input statement for reading a string from the terminal?

(a) scanf(”%[^\n]”, str); **[LO 9.2 H]**

(b) gets(str);

(c) scanf(”%[^\n]”, str[0]);

(d) All of these are correct

9.6 A character array str[] stores the following string. What will be the return value of strlen(str) function? **[LO 9.4 E]**

N	E	W	\0

(a) 3

(b) 4

(c) 0

(d) None of these

9.7 Which of the following is an incorrect input statement for reading a string from the terminal? **[LO 9.2 M]**

(a) scanf("%[^\n]", str);

(b) gets(str);

(c) scanf("%s", &str[0]);

(d) All of these are correct

REVIEW QUESTIONS

9.1 State whether the following statements are *true* or *false*

(a) When initializing a string variable during its declaration, we must include the null character as part of the string constant, like "GOOD\0". **[LO 9.1 M]**

(b) The **gets** function automatically appends the null character at the end of the string read from the keyboard. **[LO 9.2 E]**

(c) When reading a string with **scanf** , it automatically inserts the terminating null character. **[LO 9.2 E]**

(d) The input function **gets** has one string parameter. **[LO 9.2 E]**

(e) The function **scanf** cannot be used in any way to read a line of text with the white-spaces. **[LO 9.2 M]**

(f) The function **getchar** skips white-space during input. **[LO 9.2 M]**

(g) In C, strings cannot be initialized at run time. **[LO 9.2 H]**

(h) String variables cannot be used with the assignment operator. **[LO 9.4 E]**

(i) We cannot perform arithmetic operations on character variables. **[LO 9.4 E]**

(j) The ASCII character set consists of 128 distinct characters. **[LO 9.4 E]**

(k) In the ASCII collating sequence, the uppercase letters precede lowercase letters. **[LO 9.4 M]**

(l) In C, it is illegal to mix character data with numeric data in arithmetic operations. **[LO 9.4 M]**

(m) The function call **strcpy(s2, s1);** copies string s2 into string s1. **[LO 9.4 M]**

(n) The function call **strcmp("abc", "ABC");** returns a positive number. **[LO 9.4 M]**

(o) We can assign a character constant or a character variable to an **int** type variable. **[LO 9.4 H]**

9.2 Fill in the blanks in the following statements.

(a) We can use the conversion specification _______in **scanf** to read a line of text. **[LO 9.2 E]**

(b) The function _______does not require any conversion specification to read a string from the keyboard. **[LO 9.2 E]**

(c) The **printf** may be replaced by ______function for printing strings. **[LO 9.3 E]**

(d) The function **strncat** has _____ parameters. **[LO 9.4 E]**

(e) The function _______ is used to determine the length of a string. **[LO 9.4 E]**

(f) We can initialize a string using the string manipulation function_______. **[LO 9.2 M]**

(g) To use the function **atoi** in a program, we must include the header file ____. **[LO 9.4 M]**

(h) The _________string manipulation function determines if a character is contained in a string. **[LO 9.4 M]**

(i) The function call **strcat (s2, s1);** appends _____ to ______. **[LO 9.4 M]**

(j) The function _____is used to sort the strings in alphabetical order. **[LO 9.4 H]**

9.3 Describe the limitations of using **getchar** and **scanf** functions for reading strings. **[LO 9.2 M]**

9.4 Character strings in C are automatically terminated by the *null* character. Explain how this feature helps in string manipulations. **[LO 9.4 H]**

9.5 Strings can be assigned values as follows: **[LO 9.1, 9.2, 9.4 E]**

(a) During type declaration char string[ ] = {"......."};

(b) Using **strcpy** function strcpy(string, ".......");

(c) Reading using **scanf** function scanf("%s", string);

(d) Reading using **gets** function gets(string);

Compare them critically and describe situations where one is superior to the others.

9.6 Assuming the variable **string** contains the value "The sky is the limit.", determine what output of the following program segments will be. **[LO 9.3 H]**

```
(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')
```

9.7 Which of the following statements will correctly store the concatenation of strings **s1** and **s2** in string **s3** ? **[LO 9.4 M]**

(a) s3 = strcat (s1, s2);

(b) strcat (s1, s2, s3);

(c) strcat (s3, s2, s1);

(d) strcpy (s3, strcat (s1, s2));

(e) strcmp (s3, strcat (s1, s2));

(f) strcpy (strcat (s1, s2), s3);

9.8 What will be the output of the following statement? **[LO 9.3 M]**

```
printf ("%d", strcmp ("push", "pull"));
```

9.9 Assume that s1, s2 and s3 are declared as follows: **[LO 9.4 H]**

```
char s1[10] = "he", s2[20] = "she", s3[30], s4[30];
```

What will be the output of the following statements executed in sequence?

```
printf("%s", strcpy(s3, s1));
printf("%s", strcat(strcat(strcpy(s4, s1), "or"), s2));
printf("%d %d", strlen(s2)+strlen(s3), strlen(s4));
```

9.10 What will be the output of the following segment? **[LO 9.4 E]**

```
char s1[ ] = "Kolkotta" ;
char s2[ ] = "Pune" ;
strcpy (s1, s2) ;
printf("%s", s1) ;
```

9.11 What will be the output of the following segment? **[LO 9.4 E]**

```
char s1[ ] = "NEW DELHI" ;
char s2[ ] = "BANGALORE" ;
strncpy (s1, s2, 3) ;
printf("%s", s1) ;
```

9.12 What will be the output of the following code? **[LO 9.4 E]**

```
char s1[ ] = "Jabalpur" ;
char s2[ ] = "Jaipur" ;
printf(strncmp(s1, s2, 2) );
```

9.13 What will be the output of the following code? **[LO 9.4 E]**

```
char s1[ ] = "ANIL KUMAR GUPTA";
char s2[ ] = "KUMAR";
printf (strstr (s1, s2) );
```

9.14 Compare the working of the following functions: **[LO 9.4 M]**

(a) strcpy and strncpy;

(b) strcat and strncat; and

(c) strcmp and strncmp.

DEBUGGING EXERCISES

9.1 Find errors, if any, in the following code segments: **[LO 9.4 M]**

(a)
```
char str[10]
    strncpy(str, "GOD", 3);
    printf("%s", str);
```

(b)
```
char str[10];
```

strcpy(str, “Balagurusamy”);

(c) if strstr(“Balagurusamy”, “guru”) = = 0);

printf(“Substring is found”);

(d) char s1[5], s2[10],

gets(s1, s2);

PROGRAMMING EXERCISES

9.1 Write a program, which reads your name from the keyboard and outputs a list of ASCII codes, which represent your name. **[LO 9.4 M]**

9.2 Write a program to do the following: **[LO 9.3 M]**

(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.

9.3 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 nth character. **[LO 9.4 M]**

9.4 Write a program which will read a text and count all occurrences of a particular word. **[LO 9.4 H]**

9.5 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. **[LO 9.4 H]**

9.6 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.” **[LO 9.4 H]**

9.7 A Maruti car dealer maintains a record of sales of various vehicles in the following form: **[LO 9.4 M]**

Vehicle type	Month of sales	Price
MARUTI-800	02/01	210000
MARUTI-DX	07/01	265000
GYPSY	04/02	315750
MARUTI-VAN	08/02	240000

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).

9.8 Write a program that reads a string from the keyboard and determines whether the string is a *palindrome* or not. (A string is a palindrome if it can be read from left and right with the same meaning. For example, Madam and Anna are palindrome strings. Ignore capitalization). **[LO 9.4 M]**

9.9 Write program that reads the cost of an item in the form RRRR.PP (Where RRRR denotes Rupees and PP denotes Paise) and converts the value to a string of words that expresses the numeric value in words. For example, if we input 125.75, the output should be "ONE HUNDRED TWENTY FIVE AND PAISE SEVENTY FIVE". **[LO 9.3 H]**

9.10 Develop a program that will read and store the details of a list of students in the format **[LO 9.4 H]**

Roll No.	Name	Marks obtained
. . . .	. . . .	. . . .
. . . .	. . . .	. . . .
. . . .	. .. .	. . ..

and produce the following output list:

(a) Alphabetical list of names, roll numbers and marks obtained.

(b) List sorted on roll numbers.

(c) List sorted on marks (rank-wise list)

9.11 Write a program to read two strings and compare them using the function **strncm p ( )** and print a message that the first string is equal, less, or greater than the second one. **[LO 9.4 E]**

9.12 Write a program to read a line of text from the keyboard and print out the number of occurrences of a given substring using the function **strstr ( ).** **[LO 9.4 M]**

9.13 Write a program that will copy m consecutive characters from a string s1 beginning at position n into another string s2. **[LO 9.4 E]**

9.14 Write a program to create a directory of students with roll numbers. The program should display the roll number for a specified name and vice-versa. **[LO 9.4 M]**

9.15 Given a string

char str [ ] = "123456789" ;

Write a program that displays the following: **[LO 9.3 M]**

```
    1
   232
  34543
 4567654
567898765
```

9.16 Write a C program to compare two strings without using any string function. **[LO 9.3 M]**

9.17 Write a C program to find the largest and smallest sized word in a string. **[LO 9.3 H]**

9.18 Write a C program to replace all the white spaces in a string with double white spaces. **[LO 9.3 M]**

9.19 Write a C program to enter multiple strings and display them in lexicographical order. **[LO 9.4 H]**

9.20 Write a C program to concatenate two strings without using any string function. **[LO 9.3 M]**

INTERVIEW QUESTIONS

9.1 Write the code for printing "Hello World' using printf function but the printf statement should not end with a semi-colon. **[LO 9.3 H]**

9.2 If the if block in a program contains the statement printf("Hello") while the corresponding else block contains the statement printf(" World"), then what should be the condition within the if statement so that the output of the code is 'Hello World'? **[LO 9.3 H]**

9.3 Can you write a program that does not use any semi-colon but still executes successfully? **[LO 9.1 H]**

9.4 What is strcat function? **[LO 9.4 E]**

9.5 Is there any string handling function called strstr? **[LO 9.4 E]**

E for Easy, **M** for Medium and **H** for High

Chapter 10

User-Defined Functions

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

LO 10.1 Outline user-defined functions
LO 10.2 Identify the elements of user-defined functions
LO 10.3 Explain the different categories of functions
LO 10.4 Know the concept of recursion
LO 10.5 Describe how arrays are passed to functions
LO 10.6 Discuss the relevance of storage classes on scope, visibility and lifetime of variables

INTRODUCTION

We have mentioned earlier that one of t he strengths of C language is C functions. They 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 the following:

- How a function is designed?
- How a function is integrated into a program?
- How two or more functions are put together? and
- How they communicate with one another?

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. In fact, this is one of the strengths of C language.

Need for User-Defined Functions

LO 10.1
Outline 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 independently coded programs are called *subprograms* that are much easier to understand, debug, and test. In C, such subprograms are referred to as **'functions'** .

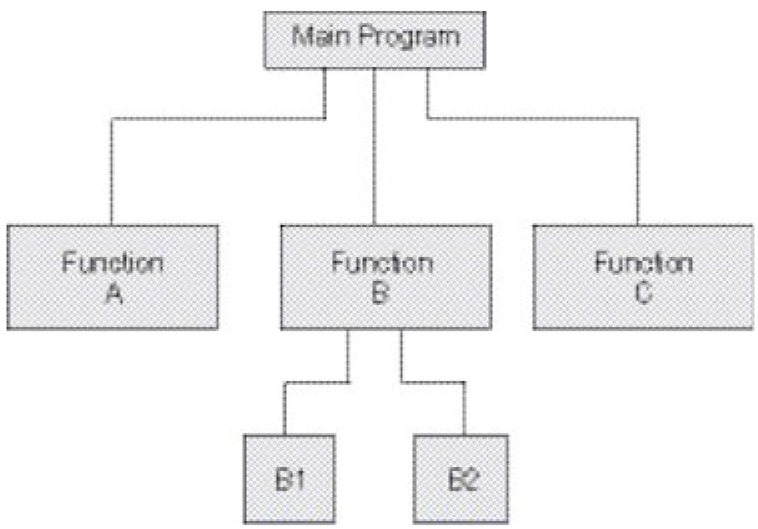


Fig. 10.1 Top-down modular programming using functions

There are times when certain type of operations or calculations are 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 “division” 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. 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 all over again from scratch.

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 known as *functions* .

Consider a set of statements as shown below:

```
void  printline(void)
{
      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:

```
void printline(void); /* declaration  */
   main( )
  {
      printline( );
      printf("This illustrates the use of C functions\n");
      printline();
  }
    void printline(void)
  {
      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() function

printline() function

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** 39 times repeatedly.

Any function can call any other function. In fact, it can also 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 functions 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. See the box “Modular Programming”.

Modular Programming

Modular programming is a strategy applied to the design and development of software systems. It is defined as organizing a large program into small, independent program segments called ***modules*** that are separately named and individually callable *program units* . These modules are carefully integrated to become a software system that satisfies the system requirements. It is basically a “divide-and-conquer” approach to problem solving.

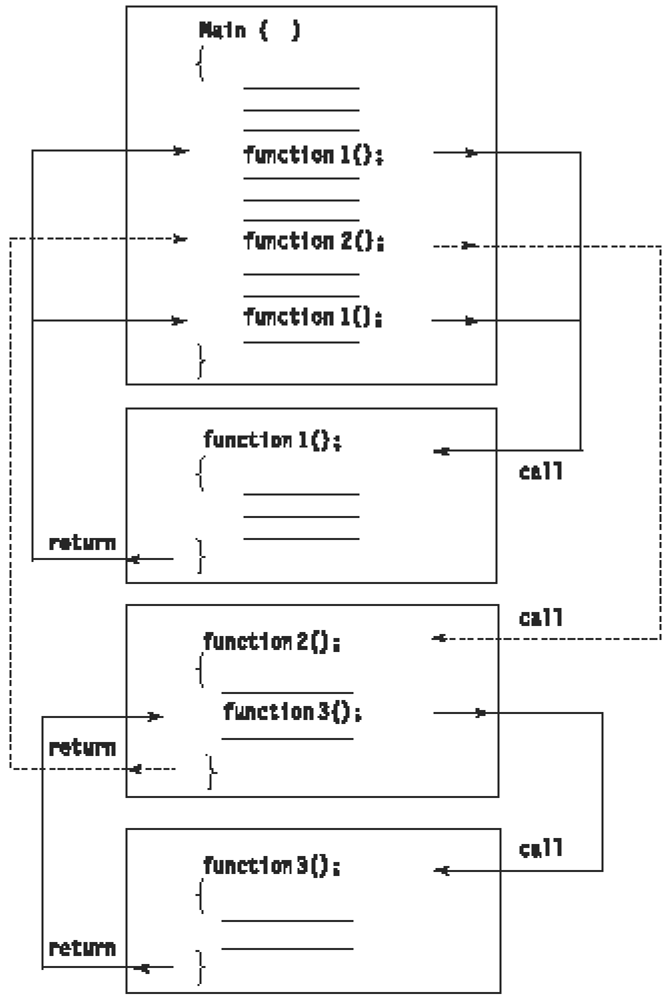


Fig. 10.2 Flow of control in a multi-function program

Modules are identified and designed such that they can be organized into a top-down hierarchical structure (similar to an organization chart). In C, each module refers to a function that is responsible for a single task.

Some characteristics of modular programming are as follows:

1. Each module should do only one thing.
2. Communication between modules is allowed only by a calling module.
3. A module can be called by one and only one higher module.
4. No communication can take place directly between modules that do not have calling-called relationship.

> **TIP**
> A function should be a cohesive unit. The coupling between two different functions should be the least.

5. All modules are designed as *single-entry, single-exit* systems using control structures.

ELEMENTS OF USER-DEFINED FUNCTIONS

LO 10.2
Identify the elements of user-defined functions

We have discussed and used a variety of data types and variables in our programs so far. However, declaration and use of these variables were primarily done inside the **main** function. As we mentioned in Chapter 5, functions are classified as one of the derived data types in C. We can therefore define functions and use them like any other variables in C programs. It is therefore not a surprise to note that there exist some similarities between functions and variables in C.

- Both function names and variable names are considered identifiers and therefore, they must adhere to the rules for identifiers.
- Like variables, functions have types (such as int) associated with them.
- Like variables, function names and their types must be declared and defined before they are used in a program.

In order to make use of a user-defined function, we need to establish three elements that are related to functions.

1. Function definition.
2. Function call.
3. Function declaration.

The *function definition* is an independent program module that is specially written to implement the requirements of the function. In order to use this function we need to invoke it at a required plac e in the program. This is known as the *function call.* The program (or a function) that calls the function is referred to as the *calling program* or *calling function.* The calling program should declare any function (like declaration of a variable) that is to be used later in the program. This is known as the *function declaration* or *function prototype.*

Definition of Functions

A *function definition,* also known as *function implementation* shall include the following elements:

1. function name;
2. function type;
3. list of parameters;
4. local variable declarations;
5. function statements; and
6. a return statement.

All the six elements are grouped into two parts, namely,

- function header (First three elements); and
- function body (Second three elements).

A general format of a function definition to implement these two parts is given below:

```
function_type  function_name(parameter list)
  {
    local variable declaration;
    executable statement1;
    executable statement2;
    . . . . .
    . . . . .
    return statement;
  }
```

The first line **function_type function_name(parameter list)** is known as the *function header* and the statements within the opening and closing braces constitute the *function body,* which is a compound statement.

Function Header

The function header consists of three parts: the function type (also known as *return* type), the function name, and the *formal* parameter list. Note that a semicolon is not used at the end of the function header.

Name and Type

The *function type* specifies the type of value (*like float or double*) that the function is expected to return to the program calling the function. If the return type is not explicitly specified, C will assume that it is an integer type. If the function is not returning anything, then we need to specify the return type as **void** . Remember, **void** is one of the fundamental data types in C. It is a good programming practice to code explicitly the return type, even when it is an integer. The value returned is the output produced by the function.

The *function name* is any valid C identifier and therefore must follow the same rules of formation as other variable names in C. The name should be appropriate to the task performed by the function. However, care must be exercised to avoid duplicating library routine names or operating system commands.

Formal Parameter List

The *parameter list* declares the variables that will receive the data sent by the calling program. They serve as input data to the function to carry out the specified task. Since they represent the actual input values, they are often referred to as *formal* parameters. These parameters can also be used to send values to the calling programs. This aspect will be covered later when we discuss more about functions. The parameters are also known as *arguments.*

The parameter list contains declaration of variables separated by commas and surrounded by parentheses. Examples:

```
float quadratic (int a, int b, int c) {. . . . }
double power (double x, int n) {. . . ..}
float mul (float x, float y) {. . . . }
int sum (int a, int b) {. . . . }
```

Remember, there is no semicolon after the closing parenthesis. Note that the declaration of parameter variables cannot be combined. That is, **int sum (int a,b)** is illegal.

A function need not always receive values from the calling program. In such cases, functions have no formal parameters. To indicate that the parameter list is empty, we use the keyword **void** between the parentheses as in

```
void printline (void)
{
. . . .
}
```

This function neither receives any input values nor returns back any value. Many compilers accept an empty set of parentheses, without specifying anything as in

```
void printline ( )
```

But, it is a good programming style to use **void** to indicate a nill parameter list.

Function Body

The *function body* contains the declarations and statements necessary for performing the required task. The body enclosed in braces, contains three parts, in the order given as follows:

1. Local declarations that specify the variables needed by the function.

2. Function statements that perform the task of the function.

3. A **return** statement that returns the value evaluated by the function.

If a function does not return any value (like the **printline** function), we can omit the **return** statement. However, note that its return type should be specified as **void.** Again, it is nice to have a return statement even for **void** functions.

Some examples of typical function definitions are:

```
(a)  float mul (float x, float y)
     {
        float result;            /* local variable */
        result = x * y;             /* computes the product */
        return (result);            /* returns the result */
     }
(b)  void sum (int a, int b)
     {
        printf ("sum = %s", a + b);  /* no local variables */
        return;                     /* optional */
     }
(c)  void display (void)
        {                           /* no local variables */
           printf ("No type, no parameters");
                                    /* no return statement */
        }
```

Note

1. *When a function reaches its return statement, the control is transferred back to the calling program. In the absence of a return statement, the closing brace acts as a* **void return**.
2. *A* local *variable is a variable that is defined inside a function and used without having any role in the communication between functions.*

RETURN VALUES AND THEIR TYPES

As pointed out earlier, 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;

Note *C 9 9, if a function is specified as returning a value, the* ***return*** *must have value associated with it.*

The second form of **return** with an expression returns the value of the expression. For example, the function

```
int mul (int x, int 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 as discussed earlier.

When a value is returned, it is automatically cast to the function's type. In functions that do computations using **doubles** , yet return **ints** , the returned value will be truncated to an integer. For instance, the function

```
int product (void)
{
      return (2.5 * 3.0);
}
```

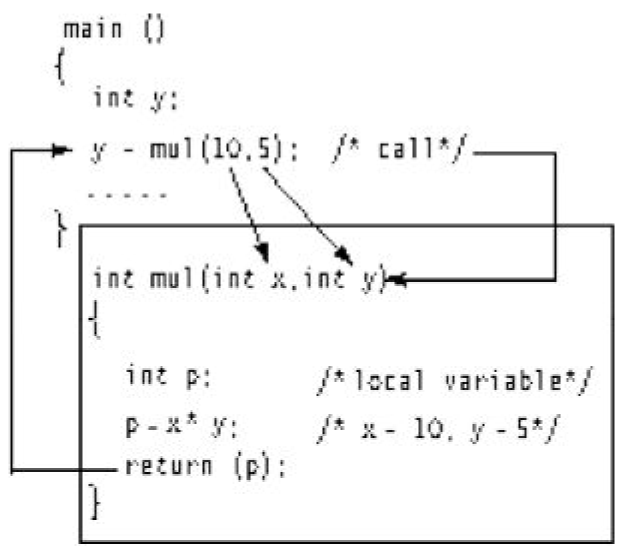
will return the value 7, only the integer part of the result.

Function Calls

A function can be called by simply using the function name followed by a list of *actual parameters* (or arguments), if any, enclosed in parentheses. Example:

```
main( )
{
   int y;
   y = mul(10,5);        /* Function call */
   printf("%d\n", y);
}
```

When the compiler encounters a function call, the control is transferred to the function **mul()** . 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 **y** . This is illustrated as follows :



The function call sends two integer values 10 and 5 to the function.

int mul(int x, int y)

which are assigned to **x** and **y** respectively. The function computes the product **x** and **y** , assigns the result to the local variable **p** , and then returns the value 25 to the **main** where it is assigned to **y** again.

There are many different ways to call a function. Listed below are some of the ways the function **mul** can be invoked.

mul (10, 5)

mul (m, 5)

mul (10, n)

mul (m, n)

mul (m + 5, 10)

mul (10, mul(m,n))

mul (expression 1, expression2)

Note that the sixth call uses its own call as its one of the parameters. When we use expressions, they should be evaluated to single values that can be passed as actual parameters.

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,

```
mu l (a,b) = 15;
```

is invalid.

A function that does not return any value may not be used in expressions; but can be called in to perform certain tasks specified in the function. The function **printline( )** discussed in Section 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.

Function Call

A function call is a postfix expression. The operator (. .) is at a very high level of precedence. (See Table 4.8) Therefore, when a function call is used as a part of an expression, it will be evaluated first, unless parentheses are used to change the order of precedence.

In a function call, the function name is the operand and the parentheses set (. .) which contains the *actual parameters* is the operator. The actual parameters must match the function's formal parameters in type, order and number. Multiple actual parameters must be separated by commas.

Note

1. *If the actual parameters are more than the formal parameters, the extra actual arguments will be discarded.*
2. *On the other hand, if the actuals are less than the formals, the unmatched formal arguments will be initialized to some garbage.*
3. *Any mismatch in data types may also result in some garbage values.*

FUNCTION DECLARATION

Like variables, all functions in a C program must be declared, before they are invoked. A *function declaration* (also known as *function prototype*) consists of four parts.

- Function type (return type).
- Function name.
- Parameter list.

- Terminating semicolon.

They are coded in the following format:

Function-type function-name **(parameter list);**

This is very similar to the function header line except the terminating semicolon. For example, **mul** function defined in the previous section will be declared as:

int mul (int m, int n); /* Function prototype */

Points to Note

1. The parameter list must be separated by commas.
2. The parameter names do not need to be the same in the prototype declaration and the function definition.
3. The types must match the types of parameters in the function definition, in number and order.
4. Use of parameter names in the declaration is optional.
5. If the function has no formal parameters, the list is written as (void).
6. The return type is optional, when the function returns **int** type data.
7. The retype must be **void** if no value is returned.
8. When the declared types do not match with the types in the function definition, compiler will produce an error.

Equally acceptable forms of declaration of **mul** function are as follows:

```
int  mul  (int, int);
     mul  (int a, int b);
     mul  (int, int);
```

When a function does not take any parameters and does not return any value, its prototype is written as:

void display (void);

A prototype declaration may be placed in two places in a program.

1. Above all the functions (including **main**).
2. Inside a function definition.

When we place the declaration above all the functions (in the global declaration section), the prototype is referred to as a *global prototype* . Such declarations are available for all the functions in the program.

When we place it in a function definition (in the local declaration section), the prototype is called a *local prototype* . Such declarations are primarily used by the functions containing them.

The place of declaration of a function defines a region in a program in which the function may be used by other functions. This region is known as the *scope* of the function. (Scope is discussed later in this chapter.) It is a good programming style to declare prototypes in the global declaration section before **main** . It adds flexibility, provides an excellent quick reference to the functions used in the program, and enhances documentation.

Prototypes: Yes or No

Prototype declarations are not essential. If a function has not been declared before it is used, C will assume that its details available at the time of linking. Since the prototype is not available, C will assume that the return type is an integer and that the types of parameters match the formal definitions. If these assumptions are wrong, the linker will fail and we will have to change the program. The moral is that we must always include prototype declarations, preferably in global declaration section.

Parameters Everywhere!

Parameters (also known as arguments) are used in following three places:

1. in declaration (prototypes),
2. in function call, and
3. in function definition.

The parameters used in prototypes and function definitions are called *formal parameters* and those used in function calls are called *actual parameters* . Actual parameters used in a calling statement may be simple constants, variables, or expressions.

TIP

Remember that the prototype of a function as provided in the declaration should exactly match the function header in the definition.

The formal and actual parameters must match exactly in type, order and number. Their names, however, do not need to match.

Category of Functions

LO 10.3
Explain the different categories 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 one return value.

Category 4: Functions with no arguments but return a value.

Category 5: Functions that return multiple values.

In the sections to follow, we shall discuss these categories with examples. Note that, from now on, we shall use the term arguments (rather than parameters) more frequently .

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.

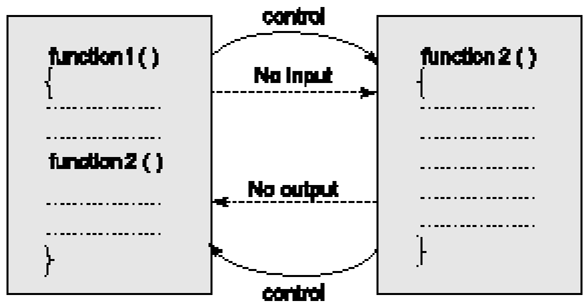


Fig. 10.3 No data communication between functions

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.

WORKED-OUT PROBLEM 10.1

E

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

```
/* Function declaration */
void printline (void);
void value (void);
  main()
  {
       printline();
       value();
       printline();
  }
  /*      Function1: printline( )         */

  void printline(void)   /* contains no arguments */
  {
       int i ;

       for(i=1; i <= 35; i++)
         printf("%c",'-');
       printf("\n");
  }
  /*       Function2: value( )           */
  void value(void)        /* 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 include 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. Return types of both **printline** and **value** are declared as **void.**

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.

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.

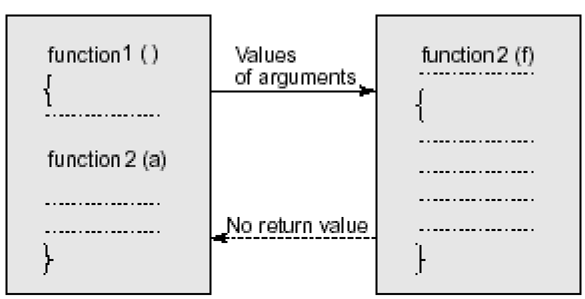


Fig. 10.5 One-way data communication

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 .

We shall modify the definitions of both the called functions to include arguments as follows:

void printline(char ch)

void value(float p, float r, int 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

void value(float p, float r, int 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 .

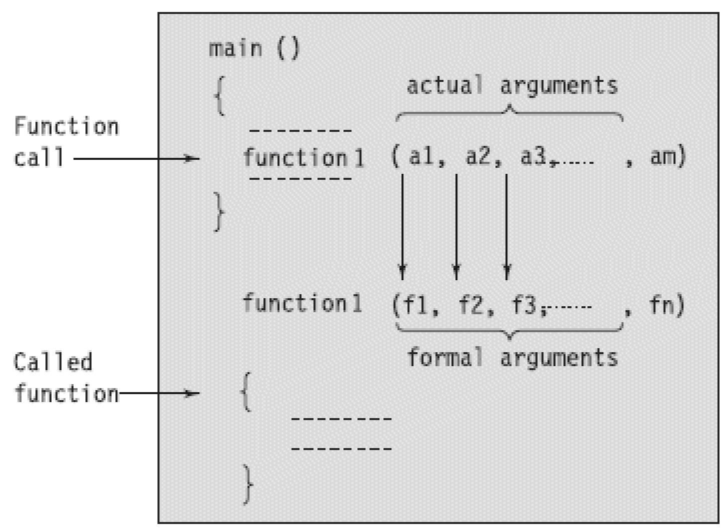


Fig. 10.6 Arguments matching between the function call and the called function

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.

WORKED-OUT PROBLEM 10.2 **M**

Modify the program of Program 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.** 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;
```

Program

```
/* prototypes */
void printline (char c);
void value (float, float, int);

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');
}
void printline(char ch)
{
     int i ;
     for(i=1; i <= 52; i++)
          printf("%c",ch);
     printf("\n");
}
void value(float p, float r, int n)
{
     int year ;
     float sum ;
     sum = p ;
     year = 1;
     while(year <= n)
     {
         sum = sum * (1+r);
         year = year +1;
     }
        printf("%f\t%f\t%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

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).

Variable Number of Arguments

Some functions have a variable number of arguments and data types which cannot be known at compile time. The **printf** 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 (int num, …)

Both the function declaration and definition should use ellipsis to indicate that the arguments are arbitrary both in number and type.

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. 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 display of 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.

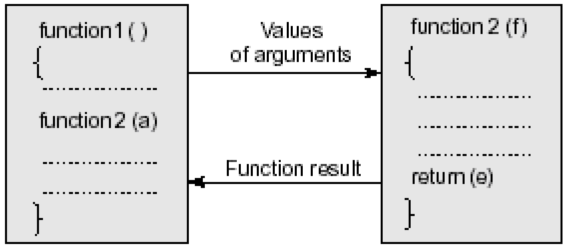


Fig. 10.8 Two-way data communication between functions

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* .

WORKED-OUT PROBLEM 10.3 M

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. 10.10 . One major change is the movement of the **printf** statement from **value** to **main.**

```
Program
          void printline (char ch, int len);
          value (float, float, int);

              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);
              }
                void printline(char ch, int len)
                {
                  int i;
                  for (i=1;i<=len;i++) printf("%c",ch);
                  printf("\n");
                }
                value(float p, float r, int n) /* default return type */
                {
                  int year;
                  float sum;
                  sum = p; year = 1;
                  while(year <=n)
                  {
                    sum = sum * (1+r);
                    year = year +1;
                  }
                  return(sum);     /* returns int part of sum */
                }
Output
          Enter principal amount, interest rate, and period
          5000    0.12    5
          ****************************************************
          5000.000000     0.1200000    5    8811.000000
          = = = = = = = = = = = = = = = = = = = = = = = = = =
```

Fig. 10.9 Functions with arguments and return values

The calculated value is passed on to **main** through statement:

return(sum);

Since, by default, the return type of **value** function is **int,** the ‘integer’ value of **sum** at this point is returned to **main** and assigned to the variable **amount** by the functional 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 actual 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 integer value of **sum** is passed back to the function-call in the **main** and the following indirect assignment occurs:

   value(principal, inrate, period) = sum;

3. The calling statement is executed normally and the returned value is thus assigned to **amount** , a **float** variable.

4. Since **amount** is a **float** variable, the returned integer part of sum is converted to floating-point value. See the output.

Another important change is the inclusion of 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;

Returning Float Values

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 Program 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.**

In all such cases, we must explicitly specify the *return type* in both the function definition and the prototype declaration.

If 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.

WORKED-OUT PROBLEM 10.4 **E**

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 prototype declaration

double power(int, int);

appears in **main** , before **power** is called.

```
Program
            main( )
            {
               int x,y;          /*input data */
               double power(int, int); /* prototype declaration*/
               printf("Enter x,y:");
               scanf("%d %d" , &x,&y);
               printf("%d to power %d is %f\n", x,y,power (x,y));

            }
            double power (int x, int 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 Power functions: Illustration of return of float values

WORKED-OUT PROBLEM 10.5

H

The program in Fig. 10.11 shows how to write a C program (float x [ ], int n) that returns the position of the first minimum value among the first n elements of the given array x.

```
Program
        #include <stdio.h>
        #include <stdlib.h>
        int minpos(float [], int);
        void main()
        {
                int n;
                float x[10] = {12.5, 3.0, 45.1, 8.2, 19.3, 10.0, 7.8, 23.7, 29.9, 5.2};
                printf("Enter the value of n: ");
                scanf("%d", &n);
                if(n < 1 || n > 10)
                {
                        printf("invalid value of n...Press any key to terminate the
                                program..");
                        exit(0);
                }
                printf("Within the first %d elements of array, the first minimum value is
                        stored at index %d". n, minpos(x,n));
        }
        int minpos(float a[], int N)
        {
                int i, index;
                float min=9999.99;
                for(i=0; i<N; i++)
                        if(a[i ] < min)
                        {
                                min = a[i];
                                index = i;
                        }
                return (index);
        }
Output
   Enter the value of n: 5
   Within the first 5 elements of array, the first minimum value is stored at index 1.
```

Fig. 10.11 Program to return the position of the first minimum value in an array

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** .

No Arguments but Returns a Value

There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. A typical example is the **getchar** function declared in the header file **<stdio.h>** . We have used this function earlier in a number of places. The **getchar** function has no parameters but it returns an integer type data that represents a character.

We can design similar functions and use in our programs. Example:

```
int get_number(void);
main
{
int m = get_number( );
printf("%d",m);
}
int get_number(void)
{
int number;
scanf("%d", &number);
return(number);
}
```

It is possible to have functions with multiple return values, which can be achieved with the help of pointers. We will discuss this in detail in Chapter 13 when we discuss pointers.

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:

```
float ratio (int x, int y, int z);
int difference (int x, int y);
main( )
{
   int a, b, c;
   scanf("%d %d %d", &a, &b, &c);
   printf("%f \n", ratio(a,b,c));
}

float ratio(int x, int y, int z)
{
   if(difference(y, z))
      return(x/(y-z));
   else
      return(0.0);
}
int difference(int p, int q)
{
      if(p != q)
         return (1);
      else
         return(0);
}
```

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).

Note that the nesting does not mean defining one function within another. Doing this is illegal.

TEST YOUR SKILLS

Write a C program that uses function to determine whether the given string is palindrome or not. **[M]**

RECURSION

LO 10.4
Know the concept of 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 is 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(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 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.

> **TIP**
> Every recursive function should have a terminating logic within it, else a stack overflow error would occur at runtime.

Another classic example of a recursive operation is that of the Fibonacci series. The n th . term of a Fibonacci series is defined using the following recurrence equation:

$F1 = 1$

$F2 = 1$

$F_n = F_{n-1} + F_{n-2}$

Thus, the terms of the series are 1, 1, 2, 3, 5, 8, 13, 21, …..

The first two terms of the series are also sometimes defined as 0 and 1, respectively.

The C function required to generate the n^{th} . term of such a series is as follows:

```
int fibonacci (int n) {
        if (n <= 0) {
                printf("\nTh term number cannot be negative");
                return -999; /*error state*/
        }
        else if(n == 1 || n == 2)
                return 1;
        else
                return (fibonacci (n - 1) + fibonacci (n - 2));
}
```

Notice that the function 'fibonacci' has been recursively called twice: once for computing the $(n–1)^{th}$. term and then again for computing the $(n–2)^{th}$. term.

Ackermann function is another example of a well-defined, total computable function. This function grows faster than even exponential functions. This function has several forms. We proceed to discuss one of the often-used variations. The Ackermann function, A(m, n), for m>=0 and n>=0 may be defined as follows:

$$A(m,n) = \begin{cases} n+1, \text{ for } m = 0 \\ A(m-1, 1), \text{ for } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)), \text{ for } m > 0 \text{ and } n > 0 \end{cases}$$

One should be very careful when writing a C function for computing the elements of the Ackermann function. Due to the multiple-exponential growth rate of the function, even for small indices such as m=4, n=3 or m=4, n=4, the Ackermann function has humongous values. The operation is also extremely resource intensive. Thus, the data type chosen for housing such enormous values should be at the very least a long int or a long double. Shown here is a recursive implementation for computing A(m, n).

```
long double Ackermann (long double m, long double n) {
        if(m<0 || n<0) {
                printf("\nm, n should be >=0");
                return -999.0;     /*error state*/
        }
        else if(m == 0)
                return n + 1;
        else if(m > 0 && n == 0)
                return Ackermann (m – 1, 1);
        else
                return Ackermann (m – 1, Ackermann (m, n – 1));
}
```

Test Your Skills

Write a C program to find the GCD of two numbers using recursion. **[M]**

Passing Arrays to Functions

One-Dimensional Arrays

LO 10.5
Describe how arrays are passed to functions

Like the values of simple variables, it is also possible to pass the values of an array to a function. To pass a one-dimensional 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 the whole array **a** to the called function. The called function expecting this call must be appropriately defined. The **largest** function header might look like:

float largest(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(float a[ ], int n);
      float value[4] = {2.5,-4.75,1.2,3.67};
      printf("%f\n", largest(value,4));
}
float largest(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 array **value** become 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** .

In C, the name of the array represents the address of its first element. By passing the array name, we are, in fact, passing the address of the array to the called function. The array in the called function now refers to the same array stored in the memory. Therefore, any changes in the array in the called function will be reflected in the original array.

Passing addresses of parameters to the functions is referred to as *pass by address* (or pass by pointers). Note that we cannot pass a whole array by value as we did in the case of ordinary variables.

WORKED-OUT PROBLEM 10.6 **H**

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 give by

$$S.D = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\bar{x} - x_i)^2}$$

Where $\bar{x}$ is the mean of the values.

```
Program
            #include     <math.h>
            #define SIZE    5
            float std_dev(float a[], int n);
            float mean (float a[], int n);
            main( )
            {
                 float value[SIZE];
                 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(float a[], int n)
            {
                 int i;
                 float 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(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. 10.12 Passing of arrays to a function

A multifunction program consisting of **main, std_dev,** and **mean** functions is shown in Fig. 10.12 . **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 global section of the program.

Three Rules to Pass an Array to a Function

1. The function must be called by passing only the name of the array.
2. In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.
3. The function prototype must show that the argument is an array.

When dealing with array arguments, we should remember one major distinction. If a function changes the values of the elements of an array, 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. However, this does not apply when an individual element is passed on as argument. Program 10.6 highlights these concepts.

SEARCHING AND SORTING

Searching and sorting are two of the most common applications involving arrays and functions. Let us try to understand the operations involved in each of them.

Searching

Searching is an operation that is used when a look-up needs to be conducted on a set of data to locate a particular element. This search operation may be conducted in a number of ways. Linear search and binary search are two of the most popular searching techniques. The **Linear search** operation works by looking for the required element at each position of the array starting with the first and moving on in a sequence. The time complexity of such a search process is O(n). The complexity gives us an idea that to retrieve an element from the array the number of comparison operations required is of the order n.

Let us consider the following program for implementing the linear search algorithm. The program uses a function, *linear_search()* that returns the position of the element, if found and returns -1 otherwise.

```
#include <stdio.h>
#include <stdlib.h>
int linear_search(int [], int, int);
main(){
      int A[100], i, n, loc, d;
      printf("\nEnter the size of the list: ");
      scanf("%d", &n);
      if(n>100){
             printf("\nSize should be less or equal to 100");
             exit(0);
      }
      printf("\nEnter the elements of the array:");
      for(i = 0; i < n; i++){/*input the numbers*/
              printf("\nEnter A[%d]: ", i);
              scanf("%d", %A[i]);
      }
      printf("\nEnter the data that needs to be searched: ");
      scanf("%d", &d);
      loc = linear_search(A, n, d);
      if(loc == -1)
              printf("\nThe data %d is not present in the array");
      else printf("\nThe data %d is present at position %d of the array", d, loc);
}/*end of main*/
```

```
int linear_search(int A[], int n, int d){
     int i;
     for(i = 0; i < n; i++)
             if(A[i] == d)
                     return i; /*search successful*/
     return -1; /*search unsuccessful*/
}/*end of linear_search function*/
```

Below is a sample output of the program:

Enter the size of the list: 5

Enter the elements of the array:

Enter A[0] = 50

Enter A[1] = 30

Enter A[2] = 10

Enter A[3] = 60

Enter A[4] = 80

Enter the data that needs to be searched: 60

The data 60 is present at position 3 of the array

Another possible output is as follows:

Enter the size of the list: 5

Enter the elements of the array:

Enter A[0] = 50

Enter A[1] = 30

Enter A[2] = 10

Enter A[3] = 60

Enter A[4] = 80

Enter the data that needs to be searched: 75

The data 75 is not present in the array

Sorting

Sorting is an operation that is needed to arrange the data set in a specified order. Bubble sort, merge sort, heap sort, radix sort and quick sort are a few sorting algorithms. We now consider a well-known sorting algorithm named **Selection sort** to arrange a set of numbers in ascending order. The algorithm chooses the minimum among all elements and places it at the first position (the minimum should be at the first position). The remaining numbers in the list are now considered. The minimum among the ones left is chosen. This chosen number is the second minimum number in the list (the least of all was chosen the previous time). This second minimum number is now moved to the second position on the list. The list of numbers to be sorted, thus, has reduced by two. The process of selecting the minimum and moving it to its appropriate position is continued till all but one element have been sorted. The lone element that is left is sorted by itself. Let us consider an example and understand the sorting process. Below is a list of 5 numbers stored in an array. The numbers and their respective positions have been shown in the table below (Table 1). The first pass of the algorithm works by selecting the minimum number on the list (in this case 1 at position 4, shown in bold) and moving it to position 0 of the list. The changed list has been shown in Table 2. The shaded column indicates that the number in the position has been already sorted. The second pass of the algorithm works by considering the remaining four elements on the list (positions 1 through 4). In this pass once again the minimum of the remaining four elements is selected (2 at position 3, shown in bold in Table 2). This minimum is moved to position 1 of the list. The changed list is shown in Table 3. The first two columns have been shaded indicating that these two positions have been already sorted. The same process is repeated twice more. The progress has been shown in Tables 3 and 4 respectively. Table 5 shows that after the first four elements have been sorted, one element is left which happens to be the largest and should be appropriately present at the last position. Thus, to sort 5 numbers 4 passes were required. In general to sort, 'n' numbers n – 1 passes are required. Each pass requires the selection of the minimum number of the remaining numbers. The time complexity of this algorithm is $O(n^2)$.

Table 1 First Pass

Number	5	7	3	2	**1**
Position	0	1	2	3	**4**

Table 2 Second Pass

Number	1	7	3	**2**	5
Position	0	1	2	**3**	4

Table 3 Third Pass

Number	1	2	**3**	7	5

Position	0	1	**2**	3	4

Table 4 Fourth Pass

Number	1	2	3	7	**5**
Position	0	1	2	3	**4**

Table 5 Final list after four passes

Number	1	2	3	5	7
Position	0	1	2	3	4

Let us now consider a C program to sort a list of n numbers:

```c
#include <stdio.h>
#include <stdlib.h>

void selection_sort(int [], int);

main(){
      int A[100], int n, i;
      printf("\nEnter the size of the list: ");
      scanf("%d", &n);
      if(n>100){
              printf("\nSize should be less or equal to 100");
             exit(0);
      }
      printf("\nEnter the elements of the array:");
      for(i = 0; i < n; i++){/*input the numbers*/
              printf("\nEnter A[%d]: ", i);
              scanf("%d:, &A[i]);
      }
      selection_sort(A, n);/*call the sorting function*/
      printf("\nThe sorted list is as follows: ");
      for(i = 0; i < n; i++)
              printf("\nA[%d] = %d", i, A[i]);
}/*end of main*/

void selection_sort(int A[], int n){
      int i, j, pos, min;
      for(i = 0; i < n – 1; i++){
              pos = i;/*assume that first of remaining numbers is the minimum*/
              min = A[i];
              for(j = i +1; j < n; j++)
                      if(A[j] < min){/*change assumption if proven wrong*/
                              min = A[j];
                              pos = j;
                      }
              A[pos] = A[i];
              A[i] = min; /*move minimum to appropriate position*/
      }/*end of outer for*/
}/*end of selection_sort function*/
```

The following is a sample output of the above program:

Enter the size of the list: 5

Enter the elements of the array:

Enter A[0] = 5

Enter A[1] = 3

Enter A[2] = 10

Enter A[3] = 6

Enter A[4] = 8

The sorted list is as follows:

A[0] = 3

A[1] = 5

A[2] = 6

A[3] = 8

A[4] = 10

WORKED-OUT PROBLEM 10.7 **M**

Write a program that uses a function to sort an array of integers.

A program to sort an array of integers using the Bubble sort algorithm is shown in Fig. 10.13 .

Program

```
void sort(int m, int x[ ]);
main()
{
     int i;
     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");
}
void sort(int m, int 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. 10.13 Sorting of array elements using a function

Two-Dimensional Arrays

Like simple arrays, we can also pass multi-dimensional arrays to functions. The approach is similar to the one we did with one-dimensional arrays. The rules are simple.

1. The function must be called by passing only the array name.

2. In the function definition, we must indicate that the array has two-dimensions by including two sets of brackets.

3. The size of the second dimension must be specified.

4. The prototype declaration should be similar to the function header.

The function given below calculates the average of the values in a two-dimensional matrix.

```
double average(int x[][N], int M, int N)
{
   int i, j;
   double sum = 0.0;
   for (i=0; i<M; i++)
        for(j=1; j<N; j++)
        sum += x[i][j];
   return(sum/(M*N));
}
```

This function can be used in a main function as illustrated below:

```
main( )
{
     int M=3, N=2;
     double average(int [ ] [N], int, int);
     double mean;
     int matrix [M][N]=
             {
                {1,2},
                {3,4},
                {5,6}
             };

   mean = average(matrix, M, N);
   . . . . . .
   . . . . . .
}
```

Passing Strings to Functions

The strings are treated as character arrays in C and therefore the rules for passing strings to functions are very similar to those for passing arrays to functions.

Basic rules are:

1. The string to be passed must be declared as a formal argument of the function when it is defined. Example:

```
void display(char item_name[ ])
{
        . . . . . .
        . . . . . .
}
```

2. The function prototype must show that the argument is a string. For the above function definition, the prototype can be written as

void display(char str[ ]);

3. A call to the function must have a string array name without subscripts as its actual argument. Example:

display (names);

where **names** is a properly declared string array in the calling function.

Test Your Skills

Write a C program to compute the sum of N numbers by passing array to a function. [E]

The Scope, Visibility, and Lifetime of Variables

LO 10.6
Discuss the relevance of storage classes on scope, visibility and lifetime of variables

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.

In C not only do all variables have a data type, they also have a *storage class.* The following variable storage classes are most relevant to functions:

1. Local variables (Automatic variables)
2. Global variables (External variables)
3. Static variables.

4. Register variables.

We shall briefly discuss the *scope, visibility* , and *longevity* of each of the above class of variables. The *scope* of variable determines over what region 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 *visibility* refers to the accessibility of a variable from the memory.

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.

WORKED-OUT PROBLEM 10.8 **M**

Write a multifunction to illustrate how automatic variables work.

A program with two subprograms **function1** and **function2** is shown in Fig. 10.14 . **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 **function1** . 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.

```
Program
            void function1(void);
            void function2(void);
            main( )
            {
                 int m = 1000;
                 function2();

                 printf("%d\n",m);  /* Third output */
            }
```

```
            void function1(void)
            {
            int m = 10;
            printf("%d\n",m);  /* First output */
            }

            void function2(void)
            {
                 int m = 100;
                 function1();
                 printf("%d\n",m);  /* Second output */
            }

Output
            10
            100
            1000
```

Fig. 10.14 Working of automatic variables

There are two consequences of the scope and longevity of **auto** variables worth remembering. First, any variable local to **main** will be normally *alive* throughout the whole program, although it is *active* only in **main** . Secondly, during recursion, the nested variables are unique **auto** variables, a situation similar to function-nested **auto** variables with identical names.

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.

WORKED-OUT PROBLEM 10.9 E

Write a multifunction program to illustrate the properties of global variables.

A program to illustrate the properties of global variables is presented in Fig. 10.15 . 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. Further, since the value of **x** is directly available, we need not use **return(x)** statements in **fu n 1** and **fun3.** However, since **fun2** has a definition of **x,** it returns its local value of **x** and therefore uses a **return** statement. In **fun2** , the global **x** is not visible. The local **x** hides its visibility here.

Program

```
int fun1(void);
int fun2(void);
int fun3(void);
int x ;       /* global */
main( )
{
     x = 10 ;      /* global x */
     printf("x = %d\n", x);
     printf("x = %d\n", fun1());
     printf("x = %d\n", fun2());
     printf("x = %d\n", fun3());
}
fun1(void)
{
     x = x + 10 ;
}
int fun2(void)
{
int x ;            /* local */
x = 1 ;
return (x);
}
fun3(void)
{
     x = x + 10 ;    /* global x */
}
```

Output

```
x = 10
x = 20
x = 1
x = 30
```

Fig. 10.15 Illustration of properties 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.

Global Variables as Parameters

Since all functions in a program source file can access global variables, they can be used for passing values between the functions. However, using global variables as parameters for passing values poses certain problems.

- The values of global variables which are sent to the called function may be changed inadvertently by the called function.
- Functions are supposed to be independent and isolated modules. This character is lost, if they use global variables.
- It is not immediately apparent to the reader which values are being sent to the called function.
- A function that uses global variables suffers from reusability.

One other aspect of a global variable is that it is available only from the point of declaration to the end of the program. Consider a program segment as shown below:

```
main( )
{
     y = 5;
     . . . .
     . . . .
}
int y;       /* global declaration */
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.

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 */
```

TIP

External or global variables, though accessible from all functions, should be used sparingly in a program.

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 **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(void);
      extern float height [ ];
      . . . . .
      . . . . .
      print_out( );
}
void print_out(void)
{
      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.

Example:

```
extern float height[ ];
main( )
{
      int i;
      void print_out(void);
      . . . . .
      . . . . .
      print_out( );
}
void print_out(void)
{
      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(void);

is equivalent to

extern void print_out(void);

Function declarations outside of any function behave the same way as variable declarations.

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

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 up to 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.

WORKED-OUT PROBLEM 10.10 E

Write a program to illustrate the properties of a static variable.

The program in Fig. 10.16 explains the behaviour of a static variable.

```
Program
            void stat(void);
            main ( )
            {
                 int i;
                 for(i=1; i<=3; i++)
                 stat( );
            }
            void stat(void)
            {
                 static int x = 0;

                 x = x+1;
                 printf("x = %d\n", x);
            }
Output
            x = 1
            x = 2
            x = 3
```

Fig. 10.16 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 non-register 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 Variables

Storage Class	Where declared	Visibility (Active)	Lifetime (Alive)
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) **extern** and the file where originally declared as global.	Entire file plus other files where variable is declared	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

Nested Blocks

A set of statements enclosed in a set of braces is known a *block* or a *compound* statement. Note that all functions including the **main** use compound *statement.* A block can have its own declarations and other statements. It is also possible to have a block of such statements inside the body of a function or another block, thus creating what is known as *nested blocks* as shown below:

```
main( )
{
        int a = 20;
        int b = 10;
        . . . . .                                  Outer
        {                                          block
                int a = 0;               Inner
                int c = a + b;           block
                . . . . .
        }
        b = a;
}
```

When this program is executed, the value c will be 10, not 30. The statement b = a; assigns a value of 20 to **b** and not zero. Although the scope of **a** extends up to the end of **main** it is not “visible” inside the inner block where the variable **a** has been declared again. The inner **a** hides the visibility of the outer **a** in the inner block. However, when we leave the inner block, the inner **a** is no longer in scope and the outer **a** becomes visible again.

Remember, the variable **b** is not re-declared in the inner block and therefore it is visible in both the blocks. That is why when the statement

int c = a + b;

is evaluated, **a** assumes a values of 0 and **b** assumes a value of 10.

Although main's variables are visible inside the nested block, the reverse is not true.

Scope Rules

Scope

The region of a program in which a variable is available for use.

Visibility

The program's ability to access a variable from the memory.

Lifetime

The lifetime of a variable is the duration of time in which a variable exists in the memory during execution.

Rules of use

1. The scope of a global variable is the entire program file.
2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.
3. The scope of a formal function argument is its own function.
4. The lifetime (or longevity) of an **auto** variable declared in **main** is the entire program execution time, although its scope is only the **main** function.
5. The life of an **auto** variable declared in a function ends when the function is exited.
6. A **static** local variable, although its scope is limited to its function, its lifetime extends till the end of program execution.

7. All variables have visibility in their scope, provided they are not declared again.
8. If a variable is redeclared within its scope again, it loses its visibility in the scope of the redeclared variable.

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 10.17 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. 10.17 Use of extern in a multifile program

The multifile program shown in Fig. 10.18 can be modified as shown in Fig. 10.17 .

file1.c

```
int m; /* global variable */

main( )
{       {
        int i;
        . . . . .
}
function1( )
{
        int j;
        . . . . .
}
```

file2.c

```
extern int m;

function2( )

          int i;
          . . . . .
}
function3( )
{
int count;
. . . . .
}
```

Fig. 10.18 Another version of a multifile program

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.**

KEY CONCEPTS

- **ARGUMENTS:** Are the set of values that are passed to a function to enable the function to perform the desired task. **[LO 10.1]**
- **BLOCK STATEMENT:** Is a set of statements enclosed within a set of braces. **[LO 10.1]**
- **FUNCTION:** Is an independently coded subprogram that performs a specific task. **[LO 10.1]**
- **MODULAR PROGRAMMING:** Is a software development approach that organizes a large program into small, independent program segments called modules. **[LO 10.1]**
- **CALLING PROGRAM:** Is the program or function that calls another function. **[LO 10.2]**
- **FUNCTION BODY:** Contains the statement block for performing the required task.
- **FUNCTION TYPE:** Specifies the type of value that the **[LO 10.2]** function will return. **[LO 10.2]**
- **PARAMETER LIST:** Is a list of variables that will receive data values at the time of function call. **[LO 10.2]**
- **PROGRAM DEFINITION:** Is an independent program module that is written to perform specific task. It is also referred as function definition. **[LO 10.2]**
- **RECURSION:** Is a scenario where a function calls itself. **[LO 10.4]**
- **EXTERNAL VARIABLE:** Is a variable that is active throughout the program. It is also referred as global variable. **[LO 10.6]**
- **LOCAL VARIABLE:** Is a variable that is active only within a specific function or statement block. It is also referred as internal variable. **[LO 10.6]**

ALWAYS REMEMBER

- A function that returns a value can be used in expressions like any other C variable. **[LO 10.1]**

- A function that returns a value cannot be used as a stand-alone statement. **[LO 10.1]**
- Where more functions are used, they may be placed in any order. **[LO 10.1]**
- It is a syntax error if the types in the declaration and function definition do not match. **[LO 10.2]**
- It is a syntax error if the number of actual parameters in the function call do not match the number in the declaration statement. **[LO 10.2]**
- It is a logic error if the parameters in the function call are placed in the wrong order. **[LO 10.2]**
- Placing a semicolon at the end of header line is illegal. **[LO 10.2]**
- Forgetting the semicolon at the end of a prototype declaration is an error. **[LO 10.2]**
- A **return** statement can occur anywhere within the body of a function. **[LO 10.2]**
- A function definition may be placed either after or before the **main** function. **[LO 10.2]**
- A **return** statement is required if the return type is anything other than **void** . **[LO 10.3]**
- If a function does not return any value, the return type must be declared **void** . **[LO 10.3]**
- If a function has no parameters, the parameter list must be declared **void** . **[LO 10.3]**
- Using **void** as return type when the function is expected to return a value is an error. **[LO 10.3]**
- Trying to return a value when the function type is marked **void** is an error. **[LO 10.3]**
- Defining a function within the body of another function is not allowed. **[LO 10.3]**
- It is an error if the type of data returned does not match the return type of the function. **[LO 10.3]**

- It will most likely result in logic error if there is a mismatch in data types between the actual and formal arguments. **[LO 10.3]**
- Functions return integer value by default. **[LO 10.3]**
- A function without a return statement cannot return a value, when the parameters are passed by value. **[LO 10.3]**
- When the value returned is assigned to a variable, the value will be converted to the type of the variable receiving it. **[LO 10.3]**
- Function cannot be the target of an assignment. **[LO 10.3]**
- A function with void return type cannot be used in the right-hand side of an assignment statement. It can be used only as a stand-alone statement. **[LO 10.3]**
- A function can have more than one return statement. **[LO 10.3]**
- A recursive function must have a condition that forces the function to return without making the recursive call; otherwise the function will never return. **[LO 10.4]**
- It is illegal to use the name of a formal argument as the name of a local variable. **[LO 10.5]**
- Variables in the parameter list must be individually declared for their types. We cannot use multiple declarations (like we do with local or global variables). **[LO 10.5]**
- Use parameter passing by values as far as possible to avoid inadvertent changes to variables of calling function in the called function. **[LO 10.5]**
- Although not essential, include parameter names in the prototype declarations for documentation purposes. **[LO 10.5]**
- A global variable used in a function will retain its value for future use. **[LO 10.6]**
- A local variable defined inside a function is known only to that function. It is destroyed when the function is exited. **[LO 10.6]**
- A global variable is visible only from the point of its declaration to the end of the program. **[LO 10.6]**

- When a variable is redeclared within its scope either in a function or in a block, the original variable is not visible within the scope of the redeclared variable. **[LO 10.6]**
- A local variable declared **static** retains its value even after the function is exited. **[LO 10.6]**
- Static variables are initialized at compile time and therefore , they are initialized only once. **[LO 10.6]**

• Avoid the use of names that hide names in outer scope. **[LO 10.6]**

BRIEF CASES

1. Calculation of Area under a Curve **[LO 10.1, 10.2, 10.3, 10.6 M]**

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 area of individual trapezoids. The area of a trapezoid is given by

Area = 0.5 * (h1 + h2) * b

where h1 and h2 are the heights of two sides and b is the width as shown in Fig. 10.19 .

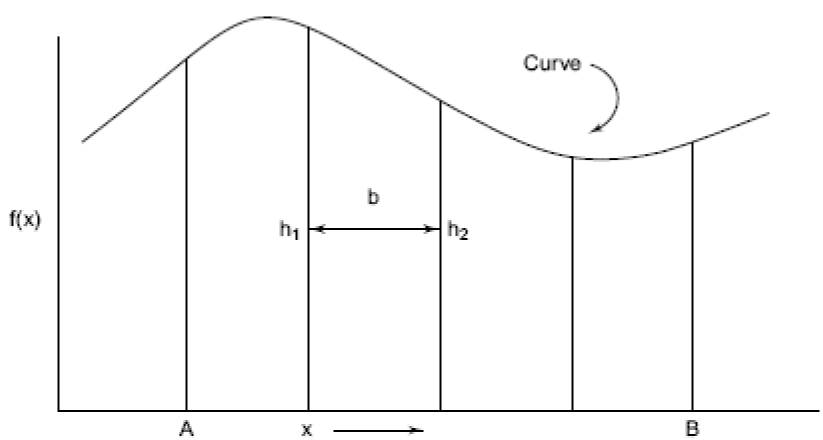


Fig. 10.19 Area under a curve

The program in Fig. 10.21 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

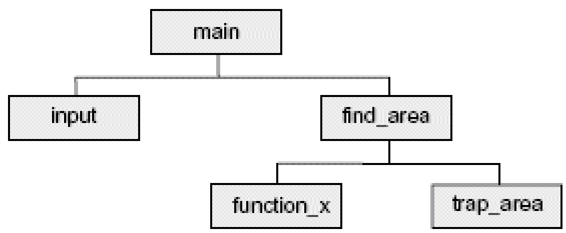


Fig. 10.20 Modular chart

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 in Fig. 10.20
.

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

```
#include <stdio.h>
float    start_point,              /* GLOBAL VARIABLES */
         end_point,
         total_area;
int      numtraps;
main( )
{
     void    input(void);
     float   find_area(float a,float b,int n); /* prototype */

     print("AREA UNDER A CURVE");
     input( );
     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);   /* prototype */
     float trap_area(float h1,float h2,float base);/*prototype*/
     base = (b-1)/n;
     lower = a;
     for(lower =a; lower <= b-base; lower = lower + base)
{
        h1   =  function_x(lower);
        h1   =  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 VARIABLE */
   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
```

```
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. 10.21 Computing area under a curve

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

10.1 Which of the following is true while passing array to a function? **[LO 10.5 M]**

(a) The function is called by passing the array name without the brackets

(b) The size of the array is not required to be specified in the formal parameter list

(c) The function prototype must show that the argument is an array

(d) All of the above

10.2 Which of the following is true about automatic variables? **[LO 10.6 M]**

(a) Automatic variables are declared inside a function in which they are to be utilised

(b) Automatic variables are created when the function is called and destroyed automatically when the function is exited

(c) A variable declared inside a function without storage class specification is, by default, an automatic variable

(d) All of the above

10.3 Which of the following is true about global variables? **[LO 10.6 M]**

(a) A global variable is available only from the point of declaration to the end of the program

(b) A global variable can be accessed by all functions in a program

(c) A global variable is initialised to Zero by default

(d) All of the above

10.4 For which of the following types of variables, the variable value persists till the end of the program.

(a) External (b) Static (c) Automatic (d) Register **[LO 10.6 E]**

10.5 In a C program, the function definition can be placed at which of the following places? **[LO 10.2 M]**

(a) Before the main function (b) After the main function

(c) Inside the main function (d) Both 1 and 2

10.6 If the number and type of a function declaration and function definition do not match, then **[LO 10.2 M]**

(a) The type of the function definition is considered

(b) The type of the function declaration is considered

(c) A compiler error is generated for the mismatch

(d) None of the above

10.7 Which of the following is true about function declaration? **[LO 10.2 E]**

(a) A function declaration statement is suffixed by a semi-colon

(b) It is not necessary to specify the variable names in the function prototype; only the argument type is sufficient

(c) A function declaration statement can be skipped as long as the function definition is present in the program

(d) All of the above

REVIEW QUESTIONS

10.1 State whether the following statements are *true* or *false* .

(a) Any name can be used as a function name. **[LO 10.1 E]**

(b) A function without a **return** statement is illegal. **[LO 10.2 E]**

(c) A function prototype must always be placed outside the calling function. **[LO 10.2 E]**

(d) The variable names used in prototype should match those used in the function definition. **[LO 10.2 E]**

(e) The return type of a function is **int** by default. **[LO 10.2 M]**

(f) When variable values are passed to functions, a copy of them are created in the memory. **[LO 10.2 M]**

(g) A function can be defined within the **main** function. **[LO 10.2 M]**

(h) A function can be defined and placed before the **main** function. **[LO 10.2 M]**

(i) C functions can return only one value under their function name. **[LO 10.3 E]**

(j) A function in C should have at least one argument. **[LO 10.3 E]**

(k) Only a **void** type function can have **void** as its argument. **[LO 10.3 E]**

(l) Program execution always begins in the main function irrespective of its location in the program. **[LO 10.3 M]**

(m) In parameter passing by pointers, the formal parameters must be prefixed with the symbo l * in their declarations. **[LO 10.3 M]**

(n) In parameter passing by pointers, the actual parameters in the function call may be variables or constants. **[LO 10.3 H]**

(o) An user-defined function must be called at least once; otherwise a warning message will be issued. **[LO 10.3 H]**

(p) A function can call itself. **[LO 10.4 E]**

(q) In passing arrays to functions, the function call must have the name of the array to be passed without brackets. **[LO 10.5 E]**

(r) In passing strings to functions, the actual parameter must be name of the string post-fixed with size in brackets. **[LO 10.5 M]**

(s) Global variables are visible in all blocks and functions in the program. **[LO 10.6 E]**

(t) Global variables cannot be declared as **auto** variables. **[LO 10.6 H]**

10.2 Fill in the blanks in the following statements.

(a) The parameters used in a function call are called ______. **[LO 10.1 E]**

(b) In prototype declaration, specifying ____ is optional. **[LO 10.2 E]**

(c) A _______ aids the compiler to check the matching between the actual arguments and the formal ones. **[LO 10.2 H]**

(d) In passing by pointers, the variables of the formal parameters must be prefixed with _______ in their declaration. **[LO 10.3 M]**

(e) By default, ______ is the return type of a C function. **[LO 10.3 M]**

(f) A function that calls itself is known as a _______ function. **[LO 10.4 E]**

(g) A variable declared inside a function is called _________. **[LO 10.6 E]**

(h) _________ refers to the region where a variable is actually available for use. **[LO 10.6 M]**

(i) If a local variable has to retain its value between calls to the function, it must be declared as _________. **[LO 10.6 M]**

(j) A variable declared inside a function by default assumes ______ storage class. **[LO 10.6 M]**

10.3 The **main** is a user-defined function. How does it differ from other user-defined functions? **[LO 10.1 H]**

10.4 Describe the two ways of passing parameters to functions. When do you prefer to use each of them? **[LO 10.5 M]**

10.5 What is prototyping? Why is it necessary? **[LO 10.2 E]**

10.6 Distinguish between the following:

(a) Actual and formal arguments **[LO 10.2 M]**

(b) & operator and * operator **[LO 10.3 M]**

(c) Global and local variables **[LO 10.6 E]**

(d) Automatic and static variables **[LO 10.6 M]**

(e) Scope and visibility of variables **[LO 10.6 M]**

10.7 Explain what is likely to happen when the following situations are encountered in a program. **[LO 10.3 H]**

(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 type of the corresponding formal argument.

(c) Data type of one of the arguments in a prototype does not match with the type of the corresponding formal parameter in the header line.

(d) The order of actual parameters in the function call is different from the order of formal parameters in a function where all the parameters are of the same type.

(e) The type of expression used in **return** statement does not match with the type of the function.

10.8 Which of the following prototype declarations are invalid? Why? **[LO 10.2 M]**

(a) int (fun) void;

(b) double fun (void)

(c) float fun (x, y, n);

(d) void fun (void, void);

(e) int fun (int a, b);

(f) fun (int, float, char);

(g) void fun (int a, int &b);

10.9 Which of the following header lines are invalid? Why? **[LO 10.2 M]**

(a) float average (float x, float y, float z);

(b) double power (double a, int n – 1)

(c) int product (int m, 10)

(d) double minimum (double x; double y;)

(e) int mul (int x, y)

(f) exchange (int *a, int *b)

(g) void sum (int a, int b, int &c)

10.10 A function to divide two floating point numbers is as follows: **[LO 10.3 M]**

```
        divide (float x, float 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)

(d) divide (2.0, 3.0)

10.11 What will be the effect on the above function calls if we change the header line as follows: **[LO 10.3 H]**

(a) int divide (int x, int y)

(b) double divide (float x, float y)

10.12 Determine the output of the following program? **[LO 10.3 H]**

```
int prod( int m, int n);
main ( )
{
     int x = 10;
     int y = 20;
     int p, q;
     p = prod (x,y);
     q = prod (p, prod (x,z));
     printf ("%d %d\n", p,q);
}
int prod( int a, int b)
{
     return (a * b);
}
```

10.13 What will be the output of the following program? **[LO 10.3 H]**

```
void test (int *a);
main ( )
{
     int x = 50;
     test ( &x);
     printf("%d\n", x);
}
void test (int *a);
{
     *a = *a + 50;
}
```

10.14 The function **test** is coded as follows: **[LO 10.3 M]**

```
int test (int number)
{
          int m, n = 0;
          while (number)
          {
                    m = number % 10;
                    if (m % 2)
                         n = n + 1;
                         number = number /10;          return (n);
          }                                    }
```

What will be the values of **x** and **y** when the following statements are executed?

int x = test (135);

int y = test (246);

10.15 Enumerate the rules that apply to a function call. **[LO 10.2 M]**

10.16 Summarize the rules for passing parameters to functions by pointers. **[LO 10.3 M]**

10.17 What are the rules that govern the passing of arrays to function? **[LO 10.5 M]**

10.18 State the problems we are likely to encounter when we pass global variables as parameters to functions. **[LO 10.6 H]**

DEBUGGING EXERCISES

10.1 Find errors, if any, in the following function definitions: **[LO 10.3 H]**

```
(a) void abc (int a, int b)
    {
                int c;
                . . . .
                return (c);
    }
(b) int abc (int a, int b)
    {
                . . . .
                . . . .
    }
(c) int abc (int a, int b)
    {
                double c = a + b;
                return (c);
    }
(d) void abc (void)
    {
                . . . .
                . . . .
                return;
    }
(e) int abc(void)
    {
                . . . .
                . . . .
                return;
    }
```

10.2 Find errors in the following function calls: **[LO 10.3 M]**

(a) void xyz ( );

(b) xyx (void);

(c) xyx (int x, int y);

(d) xyzz ( );

(e) xyz ( ) + xyz ( );

PROGRAMMING EXERCISES

10.1 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. Assume that **x** and **y** are defined as global variables. **[LO 10.2, 10.6 M]**

10.2 Write a function **space(x)** that can be used to provide a space of x positions between two output numbers. Demonstrate its application. **[LO 10.2 M]**

10.3 Use recursive function calls to evaluate **[LO 10.4 H]**

$$f(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

10.4 An n_order polynomial can be evaluated as follows: **[LO 10.5 H]**

$$P = (.....(((a_0 x + a_1)x + a_2)x + a_3)x + ... + a_n)$$

Write a function to evaluate the polynomial, using an array variable. Test it using a main program.

10.5 The Fibonacci numbers are defined recursively as follows: **[LO 10.3 M]**

$$F_1 = 1$$
$$F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2},\ n > 2$$

Write a function that will generate and print the first n Fibonacci numbers. Test the function for n = 5, 10, and 15.

10.6 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. **[LO 10.2 E]**

10.7 Write a function **prime** that returns 1 if its argument is a prime number and returns zero otherwise. **[LO 10.2 E]**

10.8 Write a function that will scan a character string passed as an argument and convert all lowercase characters into their uppercase equivalents. **[LO 10.5 H]**

10.9 Develop a top_down modular program to implement a calculator. The program should request the user to input two numbers and display one of the following as per the desire of the user: **[LO 10.2, 10.3 H]**

(a) Sum of the numbers

(b) Difference of the numbers

(c) Product of the numbers

(d) Division of the numbers

Provide separate functions for performing various tasks such as reading, calculating and displaying. Calculating module should call second level modules to perform the individual mathematical operations. The main function should have only function calls.

10.10 Develop a modular interactive program using functions that reads the values of three sides of a triangle and displays either its area or its perimeter as per the request of the user. Given the three sides a, b and c. **[LO 10.2, 10.3 H]**

$$\text{Perimeter} = a + b + c$$

$$\text{Area} = \sqrt{(s-a)(s-b)(s-c)}$$

where $$s = (a + b + c)/2$$

10.11 Write a function that can be called to find the largest element of an m by n matrix. **[LO 10.5 M]**

10.12 Write a function that can be called to compute the product of two matrices of size m by n and n by m. The main function provides the values for m and n and two matrices. **[LO 10.5 M]**

10.13 Design and code an interactive modular program that will use functions to a matrix of m by n size, compute column averages and row averages, and then print the entire matrix with averages shown in respective rows and columns. **[LO 10.3, 10.5 M]**

10.14 Develop a top-down modular program that will perform the following tasks: **[LO 10.3, 10.5 H]**

(a) Read two integer arrays with unsorted elements.

(b) Sort them in ascending order

(c) Merge the sorted arrays

(d) Print the sorted list

Use functions for carrying out each of the above tasks. The main function should have only function calls.

10.15 Develop your own functions for performing following operations on strings: **[LO 10.3, 10.5 M]**

(a) Copying one string to another

(b) Comparing two strings

(c) Adding a string to the end of another string

Write a driver program to test your functions.

10.16 Write a program that invokes a function called **find( )** to perform the following tasks: **[LO 10.5 M]**

(a) Receives a character array and a single character.

(b) Returns 1 if the specified character is found in the array, 0 otherwise.

10.17 Design a function **locate ( )** that takes two character arrays **s1** and **s2** and one integer value **m** as parameters and inserts the string **s2** into **s1** immediately after the index **m** . **[LO 10.5 H]**

Write a program to test the function using a real-life situation. (Hint: s2 may be a missing word in s1 that represents a line of text).

10.18 Write a function that takes an integer parameter **m** representing the month number of the year and returns the corresponding name of the month. For instance, if m = 3, the month is March. Test your program. **[LO 10.3 M]**

10.19 In preparing the calendar for a year we need to know whether that particular year is leap year or not. Design a function **leap( )** that receives the year as a parameter and returns an appropriate message. **[LO 10.2 E]**

What modifications are required if we want to use the function in preparing the actual calendar?

10.20 Write a function that receives a floating point value x and returns it as a value rounded to two nearest decimal places. For example, the value 123.4567 will be rounded to 123.46 (Hint: Seek help of one of the math functions available in math library). **[LO 10.2 M]**

INTERVIEW QUESTIONS

10.1 Is it possible to execute certain instructions after the main function has ended? **[LO 10.1 H]**

10.2 What is the difference between formal arguments and actual arguments? **[LO 10.2 E]**

10.3 What is the difference between 'call by value' and 'call by reference'? **[LO 10.3 M]**

10.4 Write the code for a recursive function that takes two integers as input parameters and returns their greatest common divisor (GCD). **[LO 10.4 M]**

10.5 What is a comma operator? **[LO 10.2 E]**

10.6 How will you print the string 'Hello World' in green color? **[LO 10.1 E]**

10.7 What is the default return type of a function defined without a return type? **[LO 10.2 M]**

Chapter 11

Structures and Unions

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

LO 11.1 Explain how structures are used in a program
LO 11.2 Describe how structure variables and members are manipulated
LO 11.3 Discuss structures and arrays
LO 11.4 Illustrate nested structures and 'structures and functions'
LO 11.5 Determine how structures and unions differ in terms of their storage technique

INTRODUCTION

W e have 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, we cannot use an array if we want to represent a collection of data items of different types using a single name. Fortunately, C supports a constructed data type known as *structures,* a mechanism 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. More examples of such structures are:

time	:	seconds, minutes, hours
date	:	day, month, year
book	:	author, title, price, year
city	:	name, country, population
address	:	name, door-number, street, city
inventory	:	item, stock, value
customer	:	name, telephone, city, category

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. This chapter is devoted to the study of structures and their applications in program development. Another related concept known as *unions* is also discussed.

DEFINING A STRUCTURE

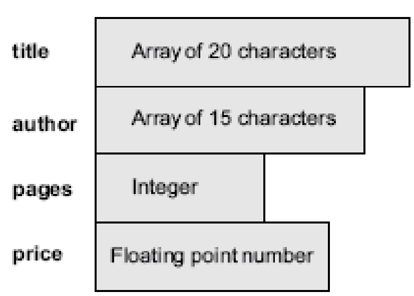
Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book database consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

LO 11.1
Explain how structures are used in a program

```
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 data fields, namely **title, author, pages** , and **price** . These fields are called *structure elements* or *members*. Each member may belong to a different 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 definition 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;
            ----        ----
            ----        ----
};
```

In defining a structure you may note the following syntax:

TIP
Structures can store heterogeneous data types to represent real-life entities.

1. The template is terminated with a semicolon.
2. While the entire definition 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.

Arrays Vs Structures

Both the arrays and structures are classified as structured data types as they provide a mechanism that enable us to access and manipulate data in a relatively easy manner. But they differ in a number of ways which are as follows:

1. An array is a collection of related data elements of same type. Structure can have elements of different types.
2. An array is a derived data type whereas a structure is a programmer-defined one.
3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it. But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

DECLARING STRUCTURE VARIABLES

After defining a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of any other data types. It includes the following elements:

1. The keyword **struct.**
2. The structure tag name.
3. List of variable names separated by commas.
4. A terminating semicolon.

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.** When the compiler comes across a declaration statement, it reserves memory space for the structure variables. It is also allowed to combine both the structure definition 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 here. For example:

```
struct
{
   ........
   ........
   ........
} book1, book2, book3;
```

declares **book1** , **book2** and **book3** as structure variables representing three books, but does not include a tag name. However, this approach is not recommended for the simple reason that a structure without a tag name cannot be referenced from anywhere else.

Type-Defined Structures

We can use the keyword **typedef** to define a structure as follows:

```
typedef struct
{  . . . . .
   type member1;
   type member2;
   . . . . .
   . . . . .
}  type_name;
```

The type_name represents structure definition associated with it and therefore, can be used to declare structure variables as shown below:

type_name variable1, variable2, ... ;

Remember that (1) the name *type_name* is the type definition name, not a variable and (2) we cannot define a variable with *typedef* declaration.

WORKED-OUT PROBLEM 11.1 **M**

Explain how complex number can be represented using structures. Write two C functions: one to return the sum of to complex numbers passed as parameters.

A complex number has two parts: real and imaginary. Structures can be used to realize complex numbers in C, as shown below:

```
struct complex /*Declaring the complex number datatype using structure*/
{
  double real;/*Real part*/
  double img;/*Imaginary part*/
};
```

Function to return the sum of two complex numbers

```
struct complex add(struct complex c1, struct complex c1)
{
  struct complex c3;
  c3.real=c1.real+c2.real;
  c3.img=c1.img+c2.img;
  return(c3);
}
```

Function to return the product of two complex numbers

```
struct complex product(struct complex c1, struct complex c1)
{
  struct complex c3;
  c3.real=c1.real*c2.real-c1.img*c2.img;
  c3.img=c1.real*c2.img+c1.img*c2.real;
  return(c3);
}
```

ACCESSING STRUCTURE MEMBERS

We can access and 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 whereas 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 = 120.50;
```

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.

WORKED-OUT PROBLEM 11.2 E

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.

Structure definition along with the program is shown in Fig. 11.1 . 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.

```
Program
            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 %f\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. 11.1 Defining and accessing structure members

STRUCTURE INITIALIZATION

Like any other data type, a structure variable can be initialized at compile time.

```
main()
{
     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;
};
struct st_record student1 - { 60, 180.75 };
struct st_record student2 - { 53, 170.60 };
.....
.....
}
```

Another method is to initialize a structure variable outside the function as shown below:

```
struct st_record
{
     int weight;
     float height;
}    student1 - {60, 180.75};
main()
{
     struct st_record student2 - {53, 170.60};
     .....
     .....
}
```

C language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual variables.

Note that the compile-time initialization of a structure variable must have the following elements:

1. The keyword **struct** .
2. The structure tag name.
3. The name of the variable to be declared.
4. The assignment operator =.
5. A set of values for the members of the structure variable, separated by commas and enclosed in braces.
6. A terminating semicolon.

Rules for Initializing Structures

There are a few rules to keep in mind while initializing structure variables at compile-time which are as follows:

1. We cannot initialize individual members inside the structure template.
2. The order of values enclosed in braces must match the order of members in the structure definition.
3. It is permitted to have a partial initialization.
   We can initialize only the first few members and leave the remaining blank. The uninitialized members should be only at the end of the list.
4. The uninitialized members will be assigned default values as follows:
   - Zero for integer and floating point numbers.

TIP

The rules applicable to assign or input values to a structure member are same as the ones applicable on the data type of the member. The only thing to keep in mind is that the member would now be qualified by the structure variable name and the (.) operator.

- '\0' for characters and strings.

COPYING AND COMPARING STRUCTURE VARIABLES

LO 11.2
Describe how structure variables and members are manipulated

Two variables of the same structure type can be copied the same way as ordinary variables. If **person1** and **person2** belong to the same structure, then the following statements are valid:

person1 = person2;

person2 = person1;

However, the statements such as

person1 == person2

person1 != person2

a re not permitted. C does not permit any logical operations on structure variables. In case, we need to compare them, we may do so by comparing members individually.

TIP
The relational operators are not applicable for structure members.

WORKED-OUT PROBLEM 11.3 M

Write a program to illustrate the comparison of structure variables.

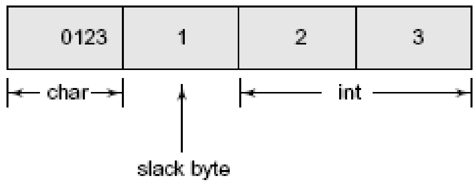
T he program shown in Fig. 11.2 illustrates how a structure variable can be copied into another of the same type. It also performs member-wise comparison to decide whether two structure variables are identical.

```
Program
            struct class
            {
                 int  number;
                 char name[20];
                 float marks;
            };

            main()
            {
                 int  x;
                 struct class student1 = {111,"Rao",72.50};
                 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. 11.2 Comparing and copying structure variables

Word Boundaries and Slack Bytes

Computer stores structures using the concept of "word boundary". The size of a word boundary is machine dependent. In a computer with two bytes word boundary, the members of a structure are stored left_aligned on the word boundary, as shown below. A character data takes one byte and an integer takes two bytes. One byte between them is left unoccupied. This unoccupied byte is known as the *slack byte.*



When we declare structure variables, each one of them may contain slack bytes and the values stored in such slack bytes are undefined. Due to this, even if the members of two variables are equal, their structures do not necessarily compare equal. C, therefore, does not permit comparison of structures. However, we can design our own function that could compare individual members to decide whether the structures are equal or not.

Operations on Individual Members

As pointed out earlier, the individual members are identified using the member operator, the *dot* . A member with the *dot operator* along with its structure variable can be treated like any other variable name and therefore can be manipulated using expressions and operators. Consider the program in Fig. 11.2 . We can perform the following operations:

```
if (student1.number -- 111)
   student1.marks +- 10.00;
float sum - student1.marks + student2.marks;
student2.marks * - 0.5;
```

We can also apply increment and decrement operators to numeric type members. For example, the following statements are valid:

student1.number ++;

++ **student1.number;**

The precedence of the *member* operator is higher than all *arithmetic* and *relational* operators and therefore no parentheses are required.

Three Ways to Access Members

We have used the dot operator to access the members of structure variables. In fact, there are two other ways. Consider the following structure:

```
typedef struct
{
        int x;
        int y;
} VECTOR;
VECTOR v, *ptr;
ptr = & v;
```

The identifier **ptr** is known as **pointer** that has been assigned the address of the structure variable n. Now, the members can be accessed in the following three ways:

- using dot notation : v **.** x
- using indirection notation : (*ptr) **.** x
- using selection notation : ptr –> x

The second and third methods will be considered in Chapter 12.

Test Your Skills

Write a C program for storing student details using structures. [E]

Arrays of Structures

LO 11.3
Discuss structures and arrays

We use structures to describe the format of a number of related variables. For example, in analyzing 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()
{
   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 = 65;
   ....
   ....
student[2].subject3 = 71;
```

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. Remember, each element of **student** array is a structure variable with three 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. 11.3 .

student [0].subject 1	45
.subject 2	68
.subject 3	81
student [1].subject 1	75
.subject 2	53
.subject 3	69
student [2].subject 1	57
.subject 2	36
.subject 3	71

Fig. 11.3 The array student inside memory

WORKED-OUT PROBLEM 11.4

M

For the **student** array discussed above, write a program to calculate the subject-wise and student-wise totals and store them as a part of the structure.

The program is shown in Fig. 11.4 . 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

```
struct marks
 {
     int  sub1;
     int  sub2;
     int  sub3;
     int  total;
  };
  main()
  {
     int  i;
     struct marks student[3] =  {{45,67,81,0},
                                 {75,53,69,0},
                                 {57,36,71,0}};
     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. 11.4 Arrays of structures: Illustration of subscripted structure variables

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-dimensional 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.

WORKED-OUT PROBLEM 11.5 **E**

Rewrite the program of Program 11.4 using an array member to represent the three subjects.

The modified program is shown in Fig. 11.5 . You may notice that the use of array name for subjects has simplified in code.

```
Program
            main()
            {
                struct  marks
                {
                    int  sub[3];
                    int  total;
            };
                struct marks student[3] =
                {45,67,81,0,75,53,69,0,57,36,71,0};

                struct marks total;
                int  i,j;
```


```
Program
            main()
            {
                struct  marks
                {
                    int  sub[3];
                    int  total;
            };
                struct marks student[3] =
                {45,67,81,0,75,53,69,0,57,36,71,0};

                struct marks total;
                int  i,j;
```


Fig. 11.5 Use of subscripted members arrays in structures

STRUCTURES WITHIN STRUCTURES

LO 11.4
Illustrate nested structures and 'structures and functions'

S tructures 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;
   char  department;
   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;
   char department;
   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 are invalid:

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;
     char department;
     struct pay allowance;
     struct pay arrears;
  };
  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.

Note *C permits nesting upto 15 levels. However, C99 allows 63 levels of nesting.*

STRUCTURES AND FUNCTIONS

We know that the main philosophy of C language is the use of functions. And 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.

1. 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.
2. 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.
3. 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 function. 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(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 back to the calling function. 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 functions must be declared in the calling function appropriately.

WORKED-OUT PROBLEM 11.6 M

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. 11.6 . 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

```
/*   Passing a copy of the entire structure        */
struct stores
{
     char    name[20];
     float   price;
     int     quantity;
};
struct stores update (struct stores product, float p, int q);
float mul (struct stores stock);
main()
{
   float    p_increment, value;
   int       q_increment;

   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("Updated values of item\n\n");
   printf("Name      : %s\n",item.name);
   printf("Price     : %f\n",item.price);
   printf("Quantity  : %d\n",item.quantity);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
   value  = mul(item);
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

   printf("\nValue of the item  =  %f\n", value);
}
struct stores update(struct stores product, float p, int q)
{
     product.price += p;
     product.quantity += q;
     return(product);
}
```

```
float mul(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. 11.6 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.

TEST YOUR SKILLS

Write a C program to demonstrate nesting of structures. [E]

UNIONS

LO 11.5
Determine how structures and unions differ in terms of their storage technique

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.

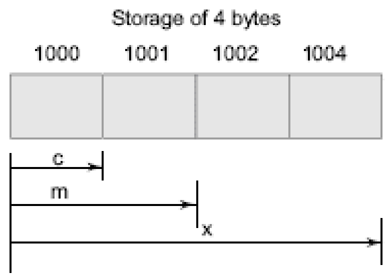


Fig. 11.7 Sharing of a storage locating by union members

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 11.7 shows how all the three variables share the same address. This assumes that a float variable requires 4 bytes of storage and an int variable requires 2 bytes of space.

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 supersedes 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.

Unions may be initialized when the variable is declared. But, unlike structures, it can be initialized only with a value of the same type as the first union member. For example, with the preceding, the declaration

union item abc = {100};

is valid but the declaration

union item abc = {10.75};

is invalid. This is because the type of the first member is **int.** Other members can be initialized by either assigning values or reading from the keyboard.

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

sizeof(y)/sizeof(x)

would give the number of elements in the array **y.**

Bit Fields

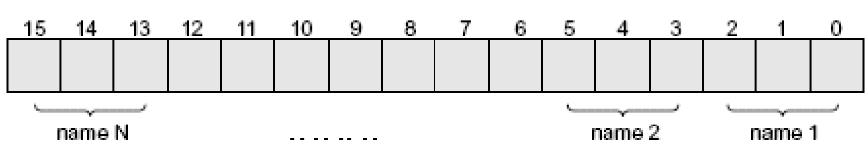
So far, we have been using integer fields of size 16 bits to store data. There are occasions where data items require much less than 16 bits space. In such cases, we waste memory space. Fortunately, C permits us to use small *bit fields* to hold data items and thereby to pack several data items in a word of memory. Bit fields allow direct manipulation of a string of preselected bits.

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.

T he 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 to 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.

Note *Other related topics such as 'Structures with Pointers' and 'Structures and Linked Lists' are discussed in Chapter 12 and Chapter 13, respectively.*

KEY CONCEPTS

- **ARRAY:** It is a fixed-size sequenced collection of elements of the same data type. **[LO 11.1]**
- **DOT OPERATOR:** This links a structure variable with a structure member. It is used to read/write member values. **[LO 11.1]**
- **STRUCTURE:** This is a user-defined data type that allows different data types to be combined together to represent a data record. **[LO 11 .1]**
- **UNION:** It is similar to a structure in syntax but differs in storage technique. Unlike structures, union members use the same memory location for storing all member values. **[LO 11.5]**
- **BIT FIELD:** This refers to a set of adjacent bits with size ranging from 1 to 16 bits. **[LO 11.5]**

ALWAYS REMEMBER

- Remember to place a semicolon at the end of definition of structures and unions. **[LO 11.1]**
- We can declare a structure variable at the time of definition of a structure by placing it after the closing brace but before the semicolon. **[LO 11.1]**
- Do not place the structure tag name after the closing brace in the definition. That will be treated as a structure variable. The tag name

must be placed before the opening brace but after the keyword **struct.** **[LO 11.1]**

- When we use **typedef** definition, the *type_name* comes after the closing brace but before the semicolon. **[LO 11.1]**
- We cannot declare a variable at the time of creating a **typedef** definition. We must use the *type_name* to declare a variable in an independent statement. **[LO 11.1]**
- It is an error to use a structure variable as a member of its own **struct** type structure. **[LO 11.1]**
- Declaring a variable using the tag name only (without the keyword struct) is an error. **[LO 11.1]**
- It is illegal to refer to a structure member using only the member name. **[LO 11.1]**
- When using **scanf** for reading values for members, we must use address operator & with non-string members. **[LO 11.1]**
- Always provide a structure tag name when creating a structure. It is convenient to use tag name to declare new structure variables later in the program. **[LO 11.1]**
- Use short and meaningful structure tag names. **[LO 11.1]**
- Avoid using same names for members of different structures (although it is not illegal). **[LO 11.1]**
- It is an error to compare two structure variables. **[LO 11.2]**
- Assigning a structure of one type to a structure of another type is an error. **[LO 11.2]**
- When accessing a member with a pointer and dot notation, parentheses are required around the pointer, like (*ptr).number. **[LO 11.2]**
- The selection operator (– >) is a single token. Any space between the symbols – and > is an error. **[LO 11.2]**
- Forgetting to include the array subscript when referring to individual structures of an array of structures is an error. **[LO 11.3]**

- When structures are nested, a member must be qualified with all levels of structures nesting it. **[LO 11.4]**
- Passing structures to functions by pointers is more efficient than passing by value. (Passing by pointers are discussed in Chapter 1 2 .) **[LO 11.4]**
- A union can store only one of its members at a time. We must exercise care in accessing the correct member. Accessing a wrong data is a logic error. **[LO 11.5]**
- It is an error to initialize a union with data that does not match the type of the first member. **[LO 11.5]**
- We cannot take the address of a bit field. Therefore, we cannot use **scanf** to read values in bit fields. We can neither use pointer to access the bit fields. **[LO 11.5]**
- Bit fields cannot be arrayed. **[LO 11.5]**

BRIEF CASES

1. Book Shop Inventory **[LO 11.1, 11.2, 11.3, 11.4, 11.5 M]**

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. 11.8 . 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

```c
#include   <stdio.h>
#include   <string.h>
struct  record
{
     char    author[20];
     char    title[30];
     float    price;
     struct

     {
        char    month[10];
        int     year;
     }
     date;
     char    publisher[10];
     int     quantity;
};
   int look_up(struct record table[],char s1[],char s2[],int m);
   void get (char string [ ] );
   main()
{
     char title[30], author[20];
     int  index, no_of_records;
     char response[10], quantity[10];
     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",1988,"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  */
```

```c
        {
              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");
    }
       void get(char string [] )
    {
       char  c;
       int  i = 0;
       do
       {
            c = getchar();
            string[i++] = c;
    }
    while(c != '\n');
    string[i-1] = '\0';
    }

    int look_up(struct record table[],char s1[],char 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. 11.8 Program of bookshop inventory

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

11.1 Which of the following is the correct way of accessing the members of a structure variable? **[LO 11.3 H]**

(a) Using dot notation, v.x

(b) Using indirection notation, (*ptr).x

(c) Using selection notation, ptr –> x

(d) All of the above

11.2 Which of the following is true about structures and functions? **[LO 11.4 M]**

(a) A pointer to a structure can be passed as a function argument.

(b) It is possible to return a structure variable from a function.

(c) The structure variables used in the actual argument and the formal argument must be of the same struct type.

(d) All of the above

11.3 Which of the following is incorrect about Union? **[LO 11.5 M]**

(a) The size of a union variable is equal to the size of its largest size member.

(b) Unlike structures, a union can be initialized only with a value of the same type as the first union member.

(c) Unlike structures, union does not support nesting.

(d) At any given point of time, a union variable can be used to store any one of its member types.

11.4 Which of the following operators can be used with structure variables? **[LO 11.2 M]**

(a) =

(b) +

(c) ==

(d) !=

11.5 What will be the output of following code:

```
void main()
{
 union U
 {
  int i;
  int j;
 };

 union U test-5;
printf("%d %d",test.i,test.j);
}
```

(a) 5, 0

(b) 5, garbage value

(c) 5, 5

(d) Compilation error

REVIEW QUESTIONS

11.1 State whether the following statements are *true* or *false* .

(a) A **struct** type in C is a built-in data type. **[LO 11.1 E]**

(b) The tag name of a structure is optional. **[LO 11.1 E]**

(c) Structures may contain members of only one data type. **[LO 11.1 E]**

(d) The keyword **typedef** is used to define a new data type. **[LO 11.1 E]**

(e) A structure variable is used to declare a data type containing multiple fields. **[LO 11.1 M]**

(f) It is legal to copy a content of a structure variable to another structure variable of the same type. **[LO 11.1 M]**

(g) Pointers can be used to access the members of structure variables. **[LO 11.1 H]**

(h) In accessing a member of a structure using a pointer p, the following two are equivalent: (*p) **.** member_name and p –> member_name **[LO 11.1 H]**

(i) We can perform mathematical operations on structure variables that contain only numeric type members. **[LO 11.2 M]**

(j) An array cannot be used as a member of a structure. **[LO 11.3 E]**

(k) A member in a structure can itself be a structure. **[LO 11.4 E]**

(l) Structures are always passed to functions by pointers. **[LO 11.4 H]**

(m) A union may be initialized in the same way a structure is initialized. **[LO 11.5 E]**

(n) A union can have another union as one of the members. **[LO 11.5 H]**

(o) A structure cannot have a union as one of its members. **[LO 11.5 H]**

11.2 Fill in the blanks in the following statements:

(a) The name of a structure is referred to as _______________. **[LO 11.1 E]**

(b) The variables declared in a structure definition are called its _______________. **[LO 11.1 E]**

(c) The ______________ can be used to create a synonym for a previously defined data type. **[LO 11.1 M]**

(d) The selection operator –> requires the use of a __________________ to access the members of a structure. **[LO 11.1 H]**

(e) A________________ is a collection of data items under one name in which the items share the same storage. **[LO 11.5 E]**

(f) In case of partial initialization of a structure variable through braces, the integer and floating point members are initialized to ________ value. **[LO 11.1 E]**

(g) In the memory storage of a structure variable, the unoccupied byte is known as _____________ byte. **[LO 11.2 M]**

(h) The size of a union variable is equal to the size of its _____________ member. **[LO 11.5 E]**

11.3 A structure tag name **abc** is used to declare and initialize the structure variables of type **struct abc** in the following statements. Which of them are incorrect? Why? Assume that the structure **abc** has three members, **int, float** , and **char** in that order. **[LO 11.1, 11.2, 11.3 E]**

(a) struct a,b,c;

(b) struct abc a,b,c

(c) abc x,y,z;

(d) struct abc a[ ];

(e) struct abc a = { };

(f) struct abc = b, { 1+2, 3.0, “xyz”}

(g) struct abc c = {4,5,6};

(h) struct abc a = 4, 5.0, “xyz”;

11.4 Given the declaration **[LO 11.1, 11.2 M]**

struct abc a,b,c;

which of the following statements are legal?

(a) scanf (“%d, &a);

(b) printf (“%d”, b);

(c) a = b;

(d) a = b + c;

(e) if (a>b)

. . . . .

11.5 Given the declaration **[LO 11.3 M]**

```
struct item_bank
{
        int number;
        double cost;
};
```

which of the following are correct statements for declaring one dimensional array of structures of type **struct item_bank?**

(a) int item_bank items[10];

(b) struct items[10] item_bank;

(c) struct item_bank items (10);

(d) struct item_bank items [10];

(e) struct items item_bank [10];

11.6 Given the following declaration **[LO 11.1, 11.3 M]**

```
typedef struct abc
{
       char x;
       int y;
       float z[10];
} ABC;
```

State which of the following declarations are invalid? Why?

(a) struct abc v 1;

(b) struct abc v 2[10];

(c) struct ABC v 3;

(d) ABC a,b,c;

(e) ABC a[10];

11.7 How does a structure differ from an array? **[LO 11.3 E]**

11.8 Explain the meaning and purpose of the following: **[LO 11.1, 11.5 M]**

(a) Template

(b) **struct** keyword

(c) **typedef** keyword

(d) **sizeof** operator

(e) Tag name

11.9 Explain what is wrong in the following structure declaration: **[LO 11.1 E]**

```
struct
{
       int number;
       float price;
}
main( )
{
       . . . . .
       . . . . .
}
```

11.10 When do we use the following? **[LO 11.5 M]**

(a) Unions

(b) Bit fields

(c) The **sizeof** operator

11.11 What is meant by the following terms?

(a) Array of structures **[LO 11.3 E]**

(b) Nested structures **[LO 11.4 M]**

(c) Unions **[LO 11.5 M]**

Give a typical example of use of each of them.

11.12 Describe with examples, the different ways of assigning values to structure members. **[LO 11.1 M]**

11.13 State the rules for initializing structures. **[LO 11.1 E]**

11.14 What is a 'slack byte'? How does it affect the implementation of structures? **[LO 11.2 M]**

11.15 Describe three different approaches that can be used to pass structures as function arguments. **[LO 11.4 H]**

11.16 What are the important points to be considered when implementing bit-fields in structures? **[LO 11.5 M]**

11.17 Define a structure called **complex** consisting of two floating-point numbers **x** and **y** and declare a variable **p** of type **complex** . Assign initial values 0.0 and 1.1 to the members. **[LO 11.1 M]**

11.18 What will be the output of the following program? **[LO 11.5 H]**

```
main ( )
{
     union x
     {
               int a;
               float b;
               double c ;
     };
     printf("%d\n", sizeof(x));
        a.x - 10;
     printf("%d%f%f\n", a.x, b.x, c.x);
        c.x - 1.23;
     printf("%d%f%f\n", a.x, b.x, c.x);
}
```

DEBUGGING EXERCISES

11.1 Given the structure definitions and declarations **[LO 11.1, 11.2 M]**

```
struct abc
{
        int a;
        float b;
};
struct xyz
{
        int x;
        float y;
};
abc a1, a2;
xyz x1, x2;
```

find errors, if any, in the following statements:

(a) a1 = x1;

(b) abc.a1 = 10.75;

(c) int m = a + x;

(d) int n = x1.x + 10;

(e) a1 = a2;

(f) if (a.a1 > x.x1) . . .

(g) if (a1.a < x1.x) . . .

(h) if (x1 != x2) . . .

11.2 What is the error in the following program? **[LO 11.1 M]**

```
typedef struct product
{
     char name [ 10 ];
     float price ;
} PRODUCT products [ 10 ];
```

PROGRAMMING EXERCISES

11.1 Define a structure data type called **time_struct** containing three members integer **hour,** integer **minute** and integer **second** . Develop a program that would assign values to the individual members and display the time in the following form: **[LO 11.1 M]**

16:40:51

11.2 Modify the above program such that a function is used to input values to the members and another function to display the time. **[LO 11.1, 11.4 M]**

11.3 Design a function **update** that would accept the data structure designed in Exercise 11.1 and increments time by one second and returns the new time. (If the increment results in 60 seconds, then the second member is set to zero and the minute member is incremented by one. Then, if the result is 60 minutes, the minute member is set to zero and the hour member is incremented by one. Finally when the hour becomes 24, it is set to zero.) **[LO 11.1, 11.2, 11.4 M]**

11.4 Define a structure data type named **date** containing three integer members **day** , **month** , and **year.** Develop an interactive modular program to perform the following tasks: **[LO 11.1, 11.2, 11.4 M]**

(a) To read data into structure mem bers by a function

(b) To validate the date entered by another function

(c) To print the date in the format

April 29, 2002

by a third function.

The input data should be three integers like 29, 4, and 2002 corresponding to day, month, and year. Examples of invalid data:

31, 4, 2002 – April has only 30 days

29, 2, 2002 – 2002 is not a leap year

11.5 Design a function **update** that accepts the **date** structure designed in Exercise 11.4 to increment the date by one day and return the new date. The following rules are applicable: **[LO 11.1, 11.2, 11.4 M]**

(a) If the date is the last day in a month, month should be incremented

(b) If it is the last day in December, the year should be incremented

(c) There are 29 days in February of a leap year

11.6 Modify the input function used in Exercise 11.4 such that it reads a value that represents the date in the form of a long integer, like 19450815 for the date 15-8-1945 (August 15, 1945) and assigns suitable values to the members **day, month** , and **year.**

Use suitable algorithm to convert the long integer 19450815 into year, month and day. **[LO 11.1, 11.2, 11.4 M]**

11.7 Add a function called **nextdate** to the program designed in Exercise 11.4 to perform the following task: **[LO 11.1, 11.2, 11.4 H]**

(a) Accepts two arguments, one of the structure **data** containing the present date and the second an integer that represents the number of days to be added to the present date.

(b) Adds the days to the present date and returns the structure containing the next date correctly.

Note that the next date may be in the next month or even the next year.

11.8 Use the **date** structure defined in Exercise 11.4 to store two dates. Develop a function that will take these two dates as input and compares them. **[LO 11.1, 11.2, 11.4 M]**

(a) It returns 1, if the **date1** is earlier than **date2**

(b) It returns 0, if **date1** is later date

11.9 Define a structure to represent a vector (a series of integer values) and write a modular program to perform the following tasks: **[LO 11.1, 11.2, 11.3, 11.4 M]**

(a) To create a vector

(b) To modify the value of a given element

(c) To multiply by a scalar value

(d) To display the vector in the form

(10, 20, 30, ...)

11.10 Add a function to the program of Exercise 11.9 that accepts two vectors as input parameters and return the addition of two vectors. **[LO 11.1, 11.2, 11.3, 11.4 H]**

11.11 Create two structures named **metric** and **British** which store the values of distances. The **metric** structure stores the values in metres and centimetres and the British structure stores the values in feet and inches. Write a program that reads values for the structure variables and adds values contained in one variable of **metric** to the contents of another variable of **British.** The program should display the result in the format of feet and inches or metres and centimetres as required. **[LO 11.1, 11.2 M]**

11.12 Define a structure named **census** with the following three members: **[LO 11.1, 11.3, 11.4 M]**

(a) A character array city [ ] to store names

(b) A long integer to store population of the city

(c) A float member to store the literacy level

Write a program to do the following:

(a) To read details for 5 cities randomly using an array variable

(b) To sort the list alphabetically

(c) To sort the list based on literacy level

(d) To sort the list based on population

(e) To display sorted lists

11.13 Define a structure that can describe an hotel. It should have members that include the name, address, grade, average room charge, and number of rooms. **[LO 11.1, 11.2 M]**

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

11.14 Define a structure called **cricket** that will describe the following information: **[LO 11.1, 11.2, 11.3 M]**

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.

11.15 Design a structure **student_record** to contain name, date of birth, and total marks obtained. Use the **date** structure designed in Exercise 11.4 to represent the date of birth.

[LO 11.1, 11.2, 11.4 M]

Develop a program to read data for 10 students in a class and list them rank-wise.

[LO 11.2 E]

11.16. Write a C program that prints the size of a structure data type.

[LO 11.5 E]

11.17. Write a C program that prints the size of a structure and union data type that have same number and type of members.

[LO 11.2 H]

11.18. Write a C program for demonstrating operations on individual structure members using pointer notation. **[LO 11.2 H]**

INTERVIEW QUESTIONS

11.1 What is the difference between structure and union in terms of object size? **[LO 11.5 E]**

11.2 Which operators have the highest and lowest precedence? **[LO 11.2 M]**

11.3 What is a self-referential structure? **[LO 11.3 H]**

11.4 Give some examples of user-defined data types in C. **[LO 11.2 E]**

11.5 What are the different ways of creating new data types in C. **[LO 11.5 M]**

11.6 Is it a valid structure definition? **[LO 11.3 H]**

```
struct s
{
    int n;
    struct s next;
};
```

Chapter 12

Pointers

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

LO 12.1 Know the concept of pointers
LO 12.2 Determine how pointer variables are used in a program
LO 12.3 Describe chain of pointers
LO 12.4 Illustrate pointer expressions
LO 12.5 Discuss pointers and arrays
LO 12.6 Explain how pointers are used with functions and structures

INTRODUCTION

A p ointer is a derived data type in C. It is built from one of the fundamental data types available in C. Pointers contain memory addresses as their values. Since these memory addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.

Pointers are undoubtedly one of the most distinct and exciting features of C language. It has added power and flexibility to the language. Although they appear little confusing and difficult to understand for a beginner, they are a powerful tool and handy to use once they are mastered.

Pointers are used frequently in C, as they offer a number of benefits to the programmers. They include:

1. Pointers can be used to return multiple values from a function via function arguments.
2. Pointers permit references to functions and thereby facilitating passing of functions as arguments to other functions.

3. The use of pointer arrays to character strings results in saving of data storage space in memory.
4. Pointers allow C to support dynamic memory management.
5. Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.

Of course, the real power of C lies in the proper use of pointers. In this chapter, we will examine the pointers in detail and illustrate how to use them in program development. Chapter 14 examines the use of pointers for creating and managing linked lists.

UNDERSTANDING POINTERS

The computer's memory is a sequential collection of *storage cells* as shown in Fig. 12.1 . 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.

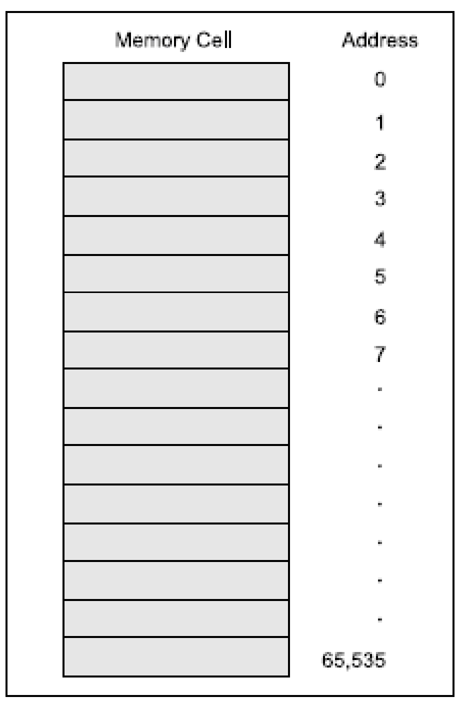


Fig. 12.1 Memory organisation

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

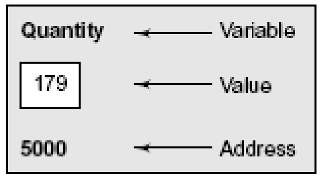


Fig. 12.2 Representation of a variable

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. 12.2 . (Note that the address of a variable is the address of the first bye occupied by that 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, that can be stored in memory, like any other variable. Such variables that hold memory addresses are called *pointer variables* . A pointer variable is, therefore, nothing but a variable that contains an address, which is a location of another variable in memory.

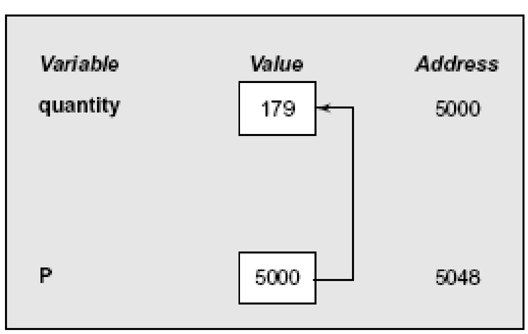


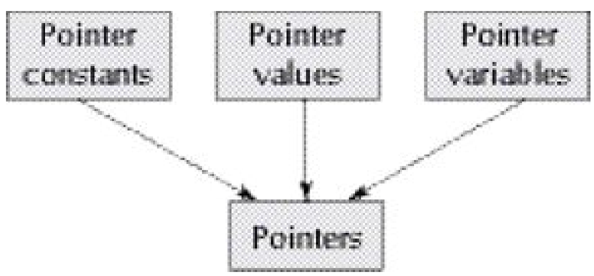
Fig. 12.3 Pointer variable

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.12.3. 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'. (We are not really concerned about the actual values of pointer variables. They may be different everytime we run the program. What we are concerned about is the relationship between the variables **p** and **quantity** .)

Underlying Concepts of Pointers

Pointers are built on the three underlying concepts as illustrated below:



Memory addresses within a computer are referred to as *pointer constants.* We cannot change them; we can only use them to store data values. They are like house numbers.

We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The value thus obtained is known as *pointer value.* The pointer value (i.e. the address of a variable) may change from one run of the program to another.

Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a *pointer variable.*

Accessing the Address of a Variable

LO 12.1
Know the concept of pointers

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 o f '.

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** .

WORKED-OUT PROBLEM 12.1 E

Write a program to print the address of a variable along with its value.

The program shown in Fig. 12.4 , declares and initializes four variables and then prints out these values with their respective storage locations. Note that we have used %u format for printing address values. Memory addresses are unsigned integers.

```
Program
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. 12.4 Accessing the address of a variable

DECLARING POINTER VARIABLES

LO 12.2
Determine how pointer variables are used in a program

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,

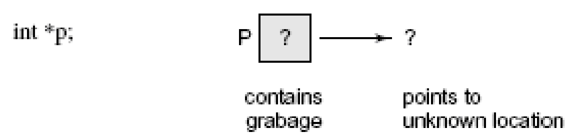
int *p; /* integer pointer */

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; / * float pointer */

declares **x** as a pointer to a floating-point variable.

The declarations cause the compiler to allocate memory locations for the pointer variables **p** and **x** . Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations as shown:



> **TIP**
> All pointer variables are of the same size irrespective of their type.

Pointer Declaration Style

Pointer variables are declared similarly as normal variables except for the addition of the unary * operator. This symbol can appear anywhere between the type name and the printer variable name. Programmers use the following styles:

```
int*     p;      /* style 1 */
int      *p;     /* style 2 */
int      * p;    /* style 3 */
```

However, the style 2 is becoming increasingly popular due to the following reasons:

1. This style is convenient to have multiple declarations in the same statement. Example:

   int *p, x, *q;

2. This style matches with the format used for accessing the target values. Example:

int x, *p, y;

x = 10;

p = & x;

y = *p; /* accessing x through p */

***p = 20;** /* assigning 20 to x */

We use in this book the style 2, namely,

int *p;

Initialization of Pointer Variables

The process of assigning the address of a variable to a pointer variable is known as *initialization* . As pointed out earlier, all uninitialized pointers will have some unknown values that will be interpreted as memory addresses. They may not be valid addresses or they may point to some values that are wrong. Since the compilers do not detect these errors, the programs with uninitialized pointers will produce erroneous results. It is therefore important to initialize pointer variables carefully before they are used in the program.

Once a pointer variable has been declared we can use the assignment operator to initialize the variable. Example:

```
int quantity;
int *p;                /* declaration  */
p = &quantity;         /* initialization  */
```

We can also combine the initialization with the declaration. That is,

int *p = &quantity;

is allowed. The only requirement here is that the variable **quantity** must be declared before the initialization takes place. Remember, this is an initialization of **p** and not ***p** .

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;          /* wrong */
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.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step. For example,

int x, *p = &x; /* three in one */

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** . And also remember that the target variable **x** is declared first. The statement

int *p = &x, x;

is not valid.

We could also define a pointer variable with an initial value of NULL or 0 (zero). That is, the following statements are valid

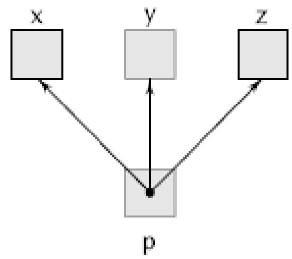
int *p = NULL;

int *p = 0;

Pointer Flexibility

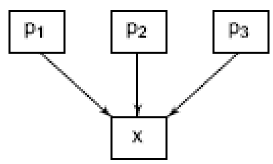
Pointers are flexible. We can make the same pointer to point to different data variables in different statements. Example;

```
int x, y, z, *p;
. . . . .
p = &x;
. . . . .
p = &y;
. . . . .
p = &z;
. . . . .
```



We can also use different pointers to point to the same data variable. Example; int x;

```
int *p1 = &x;
int *p2 = &x;
int *p3 = &x;
. . . . .
. . . . .
```



With the exception of NULL and 0, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

int *p = 5360; / *absolute address */

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.* Another name for the indirection operator is the *dereferencing 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 operato r * 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. Program 12.2 illustrates the distinction between pointer value and the value it points to.

WORKED-OUT PROBLEM 12.2 **E**

Write a program to illustrate the use of indirection operator '*' to access the value pointed to by a pointer.

The program and output are shown in Fig. 12.5 . 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
                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", 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
                4104 is stored at addr 4106
                10   is stored at addr 4108
                Now x = 25
```

Fig. 12.5 Accessing a variable through its pointer

The actions performed by the program are illustrated in Fig. 12.6 . 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** .

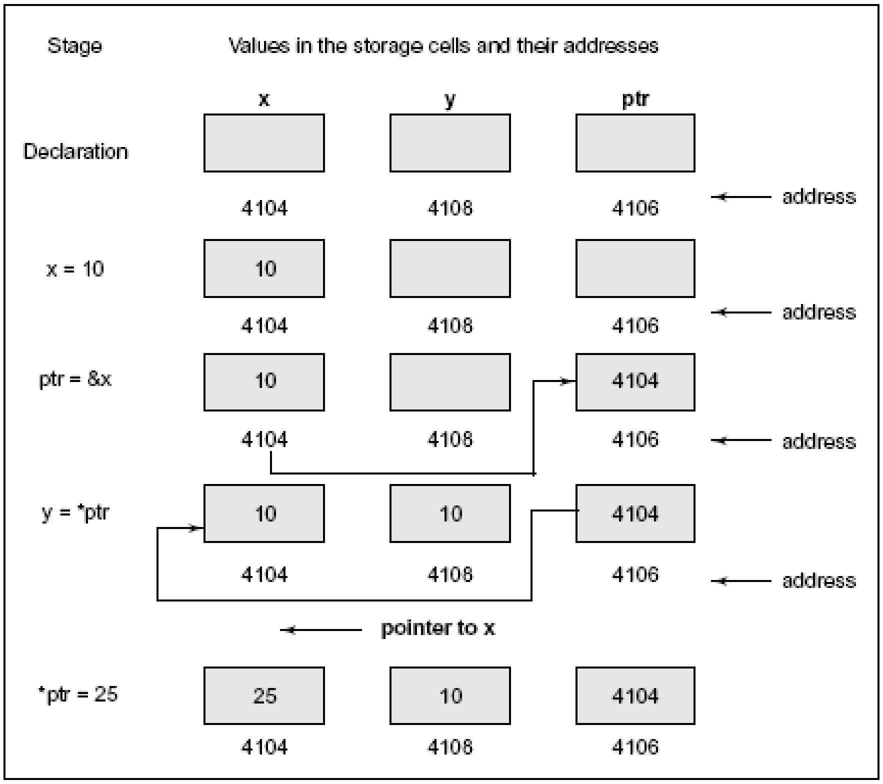


Fig. 12.6 Illustration of pointer assignments

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 o perator* .

Chain of Pointers

It is possible to make a pointer to point to another pointer, thus creating a chain of pointers as shown.

LO 12.3
Describe chain of pointers

Here, the pointer variable **p2** contains the address of the pointer variable **p1** , which points to the location that contains the desired value. This is known as *multiple indirections* .

A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name. Example:

int 10p2;

This declaration tells the compiler that **p2** is a pointer to a pointer of **int** type. Remember, the pointer **p2** is not a pointer to an integer, but rather a pointer to an integer pointer.

We can access the target value indirectly pointed to by pointer to a pointer by applying the indirection operator twice. Consider the following code:

```
main ( )
{
            int x, *p1,     **p2;
            x = 100;
            p1 = &x;        /* address of  x */
            p2 = &p1        /* address of  p1 */
            printf ("%d",   **p2);
}
```

This code will display the value 100. Here, **p1** is declared as a pointer to an integer and **p2** as a pointer to a pointer to an integer.

POINTER EXPRESSIONS

LO 12.4
Illustrate 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:

```
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.

TIP
Addition, Multiplication, Division, Module operations involving two pointers are prohibited

WORKED-OUT PROBLEM 12.3 **M**

Write a program to illustrate the use of pointers in arithmetic operations.

The program in Fig. 12.7 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
            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. 12.7 Evaluation of pointer expressions

POINTER INCREMENTS AND SCALE FACTOR

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 2804, and not 2801. That is, when we increment a pointer, its value is increased by the 'length' of the data type it points to (It is being assumed that int is a 4 byte type in a 32 bit computer). This length is called the *scale factor* . For a 32 bit computer, the lengths of various data types are as follows:

char	1 byte
int	4 bytes
float	4 bytes
long int	8 bytes
double	8 bytes

TIP

Pointers are incremented/decremented by one unit. The unit size depends upon the data type of the pointer.

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. (32 bit systems use 4 bytes for storing integers and 2 bytes for short integers).

Rules of Pointer Operations

The following rules apply when performing operations on pointer variables:

1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the values of another pointer variable.
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.
5. An integer value may be added or subtracted from a pointer variable.
6. When two pointers point to the same array, one pointer variable can be subtracted from another.

7. When two pointers point to the objects of the same data types, they can be compared using relational operators.
8. A pointer variable cannot be multiplied by a constant.
9. Two pointer variables cannot be added.
10. A value cannot be assigned to an arbitrary address (i.e., &x = 10; is illegal).

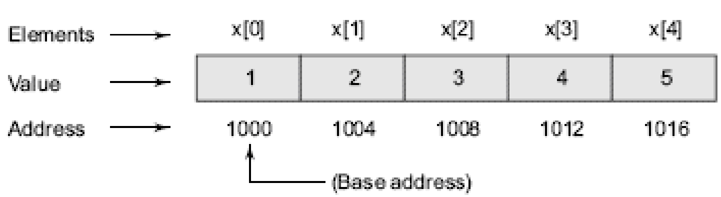
POINTERS AND ARRAYS

LO 12.5
Discuss 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:

int x[5] = {1, 2, 3, 4, 5};

Suppose the address of x is 1000 and assuming that each integer requires 4 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];

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 as:

p = &x[0] (= 1000)

p + 1 = &x[1] (= 1004)

p + 2 = &x[2] (= 1008)

p + 3 = &x[3] (= 1012)

p + 4 = &x[4] (= 1016)

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 x scale factor of int)
                = 1000 + (3 * 4)
                = 1012
```

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 Worked Out Problem 12.4 illustrates the use of pointer accessing method.

WORKED-OUT PROBLEM 12.4 **M**

Write a program using pointers to compute the sum of all elements stored in an array.

The program shown in Fig. 12.8 illustrates how a pointer can be used to traverse an array element. 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. The program has been run on Linux which is a 32 bit system.

```
Program
            main()
            {
                 int *p, sum, i;
                 int x[5] = {5,9,6,3,7};
                 i  = 0;
                 p  = x;     /* initializing with base address of x */
                 printf("Element   Value   Address\n\n");
                 while(i < 5)
            {
                 printf(" x[%d] %d %u\n", i, *p, p);
                 sum = sum + *p;   /* accessing array element  */
                 i++, p++;        /* incrementing pointer     */
            }
            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          170
                   x[2]         6          174
                   x[3]         3          178
                   x[4]         7          182
                   Sum    =   55
                   &x[0]  =   166
                   p      =   186
```

Fig. 12.8 Accessing one-dimensional array elements using the pointer

It is possible to avoid the loop control variable **i** as shown:

```
.....
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.

Pointers can be used to manipulate two-dimensional arrays as well. We know that in a one-dimensional array **x,** the expression

$$*(x+i) \text{ or } *(p+i)$$

represents the element **x[i]** . Similarly, an element in a two-dimensional array can be represented by the pointer expression as follows:

$$*(*(a+i)+j) \text{ or } *(*(p+i)j)$$

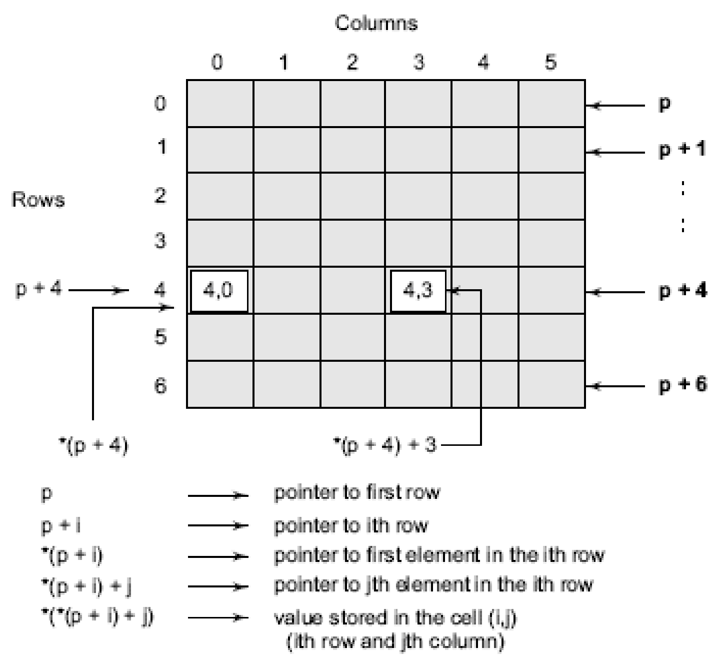
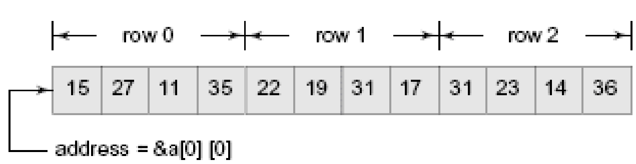


Fig. 12.9 Pointers to two-dimensional arrays

Figure 12.9 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:



If we declare **p** as an **int** pointer with the initial address of &a[0][0], then

a[i][j] is equivalent to $*(p+4 \times i+j)$

You may notice that, if we increment **i** by 1, the **p** is incremented by 4, the size of each row. Then the element **a[2][3]** is given by **$*(p+2 \times 4+3) = *(p+11)$** .

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.

Pointers and Character Strings

We have seen in Chapter 8 that strings are treated like character arrays and therefore, they are declared and initialized as follows:

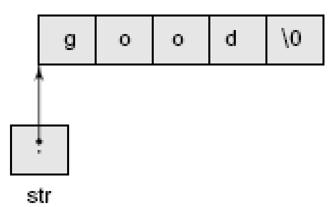
char str [5] = "good";

The compiler automatically inserts the null character '\0' at the end of the string. C supports an alternative method to create strings using pointer variables of type **char** . Example:

char *str = "good";

This creates a string for the literal and then stores its address in the pointer variable **str** .

The pointer **str** now points to the first character of the string "good" as:



We can also use the run-time assignment for giving values to a string pointer. Example

char * string1;

string1 = "good";

Note that the assignment

string1 = “good”;

is not a string copy, because the variable **string1** is a pointer, not a string.

(As pointed out in Chapter 9, C does not support copying one string to another through the assignment operation.)

We can print the content of the string **string1** using either **printf** or **puts** functions as follows:

printf(“%s”, string1);

puts (string1);

Remember, although **string1** is a pointer to the string, it is also the name of the string. Therefore, we do not need to use indirection operator * here.

Like in one-dimensional arrays, we can use a pointer to access the individual characters in a string. This is illustrated by the Worked-Out Problem 12.5.

WORKED-OUT PROBLEM 12.5 E

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.12.10. The statement

char *cptr = name;

```
Program
        main()
        {
             char  *name;
             int   length;
             char  *cptr = name;
             name  = "DELHI";
             printf ("%s\n", name);
             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
        DELHI
        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. 12.10 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

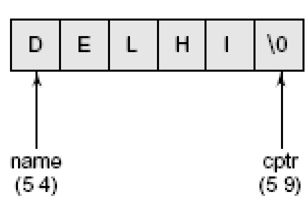
while(*cptr != '\0')

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";

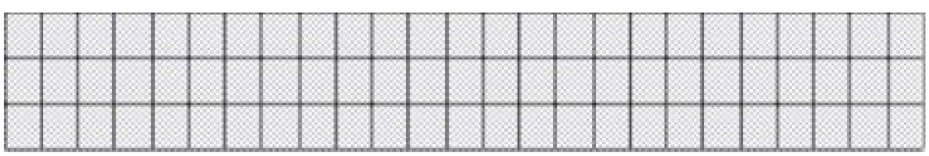
do not work.

ARRAY OF POINTERS

One important use 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,

```
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:

name [0] ⟶ New Zealand
name [1] ⟶ Australia
name [2] ⟶ India

This declaration allocates only 28 bytes, sufficient to hold all the characters as shown

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 jth character in the ith 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.

Test Your Skills

Write a C program to print the addresses and values of a 2-dimensional array. [E]

Functions that Return Multiple Values

Up till now, we have illustrated functions that return just one value using a return statement. That is because, a return statement can return only one value. Suppose, however, that we want to get more information from a function. We can achieve this in C using the arguments not only to receive information but also to send back information to the calling function. The arguments that are used to "send out" information are called *output parameters.*

The mechanism of sending back information through arguments is achieved using what are known as the *address operator* (&) and *indirection operator* (*). Let us consider an example to illustrate this.

```
void mathoperation (int x, int y, int *s, int *d);
main( )
{
     int x = 20, y = 10, s, d;
     mathoperation(x,y, &s, &d);

     printf("s=%d\n d=%d\n", s,d);
}
void mathoperation (int a, int b, int *sum, int *diff)
{
     *sum  = a+b;
     *diff = a-b;
}
```

The actual arguments **x** and **y** are input arguments, **s** and **d** are output arguments. In the function call, while we pass the actual values of **x** and **y** to the function, we pass the addresses of locations where the values of **s** and **d** are stored in the memory. (That is why, the operator & is called the address operator.) When the function is called the following assignments occur:

value of	x to a
value of	y to b
address of	s to sum
address of	d to diff

Note that indirection operator * in the declaration of **sum** and **diff** in the header indicates these variables are to store addresses, not actual values of variables. Now, the variables **sum** and **diff** point to the memory locations of **s** and **d** respectively.

(The operator * is known as indirection operator because it gives an indirect reference to a variable through its address.)

In the body of the function, we have two statements:

* sum = a+b;

* diff = a-b;

The first one adds the values **a** and **b** and the result is stored in the memory location pointed to by **sum.** Remember, this memory location is the same as the memory location of **s.** Therefore, the value stored in the location pointed to by **sum** is the value of **s.**

Similarly, the value of a–b is stored in the location pointed to by **diff,** which is the same as the location **d.** After the function call is implemented, the value of **s** is a+b and the value of **d** is a–b. Output will be:

s = 30

d = 10

The variables ***sum** and ***diff** are known as *pointers* and **sum** and **diff** as *pointer* variables. Since they are declared as **int** , they can point to locations of **int** type data.

The use of pointer variables as actual parameters for communicating data between functions is called "pass by pointers" or "call by address or reference".

Rules for Pass by Pointers

1. The types of the actual and formal arguments must be same.
2. The actual arguments (in the function call) must be the addresses of variables that are local to the calling function.
3. The formal arguments in the function header must be prefixed by the indirection operator *.
4. In the prototype, the arguments must be prefixed by the symbol *.
5. To access the value of an actual argument in the called function, we must use the corresponding formal argument prefixed with the indirection operator *.

Pass by Value versus Pass by Pointers

The technique used to pass data from one function to another is known as *parameter passing* . Parameter passing can be done in following two ways:

- Pass by value (also known as call by value).
- Pass by Pointers (also known as call by pointers).

In *pass by value,* values of actual parameters are copied to the variables in the parameter list of the called function. The called function works on the copy and not on the original values of the actual parameters. This ensures that the original data in the calling function cannot be changed accidentally.

In *pass by pointers* (also known as pass by address), the memory addresses of the variables rather than the copies of values are sent to the called function. In this case, the called function directly works on the data in the calling function and the changed values are available in the calling function for its use.

Pass by pointers method is often used when manipulating arrays and strings. This method is also used when we require multiple values to be returned by the called function.

POINTERS AS FUNCTION ARGUMENTS

> **LO 12.6**
> Explain how pointers are used with functions and structures

We have seen 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. We used this method when discussing functions that return multiple values.

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 variables 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);   /* call by reference or address */
        printf("%d\n",x);
}
change(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.

> **Note** *C99 adds a new qualifier* ***restrict*** *to the pointers passed as function parameters. See the Appendix "C99 Features".*

WORKED-OUT PROBLEM 12.6 M

Write a function using pointers to exchange the values stored in two locations in the memory.

The program in Fig. 12.11 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
            void exchange (int *, int *);    /* prototype */
            main()
            {
                 int  x, y;
                 x - 100;
                 y - 200;
                 printf("Before exchange  : x - %d   y - %d\n\n", x, y);
                 exchange(&x,&y);     /* call */
                 printf("After exchange   : x - %d   y - %d\n\n", x, y);
            }
            exchange (int *a, int *b)
            {
                 int t;
                 t - *a;    /* Assign the value at address a to t */
                 *a - *b;   /* put b into a */
                 *b - t;    /* put t into b */
            }

Output

            Before exchange  : x - 100   y - 200
            After exchange   : x - 200   y - 100
```

Fig. 12.11 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.

The use of pointers to access array elements is very common in C. We have used a pointer to traverse array elements in Program 12.4. We can also use this technique in designing user-defined functions discussed in Chapter 9. Let us consider the problem sorting an array of integers discussed in Program 9.6.

The function **sort** may be written using pointers (instead of array indexing) as shown:

```
void sort (int m, int *x)
{     int i j, temp;
      for (i=1; i<= m-1; i++)
            for (j=1; j<= m-1; j++)
            if (*(x+j-1) >= *(x+j))
            {
                  temp = *(x+j-1);
                  *(x+j-1) = *(x+j);
                  *(x+j) = temp;
                  }
}
```

Note that we have used the pointer x (instead of array x[ ]) to receive the address of array passed and therefore the pointer x can be used to access the array elements (as pointed out in Section 12.10). This function can be used to sort an array of integers as follows:

```
            . . . . .
int score[4] = {45, 90, 71, 83};
            . . . . .
sort(4, score); /* Function call */
            . . . . .
```

The calling function must use the following prototype declaration.

void sort (int, int *);

This tells the compiler that the formal argument that receives the array is a pointer, not array variable.

Pointer parameters are commonly employed in string functions. Consider the function copy which copies one string to another.

```
copy(char *s1, char *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.

WORKED-OUT PROBLEM 12.7 **M**

The program of Fig. 12.12 shows how to calculate the sum of two numbers which are passed as arguments using the call by reference method.

Program

```
#include<stdio.h>

void swap (int *,int *);

main()
{
    int x-0;
    int y-20;

    printf("\nValue of X and Y before swapping are X-%d and Y-%d", x,y);
    swap(&x, &y);
    printf("\n\nValue of X and Y after swapping are X-%d and Y-%d", x,y);
}

void swap(int *p, int *q) /*Value of x and y are transferred using call by reference*/
{
    int r;

    r-*p;
    *p-*q;
    *q-r;
}
```

Output

```
Value of X and Y before swapping are X-10 and Y-20
Value of X and Y after swapping are X-20 and Y-10
```

Fig. 12.12 Program to pass the arguments using call by reference method

FUNCTIONS RETURNING POINTERS

We have seen so far that a function can return a single value by its name or return multiple values through pointer parameters. Since pointers are a data type in C, we can also force a function to return a pointer to the calling function. Consider the following code:

int *larger (int *, int *); /* prototype */

```
int *larger (int *, int *);     /* prototype */
main ( )
{
        int a - 10;
        int b - 20;
        int *p;
        p - larger(&a, &b);  /Function call */
        printf ("%d", *p);
}
int *larger (int *x, int *y)
{
     if (*x>*y)
          return (x);  / *address of a */
     else
          return (y);  /* address of b */
}
```

The function **larger** receives the addresses of the variables **a** and **b** , decides which one is larger using the pointers **x** and **y** and then returns the address of its location. The returned value is then assigned to the pointer variable **p** in the calling function. In this case, the address of **b** is returned and assigned to **p** and therefore the output will be the value of **b** , namely, 20.

Note that the address returned must be the address of a variable in the calling function. It is an error to return a pointer to a local variable in the called function.

POINTERS TO FUNCTIONS

A function, like a variable, has a type and 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 mul(int, int);
double (*p1)();
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) /* Function call */

is equivalent to

mul(x,y)

Note the parentheses around ***p1.**

WORKED-OUT PROBLEM 12.8 **H**

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. 12.13 . The printing is done by the function **table** by evaluating the function passed to it by the **main.**

```
Program
        #include <math.h>
        #define PI 3.1415926
        double y(double);
        double cos(double);
        double table (double(*f)(), double, double, double);

        main()
        { printf("Table of y(x) = 2*x*x-x+1\n\n");
          table(y, 0.0, 2.0, 0.5);
          printf("\nTable of cos(x)\n\n");
          table(cos, 0.0, PI, 0.5);
        }
        double table(double(*f)(),double min, double max, double step)
        {    double a, value;
             for(a = min; a <= max; a += step)
             {
                value = (*f)(a);
                printf("%5.2f  %10.4f\n", a, value);
             }
        }
        double y(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. 12.13 Use of pointers to functions

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);

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.

Compatibility and Casting

A variable declared as a pointer is not just a *pointer type* variable. It is also a pointer to a *specific* fundamental data type, such as a character. A pointer therefore always has a type associated with it. We cannot assign a pointer of one type to a pointer of another type, although both of them have memory addresses as their values. This is known as *incompatibility* of pointers.

All the pointer variables store memory addresses, which are compatible, but what is not compatible is the underlying data type to which they point to. We cannot use the assignment operator with the pointers of different types. We can however make explicit assignment between incompatible pointer types by using **cast** operator, as we do with the fundamental types. Example:

```
int x;
char *p;
p = (char *) & x;
```

In such cases, we must ensure that all operations that use the pointer **p** must apply casting properly.

We have an exception. The exception is the void pointer (void *). The void pointer is a ***generic pointer*** that can represent any pointer type. All pointer types can be assigned to a void pointer and a void pointer can be assigned to any pointer without casting. A void pointer is created as follows:

void *vp;

Remember that since a void pointer has no object type, it cannot be de-referenced.

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* (also known as *member selection 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 *.

WORKED-OUT PROBLEM 12.9 **H**

Write a program to illustrate the use of structure pointers.

A pro gram to illustrate the use of a structure pointer to manipulate the elements of an array of structures is shown in Fig. 12.14 . 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
            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("%s %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. 12.14 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;      /* pointer inside the struct */
}  ptr;           /* struct type pointer */
```

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(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.**

TEST YOUR SKILLS

Write a C program that manipulates local integer variable values from a different function. [E]

TROUBLES WITH POINTERS

Pointers give us tremendous power and flexibility. However, they could become a nightmare when they are not used correctly. The major problem with wrong use of pointers is that the compiler may not detect the error in most cases and therefore the program is likely to produce unexpected results. The output may not given us any clue regarding the use of a bad pointer. Debugging therefore becomes a difficult task.

We list here some pointer errors that are more commonly committed by the programmers.

- Assigning values to uninitialized pointers

```
int * p, m = 100 ;
*p = m ; /* Error */
```

- Assigning value to a pointer variable

```
int *p, m = 100 ;
p = m; /* Error */
```

- Not dereferencing a pointer when required

```
int *p, x = 100;
p = &x;
printf("%d",p); /* Error */
```

- Assigning the address of an uninitialized variable

```
int m, *p
p = &m; /* Error */
```

- Comparing pointers that point to different objects

```
char name1 [ 20 ], name2 [ 30 ];
   char *p1 = name1;
   char *p2 = name2;
   if(p1 > p2).......        /* Error */
```

We must be careful in declaring and assigning values to pointers correctly before using them. We must also make sure that we apply the address operator & and referencing operator * correctly to the pointers. That will save us from sleepless nights.

KEY CONCEPTS

- **MEMORY:** This is a sequential collection of storage cells with each cell having an address value associated with it. **[LO 12.1]**
- **POINTER:** It is used to store the memory address as value. **[LO 12.1]**
- **POINTER VARIABLE:** It is a variable that stores the memory address of another variable. **[LO 12.2]**
- **CALL BY REFERENCE:** It is the process of calling a function using pointers to pass the addresses of variables. **[LO 12.6]**
- **CALL BY VALUE:** It is the process of passing the actual value of variables. **[LO 12.6]**

ALWAYS REMEMBER

- Only an address of a variable can be stored in a pointer variable. **[LO 12.1]**
- Do not store the address of a variable of one type into a pointer variable of another type. **[LO 12.1]**
- The value of a variable cannot be assigned to a pointer variable. **[LO 12.1]**
- A very common error is to use (or not to use) the address operator (&) and the indirection operator (*) in certain places. Be careful. The compiler may not warn such mistakes. **[LO 12.1]**
- Remember that the definition for a pointer variable allocates memory only for the pointer variable, not for the variable to which it is pointing. **[LO 12.1]**
- A pointer variable contains garbage until it is initialized. Therefore , we must not use a pointer variable before it is assigned, the address of a variable. **]** **[LO 12.2**

- It is an error to assign a numeric constant to a pointer variable.] [LO 12.2
- It is an error to assign the address of a variable to a variable of any basic data types.] [LO 12.2
- A proper understanding of a precedence and associativity rules is very important in pointer applications. For example, expressions like *p++, *p[ ], (*p)[ ], (p).member should be carefully used.] [LO 12.4
- Be careful while using indirection operator with pointer variables. A simple pointer uses single indirection operator (*ptr) while a pointer to a pointer uses additional indirection operator symbol (23ptr).] [LO 12.4
- 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 size, except the first, which is optional.] [LO 12.5
- If we want a called function to change the value of a variable in the calling function, we must pass the address of that variable to the called function.] [LO 12.6
- When we pass a parameter by address, the corresponding formal parameter must be a pointer variable.] [LO 12.6
- It is an error to assign a pointer of one type to a pointer of another type without a cast (with an exception of void pointer).] [LO 12.6

BRIEF CASES

1. Processing of Examination Marks [LO 12.2, 12.5, 12.6 H]

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. 12.15 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

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.

Program

```
#define  STUDENTS  5
#define  SUBJECTS  4
#include <string.h>

main()
{
  char name[STUDENTS][20];
  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(char *string[ ],
         int array [ ] [SUBJECTS +1], int m, int n)
  {
      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(int array [ ] [SUBJECTS +1], int m, int n)
  {
      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(char *string [ ],
              int array [ ] [SUBJECTS + 1]
              int m,
```

```
                 int n)
   {
      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(char *string[ ],
            int array [] [SUBJECTS + 1],
            int m,
            int n)
{
      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(int *p, int *q)
```

```
    {
        int  temp;
        temp = *p;
        *p   = *q;
        *q   = temp;
    }

    /*    Exchange of strings        */
    swap_string(char s1[ ], char 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[i];
          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. 12.15 Preparation of the rank list of a class of students

2. Inventory Updating [LO 12.2, 12.6 M]

The price and quantity of items stocked in a store changes every day. They may either increase or decrease. The program in Fig. 12.16 reads the incremental values of price and quantity and computes the total value of the items in stock.

The program illustrates the use of structure pointers as function parameters. **&item,** the address of the structure **item,** is passed to the functions **update()** and **mul().** The formal arguments **product** and **stock,** which receive the value of **&item,** are declared as pointers of type **struct stores.**

Program

```
struct stores
{
     char  name[20];
     float price;
     int   quantity;
};
main()
{
     void update(struct stores *, float, int);
     float        p_increment, value;
     int           q_increment;

     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, &q_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(struct stores *product, float p, int q)
{
     product->price += p;
     product->quantity += q;
}
float mul(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. 12.16 Use of structure pointers as function parameters

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

12.1 A pointer in C language is **[LO 12.1 E]**

(a) address of some location

(b) useful to describe linked list

(c) can be used to access elements of an array

(d) All of the above.

12.2 Which constant values can be assigned to a pointer variable? **[LO 12.4 M]**

(a) 0

(b) Null

(c) None

(d) Both 1 and 2

12.3 Which of the following operations is allowed to be performed on a pointer variable (e.g. ptr)? **[LO 12.4 M]**

(a) ptr+1

(b) ptr-1

(c) ptr++ and ptr--

(d) All of these

12.4 Which of the following is a valid indirection notation for a pointer? **[LO 12.3 H]**

(a) *ptr

(b) 26ptr

(c) 26*ptr

(d) All of these

12.5 Which of the following is incorrect about pointers? **[LO 12.1 M]**

(a) A pointer variable stores NULL value until it is initialized.

(b) A pointer variable cannot be assigned any numeric constant except 0.

(c) Except for the case of void pointer, it is an error to assign a pointer of one type to a pointer of another type without a cast.

(d) The address operator (&) is not required to be used while assigning the base address of an array to a pointer variable.

REVIEW QUESTIONS

12.1 State whether the following statements are *true* or *false* .

(a) Pointer constants are the addresses of memory locations. **[LO 12.1 M]**

(b) The underlying type of a pointer variable is void. **[LO 12.1 M]**

(c) Pointer variables are declared using the address operator. **[LO 12.2 E]**

(d) It is possible to cast a pointer to float as a pointer to integer. **[LO 12.2 M]**

(e) Pointers to pointers is a term used to describe pointers whose contents are the address of another pointer. **[LO 12.3 E]**

(f) A pointer can never be subtracted from another pointer. **[LO 12.4 E]**

(g) An integer can be added to a pointer. **[LO 12.4 M]**

(h) Pointers cannot be used as formal parameters in headers to function definitions. **[LO 12.6 E]**

(i) When an array is passed as an argument to a function, a pointer is passed. **[LO 12.6 M]**

(j) Value of a local variable in a function can be changed by another function. **[LO 12.6 M]**

12.2 Fill in the blanks in the following statements:

(a) A pointer variable contains as its value the _____ of another variable. **[LO 12.1 E]**

(b) The _____operator returns the value of the variable to which its operand points. **[LO 12.1 E]**

(c) The ______operator is used with a pointer to de-reference the address contained in the pointer. **[LO 12.2 M]**

(d) The pointer that is declared as ______cannot be de-referenced. **[LO 12.2 M]**

(e) The only integer that can be assigned to a pointer variable is ______. **[LO 12.4 M]**

(f) The process of calling a function using pointers to pass the addresses of variables is known as ___________. **[LO 12.6 E]**

(g) ___________ pointer is called generic pointer and can be used to represent any pointer type. **[LO 12.4 M]**

12.3 What is a pointer? How can it be initialized? **[LO 12.1, 12.2 E]**

12.4 Explain the effects of the following statements: **[LO 12.2 M]**

(a) int a, *b = &a;

(b) int p, *p;

(c) char *s;

(d) a = (float *) &x);

(e) double(*f)();

12.5 Distinguish between (*m)[5] and *m[5]. **[LO 12.5 M]**

12.6 Given the following declarations: **[LO 12.4 H]**

```
int x = 10, y = 10;
int *p1 = &x, *p2 = &y;
```

What is the value of each of the following expressions?

(a) (*p1) ++

(b) – – (*p2)

(c) *p1 + (*p2) – –

(d) + + (*p2) – *p1

12.7 Describe typical applications of pointers in developing programs. **[LO 12.1 E]**

12.8 What are the arithmetic operators that are permitted on pointers? **[LO 12.4 E]**

12.9 What is printed by the following program? **[LO 12.2, 12.3 M]**

```
int m = 100';
int * p1 = &m;
int **p2 = &p1;
printf("%d", **p2);
```

12.10 Assuming **name** as an array of 15 character length, what is the difference between the following two expressions? **[LO 12.4, 12.5 E]**

(a) name + 10; and

(b) *(name + 10).

12.11 What is the output of the following segment? **[LO 12.5 H]**

```
int m[2];
*(m+1) = 100;
*m = *(m+1);
printf("%d", m [0]);
```

12.12 What is the output of the following code? **[LO 12.4, 12.5 M]**

```
int m [2];
int *p = m;
m [0] = 100 ;
m [1] = 200 ;
printf("%d %d", ++*p, *p);
```

12.13 What is the output of the following program? **[LO 12.5, 12.6 M]**

```
int f(char *p);
main ( )
{
     char str[ ] = "ANSI";
     printf("%d", f(str) );
}
int f(char *p)
{
     char *q = p;
     while (*++p)
            ;
     return (p-q);
}
```

12.14 Given below are two different definitions of the function **search( )** **[LO 12.3, 12.6 M]**

```
a) void search (int* m[ ], int x)
   {
   }
b) void search (int ** m, int x)
   {
   }
```

Are they equivalent? Explain.

12.15 Do the declarations **[LO 12.5 M]**

```
char s [ 5 ] ;
char *s;
```

represent the same? Explain.

12.16 Which one of the following is the correct way of declaring a pointer to a function? Why? **[LO 12.5, 12.6 E]**

(a) int (*p) (void) ;

(b) int *p (void);

DEBUGGING EXERCISES

12.1 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. **[LO 12.2, 12.4 M]**

(a) p1 = &m;

(b) p2 = n;

(c) *p1 = &n;

(d) p2 = &*&m;

(e) m = p2–p1;

(f) p1 = &p2;

(g) m = *p1 + *p2++;

12.2 Find the error, if any, in each of the following statements: **[LO 12.2, 12.3 E]**

(a) int x = 10;

(b) int *y = 10;

(c) int a, *b = &a;

(d) int m;

int 29x = &m;

12.3 What is wrong with the following code? **[LO 12.2, 12.3 M]**

```
int **p1, *p2;
p2 - &p1;
```

PROGRAMMING EXERCISES

12.1 Write a program using pointers to read in an array of integers and print its elements in reverse order. **[LO 12.5 M]**

12.2 We know that the roots of a quadratic equation of the form **[LO 12.6 M]**

$$ax^2 + bx + c = 0$$

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.

12.3 Write a function that receives a sorted array of integers and an integer value, and inserts the value in its correct place. **[LO 12.5, 12.6 H]**

12.4 Write a function using pointers to add two matrices and to return the resultant matrix to the calling function. **[LO 12.5, 12.6 M]**

12.5 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. **[LO 12.5, 12.6 H]**

12.6 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. **[LO 12.5, 12.6 M]**

12.7 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 **strcmp** and **swap.sort** in turn should call these functions via the pointers. **12.5, 12.6 H]** **[LO**

12.8 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. **[LO 12.2, 12.4, 12.6 H]**

(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.)

12.9 Write a function (using a pointer parameter) that reverses the elements of a given array. **[LO 12.5, 12.6 H]**

12.10 Write a function (using pointer parameters) that compares two integer arrays to see whether they are identical. The function returns 1 if they are identical, 0 otherwise. **[LO 12.5, 12.6 M]**

12.11 Write a C program to insert a substring into another string by using function and pointers. **[LO 12.6 H]**

12.12 Write a C program that uses the pointer increment operations to demonstrate the scale factor. **[LO 12.4 M]**

12.13 Write a C program that displays the addresses and values pointed by an array of pointers. **[LO 12.5 E]**

12.14 Write a C program that uses pointers to demonstrate initialization of structure members. **[LO 12.6 E]**

INTERVIEW QUESTIONS

12.1 Which format specifiers are used to print the address of a pointer? **[LO 12.1 M]**

12.2 What is pointer to a pointer? **[LO 12.3 E]**

12.3 What will be the output of following code:

```
char *ptr;
printf("%d %d",sizeof(*ptr),sizeof(ptr));
```

[LO 12.4 H]

12.4 What is a null pointer? **[LO 12.1 E]**

12.5 What is function pointer? **[LO 12.6 E]**

12.6 What is a void pointer? **[LO 12.4 E]**

12.7 What is the difference between uninitialized pointer and NULL pointer? **[LO 12.1 M]**

Chapter 13

File Management in C

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

LO 13.1 Describe opening and closing of files
LO 13.2 Discuss input/output operations on files
LO 13.3 Determine how error handling is performed during I/O operations
LO 13.4 Explain random access to files
LO 13.5 Know the command line arguments

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.

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.

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.

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 the following:

LO 13.1
Describe opening and closing of files

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:

TIP
A file mode is used to specify the operation for which the file is being opened.

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.

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.

INPUT/OUTPUT OPERATIONS ON FILES

LO 13.2
Discuss 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.

> **TIP**
> All file manipulation involves three steps – Opening, Performing read/write and Closing.

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.

WORKED-OUT PROBLEM 13.1 E

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.13.1. 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

```
#include  <stdio.h>

main()
{
     FILE *f1;
     char c;
     printf("Data Input\n\n");
     /* Open the file INPUT */
     f1 = fopen("INPUT", "w");

     /* Get a character from keyboard   */
     while((c=getchar()) != EOF)

          /* Write a character to INPUT  */
          putc(c,f1);
     /* Close the file INPUT   */
     fclose(f1);
     printf("\nData Output\n\n");
     /* Reopen the file INPUT    */
     f1 = fopen("INPUT","r");

     /* Read a character from INPUT*/
     while((c=getc(f1)) != EOF)

            /* Display a character on screen */
            printf("%c",c);

     /* Close the file INPUT       */
     fclose(f1);
}
```

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. 13.1 Character oriented read/write operations on a file

The file INPUT is again 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 getw and putw Functions

The **getw** and **putw** are integer-oriented functions. They are similar to the **getc** and **putc** functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general forms of **getw** and **putw** are as follows:

```
putw(integer, fp);
getw(fp);
```

Worked-Out Problem 13.2 illustrates the use of **putw** and **getw** functions.

WORKED-OUT PROBLEM 13.2 E

A file named DATA contains a series of integer numbers. Code a program to read these numbers and then write all 'odd' numbers to a file to be called ODD and all 'even' numbers to a file to be called EVEN.

The program is shown in Fig. 13.2 . It uses three files simultaneously and therefore, we need to define three-file pointers **f1, f2** and **f3.**

First, the file DATA containing integer values is created. The integer values are read from the terminal and are written to the file **DATA** with the help of the statement

putw(number, f1);

Notice that when we type –1, the reading is terminated and the file is closed. The next step is to open all the three files, **DATA** for reading, **ODD** and **EVEN** for writing. The contents of **DATA** file are read, integer by integer, by the function **getw(f1)** and written to **ODD** or **EVEN** file after an appropriate test. Note that the statement

(number = getw(f1)) != EOF

reads a value, assigns the same to **number** , and then tests for the end-of-file mark.

Finally, the program displays the contents of ODD and EVEN files. It is important to note that the files **ODD** and **EVEN** opened for writing are closed before they are reopened for reading.

Program

```
#include  <stdio.h>
main()
{
     FILE  *f1, *f2, *f3;
     int   number, i;

     printf("Contents of DATA file\n\n");
     f1 = fopen("DATA", "w");      /* Create DATA file    */
     for(i = 1; i <= 30; i++)
     {
          scanf("%d", &number);
          if(number == -1) break;
          putw(number,f1);
     }
     fclose(f1);

     f1 = fopen("DATA", "r");
     f2 = fopen("ODD", "w");
     f3 = fopen("EVEN", "w");

     /* Read from DATA file */
     while((number = getw(f1)) != EOF)
     {
          if(number %2 == 0)
             putw(number, f3);   /*  Write to EVEN file  */
          else
             putw(number, f2);   /*  Write to ODD file   */
     }
     fclose(f1);
     fclose(f2);
     fclose(f3);

     f2 = fopen("ODD","r");
     f3 = fopen("EVEN", "r");

     printf("\n\nContents of ODD file\n\n");
     while((number = getw(f2)) != EOF)
        printf("%4d", number);

     printf("\n\nContents of EVEN file\n\n");
     while((number = getw(f3)) != EOF)
        printf("%4d", number);

     fclose(f2);
     fclose(f3);

}
```

Output

```
Contents of DATA file
111 222 333 444 555 666 777 888 999 000 121 232 343 454 565 -1

Contents of ODD file
111 333 555 777 999 121 343 565

Contents of EVEN file
222 444 666 888   0 232 454
```

Fig. 13.2 Operations on integer data

The fprintf and fscanf Functions

So far, we have seen functions, that can handle only one character or integer at a time. Most compilers support two other functions, namely **fprintf** and **fscanf** , 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

```
fprintf(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.**

WORKED-OUT PROBLEM 13.3 H

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.13.3. 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.…é

Program

```
#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 name  Number   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
```

C-3	247	31.75	104	3302.00

Fig. 13.3 Operations on mixed data types

Test Your Skills

Write a C program that reads the contents of one file and writes its reverse in another file. **[M]**

Error Handling During I/O Operations

LO 13.3
Determine how error handling is performed during I/O operations

It is possible that an error may occur during I/O operations on a file. Typical error situations include the following:

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 up to 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");
```

WORKED-OUT PROBLEM 13.4 E

Write a program to illustrate error handling in file operations.

The program shown in Fig. 13.4 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 non-zero integer when the entire data has been read, and hence the program prints the message “Ran out of data” and terminates further reading.

Program

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

main()
{
        char filename[10];
        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");
        scanf("%s", filename);

        if((fp2 = fopen(filename,"r")) == NULL)
        {
                printf("Cannot open the file.\n");
                exit(0);
        }
        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
TEST
10
20
30
40
```

```
50
60
70
80
90
100

Ran out of data.
```

Fig. 13.4 Illustration of error handling in file operations

RANDOM ACCESS TO FILES

LO 13.4
Explain 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 return 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.

Examples in Table 13.2 illustrate the operations of the **fseek** function:

Table 13.2 Operations of 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.

WORKED-OUT PROBLEM 13.5 E

Write a program that uses the functions **ftell** and **fseek**.

A program employing **ftell** and **fseek** functions is shown in Fig. 13.5 . We have created a file **RANDOM** with the following contents:

Position ---->	0	1	2	. . .	25
Character stored ---->	A	B	C	. . .	Z

We are reading the file twice. First, we are reading the content of every fifth position and printing its value along with its position on the screen. The second time, we are reading the contents of the file from the end and printing 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.

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

```
#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. 13.5 Illustration of fseek and ftell functions

WORKED-OUT PROBLEM 13.6 **M**

Write a program to append additional items to the file INVENTORY created in Program 13.3 and print the total contents of the file.

The program is shown in Fig. 13.6 . 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

```
#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;
     void append (struct invent_record *x, file *y);
     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 %8.2f %8d\n",
          item.name, item.number, item.price, item.quantity);
  }
  fclose(fp);
}
void append(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. 13.6 Adding items to an existing file

COMMAND LINE ARGUMENTS

LO 13.5
Know the 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

where **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(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.

WORKED-OUT PROBLEM 13.7 **H**

Write a C program to reverse the first n character in a file. The file name and the value of n are specified on the command line. Incorporate validation of arguments, that is, the program should check that the number of arguments passed and the value of n that are meaningful.

Program

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

void main (int argc, char *argv[ ])
{
            FILE *fs;
            char str[100];
            int i, n, j;
            if(argc!-3)  /*Checking the number of arguments given at command line*/
            {
                      puts("Improper number of arguments.");
                      exit(0);
            }
            n-atoi(argv[2]);
            fs - fopen(argv[1], 'r');  /*Opening the source file in read mode*/
            if(fs--NULL)
            {
                      printf("Source file cannot be opened.");
                      exit(0);
            }
            i-0;
            while(1)
            {
                    if((str[i]-fgetc(fs))!-EOF)  /*Reading contents of file character by
                                                                          character*/
                                   j-i+1:
                      else
                                   break;
            }
            fclose(fs);
            fs-fopen(argv[1],'w');  /*Opening the file in write mode*/
            if(n<0||n>strlen(str))
            {
                       printf("Incorrect value of n. Program will terminate...\n\n");
                       exit(1);
            }
            j-strlen(str);
            for (i-1;i<-n;i++, j--)
                      fputc(str[j],fs);
            fclose(fs);
           printf("\n%d characters of the file successfully printed in reverse order",n);
}
```

Output

```
D:\TC\BIN\program source.txt 5
5 characters of the file successfully printed in reverse order
```

Fig. 13.7 Program to reverse n characters in a file

WORKED-OUT PROBLEM 13.8

E

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 13.8 shows the use of command line arguments. The command line is

F12_7 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

```
#include  <stdio.h>

main(int argc, char *argv[])
{
    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_7 TEXT AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGG

No. of arguments in Command line = 9

Contents of TEXT file

AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG
```

```
C:\C\F12_7.EXE
 TEXT
    AAAAAA
          BBBBBB
                CCCCCC
                      DDDDDD
                            EEEEEE
                                  FFFFFF
                                        GGGGGG
```

Fig. 13.8 Use of command line arguments

Test Your Skills

Write a C program that uses command line arguments to open and read a file. [E]

KEY CONCEPTS

- **FILENAME:** It is a string of characters that make up a valid filename for the operating system. **[LO 13.1]**
- **fseek :** It is a function that sets the position to a desired point in the file. **[LO 13.4]**
- **ftell :** It is a function that returns the current position in the file. **[LO 13.4]**
- **rewind :** It is a function that sets the position to the beginning of the file. **[LO 13.4]**
- **COMMAND LINE ARGUMENT:** It is a parameter supplied to a program from command prompt when the program is invoked. **[LO 13.5]**

ALWAYS REMEMBER

- Do not try to use a file before opening it. **[LO 13.1]**
- Remember, when an existing file is open using 'w' mode, the contents of file are deleted. **[LO 13.1]**
- When a file is used for both reading and writing, we must open it in 'w+' mode. **[LO 13.1]**
- It is an error to omit the file pointer when using a file function. **[LO 13.1]**

- It is an error to open a file for reading when it does not exist. **[LO 13.1]**
- It is an error to access a file with its name rather than its file pointer. **[LO 13.1]**
- It is a good practice to close all files before terminating a program. **[LO 13.1]**
- EOF is integer type with a value –1. Therefore, we must use an integer variable to test EOF. **[LO 13.2]**
- It is an error to try to read from a file that is in write mode and vice versa. **[LO 13.2]**
- To avoid I/O errors while working with files, it is a good practice to include error handling code in programs by using functions such as feof and ferror. **[LO 13.3]**
- It is an error to attempt to place the file marker before the first byte of a file. **[LO 13.4]**
- It is a good practice to check the return value of fseek function every time it is invoked. A positive return value ensures that the file pointer is within the file boundaries. **[LO 13.5]**

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

13.1 Which of the following functions are used to write an integer to a file? **[LO 13.2 E]**

(a) fprintf()

(b) putw()

(c) putc()

(d) puts()

13.2 Which of the following symbols are used to indicate EOF? **[LO 13.2 M]**

(a) ^z

(b) ~z

(c) EOF

(d) None of these

13.3 While reading and writing integer data to a file, an EOF is indicated by which of the following integer values? **[LO 13.2 M]**

(a) –1

(b) 0

(c) –9999

(d) None of these

13.4 Which of the following status-enquiry functions is used to detect I/O errors during file handling?

(a) feof() **[LO 13.3 E]**

(b) error()

(c) ferror()

(d) None of these

REVIEW QUESTIONS

13.1 State whether the following statements are *true* or *false* .

(a) A file must be opened before it can be used. **[LO 13.1 E]**

(b) All files must be explicitly closed. **[LO 13.1 E]**

(c) Files are always referred to by name in C programs. **[LO 13.1 E]**

(d) Using **fseek** to position a file beyond the end of the file is an error. **[LO 13.1 M]**

(e) Function **fseek** may be used to seek from the beginning of the file only. **[LO 13.4 E]**

(f) If an existing file is opened in write-mode, then the current contents of the file are deleted.

(g) If a non-existent file is opened in read-mode, then it **[LO 13.1 E]** results in an error. **[LO 13.1 E]**

(h) If a file is not closed through an explicit fclose() statement, then the file remains open even after the termination of the program. **[LO 13.2 M]**

(i) Both scanf() and fscanf() functions return the number of items that are successfully read.

13.2 Fill in the blanks in the following statements. **[LO 13.2 M]**

(a) The mode ________ is used for opening a file for updating. **[LO 13.1 E]**

(b) The function ______ is used to write data to randomly accessed file. **[LO 13.2 M]**

(c) The function _____gives the current position in the file. **[LO 13.4 E]**

(d) The function ________ may be used to position a file at the beginning. **[LO 13.4 M]**

(e) fprintf() ___________ function is used to write mixed data to a file. **[LO 13.2 E]**

(f) feof() __________ function is used to test the end of file condition i.e. whether the end of the file has reached or not. **[LO 13.3 M]**

13.3 Describe the use and limitations of the functions **getc** and **putc.** **[LO 13.2 E]**

13.4 What is the significance of EOF? **[LO 13.2 M]**

13.5 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? **[LO 13.1 H]**

13.6 Distinguish between the following functions:

(a) getc and getchar **[LO 13.2 E]**

(b) printf and fprintf **[LO 13.2 E]**

(c) feof and ferror **[LO 13.3 M]**

13.7 How does an append mode differ from a write mode? **[LO 13.1 M]**

13.8 What are the common uses of **rewind** and **ftell** functions? **[LO 13.4 E]**

13.9 Explain the general format of **fseek** function? **[LO 13.4 E]**

13.10 What is the difference between the statements **rewind(fp);** and **fseek(fp,0L,0);?** **[LO 13.4 M]**

13.11 What does the following statement mean? **[LO 13.1 H]**

FILE(*p) (void)

13.12 What does the following statement do? **[LO 13.2 M]**

```
while ( (c = getchar( ) != EOF )
putc(c, fl);
```

13.13 What does the following statement do? **[LO 13.2 M]**

```
while ( (m = getw(fl) ) != EOF)
printf("%5d", m);
```

13.14 What does the following segment do? **[LO 13.2 H]**

```
. . . .
for (i = 1; i <= 5; i++ )
{
fscanf(stdin, "%s", name);
fprintf(fp, "%s", name);
}
. . . .
```

13.15 What is the purpose of the following functions? **[LO 13.3 E]**

(a) feof ()

(b) ferror ( )

13.16 Give examples of using **feof** and **ferror** in a program. **[LO 13.3 M]**

13.17 Can we read from a file and write to the same file without resetting the file pointer? If not, why? **[LO 13.2 H]**

13.18 When do we use the following functions? **[LO 13.4 M]**

(a) free ( )

(b) rewind ( )

13.19 Describe an algorithm that will append the contents of one file to the end of another file. **[LO 13.1 E]**

DEBUGGING EXERCISES

13.1 Find error, if any, in the following statements: **[LO 13.1 M]**

```
FILE fptr;
fptr = fopen ("data", "a+");
```

PROGRAMMING EXERCISES

13.1 Write a program to copy the contents of one file into another. **[LO 13.2 E]**

13.2 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. **[LO 13.2 M]**

13.3 Write a program that compares two files and returns 0 if they are equal and 1 is they are not. **[LO 13.2 E]**

13.4 Write a program that appends one file at the end of another. **[LO 13.4 E]**

13.5 Write a program that reads a file containing integers and appends at its end the sum of all the integers. **[LO 13.4 M]**

13.6 Write a program that prompts the user for two files, one containing a line of text known as source file and other, an empty file known as target file and then copies the contents of source file into target file. **[LO 13.1, 13.2, 13.5 H]**

Modify the program so that a specified character is deleted from the source file as it is copied to the target file.

13.7 Write a program that requests for a file name and an integer, known as offset value. The program then reads the file starting from the location specified by the offset value and prints the contents on the screen. **[LO 13.4 M]**

Note: If the offset value is a positive integer, then printing skips that many lines. If it is a negative number, it prints that many lines from the end of the file. An appropriate error message should be printed, if anything goes wrong.

13.8 Write a program to create a sequential file that could store details about five products. Details include product code, cost and number of items available and are provided through keyboard. **[LO 13.2 M]**

13.9 Write a program to read the file created in Exercise 13.8 and compute and print the total value of all the five products. **[LO 13.2 M]**

13.10 Rewrite the program developed in Exercise 13.8 to store the details in a random access file and print the details of alternate products from the file. Modify the program so that it can output the details of a product when its code is specified interactively. **[LO 13.2, 13.4 H]**

13.11 Write a C program that uses file handling methods to store records of mixed data in a file. **[LO 13.2 H]**

13.12 Write a C program that uses getw function to read integer values from one file. Subsequently, it uses the putw function to write the integer values in reverse order in another file. **[LO 13.2 M]**

13.13 Write a C program that reads characters from a file and prints their ASCII codes. **[LO 13.2 E]**

13.14 Write a C program that concatenates the contents of two files and writes then in the third file. **[LO 13.2 H]**

13.15 Write a C program that uses fscanf function to read integer values from a file, computes the square of each integer value and places the resultant values in a different file. **[LO 13.2 H]**

INTERVIEW QUESTIONS

13.1 What is the difference between printf and cprintf functions? **[LO 13.2 H]**

13.2 What is the difference between getc, getw and fscanf functions? **[LO 13.2 M]**

13.3 What is the purpose of ftell function? **[LO 13.4 E]**

13.4 What is the purpose of fseek function? **[LO 13.4 E]**

13.5 What is the difference between printf and sprintf functions? **[LO 13.2 M]**

E for Easy, **M** for Medium and **H** for High

Chapter **14**

Dynamic Memory Allocation and Linked Lists

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

- **LO 14.1** Describe dynamic memory allocation
- **LO 14.2** Differentiate between malloc and calloc
- **LO 14.3** Explain how allocated space is altered or released
- **LO 14.4** Discuss the concept of linked lists
- **LO 14.5** Determine how a linked list is implemented

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 concept 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.

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.

LO 14.1
Describe dynamic memory allocation

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 14.1. These functions help us build complex application programs that use the available memory intelligently.

TIP
Dynamic memory allocation is useful when the size required for data items is not known at compile time.

Table 14.1 Memory Allocation Functions

Function	*Task*
malloc	Allocates request 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 14.1 shows the conceptual view of storage of a C program in memory.

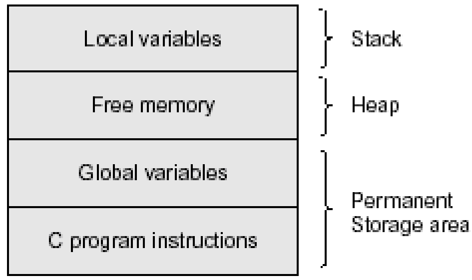


Fig. 14.1 Storage of a C program

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: malloc

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:

LO 14.2
Differentiate between malloc and calloc

```
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:

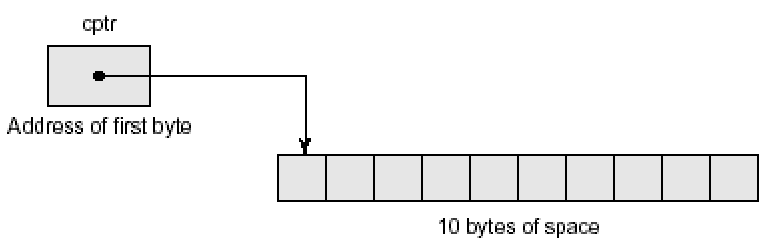
x = (int *) malloc (100 *sizeof(int));

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 as:



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.14.2.

WORKED-OUT PROBLEM 14.1 **E**

Write a program that uses a table of integers whose size will be specified interactively at run time.

The program is given in Fig.14.2. 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
#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("\n Address 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. 14.2 Memory allocation with malloc

ALLOCATING MULTIPLE BLOCKS OF MEMORY: calloc

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);
}
```

WORKED-OUT PROBLEM 14.2

Write a C program to input a list of numbers within the range 0-4 from the user and then find the frequency of each one of them.

Program

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

main(){

        int *num, *freq, i, size;

        printf("\nEnter the size of the list: ");
        scanf("%d", &size);
        num = (int *)malloc(size * sizeof(int));
        printf("\nEnter the numbers: ");
        for(i = 0; i < size; i++) {
                printf("\nEnter num[%d]: ", i);
                scanf("%d", &num[i]);
                if(num[i]<0 || num[i]>4) {
                        printf("\nNumber should be within range (0-4)");
                        i--;
                        continue;
                }
        }
        freq = (int *)calloc(5, sizeof(int));
        for(i = 0; i<size; i++)
                freq[num[i]]++;
        printf("\nThe frequencies of the numbers are: ");
        for(i = 0; i<5; i++)
                printf("\nfreq[%d] = %d", i, freq[i]);
}
```

Output

```
Enter the size of the list: 10
Enter the numbers:
Enter num[0]: 2
Enter num[1]: 3
Enter num[2]: 4
Enter num[3]: 2
Enter num[4]: 7
Number should be within range (0-4)
Enter num[4]: 1
Enter num[5]: 0
Enter num[6]: 1
Enter num[7]: 2
Enter num[8]: 4
Enter num[9]: 3
The frequencies of the numbers are:
freq[0] = 1
freq[1] = 2
freq[2] = 3
freq[3] = 2
freq[4] = 2
```

Fig. 14.3 Memory Allocation with calloc

RELEASING THE USED SPACE: free

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.

LO 14.3
Explain how allocated space is altered or released

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:

TIP
It is the programmers responsibility to free space allocated using dynamic memory allocation after use.

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. We should remember two things here:

1. It is not the pointer that is being released but rather what it points to.
2. To release an array of memory that was allocated by **calloc** we need only to release the pointer once. It is an error to attempt to release elements individually.

The use of **free** function has been illustrated in Worked-Out Problem 14.3.

ALTERING THE SIZE OF A BLOCK: realloc

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 or 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 necessary to test the success of operation before proceeding further. This is illustrated in the program of Worked-Out Problem 14.3.

WORKED-OUT PROBLEM 14.3 **M**

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. 14.4 . 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
            #include <string.h>
            #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. 14.4 Reallocation and release of memory space

Self-Referential Structures

Self-Referential structures are those in which at least one member is a pointer to another structure of the same type as itself. It is self-referential in the sense that it **references** another structure of the same type as **itself** . A typical declaration of a self-referential structure is as follows:

```
struct example {
            declaration of other members;
            struct example *self_type_pointer;
};
```

The pointer, named self_type_pointer whose type is same as the structure itself makes it a self-referential structure. Consider the program below that demonstrates the use of a self-referential structure:

```
#include <stdio.h>
struct demo {
            int num;
            struct demo *other;
};
main(){
            struct demo first, second;
            first.num = 25;
            first.other = &second; /*first.other points to the structure, second*/
            second.num = 35;
            second.other = &first; /*second.other points to the structure, first*/
            printf("\nFirst Number = %d", first.num); /*displays num of first*/
            printf("\nFirst Number = %d", (second.other)->num); /*displays num of first
                                                                  through other of second*/
            printf("\nSecond Number = %d', second.num); /*displays num of second*/
            printf("\nSecond Number = %d', (first.other)->num); /*displays num second
                                                                  through other of first*/
            printf("\nAddress of first = %u", &first);
            printf("\nAddress of first = %u", second.other); /*same as &first*/
            printf("\nAddress of second = %u", &second);
            printf("\nAddress of second = %u", first.other); /*same as &second*/
}/*end of main*/
```

Consider that the structure, *first* has been allocated space starting at address 1000 and the structure, *second* has been allocated space starting at address 2000.

When the above program is run the output will be as follows:

First number = 25

First number = 25

Second number = 35

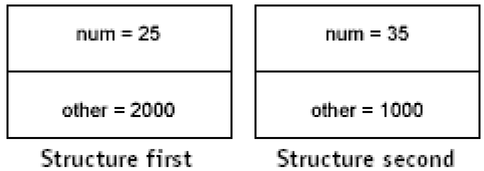
Second number = 35

Address of first = 1000

Address of first = 1000

Address of second = 2000

Address of second = 2000



Mark the use of the arrow operator (->) with *first.other* and *second.other* as both of them are pointer members.

Self-referential structures have a lot of applications. The ones that stand out are the use of self-referential structures for representing linked data structures such as linked lists, binary trees, etc.

CONCEPTS OF LINKED LISTS

LO 14.4
Discuss the concept of 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 real-life 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. 14.5 . 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.

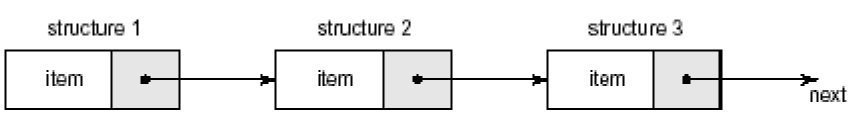
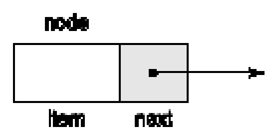


Fig. 14.5 A linked list

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;
};
```

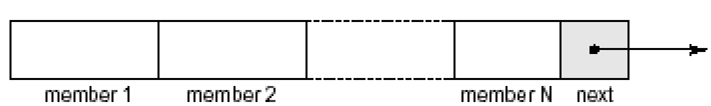
The first member is an integer item and the second a pointer to the next node in the list as shown below. Remember, the **item** is an integer here only for simplicity, and could be any complex data type.



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** .



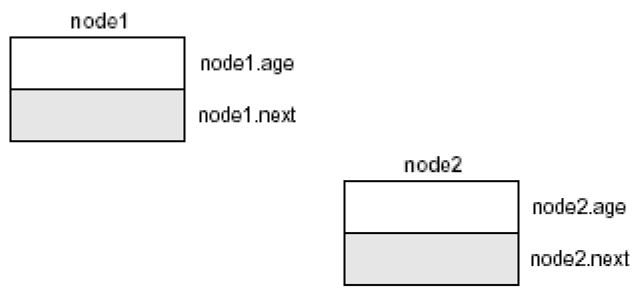
Let use consider a simple example to illustrate the concept of linking. Suppose we define a structure as follows:

```
struct link_list
{
     float age:
     struct link_list *next;
};
```

For simplicity, let as 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;

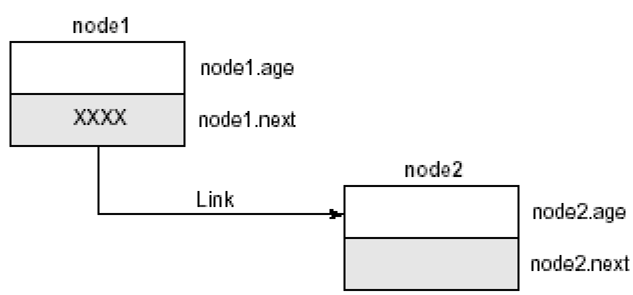
This statement creates space for two nodes each containing two empty fields as shown:



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:

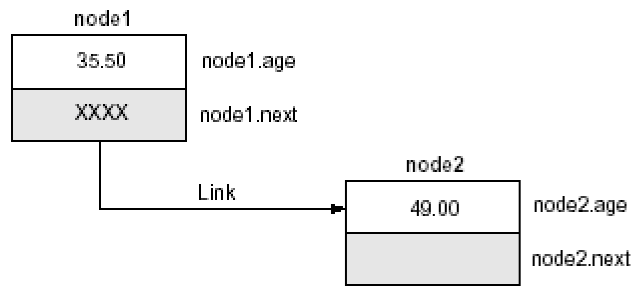


"xxxx" is the address of **node2** where the value of the variable **node2.age** will be stored. Now let us assign values to the field age.

node1.age = 35.50;

node2.age = 49.00;

The result is as follows:



We may continue this process to create a liked list of any number of values.

For example:

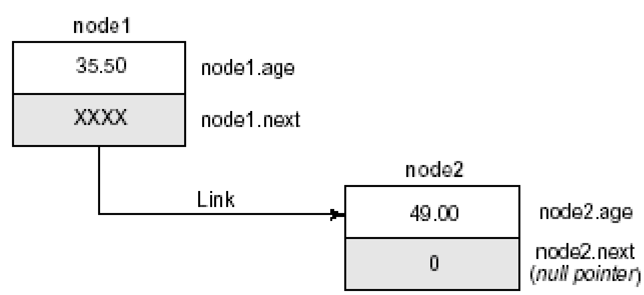
node2.next = &node3;

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 as shown:



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);

Advantages of Linked Lists

A linked list is *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 most important advantage is that the linked lists provide flexibility is allowing the items to be rearranged efficiently. It is easier to insert or delete items by rearranging the links. This is shown in Fig. 14.6.

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 linked 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.

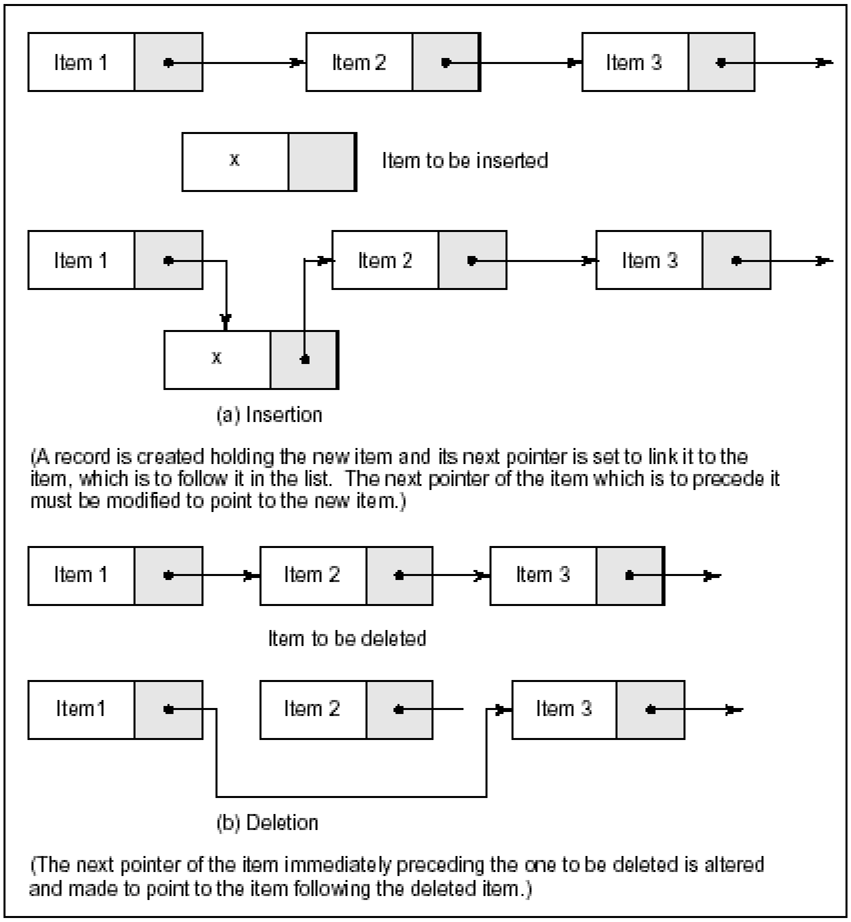


Fig. 14.6 Insertion into and deletion from a linked list

TYPES OF LINKED LISTS

There are different types of lined lists. The one we discussed so far is known as *linear singly* linked list. The other *linked* lists are as follows:

- 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 list 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 circular form. Figure 14.7 illustrates various kinds of linked lists.

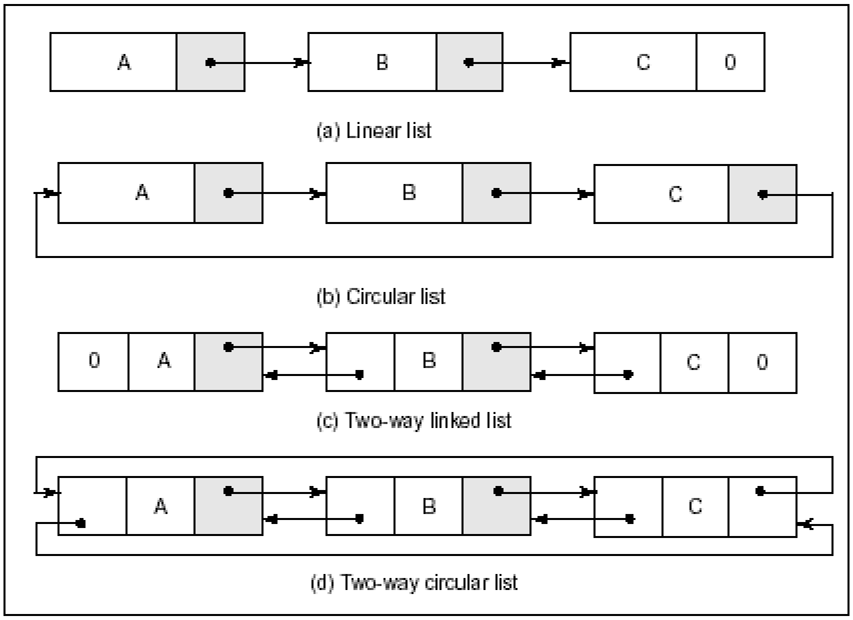


Fig. 14.7 Different types of linked lists

POINTERS REVISITED

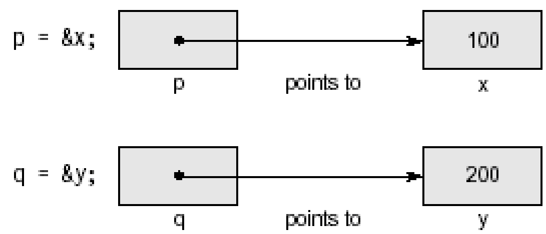
LO 14.5
Determine how a linked list is implemented

The concept of pointers was discussed in Chapter 12. 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 illustrate 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

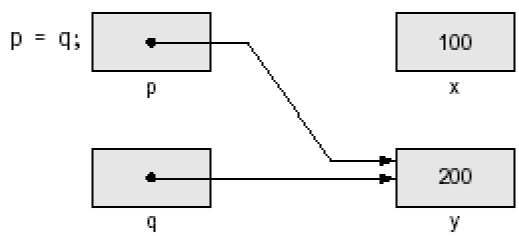


The pointer **p** contains the address of **x** and **q** contains the address of y.

***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.

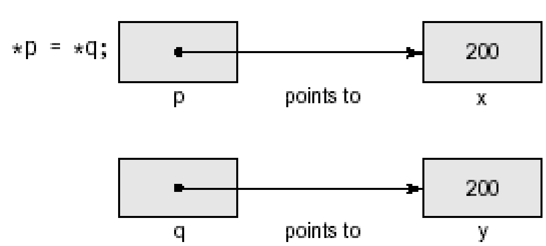


Both the pointer variables point to the same variable.

***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 address 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));

CREATING A LINKED LIST

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 sub list).
5. Looking up an item for editing or printing.
6. Inserting an item.

7. Deleting an item.
8. Concatenating two lists.

In Section 14.7 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** .)

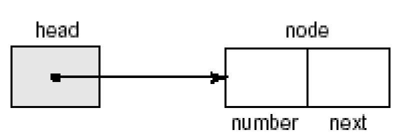
Anonymous locations are created using pointers and dynamic memory allocation functions such as **malloc** . We use a pointer **head** to create and access anonymous nodes. Consider the following:

```
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

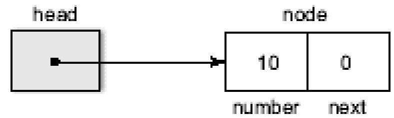
head = (node *) malloc(sizeof(node));

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 self-replacem ent statement such as:

head = head –> next;

The Worked-Out Problem 14.4 shows creation of a complete linked list and printing of its contents using recursion.

WORKED-OUT PROBLEM 14.4 **H**

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. 14.8 first allocates a block of memory dynamically for the first node 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. Not 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 return 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

```
#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");
        printf(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);  */ Recursion occurs */
        }
        return;
}
void print(node *list)
{
        if(list->next != NULL)
        {
             printf("%d-->",list ->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. 14.8 Creating a linear linked 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 the *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 sentinel node, if it exists).

WORKED-OUT PROBLEM 14.5

Write a function to insert a given item before a specified node known as key node.

The function **insert** shown in Fig. 14.9 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:

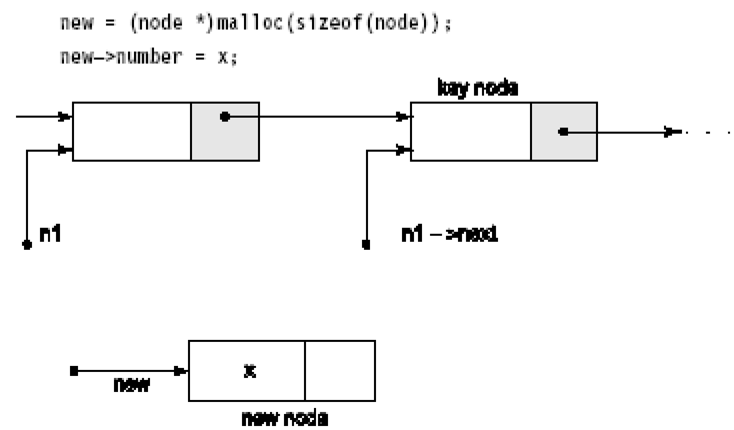
```
new->number = x;
new->next = head;
head = new;
```

```
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 if last) ");
            scanf("%d", &key);

            if(head->number == key)      /* new node is first */
            {
                new = (node *)malloc(size of(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 *lists, 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);
}
```

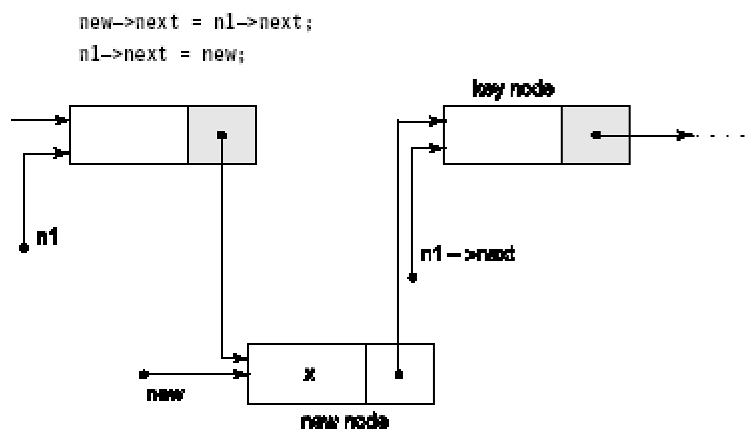
Fig. 14.9 A function for inserting an item into a linked list

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 as:

Before insertion



After insertion



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 deletion 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

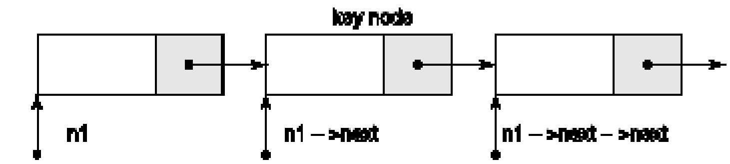
The memory space of deleted node may be released for re-use. As in the case of insertion, the process of deletion also involves search for the item to be deleted.

WORKED-OUT PROBLEM 14.6 **H**

Write a function to delete a specified node.

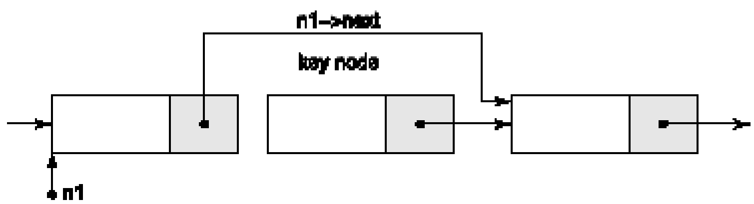
A function to delete a specified node is given in Fig. 14.10 . The function first checks whether the specified item belongs to the first node. If yes, then the pointer to the second node is temporarily assigned the pointer variable **p** , the memory space occupied by the first node is freed and the 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 the 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 if freed. The figure below shows the relative position of the key node.



The execution of the following code deletes the key node.

```
p = n1->next->next;
free (n1->next);
n1->next = p;
```



```
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("\n What 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
            {
              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 keynode */

                free(n1->next);         /* free key node */
                n1->next = p;           /* establish link */
              }
            }
            return(head);
}
            /* USE FUNCTION find() HERE */
```

Fig. 14.10 A function for deleting an item from linked list

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 insertion 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. 14.11 a). 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 the beginning of list, 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. 14.11 b). Stacks are sometimes referred to as "last in, first out" (LIFO) structure.

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.

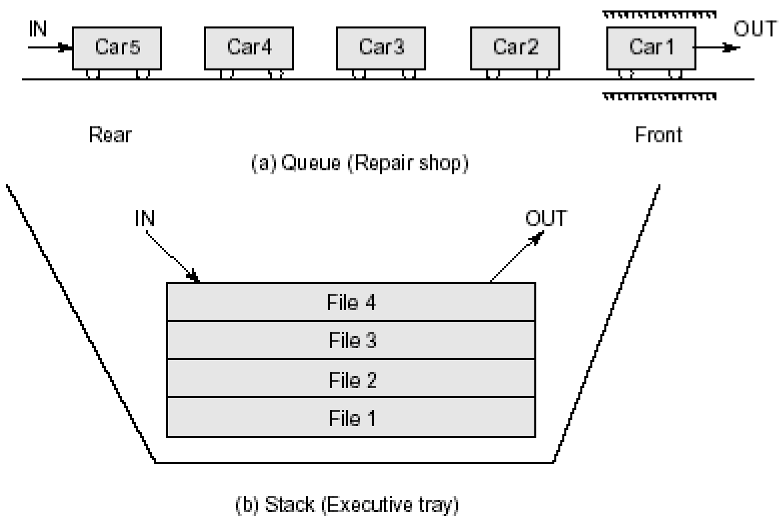


Fig. 14.11 Application of linked lists

TEST YOUR SKILLS

Write a C program to implement a stack using linked lists. **[H]**

KEY CONCEPTS

- **DYNAMIC MEMORY ALLOCATION:** It is the process of allocating memory space at run time. **[LO 14.1]**
- **HEAP:** It is the free memory space that is available for dynamic allocation during program execution. **[LO 14.1]**

- **STACK:** It is a push-down list that stores the elements based on last in, first out (LIFO) principle. **[LO 14.1]**
- **calloc FUNCTION:** It is a function that allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory. **[LO 14.2]**
- **malloc FUNCTION :** It is a function that allocates requested size of bytes and returns a pointer to the first byte of the allocated space. **[LO 14.2]**
- **sizeof OPERATOR :** It is used to determine the size of a data type. **[LO 14.2]**
- **re alloc FUNCTION :** It is a function that modifies the previously allocated memory space. **[LO 14.3]**
- **LINKED LIST:** It is a collection of structures ordered not by their physical placement in memory but by the logical links that are stored as a part of the data in the structure itself. **[LO 14.4]**
- **NULL POINTER :** It is a collection of structures ordered not by their physical placement in memory but by the logical links that are stored as a part of the data in the structure itself. **[LO 14.4]**

ALWAYS REMEMBER

- Release the dynamically allocated memory when it is no longer required to avoid any possible "memory leak". **[LO 14.1]**
- Use the **sizeof** operator to determine the size of a linked list. **[LO 14.2]**
- When using memory allocation functions **malloc** and **calloc,** test for a NULL pointer return value. Print appropriate message if the memory allocation fails. **[LO 14.2]**
- Never call memory allocation functions with a zero size. **[LO 14.2]**
- It is an error to assign the return value from **malloc** or **calloc** to anything other than a pointer. **[LO 14.2]**
- It is an error to release individually the elements of an array created with **calloc** . **[LO 14.2]**

- Using **free** function to release the memory not allocated dynamically with **malloc** or **calloc** is an error. **[LO 14.3]**
- Use of a invalid pointer with **free** may cause problems and, sometimes, system crash. **[LO 14.3]**
- Using a pointer after its memory has been released is an error. **[LO 14.3]**
- It is a logic error to set a pointer to NULL before the node has been released. The node is irretrievably lost. **[LO 14.3]**
- It is an error to declare a self-referential structure without a structure tag. **[LO 14.4]**
- It is a logic error to fail to set the link fie l d in the last node to null. **[LO 14.5]**

BRIEF CASES

---

1. Insertion in a Sorted List [LO 14.2, 14.5 M]

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 Worked-Out Problem 14.4. Figure 14.12 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

```
#include <stdlib.h>
#include <stdio.h>
#define NULL 0

struct linked_list
{
     int number;
     struct linked-list *next;
};
typedef struct linked_lit 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);
     printf("\n\n");
     printf("Input number to be inserted: ");
     scanf("%d", &n);
     head = inert(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(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 -->", list->number);
```

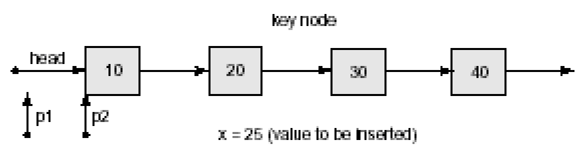
```
        if(list ->next->next = = NULL)
          printf("%d", list->next->number);

        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 -999 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
```

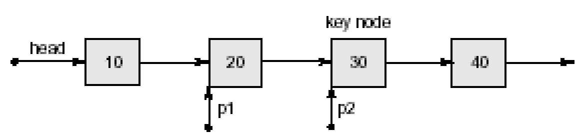
Fig. 14.12 Inserting a number in a sorted linked list

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.

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.



At the start of the search



When key node is found

2. Building a Sorted List **[LO 14.2, 14.5 H]**

The program in Fig. 14.12 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. 14.13. 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

```
#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);
      print("\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 */
                if (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. 14.13 Creation of sorted list from a given list of numbers

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

14.1 Which of the following memory management function is used to release the already allocated memory space? **[LO 14.3 E]**

(a) delete()

(b) release()

(c) free()

(d) None of these

14.2 Which of the following is the core principle on which stack is based? **[LO 14.5 E]**

(a) First in first out

(b) Last in first out

(c) No restriction in insertion or deletion of element

(d) None of these

14.3 ___________ is the free memory space that is available for dynamic allocation during program execution. **[LO 14.1 E]**

(a) Stack

(b) Queue

(c) Heap

(d) Permanent storage area

14.4 In case of _____________ linked list, the last node points to the first node in the list. **[LO 14.5 E]**

(a) Singly

(b) Doubly

(c) Circular

(d) None of these

REVIEW QUESTIONS

14.1 State whether the following statements are *true* or *false*

(a) Dynamically allocated memory can only be accessed using pointers. **[LO 14.1 M]**

(b) **calloc** is used to change the memory allocation previously allocated with **malloc** . **[LO 14.2 E]**

(c) Memory should be freed when it is no longer required. **[LO 14.3 E]**

(d) To ensure that it is released, allocated memory should be freed before the program ends. **[LO 14.3 E]**

(e) Only one call to free is necessary to release an entire array allocated with **calloc** . **[LO 14.3 M]**

(f) The link field in a linked list always points to successor. **[LO 14.4 E]**

(g) The first step in a adding a node to a linked list is to allocate memory for the next node. **[LO 14.5 M]**

(h) When realloc() function reserves a new block of memory in place of old block, then the contents of the old block are deleted. **[LO 14.3 M]**

(i) In case of a linked list, it is not necessary that the list nodes are physically placed together in the memory just like arrays. **[LO 14.4 M]**

(j) A linked-list implemented using dynamic memory allocation grows or shrinks in size as per requirement; thus prevent memory wastage. **[LO 14.4 E]**

14.2 Fill in the blanks in the following statements

(a) A_______________identifies the last logical node in a linked list. **[LO 14.4 M]**

(b) Function _______________ is used to dynamically allocate memory to arrays. **[LO 14.2 E]**

(c) A________________ is an ordered collection of data in which each element contains the location of the next element. **[LO 14.4 E]**

(d) Stacks are referred to as ________________. **[LO 14.1 M]**

(e) Data structures which contain a member field that points to the same structure type are called __________________ structures. **[LO 14.4 M]**

(f) If a memory management function fails to reserve required memory space, then it returns _______ value. **[LO 14.2 E]**

(g) malloc function reserves a block of memory of specified size and returns a pointer of type _______. **[LO 14.2 M]**

(h) ___________ pointer is use to initialize a pointer that points to nothing. **[LO 14.4 E]**

14.3 What is a linked list? How is it represented? **[LO 14.4 E]**

14.4 What is dynamic memory allocation? How does it help in building complex programs? **[LO 14.1 E]**

14.5 What is the principal difference between the functions **malloc** and **calloc** ? **[LO 14.2 M]**

14.6 Why a linked list is called a dynamic data structure? What are the advantages of using linked lists over arrays? **[LO 14.4 M]**

14.7 Describe different types of linked lists. **[LO 14.4 E]**

14.8 The following code is defined in a header file *list.h* **[LO 14.5 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.

14.9 What does the following code achieve? **[LO 14.2 E]**

```
int * p ;
p = malloc (sizeof (int) ) ;
```

14.10 What does the following code do? **[LO 14.2 E]**

```
float *p;
p = calloc (10,sizeof(float) ) ;
```

14.11 What is the output of the following code? **[LO 14.2 M]**

```
int i, *ip ;
ip = calloc ( 4, sizeof(int) );
for (i = 0 ; i < 4 ; i++)
          * ip++ = i * i;
for (i = 0 ; i < 4 ; i++)
          printf("%d\n", *—ip );
```

14.12 What is printed by the following code? **[LO 14.2 M]**

```
int *p;
 p =  malloc (sizeof (int) );
*p =  100 ;
 p =  malloc (sizeof (int) );
*p =  111;
 printf("%d", *p);
```

14.13 What is the output of the following segment? **[LO 14.5 M]**

```
struct node
{
   int m ;
   struct node *next;
}  x, y, z, *p;
 x.m = 10 ;
 y.m = 20 ;
 z.m = 30 ;
 x.next = &y;
 y.next = &z;
 z.next = NULL;
 p = x.next;
 while (p != NULL)
{
 printf("%d\n", p -> m);
 p = p -> next;
}
```

DEBUGGING EXERCISES

14.1 Find errors, if any, in the following memory management statements:

(a) *ptr = (int *)malloc(m, sizeof(int)); **[LO 14.2 M]**

(b) table = (float *)calloc(100); **[LO 14.2 M]**

(c) node = free(ptr); **[LO 14.3 M]**

14.2 Identify errors, if any, in the following structure definition statements: **[LO 14.5 E]**

```
struct
{
     char name[30]
     struct *next;
};
typedef struct node;
```

PROGRAMMING EXERCISES

14.1 In Worked-Out Problem 14.4, we have used print() in recursive mode. Rewrite this function using iterative technique in for loop. **[LO 14.5 M]**

14.2 Write a menu driven program to create a linked list of a class of students and perform the following operations: **[LO 14.5 H]**

(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 Review Question 14.8.

14.3 Write recursive and non-recursive functions for reversing the elements in a linear list. Compare the relative efficiencies of them. **[LO 14.5 M]**

14.4 Write an interactive program to create linear linked lists of customer names and their telephone numbers. The program should be menu driven and include features for add ing a new customer and deleting an existing customer. **[LO 14.5 M]**

14.5 Modify the above program so that the list is always maintained in the alphabetical order of customer names. **[LO 14.5 H]**

14.6 Develop a program to combine two sorted lists to produce a third sorted lists which contains one occurrence of each of the elements in the original lists. **[LO 14.5 M]**

14.7 Write a program to create a circular linked list so that the input order of data item is maintained. Add function to carry out the following operations on circular linked list. **[LO 14.5 H]**

a. Count the number of nodes

b. Write out contents

c. Locate and write the contents of a given node

14.8 Write a program to construct an ordered doubly linked list and write out the contents of a specified node. **[LO 14.5 M]**

14.9 Write a function that would traverse a linear singly linked list in reverse and write out the contents in reverse order. **[LO 14.5 M]**

14.10 Given two ordered singly linked lists, write a function that will merge them into a third ordered list. **[LO 14.5 M]**

14.11 Write a function that takes a pointer to the first node in a linked list as a parameter and returns a pointer to the last node. NULL should be returned if the list is empty. **[LO 14.5 M]**

14.12 Write a function that counts and returns the total number of nodes in a linked list. **[LO 14.5 E]**

14.13 Write a function that takes a specified node of a linked list and makes it as its last node. **[LO 14.5 M]**

14.14 Write a function that computers and returns the length of a circular list. **[LO 14.5 E]**

14.15 Write functions to implement the following tasks for a doubly linked list. **[LO 14.5 H]**

(a) To insert a node.

(b) To delete a node.

(c) To find a specified node.

INTERVIEW QUESTIONS

14.1 You need to print the size of a variable without using the sizeof operator. How would you do that?

14.2 What will be the output of following code? **[LO 14.1 H]**

```
void main()
{
printf("Address of main = %p",main);
}
```

[LO 14.1 H]

14.3 What is a wild pointer? **[LO 14.4 M]**

14.4 What is the difference between NULL and NUL? **[LO 14.4 H]**

14.5 What is a dangling pointer? **[LO 14.3 H]**

14.6 What is a restricted pointer? **[LO 14.1 H]**

14.7 What is the main difference between stack and queue? **[LO 14.5 M]**

E for Easy, **M** for Medium and **H** for High

Chapter 15

The Preprocessor

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

- **LO 15.1** Describe macro substitution
- **LO 15.2** Discuss file inclusion directive
- **LO 15.3** Determine how compiler control directives are used
- **LO 15.4** Identify the preprocessor directives introduced as part of ANSI additions

INTRODUCTION

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 15.1.

Table 15.1 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 which are as follows:

1. Macro substitution directives.
2. File inclusion directives.
3. Compiler control directives.

Macro Substitution

LO 15.1
Describe 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 as follows:

1. Simple macro substitution.
2. Argumented macro substitution.
3. Nested macro substitution.

Simple Macro Substitution

Simple string replacement is commonly used to define constants. Examples of definition of constants are as follows:

```
#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. A definition, such as

#define M 5

will replace all occurrences of M with 5, starting from the line of definition to the end of the program. However, a macro inside a string does not get replaced. Consider the following two lines:

total = M * value;

printf("M = %d\n", M);

These two lines would be changed during preprocessing as follows:

total = 5 * value;

printf("M = %d\n", 5);

Notice that the string **"M=%d\n"** is left unchanged.

A macro definition can include more than a simple constant value. It can include expressions as well. Following are valid definitions:

```
#define    AREA      5 * 12.46
#define    SIZE      sizeof(int) * 4
#define    TWO_PI    2.0 * 3.1415926
```

Whenever we use expressions for replacement, care should be taken to prevent an unexpected order of evaluation. Consider the evaluation of the equation

ratio = D/A;

where D and A are macros defined as follows:

```
#define identifier string
```

The result of the preprocessor's substitution for D and A is:

ratio = 45–22/78+32;

This is certainly different from the expected expression

(45 – 22)/(78+32)

Correct results can be obtained by using parentheses around the strings as:

```
#define   D    (45 – 22)
#define   A    (78 + 32)
```

It is a wise practice to use parentheses for expressions used in macro definitions.

As mentioned earlier, the preprocessor performs a literal text substitution, whenever the defined name occurs. This explains why we cannot use a semicolon to terminate the #define statement. This also suggests that we can use a macro to define almost anything. For example, we can use the definitions

```
#define    TEST     if (x > y)
#define    AND
#define    PRINT    printf("Very Good. \n");
```

to build a statement as follows:

TEST AND PRINT

The preprocessor would translate this line to

if(x>y) printf("Very Good.\n");

Some tokens of C syntax are confusing or are error-prone. For example, a common programming mistake is to use the token = in place of the token == in logical expressions. Similar is the case with the token &&.

Following are a few definitions that might be useful in building error free and more readable programs:

```
#define    EQUALS        ==
#define    AND           &&
#define    OR            ||
#define    NOT_EQUAL     !=
#define    START         main() {
#define    END           }
#define    MOD           %
#define    BLANK_LINE    printf("\n");
#define    INCREMENT     ++
```

An example of the use of syntactic replacement is:

```
START
... ....
... ....
if(total EQUALS 240 AND average EQUALS 60)
INCREMENT count;
... ....
... ....
END
```

Macros with Arguments

The preprocessor permits us to define more complex and more useful form of replacements. It takes the form:

```
#define    identifier(f1, f2, . . . . . . . fn)    string
```

Notice that there is no space between the macro *identifier* and the left parentheses. The identifiers f1, f2, … … .,fn are the formal macro arguments that are analogous to the formal arguments in a function definition.

There is a basic difference between the simple replacement discussed above and the replacement of macros with arguments. Subsequent occurrence of a macro with arguments is known as a *macro call* (similar to a function call). When a macro is called, the preprocessor substitutes the string, replacing the formal parameters with the actual parameters. Hence, the string behaves like a template.

A simple example of a macro with arguments is

```
#define          CUBE(x)                    (x*x*x)
```

If the following statement appears later in the program

volume = CUBE(side);

Then the preprocessor would expand this statement to:

volume = (side * side * side);

Consider the following statement:

volume = CUBE(a+b);

This would expand to:

volume = (a+b * a+b * a+b);

which would obviously not produce the correct results. This is because the preprocessor performs a blind test substitution of the argument a+b in place of x. This shortcoming can be corrected by using parentheses for each occurrence of a formal argument in the *string* .

Example:

```
#define          CUBE(x)                    ( (x) * (x) *(x) )
```

This would result in correct expansion of **CUBE(a+b)** as:

volume = ((a+b) * (a+b) * (a+b));

Remember to use parentheses for each occurrence of a formal argument, as well as the whole *string* .

Some commonly used definitions are:

```
#define    MAX(a,b)       (((a) > (b)) ? (a) : (b))
#define    MIN(a,b)       (((a) < (b)) ? (a) : (b))
#define    ABS(x)         (((x) > 0) ? (x) : (–(x)))
#define    STREQ(s1,s2)   (strcmp((s1), (s2)) == 0)
#define    STRGT(s1,s2)   (strcmp((s1), (s2)) > 0)
```

Nesting of Macros

We can also use one macro in the definition of another macro. That is, macro definitions may be nested. For instance, consider the following macro definitions.

```
#define    M           5
#define    N           M+1
#define    SQUARE(x)   ( (x) * (x) )
#define    CUBE(x)     (SQUARE (x) * (x))
#define    SIXTH(x)    (CUBE(x) * CUBE(x))
```

The preprocessor expands each **#define** macro, until no more macros appear in the text. For example, the last definition is first expanded into

((SQUARE(x) * (x)) * (SQUARE(x) * (x)))

Since **SQUARE (x)** is still a macro, it is further expanded into

((((x)*(x)) * (x)) * (((x) * (x)) * (x)))

which is finally evaluated as x ^{6} .

Macros can also be used as parameters of other macros. For example, given the definitions of M and N, we can define the following macro to give the maximum of these two:

#define MAX(M,N) (((M) > (N)) ? (M) : (N))

Macro calls can be nested in much the same fashion as function calls. Example:

```
#define    HALF(x)     ( (x)/2.0)
#define    Y           HALF(HALF(x))
```

Similarly, given the definition of MAX(a,b) we can use the following nested call to give the maximum of the three values x,y, and z:

MAX (x, MAX(y,z))

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.

Test Your Skills

Write a C program that implements a macro named GREATER(x,y), which prints TRUE or FALSE depending whether x is greater than y or not. **[M]**

File Inclusion

LO 15.2
Discuss file inclusion directive

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:

LO 15.3
Determine how compiler control directives are used

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.

Situation 1

This situation refers to the conditional definition of a macro. We want to ensure that the macro TEST is always defined, irrespective of whether it has been defined in the header file or not. This can be achieved as follows:

```
#include        "DEFINE.H"
#ifndef         TEST
#define         TEST 1
#endif
--- ---
```

DEFINE.H is the header file that is supposed to contain the definition of **TEST** macro. The directive.

#ifndef TEST

searches for the definition of **TEST** in the header file and *if not defined* , then all the lines between the **#ifndef** and the corresponding **#endif** directive are left ‘active’ in the program. That is, the preprocessor directive

 # define TEST is processed.

In case, the TEST has been defined in the header file, the **#ifndef** condition becomes false, therefore the directive **#define TEST** is ignored. Remember, you cannot simply write

 # define TEST 1

because if **TEST** is already defined, an error will occur.

Similar is the case when we want the macro **TEST** never to be defined. Looking at the following code:

```
… … …
#ifdef          TEST
#undef          TEST
#endif
… …
… …
```

This ensures that even if **TEST** is defined in the header file, its definition is removed. Here again we cannot simply say

#undef TEST

because, if TEST is not defined, the directive is erroneous.

Situation 2

The main concern here is to make the program portable. This can be achieved as follows:

```
... ...
... ...
main()
{
        ... ...
        ... ...
    #ifdef MOTOROLA
    {
        ... ...
        ... ...        code for MOTOROLA
        ... ...
    }
    #else
    {
        ... ...
        ... ...        code for INTEL machine
        ... ...
    }
    #endif
        ... ..
        ... ..
}
```

If we want the program to run on MOTOROLA, we include the directive

#define MOTOROLA

in the program; otherwise we don’t. Note that the compiler control directives are inside the function. Care must be taken to put the # character at column one. The compiler compiles the code for MOTOROLA computer if MOTOROLA is defined, or the code for the INTEL machine if it is not.

Situation 3

This is similar to the above situation and therefore the control directives take the following form:

```
#ifdef    ABC
        group-A lines
#else
        group-B lines
#endif
```

Group-A lines are included if the customer **ABC** is defined. Otherwise, group-B lines are included.

Situation 4

Debugging and testing are done to detect errors in the program. While the Compiler can detect syntactic and semantic errors, it cannot detect a faulty algorithm where the program executes, but produces wrong results.

The process of error detection and isolation begins with the testing of the program with a known set of test data. The program is divided down and **printf** statements are placed in different parts to see intermediate results. Such statements are called debugging statements and are not required once the errors are isolated and corrected. We can either delete all of them or, alternately, make them inactive using control directives as:

```
… …
… …
#ifdef TEST
{
        printf("Array elements\n");
        for (i = 0; i< m; i++)
             printf("x[%d] = %d\n", i, x[i]);
}
#endif
… ..
… ..
#ifdef  TEST
        printf(….);
#endif
… …
```

The statements between the directives **#ifdef** and **#endif** are included only if the macro **TEST** is defined. Once everything is OK, delete or undefine the **TEST** . This makes the **#ifdef TEST** conditions false and therefore all the debugging statements are left out.

The C preprocessor also supports a more general form of test condition - **#if** directive. This takes the following form:

```
#if constant expression
{
        statement-1;
        statement-2;
        … …
        … …
}
#endif
```

The *constant-expression* may be any logical expression such as:

```
TEST <= 3
(LEVEL == 1 || LEVEL == 2)
MACHINE == 'A'
```

778899101011111212*

If the result of the constant-expression is nonzero (true), then all the statements between the **#if** and **#endif** are included for processing; otherwise they are skipped. The names **TEST, LEVEL** , etc. may be defined as macros.

ANSI Additions

ANSI committee has added some more preprocessor directives to the existing list given in Table 15.1. They are as follows:

LO 15.4
Identify the preprocessor directives introduced as part of ANSI additions

#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)

KEY CONCEPTS

- **MACRO CALL:** It substitutes the macro identifier with the corresponding string, replacing formal parameters with actual parameters. **[LO 15.1]**
- **MACRO SUBSTITUTION:** It is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. **[LO 15.1]**

- **PREPROCESSOR:** It is a program that processes the source code before it passes through the compiler. **[LO 15.1]**
- **CONDITIONAL COMPILATION :** It is a feature that allows for a particular line or group of lines of code to be compiled if certain condition is met. **[LO 15.3]**
- **STRINGIZING OPERATOR:** It allows a formal argument within a macro function definition to be converted to a string. **[LO 15.4]**

ALWAYS REMEMBER

- Use macros to manage changes made to a program code in a systematic manner . **[LO 15.1]**
- Use #undef to undefine a macro so that it is restricted to only a specific part in a program . **[LO 15.1]**
- Remember that #include <filename> directive searches the file only in the standard directories as defined by the compiler. However, the #include "filename" directive searches the file in the program's source directory; and if the file is not located, it then search for the file in the standard directories . **[LO 15.2]**
- Use #ifndef to endif directives for conditional compilation of program code. **[LO 15.3]**

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

15.1 Which of the following is a file inclusion preprocessor directive? **[LO 15.2 E]**

(a) #include

(b) #define

(c) #pragma

(d) Both 1 and 2

15.2 Which of the following statements is incorrect about compiler control directives? **[LO 15.3 M]**

(a) #ifdef directive must be followed by #endif directive.

(b) #ifndef directive must be followed by #endif directive.

(c) #undef directive undefines macro.

(d) #if directive must be followed by a #else directive.

15.3 Which of the following directives provides alternative test facility by establishing an “if..else..if..” sequence? **[LO 15.3 M]**

(a) #elif

(b) #if

(c) #else

(d) #pragma

REVIEW QUESTIONS

15.1 State whether the following statements are *true* or *false* .

(a) The keyword **#define** must be written starting from the first column. **[LO 15.1 E]**

(b) Like other statements, a processor directive must end with a semicolon. **[LO 15.1 E]**

(c) All preprocessor directives begin with **#** . **[LO 15.1 E]**

(d) We cannot use a macro in the definition of another macro. **[LO 15.1 M]**

(e) The preprocessor directives are executed by the compiler with the same priority as other statements within the main function. **[LO 15.1 E]**

(f) #include <stdio.h> directive checks the presence of stdio.h file in the standard directory. **[LO 15.2 M]**

15.2 Fill in the blanks in the following statements.

(a) The ___________________ directive discords a macro. **[LO 15.3 E]**

(b) The operator ________________ is used to concatenate two arguments. **[LO 15.4 M]**

(c) The operator ________________ converts its operand. **[LO 15.4 M]**

(d) The ______________ directive causes an implementation-oriented action. **[LO 15.4 E]**

(e) The #include "TEST.C" directive checks the presence of TEST.C file in the ________ directory first. If the file is not present, then the compiler checks the __________ directory to check for the file. **[LO 15.2 M]**

(f) _____________ directive checks if a macro is already defined in the header file or not. **[LO 15.3 E]**

(g) ____________ directive stops the compilation process if an error is encountered. **[LO 15.4 E]**

(h) ___________ directive undefines a macro so that it is restricted to only a specific part in the program. **[LO 15.3 M]**

15.3 Explain the facilities provided by the C preprocessor with examples. **[LO 15.1 E]**

15.4 What is a macro and how is it different from a C variable name? **[LO 15.1 E]**

15.5 What precautions one should take when using macros with argument? **[LO 15.1 M]**

15.6 What are the advantages of using macro definitions in a program? **[LO 15.1 M]**

15.7 When does a programmer use **#include** directive? **[LO 15.2 E]**

15.8 The value of a macro name cannot be changed during the running of a program. Comment? **[LO 15.1 M]**

15.9 What is conditional compilation? How does it help a programmer? **[LO 15.3 E]**

15.10 Distinguish between **#ifdef** and **#if** directives. **[LO 15.3 M]**

15.11 Comment on the following code fragment: **[LO 15.3 M]**

```
#if 0
{
        line-1;
        line-2;
        .. ...
        .. ...
        line-n;
}
#endif
```

15.12 Enumerate the differences between functions and parameterized macros. **[LO 15.1 H]**

15.13 In **#include** directives, some file names are enclosed in angle brackets while others are enclosed in double quotation marks. Why? **[LO 15.2 M]**

15.14 Why do we recommend the use of parentheses for formal arguments used in a macro definition? Give an example. **[LO 15.1 M]**

DEBUGGING EXERCISES

15.1 Identify errors, if any, in the following macro definitions:

(a) #define until(x) while(!x) **[LO 15.1 H]**

(b) #define ABS(x) (x > 0) ? (x) : (–x) **[LO 15.1 H]**

(c) #ifdef(FLAG) **[LO 15.3 E]**

#undef FLAG

#endif

(d) #if n == 1 update(item) **[LO 15.3 M]**

#else print-out(item)

#endif

PROGRAMMING EXERCISES

15.1 Define a macro PRINT_VALUE that can be used to print two values of arbitrary type. **[LO 15.1 E]**

15.2 Write a nested macro that gives the minimum of three values. **[LO 15.1 M]**

15.3 Define a macro with one parameter to compute the volume of a sphere. Write a program using this macro to compute the volume for spheres of radius 5, 10 and 15 metres. **[LO 15.1 M]**

15.4 Define a macro that receives an array and the number of elements in the array as arguments. Write a program using this macro to print out the elements of an array. **[LO 15.1 H]**

15.5 Using the macro defined in the exercise 15.4, write a program to compute the sum of all elements in an array. **[LO 15.1 H]**

15.6 Write symbolic constants for the binary arithmetic operators +, –, * and /. Write a short program to illustrate the use of these symbolic constants. **[LO 15.1 M]**

15.7 Define symbolic constants for { and } and printing a blank line. Write a small program using these constants. **[LO 15.1 H]**

15.8 Write a program to illustrate the use of stringizing operator. **[LO 15.4 M]**

15.9 Write a simple C program to demonstrate how a user-created file in included in a C program. **[LO 15.2 M]**

15.10 Write a C program that uses the #error directive to stop the program execution when an error is encountered. **[LO 15.4 M]**

15.11 Write a C program that uses macro substitution to compute the area of a rectangle. **[LO 15.1 E]**

15.12 Write a C program to implement a macro named NOTEQUAL that performs logical not operation. **[LO 15.1 M]**

INTERVIEW QUESTIONS

15.1 What will be the type of i1 and i2 variables in below code?

```
#define INTPTR int*
void main()
{
INTPTR i1,i2;
}
```

[LO 15.1 H]

15.2 What is the difference between #include <file> and #include “file”? **[LO 15.2 M]**

15.3 What will be the output of following program?

```
#include<stdio.h>
#define X -Y
#define Y -Z
#define Z 10
int main()
{
printf("The value of X is %d\n", X);
```

```
return 0;
}
```

[LO 15.1 M]

Chapter 16

Developing a C Program: Some Guidelines

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

- LO 16.1 Discuss program design
- LO 16.2 Describe program coding
- LO 16.3 Identify common programming errors
- LO 16.4 Explain how program testing and debugging is performed

INTRODUCTION

We have discussed so far various features of C language and are ready to write and execute programs of modest complexity. However, before attempting to develop complex programs, it is worthwhile to consider some programming techniques that would help design efficient and error-free programs.

The program development process includes three important stages, namely, program design, program coding and program testing. All the three stages contribute to the production of high-quality programs. In this chapter we shall discuss some of the techniques used for program design, coding and testing.

PROGRAM DESIGN

LO 16.1 Discuss program design

Program design is the foundation for a good program and is therefore an important part of the program development cycle. Before coding a program, the program should be well conceived and all aspects of the program design should be considered in detail.

Program design is basically concerned with the development of a strategy to be used in writing the program, in order to achieve the solution of a problem. This includes mapping out a solution procedure and the form the program would take. The program design involves the following four stages:

1. Problem analysis.
2. Outlining the program structure.
3. Algorithm development.
4. Selection of control structures.

Problem Analysis

Before we think of a solution procedure to the problem, we must fully understand the nature of the problem and what we want the program to do. Without the comprehension and definition of the problem at hand, program design might turn into a hit-or-miss approach. We must carefully decide the following at this stage;

What kind of data will go i n ? ;

What kind of outputs are needed?; and

What are the constraints and conditions under which the program has to operate?

Outlining the Program Structure

Once we have decided what we want and what we have, then the next step is to decide how to do it. C as a structured language lends itself to a *top-down* approach. Top-down means decomposing of the solution procedure into tasks that form a hierarchical structure, as shown in Fig. 16.1 . The essence of the top-down design is to cut the whole problem into a number of independent constituent tasks, and then to cut the tasks into smaller subtasks, and so on, until they are small enough to be grasped mentally and to be coded easily. These tasks and subtasks can form the basis of functions in the program.

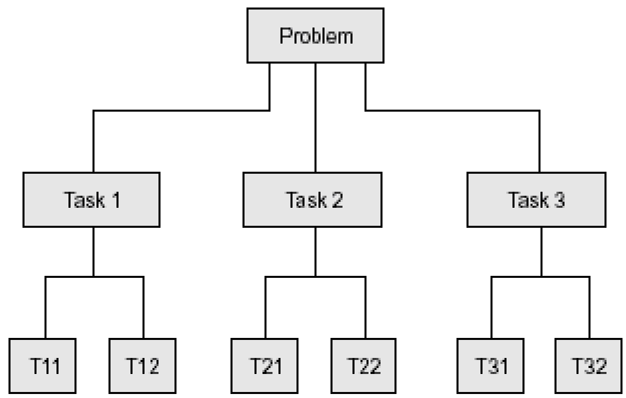


Fig. 16.1 Hierarchical structure

An important feature of this approach is that at each level, the details of the design of lower levels are hidden. The higher-level functions are designed first, assuming certain broad tasks of the immediately lower-level functions. The actual details of the lower-level functions are not considered until that level is reached. Thus the design of functions proceeds from top to bottom, introducing progressively more and more refinements.

This approach will produce a readable and modular code that can be easily understood and maintained. It also helps us classify the overall functioning of the program in terms of lower-level functions.

Algorithm Development

After we have decided a solution procedure and an overall outline of the program, the next step is to work out a detailed definite, step-by-step procedure, known as *algorithm* for each function. The most common method of describing an algorithm is through the use of *flowcharts* . The other method is to write what is known as *pseudocode* . The flow chart presents the algorithm pictorially, while the pseudocode describe the solution steps in a logical order. Either method involves concepts of logic and creativity.

Since algorithm is the key factor for developing an efficient program, we should devote enough attention to this step. A problem might have many different approaches to its solution. For example, there are many sorting techniques available to sort a list. Similarly, there are many methods of finding the area under a curve. We must consider all possible approaches and select the one, which is simple to follow, takes less execution time, and produces results with the required accuracy.

Control Structures

A complex solution procedure may involve a large number of control statements to direct the flow of execution. In such situations, indiscriminate use of control statements such as **goto** may lead to unreadable and uncomprehensible programs. It has been demonstrated that any algorithm can be structured, using the three basic control structure, namely, sequence structure, selection structure, and looping structure.

Sequence structure denotes the execution of statements sequentially one after another. Selection structure involves a decision, based on a condition and may have two or more branches, which usually join again at a later point. **if . . . .else** and **switch** statements in C can be used to implement a selection structure. Looping structure is used when a set of instructions is evaluated repeatedly. This structure can be implemented using **do** , **while** , or **for** statements.

A well-designed program would provide the following benefits:

1. Coding is easy and error-free.
2. Testing is simple.
3. Maintenance is easy.
4. Good documentation is possible.

TIP

Program writing should be preceded by algorithm/flowchart construction.

5. Cost estimates can be made more accurately.
6. Progress of coding may be controlled more precisely.

PROGRAM CODING

LO 16.2
Describe program coding

The algorithm developed in the previous section must be translated into a set of instructions that a computer can understand. The major emphasis in coding should be simplicity and clarity. A program written by one may have to be read by others later. Therefore, it should be readable and simple to understand. Complex logic and tricky coding should be avoided. The elements of coding style include:

- Internal documentation.
- Construction of statements.
- Generality of the program.
- Input/output formats.

Internal Documentation

Documentation refers to the details that describe a program. Some details may be built-in as an integral part of the program. These are known as *internal documentation* .

Two important aspects of internal documentation are, selection of meaningful variable names and the use of comments. Selection of meaningful names is crucial for understanding the program. For example,

area = breadth * length

is more meaningful than

a = b * l;

Names that are likely to be confused must be avoided. The use of meaningful function names also aids in understanding and maintenance of programs.

Descriptive comments should be embedded within the body of source code to describe processing steps.

The following guidelines might help the use of comments judiciously:

1. Describe blocks of statements, rather than commenting on every line.
2. Use blank lines or indentation, so that comments are easily readable.
3. Use appropriate comments; an incorrect comment is worse than no comment at all.

Statement Construction

Although the flow of logic is decided during design, the construction of individual statements is done at the coding stage. Each statement should be simple and direct. While multiple statements per line are allowed, try to use only one statement per line with necessary indentation. Consider the following code:

```
if(quantity>0){code = 0; quantity = rate;}
else { code = 1; sales = 0:)
```

Although it is perfectly valid, it could be reorganized as follows:

```
if(quantity>0)
{
     code = 0;
     quantity = rate;
}
else
{
     code = 1;
     sales = 0:
}
```

The general guidelines for construction of statements are:

1. Use one statement per line.

2. Use proper indentation when selection and looping structures are implemented.
3. Avoid heavy nesting of loops, preferably not more than three levels.
4. Use simple conditional tests; if necessary break complicated conditions into simple conditions.
5. Use parentheses to clarify logical and arithmetic expressions.
6. Use spaces, wherever possible, to improve readability.

Input/Output Formats

Input/output formats should be simple and acceptable to users. A number of guidelines should be considered during coding.

1. Keep formats simple.
2. Use end-of-file indicators, rather than the user requiring to specify the number of items.
3. Label all interactive input requests.
4. Label all output reports.
5. Use output messages when the output contains some peculiar results.

Generality of Programs

Care should be taken to minimize the dependence of a program on a particular set of data, or on a particular value of a parameter. Example:

```
for(sum = 0, i=1; i <= 10; i++)
sum = sum + i;
```

This loop adds numbers 1,2, …., 10. This can be made more general as follows:

```
sum =0;
for(i =m; i <=n; i = i+ step);
     sum = sum + i;
```

The initial value **m** , the final value **n** , and the increment size **step** can be specified interactively during program execution. When **m** =2, **n** =100, and **step** =2, the loop adds all even numbers up to, and including 100.

> **TIP**
> Proper Indentation should be used in statement blocks to enhance the readability and to make debugging easier.

COMMON PROGRAMMING ERRORS

> **LO 16.3**
> Identify common programming errors

By now you must be aware that C has certain features that are easily amenable to bugs. Added to this, it does not check and report all kinds of run-time errors. It is therefore, advisable to keep a track of such errors and to see that these known errors are not present in the program. This section examines some of the more common mistakes that a less experienced C programmer could make.

Missing Semicolons

Every C statement must end with a semicolon. A missing semicolon may cause considerable confusion to the compiler and result in 'misleading' error messages. Consider the following statements:

a = x+y

b = m/n;

The compiler will treat the second line as a part of the first one and treat b as a variable name. You may therefore get an "undefined name" error message in the second line. Note that both the message and location are incorrect. In such situations where there are no errors in a reported line, we should check the preceding line for a missing semicolon.

There may be an instance when a missing semicolon might cause the compiler to go ‘crazy’ and to produce a series of error messages. If they are found to be dubious errors, check for a missing semicolon in the beginning of the error list.

Misuse of Semicolon

Another common mistake is to put a semicolon in a wrong place. Consider the following code:

```
for(i = 1; i<=10; i++);
sum = sum + i;
```

This code is supposed to sum all the integers from 1 to 10. But what actually happens is that only the ‘exit’ value of i is added to the sum. Other examples of such mistake are:

```
1. while (x < Max);
   {
   }
2. if(T>= 200);
   grade = 'A';
```

A simple semicolon represents a null statement and therefore it is syntactically valid. The compiler does not produce any error message. Remember, these kinds of errors are worse than syntax errors.

Use of = Instead of = =

It is quite possible to forget the use of double equal sings when we perform a relational test. Example:

```
if(code = 1)
count ++;
```

It is a syntactically valid statement. The variable code is assigned 1 and then, because code = 1 is true, the count is incremented. In fact, the above statement does not perform any relational test on code. Irrespective of the previous value of code, **count ++;** is always executed.

Similar mistakes can occur in other control statements, such as **for** and **while** . Such a mistake in the loop control statements might cause infinite loops.

Missing Braces

It is common to forget a closing brace when coding a deeply nested loop. It will be usually detected by the compiler because the number of opening braces should match with the closing ones. However, if we put a matching brace in a wrong place, the compiler won't notice the mistake and the program will produce unexpected results.

Another serious problem with the braces is, not using them when multiple statements are to be grouped together. For instance, consider the following statements:

```
for(i=1; i<= 10; i++)
   sum1 = sum1 + i;
   sum2 = sum2 + i*i;
printf("%d %d\n", sum1,sum2);
```

This code is intended to compute **sum1, sum2** for i varying from 1 to 10, in steps of 1 and then to print their values. But, actually the **for** loop is interpreted by the compiler as follows because of the missing braces:

```
for(i=1; i<= 10; i++)
{
        sum1 = sum1 +i;
}
        sum2 = sum2 + i*i;
printf("%d %d\n", sum1,sum2);
```

The for loop treats only the first statement, namely

sum1 = sum1 + i;

as its body and therefore the statement

sum2 = sum2 + i*i;

is evaluated only once when the loop is exited. The correct way to code this segment is to place braces as follows:

```
for(i=1; i<=10; i++)
{
        sum1 = sum1 + i;
        sum2 = sum2 +i*i;
}
printf("%d %d\n", sum1, sum2);
```

Misusing Quotes

It is likely that we use single quotes whenever we handle single characters. Care should be exercised to see that the associated variables are declared properly. For example, the statement

city = 'M';

would be invalid if **city** has been declared as a **char** variable with dimension (i.e., pointer to **char**).

Improper Comment Characters

Every comment should start with a /* and end with a */ . Anything between them is ignored by the compiler. If we miss out the closing */ , then the compiler searches for a closing */ further down in the program, treating all the lines as comments. In case, it fails to find a closing */ , we may get an error message. Consider the following lines:

```
. . . . .
/* comment line 1
statement1;
statement2;
/* comment line 2 */
statement 3;
. . . . .
```

Since the closing */ is missing in the comment line 1, all the statements that follow, until the closing comment */ in comment line 2 are ignored.

We should remember that C does not support nested comments. Assume that we want to comment out the following segment:

```
. . . . . .
x = a–b;
Y = c–d;
/* compute ratio */
ratio = x/y;
. . . . . .
. . . . . .
```

we may be tempted to add comment characters as follows:

```
/*  x = a–b;
y = c–d;
/*  Compute ratio */
ratio = x/y; */
```

This is incorrect. The first opening comment matches with the first closing comment and therefore the lines between these two are ignored. The statement

ratio = x/y;

is not commented out. The correct way to comment out this segment is shown as:

```
/* x = a–b;
y = c–d;
compute ratio
ratio = x/y; */
```

Undeclared Variables

C requires every variable to be declared for its type, before it is used. During the development of a large program, it is quite possible to use a variable to hold intermediate results and to forget to declare it.

Forgetting the Precedence of Operators

Expressions are evaluated according to the precedence of operators. It is common among beginners to forget this. Consider the statement

```
if((value = product()) >=100)
     tax = 0.05 * value;
```

> **TIP**
> Brackets should be used wherever applicable to specify the precedence of operators. It increases the readability of programs.

The call **product ( )** returns the product of two numbers, which is compared to 100. If it is equal to or greater than 100, the relational test is true, and a 1 is assigned to **value** , otherwise a 0 is assigned. In either case, the only values **value** can take on are 1 or 0. This certainly is not what the programmer wanted.

The statement was actually expected to assign the value returned by **product( )** to **value** and then compare **value** with 100. If **value** was equal to or greater than 100, tax should have been computed, using the statement

tax = 0.05 * value;

The error is due to the higher precedence of the relational operator compared to the assignment operator. We can force the assignment to occur first by using parentheses as follows:

```
if ((value = product()) >= 100)
        tax = 0.05 * value;
```

Similarly, the logical operators **&&** and || have lower precedence than arithmetic and relational operators and among these two, **&&** has higher precedence than || . Try, if there is any difference between the following statements:

```
1. if(p > 50|| c > 50 && m > 60 && T > 180)
   x = 1;
2. if((p > 50|| c > 50) && m > 60 && T > 180)
   x = 1;
3. if((p > 50|| c > 50 && m > 60) && T > 180)
   x = 1;
```

Ignoring the Order of Evaluation of Increment/Decrement Operators

We often use increment or decrement operators in loops. Example

```
... ...
i = 0;
while ((c = getchar()) != '\n')
{
    string[i++] = c;
}
string[i-1] = '\n';
```

The statement **string[i++]** = c; is equivalent to:

string[i] = c;
i = i+1;

This is not the same as the statement **string[++i]** = c; which is equivalent to

i =i+1;
string[i] = c;

Forgetting to Declare Function Parameters

Remember to declare all function parameters in the function header.

Mismatching of Actual and Formal Parameter Types in Function Calls

When a function with parameters is called, we should ensure that the type of values passed, match with the type expected by the called function. Otherwise, erroneous results may occur. If necessary, we may use the *type* cast operator to change the type locally. Example:

y = cos((double)x);

Nondeclaration of Functions

Every function that is called should be declared in the calling function for the types of value it returns. Consider the following program:

```
main()
{
     float a =12.75;
     float b = 7.36;
     printf("%lf\n", division(a,b));
}
double division(float x, float y)
{
     return(x/y);
}
```

The function returns a **double** type value but this fact is not known to the compiler and therefore it expects to receive an **int** type value. The program during compilation produces an error message such as "redefinition".

The function **division** is like any other variable for the **main** and therefore it should be declared as **double** in the main.

Now, let us assume that the function division is coded as follows:

```
division(float x, float y)
{
   return(x/y);
}
```

Although the values x and y are floats and the result of x/y is also float, the function returns only integer value because no type specifier is given in the function definition. This is wrong too. The function header should include the type specifier to force the function to return a particular type of value.

TIP

All function definitions and calls should be preceded by their declaration.

Missing & Operator in scanf Parameters

All non-pointer variables in a scanf call should be preceded by an **&** operator. If the variable code is declared as an integer, then the statement

scanf("%d", code);

is wrong. The correct one is scanf("%d", **&** code);

Remember, the compiler will not detect this error and you may get a crazy output.

Crossing the Bounds of an Array

All C indices start from zero. A common mistake is to start the index from 1. For example, the segment

```
int x[10], sum i;
Sum = 0;
for (i = 1; i < = 10; i++)
     sum = sum + x[i];
```

would not find the correct sum of the elements of array x. The for loop expressions should be corrected as follows:

for(i=0;i<10;i++)

Forgetting a Space for Null character in a String

All character arrays are terminated with a null character and therefore their size should be declared to hold one character more than the actual string size.

Using Uninitialized Pointers

An uninitialized pointer points to garbage. The following program is wrong:

```
main()
{
        int a, *ptr;
        a = 25;
        *ptr = a+5;
}
```

The pointer **ptr** has not been initialized.

Missing Indirection and Address Operators

Another common error is to forget to use the operators ***** and **&** in certain places. Consider the following program:

```
main()
{
        int m, *p1;
        m = 5;
        p1 = m;
        printf("%d\n", *p1);
}
```

This will generate an error message by the compiler, because the pointer assignment

p1 = m;

is wrong. It should be:

p1 = &m;

Consider the following expression:

y = p1 + 10;

Perhaps, **y** was expected to be assigned the value at location **p1** plus 10. But it does not happen. This statement too will generate a compilation error. The above expression should be rewritten as:

y = *p1 + 10;

Missing Parentheses in Pointer Expressions

The following two statements are not the same:

x = *p1 + 1;

x = *(p1 + 1);

The first statement would assign the value at location **p1** plus 1 to **x,** while the second would assign the value at location **p1 + 1** .

Omitting Parentheses around Arguments in Macro Definitions

This would cause incorrect evaluation of expression when the macro definition is substituted.

Example:	**# define** f(x) x * x + 1
The call	**y = f(a+b);**
will be evaluated as	**y = a+b * a+b+1;** which is wrong.

Some other mistakes that we commonly make are as follows:

1. Wrong indexing of loops.
2. Wrong termination of loops.
3. Unending loops.
4. Use of incorrect relational test.
5. Failure to consider all possible conditions of a variable.

6. Trying to divide by zero.
7. Mismatching of data specifications and variables in **scanf** and **printf** statements.
8. Forgetting truncation and rounding off errors.

PROGRAM TESTING AND DEBUGGING

LO 16.4
Explain how program testing and debugging is performed

Testing and debugging refer to the tasks of detecting and removing errors in a program, so that the program produces the desired results on all occasions. Every programmer should be aware of the fact that rarely does a program run perfectly the first time. No matter how thoroughly the design is carried out, and no matter how much care is taken in coding, one can never say that the program would be 100 per cent error-free. It is therefore, necessary to make efforts to detect, isolate, and correct any errors that are likely to be present in the program.

Types of Errors

We have discussed a number of common errors. There might be many other errors, some obvious and others not so obvious. All these errors can be classified under four types, namely, syntax errors, run-time errors, logical errors, and latent errors.

Syntax errors

Any violation of rules of the language results in syntax errors. The compiler can detect and isolate such errors. When syntax errors are present, the compilation fails and is terminated after listing the errors and the line numbers in the source program, where the errors have occurred. Remember, in some cases, the line number may not exactly indicate the place of the error. In other cases, one syntax error may result in a long list of errors. Correction of one or two errors at the beginning of the program may eliminate the entire list.

Run-time errors

Errors such as a mismatch of data types or referencing an out-of-range array element go undetected by the compiler. A program with these mistakes will run, but produce erroneous results and therefore, the name run-time errors is given to such errors. Isolating a run-time error is usually a difficult task.

Logical errors

As the name implies, these errors are related to the logic of the program execution. Such actions as taking a wrong path, failure to consider a particular condition, and incorrect order of evaluation of statements belong to this category. Logical errors do not show up as compiler-generated error messages. Rather, they cause incorrect results. These errors are primarily due to a poor understanding of the problem, incorrect translation of the algorithm into the program and a lack of clarity of hierarchy of operators. Consider the following statement:

```
if(x ==y)
printf("They are equal\n");
```

when **x** and **y** are float types values, they rarely become equal, due to truncation errors. The printf call may not be executed at all. A test like **while(x != y)** might create an infinite loop.

Latent errors

It is a 'hidden' error that shows up only when a particular set of data is used. For example, consider the following statement:

ratio = (x+y)/(p–q);

An error occurs only when **p** and **q** are equal. An error of this kind can be detected only by using all possible combinations of test data.

Testing is the process of reviewing and executing a program with the intent of detecting errors, which may belong to any of the four kinds discussed above. We know that while the compiler can detect syntactic and semantic errors, it cannot detect run-time and logical errors that show up during the execution of the program. Testing, therefore, should include necessary steps to detect all possible errors in the program. It is, however, important to remember that it is impractical to find all errors. Testing process may include the following two stages:

1. Human testing.
2. Computer-based testing.

Human testing is an effective error-detection process and is done before the computer-based testing begins. Human testing methods include code inspection by the programmer, code inspection by a test group, and a review by a peer group. The test is carried out statement by statement and is analyzed with respect to a checklist of common programming errors. In addition to finding the errors, the programming style and choice of algorithm are also reviewed.

Computer-based testing involves two stages, namely *compiler testing* and *run-time testing*. Compiler testing is the simplest of the two and detects yet undiscovered syntax errors. The program executes when the compiler detects no more errors. Should it mean that the program is correct? Will it produce the expected results? The answer is negative. The program may still contain run-time and logic errors.

Run-time errors may produce run-time error messages such as “null pointer assignment” and “stack overflow”. When the program is free from all such errors, it produces output ,which might or might not be correct. Now comes the crucial test, the test for the *expected output* . The goal is to ensure that the program produces expected results under all conditions of input data.

Test for correct output is done using *test data* with known results for the purpose of comparison. The most important consideration here is the design or invention of effective test data. A useful criteria for test data is that all the various conditions and paths that the processing may take during execution must be tested.

Program testing can be done either at module (function) level or at program level. Module level test, often known as *unit test* , is conducted on each of the modules to uncover errors within the boundary of the module. Unit testing becomes simple when a module is designed to perform only one function.

Once all modules are unit tested, they should be *integrated together* to perform the desired function(s). There are likely to be interfacing problems, such as data mismatch between the modules. An *integration test* is performed to discover errors associated with interfacing.

Program Debugging

Debugging is the process of isolating and correcting the errors. One simple method of debugging is to place print statements throughout the program to display the values of variables. It displays the dynamics of a program and allows us to examine and compare the information at various points. Once the location of an error is identified and the error corrected, the debugging statements may be removed. We can use the conditional compilation statements, discussed in Chapter 15, to switch on or off the debugging statements.

Another approach is to use the process of deduction. The location of an error is arrived at using the process of elimination and refinement. This is done using a list of possible causes of the error.

The third error-locating method is to *backtrack* the incorrect results through the logic of the program until the mistake is located. That is, beginning at the place where the symptom has been uncovered, the program is traced backward until the error is located.

Program Efficiency

Two critical resources of a computer system are execution time and memory. The efficiency of a program is measured in terms of these two resources. Efficiency can be improved with good design and coding practices.

Execution Time

The execution time is directly tied to the efficiency of the algorithm selected. However, certain coding techniques can considerably improve the execution efficiency. The following are some of the techniques, which could be applied while coding the program.

1. Select the fastest algorithm possible.
2. Simplify arithmetic and logical expressions.
3. Use fast arithmetic operations, whenever possible.
4. Carefully evaluate loops to avoid any unnecessary calculations within the loops.
5. If possible, avoid the use of multi-dimensional arrays.
6. Use pointers for handling arrays and strings.

However, remember the following, while attempting to improve efficiency.

1. Analyse the algorithm and various parts of the program before attempting any efficiency changes.
2. Make it work before making it faster.
3. Keep it right while trying to make it faster.
4. Do not sacrifice clarity for efficiency.

Memory Requirement

Memory restrictions in the micro-computer environment is a real concern to the programmer. It is therefore, desirable to take all necessary steps to compress memory requirements.

1. Keep the program simple. This is the key to memory efficiency.
2. Use an algorithm that is simple and requires less steps.
3. Declare arrays and strings with correct sizes.
4. When possible, limit the use of multi-dimensional arrays.
5. Try to evaluate and incorporate memory compression features available with the language.

KEY CONCEPTS

- **PROGRAM DESIGN:** It is the process of analyzing the problem and designing the program logic. **[LO 16.1]**
- **BACKTRACK:** It is an error-locating method that traces the program backwards to locate the error. **[LO 16.4]**
- **LATENT ERROR:** It is a hidden error that occurs only when a particular set of data is used. **[LO 16.4]**
- **LOGICAL ERROR:** It is an error caused by incorrect programming logic. **[LO 16.4]**
- **RUNTIME ERROR:** It is an error that occurs at the time of running of a program. **[LO 16.4]**

ALWAYS REMEMBER

- While developing a large program, ensure that its design is captured in the program structure and algorithm . **[LO 16.1]**
- Follow good design and coding practices to ensure program efficiency . **[LO 16.1]**
- Avoid common programming errors such as, missing or misplaced semicolons, mismatched braces, undeclared variables, mismatched function arguments, referring locations outside of array boundary, etc . **[LO 16.3]**

- Test the program properly to detect and remove latent errors .

[LO 16.3]

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

16.1 Which of the following types of errors are considered hidden errors that become evident only when specific data set is used? **[LO 16.3 M]**

(a) Syntax error

(b) Logical error

(c) Run-time error

(d) Latent error

16.2 Which of the following errors should get detected during the compilation stage? **[LO 16.3 M]**

(a) Missing semi-colon

(b) Using = instead of ==

(c) Missing braces

(d) Missing quotes

16.3 Which of the following are key elements of program design? **[LO 16.1 E]**

(a) Algorithm

(b) Pseudocode

(c) Flowchart

(d) All of the above

REVIEW QUESTIONS

16.1 State whether the following statements are true or false.

(a) C allows the programmer to use some control structures interchangeably. For example, switch and if-else, for and while loop, etc. **[LO 16.2 E]**

(b) A semi-colon placed at the end of for statement results in compilation error. **[LO 16.3 M]**

16.2 Fill in the blanks in the following statements.

(a) To enforce specific order of precedence in an expression, we can make use of _________. **[LO 16.3 E]**

(b) __________ is an error-locating method that traces the program backwards to locate the error. **[LO 16.3 M]**

(c) If a program has missing braces, then it will be detected during the ___________ stage. **[LO 16.3 E]**

16.3 Discuss the various aspects of program design. **[LO 16.1 E]**

16.4 How does program design relate to program efficiency? **[LO 16.2 M]**

16.5 Readability is more important than efficiency, Comment. **[LO 16.1 M]**

16.6 Distinguish between the following: **[LO 16.4 M]**

(a) Syntactic errors and semantic errors.

(b) Run-time errors and logical errors.

(c) Run-time errors and latent errors.

(d) Debugging and testing.

(e) Compiler testing and run-time testing.

16.7 A program has been compiled and linked successfully. When you run this program you face one or more of the following situations. **[LO 16.4 M]**

(a) Program is executed but no output.

(b) It produces incorrect answers.

(c) It does not stop running.

Comment on the reasons for the occurrence of these situations.

16.8 List five common programming mistakes. Write a small program containing these errors and try to locate them with the help of computer. **[LO 16.3 E]**

16.9 In a program, two values are compared for convergence, using the statement **[LO 16.3 M]**

if((x–y) < 0.00001) ...

Does the statement contain any error? If yes, explain the error.

16.10 A program contains the following if statements: **[LO 16.4 H]**

```
... ..
... ..
if(x>1&&y == 0)p = p/x;
if(x == 5|| p > 2) p = p+2;
... ..
... ..
```

Draw a flow chart to illustrate various logic paths for this segment of the program and list test data cases that could be used to test the execution of every path shown.

16.11 Given below is a function to compute the yth power of an integer x. **[LO 16.4 H]**

```
power(int x, int y)
{
   int p;
   p = y;
   while(y > 0)
         x *= y --;
   return(x);
}
```

This function contains some bugs. Write a test procedure to locate the errors with the help of a computer.

16.12 A program reads three values from the terminal, representing the lengths of three sides of a box namely length, width and height and prints a message stating whether the box is a cube, rectangle, or semi-rectangle. Prepare sets of data that you feel would adequately test this program. **[LO 16.4 M]**

Appendix I

Bit-Level Programming

INTRODUCTION

One of the unique features of C language as compared to other high-level languages is that it allows direct manipulation of individual bits within a word. Bit-level manipulations are used in setting a particular bit or group of bits to 1 or 0. They are also used to perform certain numerical computations faster. As pointed out in Chapter 4, C supports the following operators:

1. Bitwise logical operators.
2. Bitwise shift operators.
3. One's complement operator.

All these operators work only on integer type operands.

BITWISE LOGICAL OPERATORS

There are four logical bitwise operators. They are as follows:

- Bitwise AND (&)
- Bitwise OR (|)
- Bitwise complement/ NOT (~)
- Bitwise exclusive OR (^)

All except the NOT are binary operators and require two integer-type operands. The NOT (~) is a unary operator. These operators work on their operands bit by bit starting from the least significant (i.e. the rightmost) bit, setting each bit in the result as shown in Table I.1.

Table I.1 Result of logical bitwise operations

op1	op2	op1 & op2	op1 | op2	~op1	op1 ^ op2
1	1	1	1	0	0
1	0	0	1	0	1
0	1	0	1	1	1
0	0	0	0	1	0

Bitwise AND

The bitwise AND operator is represented by a single ampersand (&) and is surrounded on both sides by integer expressions. The result of ANDing operation is 1 if both the bits have a value of 1; otherwise it is 0. Let us consider two variables **x** and **y** whose values are 13 and 25. The binary representation of these two variables are as follows:

x - - -> 0000 0000 0000 1101

y - - -> 0000 0000 0001 1001

If we execute statement

```
z = x & y ;
```

then the result would be:

z - - -> 0000 0000 0000 1001

Although the resulting bit pattern represents the decimal number 9, there is no apparent connection between the decimal values of these three variables.

Bitwise ANDing is often used to test whether a particular bit is 1 or 0. For example, the following program tests whether the fourth bit of the variable **flag** is 1 or 0.

```
#define TEST 8 /* represents 00........01000 */
main()
{
          int flag;
          ....
          ....
          if((flag & TEST) != 0) /* test 4th bit */
          {
             printf(" Fourth bit is set \n");
          }
          ....
          ....
```

Note that the bitwise logical operators have lower precedence than the relational operators and therefore additional parentheses are necessary as shown above.

The following program tests whether a given number is odd or even.

```
main()
{
          int test = 1;
          int number;
          printf("Input a number \n");
          scanf("%d", &number);
          while (number != -1)
          {
               if(number & test)
                    print("Number is odd\n\n");
               else
                    printf("Number is even\n\n");
                    printf("Input a number \n");
                    scanf("%d", &number);
          }
}
```

Output

```
Input a number
20
Number is even
Input a number
9
Number is odd
Input a number
-1
```

Bitwise OR

The bitwise OR is represented by the symbol | (vertical bar) and is surrounded by two integer operands. The result of OR operation is 1 if *at least* one of the bits has a value of 1; otherwise it is zero. Consider the variables **x** and **y** discussed above.

```
x - - ->        0000 0000 0000 1101
y - - ->        0000 0000 0001 1001
               ____________________
x|y - - ->      0000 0000 0001 1101
               ____________________
```

The bitwise inclusion OR operation is often used to set a particular bit to 1 in a flag. Example:

```
#define SET 8
main()
{
      int flag;
      ....
      ....
      flag = flag | SET;
      if (( flag & SET) != 0)
      {
         printf("flag is set \n");
      }
      ....
      ....
}
```

The statement

flag = flag | SET;

causes the fourth bit of flag to set 1 if it is 0 and does not change it if it is already 1.

Bitwise NOT/complement

The complement operator ~ (also called the one's complement operator) is a unary operator and inverts all the bits represented by its operand. That is, 0s become 1s and 1s become 0s.

Example:

x = 1001 0110 1100 1011

~x = 0110 1001 0011 0100

This operator is often combined with the bitwise AND operator to turn off a particular bit. For example, the statement

```
x = 8; /* 0000 0000 0000 1000 */
flag = flag & ~x;
```

would turn off the fourth bit in the variable **flag** .

Bitwise Exclusive OR (XOR)

The bitwise exclusive OR is represented by the symbol ^. The result of exclusive OR is 1 if only one of the bits is 1; otherwise it is 0. Consider again the same variables x and y discussed above.

```
  x = 0000  0000  0000  1101
  y = 0000  0000  0001  1001
x^y = 0000  0000  0001  0100
```

The XOR operation can also be implemented using the basic bitwise logical operators – AND, OR and NOT as follows:

```
x^y = (x & (~y)) | ((~x) & y)
x = 0000 0000 0000 1101
y = 0000 0000 0001 1001
~x = 1111 1111 1111 0010
~y = 1111 1111 1110 0110
x & (~y) = 0000 0000 0000 0100
(~x) & y = 0000 0000 0001 0000
(x & (~y)) | ((~x) & y) = 0000 0000 0001 0100
```

It should be noted that the result is same as that obtained by computing x^y.

Bitwise Shift Operators

The shift operators are used to move bit patterns either to the left or to the right. The shift operators are represented by the symbols << and >> and are used in the following form:

Left shift: *op << n*

Right shift: *op >> n*

op is the integer expression that is to be shifted and *n* is the number of bit positions to be shifted.

The left-shift operation causes all the bits in the operand *op* to be shifted to the left by *n* positions. The leftmost *n* bits in the original bit pattern will be lost and the rightmost *n* bit positions that are vacated will be filled with 0s.

Similarly, the right-shift operation causes all the bits in the operand *op* to be shifted to the right by *n* positions. The rightmost *n* bits will be lost. The leftmost *n* bit positions that are vacated will be filled with zero, if the *op* is an *unsigned integer* . If the variable to be shifted is **signed** , then the operation is machine dependent.

Both the operands *op* and *n* can be constants or variables. There are two restrictions on the value of *n* . It may not be negative and it may not exceed the number of bits used to represent the left operand *op* .

Let us suppose **x** is an unsigned integer whose bit pattern is,

```
                              0100 1001 1100 1011
then,                                          vacated
                                                 positions
                              x << 3 = 0100 1110 0101 1000
                              x >> 3 = 0000 1001 0011 1001
                                       vacated
                                       positions
```

Shift operators are often used for multiplication and division by powers of two.

Consider the following statement:

x = y << 1;

This statement shifts one bit to the left in **y** and then the result is assigned to **x** . The decimal value of **x** will be the value of **y** multiplied by 2. Similarly, the statement

x = y >> 1;

shifts **y** one bit to the right and assigns the result to **x** . In this case, the value of **x** will be the value of **y** divided by 2.

The shift operators, when combined with the logical bitwise operators, are useful for extracting data from an integer field that holds multiple pieces of information. This process is known as *masking* . Masking is discussed below.

MASKING

Masking refers to the process of extracting desired bits from (or transforming desired bits in) a variable by using logical bitwise operation. The operand (a constant or variable) that is used to perform masking is called the *mask* . Examples:

```
y = x & mask;
y = x | mask;
```

Masking is used in many different ways which are as follows:

- To decide bit pattern of an integer variable.
- To copy a portion of a given bit pattern to a new variable, while the remainder of the new variable is filled with 0s (using bitwise AND).
- To copy a portion of a given bit pattern to a new variable, while the remainder of the new variable is filled with 1s (using bitwise OR).
- To copy a portion of a given bit pattern to a new variable, while the remainder of the original bit pattern is inverted within the new variable (using bitwise *exclusive* OR).

The following function uses a mask to display the bit pattern of a variable:

```
void bit_pattern(int u)
{
      int i, x, word;
      unsigned mask;
      mask = 1;
      word = 8 * sizeof(int);
      mask =  mask << (word – 1);
               /* shift 1 to the leftmost position */
      for(i = 1; i<= word; i++)
      {
         x = (u & mask) ? 1 : 0; /* identify the bit */
         printf("%d", x); /* print bit value */
         mask >>= 1; /* shift mask by 11 position to right */
      }
   }
```

Appendix II

ASCII Values of Characters

ASCII		ASCII		ASCII		ASCII	
Value	Character	Value	Character	Value	Character	Value	Character
000	NUL	027	ESC	054	6	081	Q
001	SOH	028	FS	055	7	082	R
002	STX	029	GS	056	8	083	S
003	ETX	030	RS	057	9	084	T
004	EOT	031	US	058	:	085	U
005	ENQ	032	blank	059	;	086	V
006	ACK	033	!	060	<	087	W
007	BEL	034	“	061	=	088	X
008	BS	035	#	062	>	089	Y
009	HT	036	$	063	?	090	Z
010	LF	037	%	064	@	091	[
011	VT	038	&	065	A	092	\
012	FF	039	‘	066	B	093	]
013	CR	040	(	067	C	094	↑
014	SO	041	)	068	D	095	-
015	SI	042	*	069	E	096	←
016	DLE	043	+	070	F	097	a
017	DC1	044	,	071	G	098	b
018	DC2	045	–	072	H	099	c
019	DC3	046	.	073	I	100	d
020	DC4	047	/	074	J	101	e
021	NAK	048	0	075	K	102	f
022	SYN	049	1	076	L	103	g
023	ETB	050	2	077	M	104	h
024	CAN	051	3	078	N	105	i
025	EM	052	4	079	O	106	j
026	SUB	053	5	080	P	107	k

ASCII		ASCII		ASCII		ASCII	
Value	Character	Value	Character	Value	Character	Value	Character
108	l	113	q	118	v	123	{
109	m	114	r	119	w	124	|
110	n	115	s	120	x	125	}
111	o	116	t	121	y	126	~
112	p	117	u	122	z	127	DEL

Note *The first 32 characters and the last character are control characters; they cannot be printed.*

Appendix III

ANSI C 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 slit 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 as follows:

<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 alphanumeric. 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.
<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 1.
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 nonzero value; otherwise, return 0.
fgetc(f)	int	Enter a single character form file f.
fgets(s, i, f)	char*	Enter string s, containing i characters, from file f.
fprint(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.
fscanf(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.
gets(s)	char*	Enter string s from the standard input device.
printf(...)	int	Send data items 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.
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 double-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 of s1 > s2.
strcpy(s1, s2)	char*	Copy string s2 to string s1.
strlen(s)	int	Return the number of characters in 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 time beyond a designated base time (see the time function).
time(p)	long int	Return the number of seconds elapsed beyond a designated base time.

The following table shows some macros defined in the **stdio.h** header file:

Macros	*Data type returned*	*Task*
getc (f)	int	Enter a single character from file f
getchar (void)	int	Enter a single character from the standard input device.
putc (c, f)	int	Sends a single character to file f
putchar (c)	int	Send a single character to the standard output device.

Note *C99 adds many more header files and adds many new functions to the existing header files. For more details, refer to the manual of C99.*

Appendix IV

Projects

Project 1: Electricity Board Management System

This project helps operator (working at electricity office) to maintain records of user and request made by them. Operator will take user details and also take request details from user.

(A request may be for installing an electricity connection at newly created house, factory or Apartment.)

Operator can add user, add request, view any user and request made by him, edit request and view request. Functions have been made in separate files to keep project organised and readable. This project implements important features of C like I/O stuffs, Loop, branching, Pointers, Structures, Memory management and Functions. To run this project, create following files in a directory and run following command.

Commands

```
cd <path_to_directory>
gcc main.c application.c user.c request.c helpers.c -o application
chmod +x application
./application
```

Note: gcc command will compile all files given by command line arguments, link them and create executable file specified by -o flag

To Do

1. Make “Edit User“ feature (similar to edit request), so that operator can change name or contact no. of any user.
2. Add new request status called “COMPLETED”, which indicates successful completion of request.

Main-c

```
/**********************************************************************************************
******************************
                              Electricity board Management system

                                   Compiled on : Linux (GCC)
***********************************************************************************************
*****************************/

/*  main.c  */

#include <stdio.h>
#include 'application.h'
int main(int argc, const char * argv[]) {

    /* Entry point of Application */

    start_application();
}
```

Application-h

```
/* Application.h */

/* We are using #ifndef macro to prevent multiple inclusion of header files */

#ifndef application_h
#define application_h

/* Maximum length of user id and request id */

#define ID_LENGTH 10

/* Maximum length of string used in this project */
/* User name, comment, address etc. */

#define STR_MAX_LENGTH 256


/* All application and user interface related functions. */

/* String which will be used to separate data of each line in given file. */

extern const char * DELIMITER;

/* Start application. */

extern void start_application(void);

/* Show welcome screen of application. */

extern void welcome(void);

/* Show menu for operator with various options. */

extern void render_menu(void);

/* Add new user who want to request for electricity. */

extern void add_user_page(void);

/* Operator can view specific user and requests done by that user. */
```

```c
extern void view_user_page(void);

/* Operator can add new request which is requested by specific user. */

extern void add_request_page(void);

/* In future, if operator wants to edit request (i.e. change status or enter comments). */

extern void edit_request_page(void);

/* If operator wants to view specific request. */

extern void view_request_page(void);

/*
    Close application after operator completes his work
    input  : status of application.
    input  : message to display on the screen in case of erroneous exit.
*/

extern void close_application(int exit_status, const char * message);

/*
    Function which set cursor at X, Y on display
    input  : X Coordinate of screen.
    input  : Y Coordinate of screen.
 */

extern void gotoxy(int x, int y);

/*
    Function which display message and additional info on screen
    input  : message to be displayed.
    input  : additional information along with message.
 */

extern void show_dialogue_box(const char * message, char * info);

#endif /* application_h */
```

Application-c

```c
/*  application.c  */

#include "application.h"
#include "user.h"
#include "request.h"
#include "helpers.h"

#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>


const char * DELIMITER = ";";

void start_application(){

    /* Open user file in append mode and set `user_fp` points to file's stream. */
    user_fp = fopen(user_file_name, "a+");

    /* Open request file in append mode and set `request_fp` points to file's stream. */
    request_fp = fopen(request_file_name, "a+");

    /* If pointer cannot be set then warn operator about it and quit program. */
    if(user_fp == NULL || request_fp == NULL){
        printf("Something went wrong, while opening database file.\n");
       printf("Please make sure that file %s resides in current directory\n", user_file_name);
        getchar();
        return;
    }

    /* Fetch all users and setup array of all users */
    users = read_all_users_from_file(user_fp);

    /* Fetch all requests and setup array of all requests */
    requests = read_all_requests_from_file(request_fp);

    /* Show welcome page */
    welcome();
}

void welcome(){

    system("clear");

    gotoxy(25, 5);
    printf("Welcome To Electricity Board\n");
    gotoxy(27, 23);
    printf("Press Any Key to continue");

    /* Wait until user press some key. */
    getchar();

    render_menu();

}
```

```
void render_menu(){

    unsigned char choice = 0;
    char user_count_label[STR_MAX_LENGTH];

    system("clear");

    gotoxy(25, 5);
    printf("Select Any option from below");
    gotoxy(25, 7);
    printf("1. Add new user");
    gotoxy(25, 8);
    printf("2. View user");
    gotoxy(25, 9);
    printf("3. Add new connection request");
    gotoxy(25, 10);
    printf("4. Edit existing request");
    gotoxy(25, 11);
    printf("5. View Request");
    gotoxy(25, 12);
    printf("6. exit");

     sprintf(user_count_label, "Total Users : %lu and Total Requests : %lu", number_of_users,
number_of_requests);
    gotoxy(0, 24);
    printf("%s", user_count_label);

    gotoxy(30, 15);
    printf("Enter your choice : ");

    gotoxy(50, 15);
    scanf("%c", &choice);
    flush_buffer();

    switch (choice) {
        case '1':
            add_user_page();
            break;
        case '2':
            view_user_page();
            break;
        case '3':
            add_request_page();
            break;
        case '4':
            edit_request_page();
            break;
        case '5':
```

```
            view_request_page();
            break;
        case '6':
            close_application(EXIT_SUCCESS, "");
        default:
            close_application(EXIT_SUCCESS, "");
            break;
    }

}

void add_user_page(){

    /* Requests Id which will be stored in file. */
    unsigned long user_id = last_user_id + 1;

    /* User name. */
    char user_name[STR_MAX_LENGTH];

    /* User contact no. */
    char user_contact[ID_LENGTH + 2];

    /* User Id string, which will be shown in success dialogue box. */
    char user_id_string[STR_MAX_LENGTH];

    /* This variable will store user's data. */
    User user;

    /* User information in string, so it can be stored into file easily. */
    char * user_info_string = NULL;

    /* All labels which will be displayed on screen. */
    const char * enter_detail_label = "Enter details of user (All fields are required) ";
    const char * user_name_label = "Enter user name :  ";
    const char * user_contact_label = "Enter user Contact :  ";

    system("clear");

    gotoxy(0, 1);
    printf("%s", enter_detail_label);

    gotoxy(0, 3);
    printf("%s", user_name_label);

    gotoxy(0, 4);
    printf("%s", user_contact_label);

    /* Prepare input field for user name on row = 3, column = length(user_name_label). */
```

```
    gotoxy((int)strlen(user_name_label), 3);
    scanf("%[^\n]s", user_name);
    flush_buffer();

    /* Prepare input field for user contact on row = 4, column = length(user_contact_label). */

    gotoxy((int)strlen(user_contact_label), 4);
    scanf("%s", user_contact);
    flush_buffer();

    user.user_id = user_id;
    user.user_name = user_name;
    user.user_contact = user_contact;

    /* Delimited string representation of `user`, which will be directly stored in file. */
    /* `user_info_string` is pointing to heap memory, so release it after work is done. */

    user_info_string = get_user_string(&user);

    /* Write new user at end of file. */

    fseek(user_fp, 0, SEEK_END);
    fputs(user_info_string, user_fp);
    fputs("\n", user_fp);

    /* Free memory created by `get_user_string` */
    free(user_info_string);

    /* free memory occupied by array of users and requests */

    release_requests();
    release_users();

    /* Refresh all users */
    users = read_all_users_from_file(user_fp);

    /* Refresh all requests */
    requests = read_all_requests_from_file(request_fp);

    /* Prepare success dialogue */
    sprintf(user_id_string, "User Id : %lu", user_id);
    dialogue("User added successfully", user_id_string);
    render_menu();
}

void view_user_page(){

    /* User Id which operator wants to search. */
```

```
unsigned long user_id = 0;

/* User details.*/
User * user = NULL;

/* All Request made by above user */
Request ** user_requests = NULL;

/* Total count of all requests made by above user */
int user_requests_count = 0;

/* Line offset will be helpful to print information at desired location. */
int line_offset = 1;

system("clear");

printf("Enter user id : ");
scanf("%lu", &user_id);
flush_buffer();

/* Get user details based on `user_id` */
user = get_user(user_id);

/* If no such user found, then show error */
if(user == NULL){
    system("clear");
    show_dialogue_box("No such user exists.", "");
    render_menu();
    return;
}

system("clear");

/* Show user details and requests made by him */
printf("User Details\n");

printf("\nUser Id : %lu", user->user_id);
printf("\nUser Name : %s", user->user_name);
printf("\nUser Contact : %s", user->user_contact);

/* `user_requests` points to memory in heap, so release memory after work is done */
user_requests = get_request_of_user(user_id, &user_requests_count);

gotoxy(41,1);

if(user_requests == NULL){
    printf("Request Details\n");
    gotoxy(41, 3);
```

```
        printf("No requests made by this user");
        gotoxy(0, 24);
        printf("Press any key to return to main menu");

        /* Wait for user to press some key. */
        getchar();
        render_menu();
        return;
    }

    /* Show request details on right side */
    printf("Request Details");

    for (int i = 0 ; i < user_requests_count; i++) {
        line_offset += 2;
        gotoxy(41,line_offset++);
        printf("Request Id : %lu", user_requests[i]->request_id);
        gotoxy(41,line_offset++);
        printf("Request Status : %s", get_request_status_string(user_requests[i]->status));
        gotoxy(41,line_offset++);
        printf("Comments : %s", user_requests[i]->comment);
        gotoxy(41,line_offset++);
        printf("Address : %s", user_requests[i]->address);
    }

    /* releasing heap memory pointed to by `user_requests` */
    free(user_requests);

    gotoxy(0, 24);
    printf("Press any key to return to main menu");

    /* Wait for user to press some key. */
    getchar();
    render_menu();

}

void view_request_page(){

    /* Request ID which operator wants to search. */
    unsigned long request_id = 0;

    /* Request details. */
    Request * request = NULL;

    system("clear");

    printf("Enter request id : ");
```

```
    scanf("%lu", &request_id);
    flush_buffer();

    /* Get request details based on `request_id` */
    request = get_request(request_id);

    /* If no such request found, then show error */
    if(request == NULL){
        system("clear");
        show_dialogue_box("No such request exists.", "Please contact administration.");
        render_menu();
        return;
    }

    system("clear");

    /* Show Request details */
    printf("Request Details\n");

    printf("\nRequest Id : %lu", request->request_id);
    printf("\nRequested By : %s (id : %lu)", request->requested_by->user_name,
             request->requested_by->user_id);
    printf("\nRequest Status : %s", get_request_status_string(request->status));
    printf("\nAddress : %s", request->address);
    printf("\nComments : %s", request->comment);

    gotoxy(0, 24);

    printf("Press any key to return to main menu");

    /* Wait for user to press some key. */
    getchar();
    render_menu();

}

void add_request_page(){

    /* Requests Id which will be stored in file.  */
    unsigned long request_id = last_request_id + 1;

    /* User id of user who made request */
    unsigned long user_id = 0;

    /* Comment, if operator wants to enter one. */
    char comment[STR_MAX_LENGTH];

    /* Address of resource, where electricity lines to be installed. */
```

```
char address[STR_MAX_LENGTH];

/* Request Id string, which will be shown in success dialogue box. */
char request_id_string[STR_MAX_LENGTH];

/* User details. */
User * user = NULL;

/* Request details. */
Request request;

/* Request information in string, so it can be stored into file easily. */
char * request_info_string = NULL;

/* All labels which will be displayed on screen. */
const char * enter_detail_label = "Enter details of Request made by user ";
const char * user_id_label = "Enter user id :  ";
const char * request_comment_label = "Enter any comment (optional) :  ";
const char * request_address_label = "Enter address :  ";

system("clear");

gotoxy(0, 1);
printf("%s", enter_detail_label);

gotoxy(0, 3);
printf("%s", user_id_label);

gotoxy(0, 4);
printf("%s", request_comment_label);

gotoxy(0, 5);
printf("%s", request_address_label);

/* Prepare input field for user name on row = 3, column = length(user_name_label). */

gotoxy((int)strlen(user_id_label), 3);
scanf("%lu", &user_id);
flush_buffer();

/* Prepare input field for user contact on row = 4, column = length(user_contact_label). */

gotoxy((int)strlen(request_comment_label), 4);
/* Using fgets because string can be empty. */
fgets(comment, STR_MAX_LENGTH, stdin);

/* If user press just enter. (i.e. empty data). */
if(strcmp(comment, "\n") == 0){
```

```
        strcpy(comment, "");
    }else{
        /* If operator has entered any string, it will contain '\n' as last character. */
        /* So replace that character with NULL */
        comment[strlen(comment) - 1] = '\0';
    }

    /* Prepare input field for user contact on row = 5, column = length(user_contact_label). */

    gotoxy((int)strlen(request_address_label), 5);

    scanf("%[^\n]s", address);
    flush_buffer();

    user = get_user(user_id);

     /* If cannot find user corresponding to `user_id` then simply redirect operator to main
menu. */

    if(user == NULL){
        system("clear");
        show_dialogue_box("No such user exists.", "Please contact administration.");
        render_menu();
    }

    request.request_id = request_id;
    request.requested_by = user;
    request.status = PENDING;
    request.comment = comment;
    request.address = address;

    /* Delimited string representation of `request`, which will be stored in file directly. */
    /* `request_info_string` is pointing to heap memory, so release it after work is done */

    request_info_string = get_request_string(&request);

    /* Write new request to file. */

    fseek(request_fp, 0, SEEK_END);
    fputs(request_info_string, request_fp);
    fputs("\n", request_fp);

    /* Free memory created by `get_user_string` */

    free(request_info_string);

    /* free memory occupied by array of users and requests */
    release_requests();
```

```
    release_users();

    /* Refresh all users */
    users = read_all_users_from_file(user_fp);

    /* Refresh all requests */
    requests = read_all_requests_from_file(request_fp);

    /* Prepare success dialogue */
    sprintf(request_id_string, "Request Id : %lu", request_id);
    show_dialogue_box("Request added successfully.", request_id_string);
    render_menu();

}

void edit_request_page(){

    /* Request id for request which user want to edit. */
    unsigned long request_id = 0;

    /* New status for request. Defaults to `PENDING`. */
    request_status new_status = PENDING;

    /* New comments which user will enter.  */
    char new_comment[STR_MAX_LENGTH];

    /* Request details. */
    Request * request = NULL;

    /* Labels which will be displayed on screen. */

    char * enter_new_status_label = "Enter request status :  ";
    char * enter_new_status_suggestion_label = "(1 -> In progress, 2 -> Rejected)";
    char * enter_new_comment_label = "Enter any comments : ";

    system("clear");
    printf("Enter Request id : ");
    scanf("%lu", &request_id);
    flush_buffer();

    /* Fetch request from `requests` array. */
    request = get_request(request_id);

    /* If cannot find request corresponding to `requests_id` then simply redirect operator to
main menu. */
    if(request == NULL){
        system("clear");
        show_dialogue_box("No such request exists.", "");
```

```
        render_menu();
        return;
    }

    /* Display request parameters on screen */
    printf("\nRequest Details\n");
    printf("\nRequest Id : %lu", request->request_id);
      printf("\nRequested By : %s (id : %lu)", request->requested_by->user_name, request-
>requested_by->user_id);
    printf("\nRequest Status : %s", get_request_status_string(request->status));
    printf("\nAddress : %s", request->address);
    printf("\nComments : %s", request->comment);

    printf("\n\n******************************************************************************");

    printf("\nEnter new Details\n\n");

    printf("%s", enter_new_status_label);
    printf("\n%s", enter_new_status_suggestion_label);

    gotoxy((int)strlen(enter_new_status_label), 14);

    scanf("%d", &new_status);
    flush_buffer();

    printf("\n%s", enter_new_comment_label);
    fgets(new_comment, STR_MAX_LENGTH, stdin);

    /* If user press just enter. (i.e. empty string) */
    if(strcmp(new_comment, "\n") == 0){
        strcpy(new_comment, request->comment);
    }else{
        new_comment[strlen(new_comment) - 1] = '\0';
    }

    /* Store updated data in `request` */
    request->status = new_status;

    /* Do not copy `new_comment` into `request->comment`, */
    /* that requires expansion in size of `request->comment`, */
    /* if `new_comment` is bigger than `request->comment` in length. */
    /* Just free memory created by `request->comment` (otherwise memory will leak.) and create
new one */

    free(request->comment);
    request->comment = strdup(new_comment);
```

```
    /* Replace Data in file */
    replace_request_in_file(request_fp, request);

    /* free memory occupied by array of users and requests */
    release_requests();
    release_users();

    /* Refresh all users */
    users = read_all_users_from_file(user_fp);

    /* Refresh all requests */
    requests = read_all_requests_from_file(request_fp);

    show_dialogue_box("Request updated successfully.", "");
    render_menu();

}

void close_application(int status, const char * message){
    fclose(user_fp);
    fclose(request_fp);
    release_users();
    release_requests();
    system("clear");
    printf("%s", message);
    exit(status);
}

void show_dialogue_box(const char * message, char * id){

    int message_length = (int)strlen(message);

    const char * press_any_key = "Press any key to continue";
    int info_length = (int)strlen(press_any_key);

    int left_offset = 15;
    int length = 50;
    int line_offset = 9;

    system("clear");

    /* Line 1 */
    gotoxy(left_offset, line_offset++);
    printf("**************************************************");

    /* Line 2 */
    gotoxy(left_offset, line_offset);
    printf("*");
```

```
    gotoxy(left_offset + length - 1, line_offset++);
    printf("*");

    /* Line 3 */
    gotoxy(left_offset, line_offset);
    printf("*");
    gotoxy((80 - message_length)/2, line_offset);
    printf("%s", message);
    gotoxy(left_offset + length - 1, line_offset++);
    printf("*");

    if(strcmp(id, "") != 0){
        /* Line 4 */
        gotoxy(left_offset, line_offset);
        printf("*");
        gotoxy((int)((80 - strlen(id))/2), line_offset);
        printf("%s", id);
        gotoxy(left_offset + length - 1, line_offset++);
        printf("*");
    }

    /* Line 5 */
    gotoxy(left_offset, line_offset);
    printf("*");
    gotoxy((int)((80 - info_length)/2), line_offset);
    printf("%s", press_any_key);
    gotoxy(left_offset + length - 1, line_offset++);
    printf("*");

    /* Line 6 */
    gotoxy(left_offset, line_offset);
    printf("*");
    gotoxy(left_offset + length - 1, line_offset++);
    printf("*");

    /* Line 7 */
    gotoxy(left_offset, line_offset++);
    printf("**************************************************");

    getchar();

}

void gotoxy(int x,int y){
    printf("%c[%d;%df",0x1B,y,x);
}
```

User-h

```c
/*  user.h  */

/* We are using #ifndef macro to prevent multiple inclusion of header files */

#ifndef user_h
#define user_h

#include <stdio.h>

/* Structure which stores data of user. */

typedef struct User {

    /* Unique identifier associated with every user. */
    unsigned long user_id;

    /* User name which is of type string aka `char *`. */
    char * user_name;

    /* User contact no. is of type string aka `char *`. */
    char * user_contact;

} User;


/* User file name in which all user's data will be stored. */
extern const char * user_file_name;

/* Global variable which contains all user's data in memory. */
extern User ** users;

/* Global variable which stores total count of users. */
extern unsigned long number_of_users;

/* Global File pointer, so we can access user file in any function. */
extern FILE * user_fp;

/* Variable which keeps track of last inserted user id. */
extern unsigned long last_user_id;


/*
     Function which give `User` based on their `user_id`.
     input  : user id.
     output : Pointer to `User` which contains user's information in structure format.
 */

extern User * get_user ( unsigned long user_id );
```

```
/*
    Function which converts `User` data into string, so that we can store this o/p in file.
    input  : User information in structure format.
    output : comma separated string representation of `user_info`.
 */

char * get_user_string ( User * user );


/*
    Function which converts string (which we read from i/p file) into `User` data.
    input  : comma separated string which contains user's information.
    output : Pointer to `User` which contains all user's information in structure format.
 */

User * get_user_info ( char * user_info );

/*
     Function which reads all lines from file, converts and returns array of all users.
     input  : FILE pointer of user file.
     output : Pointer to array of all users.
 */

User ** read_all_users_from_file ( FILE * fp );

/*
     Function which replace specific user in file.
     input  : FILE pointer of user file.
     input  : User to be replaced.
 */

void replace_user_in_file ( FILE * fp , User * user );

/* Function which releases memory created by users (in heap). */

void release_users ( void );

#endif /* user_h */
```

User-c

```
/*  user.c  */

#include "user.h"
#include "helpers.h"
#include "application.h"

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

```
#include <string.h>

const char * user_file_name = "users.txt";

User ** users = NULL;

unsigned long last_user_id = 0;

FILE * user_fp = NULL;

unsigned long number_of_users = 0;


User * get_user ( unsigned long user_id ){
    for (int i = 0 ; i < number_of_users; i++) {
        if(users[i]->user_id == user_id){
            return users[i];
        }
    }

    return NULL;
}


char * get_user_string(User * user_info){

    char user_id_string[ID_LENGTH];

    char * user_info_string = (char *)malloc(sizeof(char) * STR_MAX_LENGTH);

    /* Convert user id into string. */
    sprintf(user_id_string, "%lu", user_info->user_id);

    /* Conversion of multiple strings into single delimited string. */
    strcpy(user_info_string, strcat(user_id_string, DELIMITER));
    user_info_string = strcat(user_info_string, user_info->user_name);
    user_info_string = strcat(user_info_string, DELIMITER);
    user_info_string = strcat(user_info_string, user_info->user_contact);

    /* We need to release memory created by `user_info_string`. */
    return user_info_string;

}

User * get_user_info ( char * user_info ){

    char * token = NULL;
```

```c
    /* Set `COLUMN_TAG` to 0 so that we can read new values. */
    /* `COLUMN_TAG` will help us to track which column we are going to read. */

    int COLUMN_TAG = 0;

    /* Keep copy of original `user_info` because this string will be modified. */

    char * user_info_copy = strdup(user_info);

    User * read_user = (User *) malloc(sizeof(User));

    while((token = tokenize_string(user_info_copy, DELIMITER, &user_info_copy)) != NULL){

        switch (COLUMN_TAG) {
            case 0:
                read_user->user_id = atoi(token);
                COLUMN_TAG++;
                free(token);
                break;
            case 1:
                read_user->user_name = strdup(token);
                COLUMN_TAG++;
                free(token);
                break;
            case 2:
                read_user->user_contact = strdup(token);
                COLUMN_TAG++;
                free(token);
                break;
            default:
                close_application(EXIT_FAILURE, "Error while parsing user database.....");
                break;
        }
    }

    free(user_info_copy);

    /* Check if we have read anything or not. */

    if(COLUMN_TAG != 3){
        /* If not then just release memory created by `read_user`. */
        free(read_user);
    }

    return read_user;

}
```

```
User ** read_all_users_from_file( FILE * fp ){

    /* All users (pointer to `User`) will be stored in this array. */
    User ** all_users = NULL;

    /* Read each line from file and prepare user detail. */
    User * read_user = NULL;

    /* Total number of users. */
    unsigned long user_count = 0;

    /* Read line from file and store in this variable. */
    char * user_info = (char *)malloc(sizeof(char) * STR_MAX_LENGTH);

    /* Set file pointer to beginning of file. */
    rewind(fp);

    /* Start reading file. */

    while(fgets(user_info, STR_MAX_LENGTH, fp) != NULL){

        if(strcmp(user_info, '\n') == 0)
            continue;

        /* Get User structure from string, which we have read from file */

        /* We do not need to release memory occupied by `read_user` because, */
        /* We are going to store address of this variable into `all_users` array, */
        /* which will be released at end of the program.  */

        read_user = get_user_info(user_info);

        /* If we cannot create user successfully, then just continue to next cycle. */
        if(read_user == NULL){
            continue;
        }

        /* Allocate memory for `all_users` array. */
        all_users = (User **)realloc(all_users, sizeof(User *) * (1 + user_count));

        /* Store address of user into `all_users` array. */
        all_users[user_count++] = read_user;

    }

    /* Release memory occupied by `user_info`. */
```

```
    free(user_info);

    /* Keep track of number of users in global variable. */
    last_user_id = number_of_users = user_count;

    return all_users;

}

void replace_user_in_file (FILE * fp, User * updated_user){

    User * read_user = NULL;

    /* below function can leak memory, so clear memory before returning from this function. */
    char * updated_user_string = get_user_string(updated_user);

    char line[STR_MAX_LENGTH];

    /* Open temp file in write mode. */
    FILE * write_fp = fopen("temp.txt", "w");

    /* Set file pointer to beginning of file. */
    rewind(fp);

    while ((fgets(line, STR_MAX_LENGTH, fp)) != NULL){

        /* Create user from string, which we have read from file. */
        /* Memory created by below function will be in heap. */
        /* So release that memory before loop ends. */

        if((read_user = get_user_info(line)) == NULL){
            continue;
        }

        if(read_user->user_id == updated_user->user_id){

            /* If user id matches, then replace line with `updated_user_string`. */
            fputs(updated_user_string, write_fp);

            /* As `updated_user_string` does not contain "\n", add one manually. */
            fputs("\n", write_fp);

        }else{
            /* Write `line` as it is in file. */
            /* Do not need to put extra "\n", as `line` already contains "\n". */
            fputs(line, write_fp);
        }
```

```
        free(read_user);
    }

    /* Close temp file. */
    fclose(write_fp);

    /* Close original file. */
    fclose(fp);

    /* Remove original file from disk. */
    remove(user_file_name);

    /* Rename temporary file with original file name. */
    rename("temp.txt", user_file_name);

    /* Open that file in `a+` mode so we have reached state here we left off. */
    user_fp = fopen(user_file_name, "a+");

    free(updated_user_string);

}


void release_users(){
    if(!users){
        return;
    }
    for(int i = 0 ; i < number_of_users ; i++){
        free(users[i]->user_name);
        free(users[i]->user_contact);
    }
    free(users);
    users = NULL;
}
```

Request-h

```
/*  request.h */

/* We are using #ifndef macro to prevent multiple inclusion of header files */

#ifndef request_h
#define request_h

#include "user.h"
#include <stdio.h>

/* Type of request status can be `Pending`, `In process` or `Rejected` */
```

```c
typedef enum request_status {
    PENDING,
    IN_PROCESS,
    REJECTED
} request_status;

/* Function will return string based on `request_status` */

char * get_request_status_string(request_status);


/* Structure which stores data of user */

typedef struct Request{

    /* Unique identifier associated with every request */
    unsigned long request_id;

    /* User by which connection request was created */
    User * requested_by;

    /* Current status of request */
    request_status status;

    /* Comments on request. If operator decides to reject this request, */
    /* he can optionally add reason for rejection in comments. */
    char * comment;

    /* Address at which electricity needs to be installed. */
    char * address;

} Request;


/* Request file name in which data of all requests will be stored. */
extern const char * request_file_name;

/* Global variable which contains data of all requests. */
extern Request ** requests;

/* Global variable which stores total count of all requests. */
extern unsigned long number_of_requests;

/* Global File pointers, so we can access it in any of the functions */
extern FILE * request_fp;

/* Variable which keeps track of last inserted request id. */
```

```c
extern unsigned long last_request_id;


/*
     Function which give `Request` based on their `request_id`.
     input  : request id.
     output : Pointer to `User` which contains request information in structure format.
 */

extern Request * get_request ( unsigned long request_id );

/*
    Function which converts `Request` data into string, so that we can store this o/p in file.
     input  : Request information in structure format.
     output : comma separated string representation of `request_info`.
 */

char * get_request_string ( Request * request );

/*
     Function which converts string (which we read from i/p file) into `Request` data.
     input  : comma separated string which contains request information.
     output : Pointer to `Request` which contains request information in structure format.
 */

Request * get_request_info ( char * request_info_string );

/*
     Function which reads all lines from file, converts and returns array of all requests.
     input  : FILE pointer of user file.
     output : Pointer to array of all requests.
 */

Request ** read_all_requests_from_file ( FILE * fp );

/*
     Function which replace specific request in file
     input  : FILE pointer of request file.
     input  : Request to be replaced.
 */

void replace_request_in_file ( FILE * fp , Request * request );

/*
     Function which returns all Request details, for given user id
     input  : user id.
     input  : address of variable which contains number of request.
     output : Pointer to array of all requests of specified user.
```

```
 */

Request ** get_request_of_user(unsigned long user_id, int * request_count);

/* Function which releases memory created by requests (in heap.) */

void release_requests(void);

#endif /* request_h */
```

Request-c

```
/*  request.c  */

#include "request.h"
#include "helpers.h"
#include "application.h"

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

Request ** requests = NULL;

const char * request_file_name = "requests.txt";

unsigned long number_of_requests = 0;

FILE * request_fp = NULL;

unsigned long last_request_id = 0;


Request * get_request(unsigned long request_id){
    for (int i = 0; i < number_of_requests; i++) {
        if(requests[i]->request_id == request_id){
            return requests[i];
        }
    }

    return NULL;
}

char * get_request_string(Request * request_info){

    char request_id_string[ID_LENGTH], user_id_string[ID_LENGTH], request_status[2];

    char * request_info_string = (char *)malloc(sizeof(char) * STR_MAX_LENGTH);
```

```
    if(request_info->requested_by == NULL){
        /* If we can not get who has made request, then just show message and quit program */
        close_application(EXIT_FAILURE, "Some request does not have user.");
    }

    /* sprintf will be used to convert integer into string. */

    sprintf(request_id_string, "%lu", request_info->request_id);
    sprintf(user_id_string, "%lu", request_info->requested_by->user_id);
    sprintf(request_status, "%d", request_info->status);

    /* Conversion of multiple strings into single delimited string. */

    strcpy(request_info_string, strcat(request_id_string, DELIMITER));
    request_info_string = strcat(request_info_string, user_id_string);
    request_info_string = strcat(request_info_string, DELIMITER);
    request_info_string = strcat(request_info_string, request_status);
    request_info_string = strcat(request_info_string, DELIMITER);
    request_info_string = strcat(request_info_string, request_info->comment);
    request_info_string = strcat(request_info_string, DELIMITER);
    request_info_string = strcat(request_info_string, request_info->address);

    /* We need to release memory created by `user_info_string`. */
    return request_info_string;

}

Request * get_request_info ( char * request_info ){

    char * token = NULL;

    /* Set `COLUMN_TAG` to 0 so that we can read new values. */
    /* `COLUMN_TAG` will help us to track, which column we are going to read. */

    int COLUMN_TAG = 0;

    /* Keep copy of original `request_info` because this string will be modified. */
    /* Need to free memory pointed to by `request_info_copy`. */

    char * request_info_copy = strdup(request_info);

    Request * read_request = (Request *) malloc(sizeof(Request));

    User * user = NULL;

   while((token = tokenize_string(request_info_copy, DELIMITER, &request_info_copy)) != NULL){

        switch (COLUMN_TAG) {
```

```
        case 0:
            read_request->request_id = atoi(token);
            COLUMN_TAG++;
            free(token);
            break;
        case 1:
            user = get_user(atoi(token));
            free(token);
            /* It is not necessary, that user mentioned by request does exists */
            if(!user){
                free(read_request);
                free(request_info_copy);
                return NULL;
            }
            read_request->requested_by = user;
            COLUMN_TAG++;
            break;
        case 2:
            read_request->status = atoi(token);
            COLUMN_TAG++;
            free(token);
            break;
        case 3:
            read_request->comment = strdup(token);
            COLUMN_TAG++;
            free(token);
            break;
        case 4:
            read_request->address = strdup(token);
            COLUMN_TAG++;
            free(token);
            break;
        default:
            close_application(EXIT_FAILURE, "Error while parsing user database.....");
            break;
    }
}

free(request_info_copy);

/* Check if we have read everything or not */

if(COLUMN_TAG != 5){
    /* If not then just release memory created by `read_request` */
    free(read_request);
}
```

```
    return read_request;

}

Request ** read_all_requests_from_file( FILE * fp ){

    /* All requests (pointer to `Request`) will be stored in this array. */
    Request ** all_requests = NULL;

    /* Read each line from file and prepare request detail. */
    Request * read_request = NULL;

    /* Total number of requests. */
    unsigned long request_count = 0;

    /* Read line from file and store in this variable. */
    char * request_info = (char *)malloc(sizeof(char) * STR_MAX_LENGTH);

    /* Set file pointer to beginning of file. */
    rewind(fp);

    /* Start reading file. */

    while(fgets(request_info, STR_MAX_LENGTH, fp) != NULL){

        if(strcmp(request_info, "\n") == 0)
            continue;

        /* Get Request structure from string, which we have read from file */

        /* We do not need to release memory occupied by `read_request` because, */
        /* We are going to store address of this variable into `all_requests` array, */
        /* which will be released at end of the program.  */

        read_request = get_request_info(request_info);

        /* If we can not create request successfully, then just continue to next cycle. */
        if(read_request == NULL){
            continue;
        }

        /* Allocate memory for `all_requests` array. */
          all_requests = (Request **)realloc(all_requests, sizeof(Request *) * (1 + request_
count));

        /* Store address of request into `all_requests` array. */
        all_requests[request_count++] = read_request;
```

```
    }

    /* Release memory occupied by `request_info`. */
    free(request_info);

    /* Keep track of number of users in global variable. */
    last_request_id = number_of_requests = request_count;

    return all_requests;

}

Request ** get_request_of_user(unsigned long user_id, int * request_count){

    Request ** user_requests = NULL;
    int count = 0;

    for(int i = 0 ; i < number_of_requests ; i++){
        if(requests[i]->requested_by->user_id == user_id){
          user_requests = (Request **)realloc(user_requests, sizeof(Request *) * (1 + count));
            user_requests[count++] = requests[i];
        }
    }
    /* Modify count. */
    *request_count = count;

    return user_requests;

}

void replace_request_in_file (FILE * fp, Request * updated_request){

    Request * read_request = NULL;

    /* below function can leak memory, so clear memory before returning from this function. */
    char * updated_request_string = get_request_string(updated_request);

    char line[STR_MAX_LENGTH];

    /* Open temp file in write mode. */
    FILE * write_fp = fopen("temp.txt", "w");

    rewind(fp);

    while ((fgets(line, STR_MAX_LENGTH, fp)) != NULL){

        /* Create request from string, which we have read from file. */
```

```
        /* Memory created by below function will be in heap. */
        /* So release that memory before loop ends. */

        if((read_request = get_request_info(line)) == NULL){
            continue;
        }

        if(read_request->request_id == updated_request->request_id){

            /* If request id matches, then replace line with `updated_request_string`. */
            fputs(updated_request_string, write_fp);

            /* As `updated_request_string` does not contain "\n", add one manually. */
            fputs("\n", write_fp);

        }else{
            /* Write `line` as it is in file. */
            /* Do not need to put extra "\n", as `line` already contains "\n". */
            fputs(line, write_fp);
        }

        free(read_request);
    }

    /* Close temp file. */
    fclose(write_fp);

    /* Close original file. */
    fclose(fp);

    /* Remove original file from disk. */
    remove(request_file_name);

    /* Rename temporary file with original file name. */
    rename("temp.txt", request_file_name);

    /* Open that file in `a+` mode so we have reached state here we left of. */
    request_fp = fopen(request_file_name, "a+");

    free(updated_request_string);

}


char * get_request_status_string(request_status status){

    switch (status) {
```

```
        case PENDING:
            return "Pending";
        case IN_PROCESS:
            return "In Progress";
        case REJECTED:
            return "Rejected";
        default:
            break;
    }

    return "Pending";

}

void release_requests(){
    if(!requests){
        return;
    }
    for(int i = 0 ; i < number_of_requests ; i++){
        free(requests[i]->comment);
        free(requests[i]->address);
        requests[i]->requested_by = NULL;
    }

    free(requests);
    requests = NULL;
}
```

Helpers-h

```
/*  helper.h  */

#ifndef helpers_h
#define helpers_h

/*
    Flush buffer after user enter some input.
    If we won't flush buffer then `enter` pressed by user will be stored in buffer,
    and it will be used as an input to next I/O function, which may cause issue.
*/

extern void flush_buffer(void);

/*
     Function which returns single string delimited by `delimiter` from `source`.
     Also we want to modify `source` after reading token so we have used `context` (char **),
     which is pointer to `source`, which will modify input `source`
```

```
     input  : Delimited string.
     input  : Delimiter by which string is separated.
     input  : context i.e. address of `string` (char *), so we have used char ** to catch
              address of string.
     output : Single token separated by `delimiter`.
 */


extern char * tokenize_string(char * source, char const * delimiter, char ** context);

#endif /* helpers_h */
```

Helpers-c

```
/* helpers.c */

#include "helpers.h"
#include "application.h"

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


char * tokenize_string(char * source, char const * delimiter, char ** context){

    /* Example `source` string is "A;B;C\n\0". */
    /* '\n' will be at end of the `source` as we are reading from file. */

    if(source == NULL){
        return NULL;
    }

    /* String after 1st `delimiter` occurs will be stored in this variable (";B;C\0"). */
    char * rest_string = strpbrk(source, delimiter);

    /* Creating a copy of above variable, as it will get modified */
    char copy_rest_string[STR_MAX_LENGTH];

    /* Token is string before `delimiter` ("A\0") */
    char * token = NULL;

    /* `rest_string` will be NULL when there is not any `delimiter` (example string "C\n\0") */

    if(rest_string != NULL){

        /* Make copy of `rest_string` so `copy_rest_string` will be ";B;C\n\0" */
        strcpy(copy_rest_string, rest_string);

       /* 1st character of `rest_string` is our `delimiter` so we need to replace it by "\0" */
```

```
        *rest_string = 0;

        /* With above operation `source` string will also get modified */
        /* and becomes "A\0B;C\n\0" i.e. "A\0" (string in C language is NULL terminated) */

        /* Create copy of modified `source` string which is "A\0" */
        /* As this copy is created with `strdup`, memory in heap will be created */
        /* which we will clear later . */

        token = strdup(source);

        /* `*context` is pointing to `source` */
        /* i.e. we are clearing memory pointed to by `source` */
        /* Otherwise memory will be leaked as, `source` string is getting modified */

        free(*context);

        /* point `*context` (i.e. `source`) to copied string which is ";B;C\0" */
        /* Notice +1 here, this is because we want to skip 1st `delimiter` which is at 1st
character of `copy_rest_string` */
        /* Do not need worry about memory created by below operation */
        /* because in next iteration, it will be freed */

        *context = strdup(copy_rest_string + 1);

    }else{

        /* If we can not find any `delimiter` then `source` should be like "C\n\0" */
        /* So first replace last "\n" with "\0" */

        source[strlen(source) - 1] = '\0';

        /* Create token from `source` which is now "C\0\0" i.e. "C\0" */
        token = strdup(source);

        /* Free memory pointed to by `*context` */
        free(*context);

       /* And set `*context` i.e. `source` to NULL so that we can break loop (exit condition) */

        *context = NULL;
    }

    return token;

}

void flush_buffer(){
    while (getchar() != '\n');
}
```

Fig. 1 Welcome screen

Fig. 2 Main menu

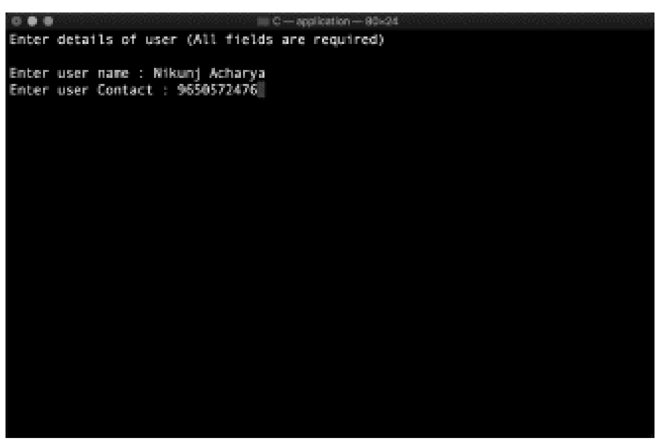


Fig. 3 Add user

Fig. 4 User success dialogue

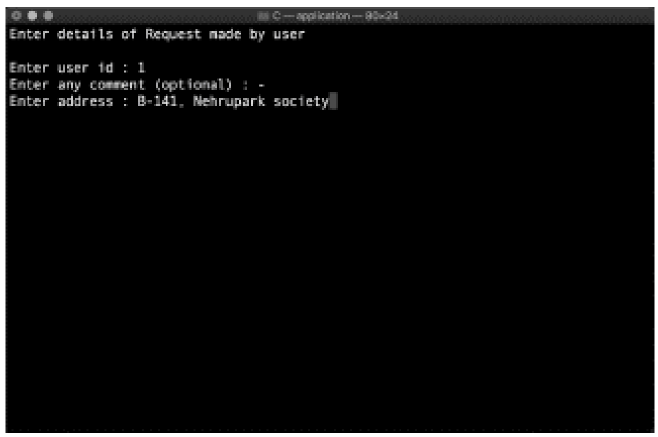


Fig. 5 Add request

Fig. 6 Request success dialogue

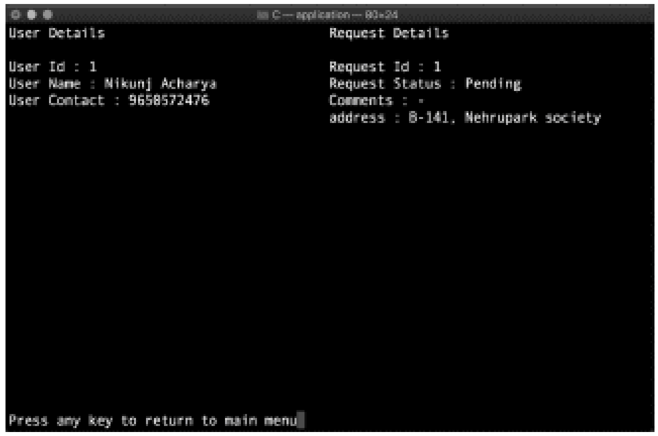


Fig. 7 View user

Fig. 8 Edit request

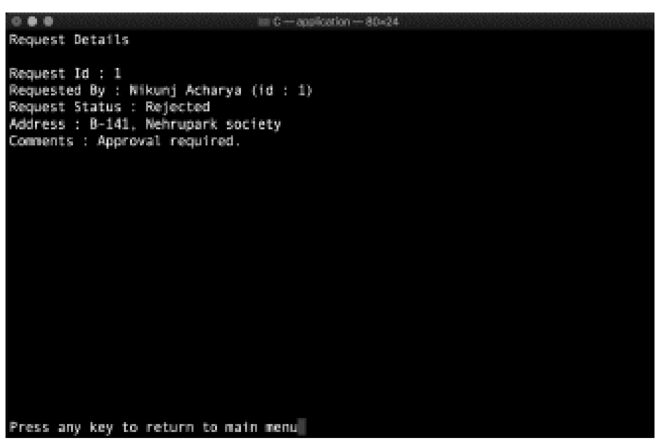


Fig. 9 View request

Project 2: Making Web Services in C

Overview

In this lesson, we will develop RESTful APIs in C. This might be interesting as usually, web services or RESTful APIs (as they are so called) are usually made using an Object-Oriented Programming language like Ruby, Java or Python. Any programming language can be used to do any task when needed but the question we need to ask is: Does it make our work easier? After all, that is the thing which program or coding intends to do, make our life easier.

Even though C is not primarily made for developing Web Services which can respond to calls over the network for HTTP methods like GET, POST, PUT or DELETE; we can still do so using some external libraries for demonstration purposes. We say demonstration purposes because it is very less probable that you will actually do so in real-life project but just so that you know, it is possible (just like everything else) and we will do it.

What will we use?

1. **Ubuntu:** To develop Web Services in C, we will use the Ubuntu platform but please note, you can make use of any other platform equally well, like Windows or a Macintosh. The commands shown in this lesson will be for Ubuntu platform but these can have some equal counterparts for other mentioned platforms as well.
2. **Onion :** We will use an external third party library called Onion . The library is dual licensed under the Apache2 license and GPLv2+, so we can make almost anything with it, use it in our commercial and free software programs, and modify it to our needs. This is a simple library which can be used to make Web Services in C with easy callback mechanism (more on this in next section).

What is Callback Mechanism?

Before we start making Web Services in C and look at how Onion library actually works, we need to understand what is callback mechanism and how does it work.

A callback is any executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at a given time [Source : Wiki]. In simple language, If the reference of a function is passed to another function as an argument to call it, then it it will be called as a Callback function. Let us consider an example here:

```
void populate_array(int *array, size_t arraySize, int (*getNextValue)(void))
{
    for (size_t i=0; i<arraySize; i++)
        array[i] = getNextValue();
}

int getNextRandomValue(void)
{
    return rand();
}
int main(void)
{
    int myarray[10];
    populate_array(myarray, 10, getNextRandomValue);
    ...
}
```

Here, the *populate_array* function takes a function pointer as its third parameter, and calls it to get the values to populate the array with. We've written the callback *getNextRandomValue* , which returns a random-ish value, and passed a pointer to it to *populate_array* . *populate_array* function will call our callback function 10 times and assign the returned values to the elements in the given array.

This means, that a callback function is something which is passed as an argument to another function and is called when an event occurs inside the function or when some logic is processed and the function being passed is needed to be called back.

For the curious:

> There are two ways that a callback is used: synchronous callback and asynchronous callback. A synchronous callback is provided to another function which is going to do some task and then return to the caller with the task completed. An asynchronous callback is provided to another function which is going to start a task and then return to the caller with the task possibly not

completed.

A synchronous callback is typically used to provide a delegate to another function to which the other function delegates some step of the task. Classic examples of this delegation are the functions bsearch() and qsort() from the Standard C Library. Both of these functions take a callback which is used during the task the function is providing so that the type of the data being searched, in the case of bsearch(), or sorted, in the case of `qsort(), does not need to be known by the function being used.

Making a Web Server in C

Now, we will start making a simple Web Server in C which can serve calls to GET endpoints in the program. Please note that the Onion library should be setup in order to use the mentioned program. The library can found at this link: *https://github.com/davidmoreno/onion* . Th e instructions to setup the library are very clearly given on the repository wiki.

We will start by providing a simple list of header libraries which we need to include in our program:

```
#include <onion/onion.h>
#include <onion/log.h>
#include <onion/version.h>
#include <signal.h>
#include <netdb.h>
#include <stdlib.h>
#include <unistd.h>
```

This include section involved libraries from the Onion library and some standard C libraries as well which help to call standard functions and use signal parameters from C binaries as well.

Even though it might look a little early but we will define two very simple callback functions now:

```
int hello(void *p, onion_request *req, onion_response *res){
          onion_response_write0(res,'pong');
          return OCS_PROCESSED;
}

onion *o - NULL;

static void shutdown_server(int _){
          if (o)
            onion_listen_stop(o);
}

onion_connection_status random_timeout(void *_, onion_request *req, onion_response *res){
          int ms-2000 * ( ((float)RAND_MAX) / rand() );
          ONION_INFO('Wait %d ms', ms);
          usleep(ms * 1000);
          ONION_INFO('Done');
          onion_response_write(res,'OK',2);
          return OCS_PROCESSED;
}
```

Let us look at what we did in the above callback functions here:

- The first function we defined is *hello* function. This is a very simple function which has some arguments which includes Onion library's request and response reference object pointers. These pointers should be present in each and every callback function to allow them to modify response object and obtain the parameters present in the request object itself.
- This function simply writes Hello World string to the Onion library's response object. Please note that even though this function returns an integer and not the response object itself, the response object is still processed and is returned to the user when the integer returned is actually **just a signal to the system that method processing for HTTP call is complete** .
- The same integer is returned in the next callback function as well, random_timeout. This is done for the same reason.
- The second callback function just times out randomly. It first finds a random number, sleeps for some milliseconds, write a String in the response object and finally signals that the processing by the callback function is done.

Now, we finally need a main function to call everything of above we wrote. But wait, where did we define that which of these callback methods will be associated with what REST endpoint? Easily guessed, this will be done in the main function as well. Let us write down the main function for the program to do all of this:

```
int main(int argc, char **argv){
    signal(SIGINT, shutdown_server);
    signal(SIGTERM, shutdown_server);

    ONION_VERSION_IS_COMPATIBLE_OR_ABORT();

    o-onion_new(O_POOL);
    onion_set_timeout(o, 5000);
    onion_set_hostname(o,'0.0.0.0');
    onion_url *urls-onion_root_url(o);

    onion_url_add_static(urls, 'static', 'Hello static world', HTTP_OK);
    onion_url_add(urls, 'timeout', random_timeout);
    onion_url_add(urls, 'ping', hello);

    onion_listen(o);
    onion_free(o);
    return 0;
}
```

We did many things in the function. Let us look at the points here:

- We provided signal parameters to the program with which server should be shut down. This is important to do to provide a way to gracefully shut down the web server without any unintended side-effects.
- Next, we check if the installed binaries on the platform are compatible with the Onion library version we are using in the program. If not, the program will abort.
- We then set some connection parameters like a timeout of 5 seconds (5000 milliseconds) and a hostname of 0.0.0.0 as well, which will allow HTTP calls to be done from anywhere to the server running this.
- We finally added some URLs. The first endpoint is a static endpoint with no callback method associated and it just returns a String and a 200 OK status code in response headers.
- The *timeout* endpoint is a GET endpoint, which when called will make a call to the callback function, *random_timeout* .
- The thirst URL we used is *ping* which will make a call to the callback function, *hello* .

- We finally started Onion to start listening using the connection parameters we defined.

Once you have Onion library setup, this program can be simply run using the command:

c server.c

Once you hit http: //localhost:8080/static , we will see the static endpoint being called. Just like that, we can call other endpoints as well.

Conclusion

Onion is a very simple C library which can be used to make a simple, yet powerful Web Server with which you can server real-time and original web content for your web application. If you want to use it in a production environment, we will suggest to look for alternatives as well because of the sophistication of other programming languages and frameworks which are primarily defined and designed to server as Web Servers.

Even though there exist more frameworks which can be used to make much more sophisticated web services, C is still a powerful language for small projects and for quick setup.

Appendix V

C99/C11 Features

C99 Features

INTRODUCTION

C, as developed and standardized by ANSI and ISO, is a powerful, flexible, portable, and elegant language. Due to its suitability for both systems and applications programming, it has become an industry-standard, general-purpose language to-day.

The standardization committee working on C language has been trying to examine each element of the language critically and see any change or enhancement is necessary in order to continue to maintain its lead over other competing languages. The committee also interacted with many user groups and elicited suggestions on improvements that are required from the point-of-view of users. The result was the new version of C, called C99.

The C99 standard incorporates enhancements and new features that are desirable for any modern computer language. Although it has borrowed some features from C++ (a progeny of C) and modified a few constructs, it retains almost all the features of ANSI C and thus continues to be a true C language.

In this appendix, we will highlight the important changes and new features added to C by the 1999 standard.

KEYWORDS

ANSI C has defined 32 keywords. C99 has added five more keywords. They are as follows:

_Bool

_Complex

_Imaginary

inline

restrict

Addition of these keywords is perhaps the most significant feature of C99. The use of these keywords are highlighted later in this appendix.

COMMENT

C99 adds what is known as the single-line comment, a feature borrowed from C++. Single-line comments begin with // (two back slashes) and end at the end of the line. Examples:

```
                                  // A comment line
if (x > y)                        // Testing
        printf(..........);       // Printing
int m;                            // Declaration
```

Single-line comments are useful when brief, line-by-line comments are needed.

DATA TYPES

C defines five basic data types, namely, **char, int, float, double,** and **void.** C99 adds three new built-in data types. They are as follows:

_Bool

_Complex

_Imaginary

C99 also allows **long** to modify **long** thus creating two more modified data types, namely, **long long int** and **unsigned long long int.**

_Bool Type

_Bool is an integer type which can held the values 1 and 0. Example:

```
_Bool x, y;
x = 1;
y = 0;
```

We know that relational and logical expressions return 0 for false and 1 for true. These values can be stored in _Bool type variables. For example,

```
_Bool b = m > n;
```

The variable b is assigned 1 if m is greater than n, otherwise 0.

_Complex and _Imaginary Types

C99 adds two keywords_ **Complex** and_ **Imaginary** to provide support for complex arithmetic that is necessary for numerical programming. The following complex types are supported:

float_Complex	float_Imaginary
double_Complex	double_Imaginary
long double_Complex	long double_Imaginary

The long long Types

The **long long int** has range of at least $-(2^{63} - 1)$ to $2^{63} - 1$. Similarly, the **unsigned long long int** has a range of 0 to $2^{64} - 1$.

DECLARATION OF VARIABLES

In C, we know that all the variables must be declared at the beginning of a block or function before any executable statements. However, C99 allow us to declare a variable at any point, just before its use. For example, the following code is legal in C99.

```
main()
{
        int m;
        m = 100;
        . . . . . . .
        . . . . . . .
        int n;                  /* Legal in C99*/
        n = 200;
        . . . .
}
```

C99 extends this concept to the declaration of control variables in **for** loops. That is, C99 permits declaration of one or more variables within the initialization part of the loop. For example, the following code is legal.

```
main()
{
        . . . . . . .
        . . . . . . .
        for (int i = 0; i<5; i + +)
        {
                . . . . . . .
                . . . . . . .
        }
        . . . . . . .
        . . . . . . .
}
```

A variable declared inside a **for** loop is local to that loop only. The value of the variable is lost, once the loop ends. (This concept is again borrowed from C++.)

I/O FORMATS

In order to handle the new data types with **long long** specification, C99 adds a new format modifier **ll** to both **scanf( )** and **printf( )** format specifications. Examples: **%lld, %llu, %lli, %llo, and %llx.**

Similarly, C99 adds **hh** modifier to d, i, o, u, and x specifications when handling **char** type values.

Handling of Arrays

C99 introduces some features that enhance the implementation of arrays.

Variable-Length Arrays

In ANSI C, we must declare array dimensions using integer constants and therefore the size of an array is fixed at compile time. C99 permits declaration of array dimensions using integer variables or any valid integer expressions. The values of these variables can be specified just before they are used. Such arrays are called *variable-length arrays.*

Example:

```
main()
{
      int m, n;
      scanf("%d %d", &m, &n);
      float matrix [ m ] [ n ];     /* variable-length array */
      . . . . . . .
      . . . . . . .
}
```

We can specify the values of m and n at run time interactively thus creating the matrix with different size each time the program is run.

Type Specification in Array Declaration

When we pass arrays as function arguments, we can qualify the dimension parameters with the keyword **static** . For example:

```
void array (int x [ static 20 ])
{
        . . . . . . .
        . . . . . . .
}
```

The qualifier **static** guarantees that the array **x** contains at least the specified number of elements.

Flexible Arrays in Structures

When designing structures, C99 permits declaration of an array without specifying any size as the last member. This is referred to as a **flexible array member** . Example:

struct find

```
{
        float x;
        int number;
        float list [  ];        /* flexible array */
};
```

FUNCTIONS IMPLEMENTATION

C99 has introduced some changes in the implementation of functions. They include the following:

- Removal of "default to **int** " rule
- Removal of "implicit function declaration"
- Restrictions on **return** statement
- Making functions **inline**

Default to int Rule

In ANSI C, when the return type of a function is not specified, the return type is assumed to be **int** . For example:

```
prod(int a, int b)        /* return type is int by default */
{
        return (a*b);
}
```

is a valid definition. The return type is assumed to be **int** by default. The implicit **int** rule is not valid in C99. It requires an explicit mention of return type, even if the function returns an integer value. The above definition must be written as:

```
int prod(int a, int b)     /* explicit type specification */
{
        return (a*b);
}
```

Another place where we use implicit **int** rule is when we declare function parameters using qualifiers. For example, function definitions such as

```
fun1( const a)            /* a is int by default */
{
        . . . . . . .
}
```

and

```
fun2 (register x, register y)     /* x and y are int */
{
        . . . . . . .
}
```

are not acceptable in C99. The parameters a, x and y must be explicitly declared as **int** , like:

const int a

const register x

Explicit Function Declaration

Although prior explicit declaration of function is not technically required in ANSI C, it is required in C99 (like in C++).

Restrictions on return Statement

In ANSI C, a non-void type function can include a **return** statement without including a value. For example, the following code is valid in ANSI C.

```
float value (float x, float y)
{
        . . . . . . .
        . . . . . . .
   return;                /* no value included */
}
```

But, in C99, if a function is specified as returning a value, its return statement must have a return value specified with it. Therefore, the above definition is not valid in C99. The **return** statement for the above function may take one of the following forms:

```
return(p);     /* p contains float value */
return(p);
return o.o;    /* when no value to be returned*/
```

Making Functions inline

The new keyword **inline** is used to optimize the function calls when a program is executed. The **inline** specifier is used in function definition as follows:

```
inline mul (int x, int y)
{
        return (x*y);
}
```

Such functions are called *inline functions.* When an inline function is invoked, the function's code is expanded inline, rather than called. This eliminates a significant amount of overhead that is required by the calling and returning mechanisms thus reducing the execution time considerably. However, the expansion "inline" may increase the size of the object code of the program when the function is invoked many times. Due to this, only small functions are made inline.

RESTRICTED POINTERS

The new keyword **restrict** has been introduced by C99 as a type qualifier that is applied only to pointers. A pointer qualified with **restrict** is referred to as a *restricted pointer* . Restricted pointers are declared as follows.

```
int *restrict p1;
void *restrict p2;
```

A pointer declared "restricted' is the only means of accessing the object it points to. (However, another pointer derived from the restricted pointer can also be used to access the object.)

Pointers with **restrict** specifier are mainly used as function parameters. They are also used as pointers that point to memory created by **malloc ( )** function.

C99 has added this feature to the prototype of many library functions, both existing and new. For details, you must refer to the functions defined in the C standard library.

COMPILER LIMITATIONS

All language compilers have limitations in terms of handling some features such as the length of significant characters, number of arguments in functions, etc. C99 has enhanced many of such limitations. They are listed below:

- Significant characters in identifiers: increased from 6 to 31
- Levels of nesting of blocks : Increased from 15 to 127
- Arguments in a function : Increased from 31 to 127
- Members in a structure : Increased from 127 to 1023

C11 Features

INTRODUCTION

C11 is the current C standard that replaces the previous standard i.e. C99. C11 mainly aims at standardizing features that are common to most of the present-day compilers. To overcome one of the limitations of C99 where there was a delay in compilers conforming to the C99 standard, C11 makes several of its features optional. This makes it easier for the compilers to conform to the C11 standard.

Language and Library Specification Changes

Let's now take a look at some of the key language and library specification changes that C11 brings to the C99 standard:

_Noreturn Function Specifier

It is used in the function declaration statement to indicate that the function does not return either by a return statement or by reaching the end of the function body. The specifier is provided in the stdnoreturn.h header file.

Some of the standard library functions that are declared using noreturn specifier include abort(), exit(), longjmp(), etc.

_Generic Keyword

It provides a method for choosing one of the given expressions at the time of compilation.

Syntax: _Generic (controlling-expression, association-list)

Here, controlling-expression is any expression, except the comma operator, and association-list is a comma-separated list of associations.

Thread local Specifier

The variables declared with _Thread_ local specifier are given thread-specific storage duration. That is, the variables are allocated when the thread begins and deallocated when the thread ends. Further, conforming to the thread-specific behavior, each thread has its own instance of the variable. The new header file <threads.h> contains _Thread_local specifier.

Enhanced Unicode Support

C11 supports certain enhancements related to Unicode support. For instance,

- New header file uchar.h comprising of UTF-16 and UTF-32 related character utilities
- u and U string literal prefixes
- u8 prefix for UTF-8 encoded literals

Anonymous Structures and Unions

C11 supports implementation of anonymous structures and unions, which are specifically useful in situations where unions and structures are nested. The following example code illustrates this:

```
struct S
{
 char c;
 union
 {
  int a; float b;
 };
};
```

Note that in the above code union is anonymous, as it has not been named.

ADDITION OF quick_exit FUNCTION

It is a new function for quick program termination. It terminates the program without cleaning the system resources.

REPLACEMENT OF gets WITH gets_s

The gets() function has been completely removed with C11 as it does not perform bounds checking resulting in buffer overflow situations. The new gets_s() function overcomes this limitation by reading at most characters from stdin. It keeps reading until a new line or end-of-file is reached and returns only a part of the read characters to the input buffer.

Alignment Specification

C11 provides four new macros for alignment of objects. These are:

_alignas

_alignof

_alignas_is_defined

_alignof_is_defined.

These macros are provided in the new stdalign.h header.

> **Note** *For a complete list of C11 standard features, you may refer the ISO/IEC 9899:2011 specification.*

Appendix VI

Graphics Programing Using C

INTRODUCTION

C is mostly favored by programmers for systems programming and developing embedded and real-time applications. However, C also possesses graphics programming capabilities. The header file *graphics.h* comprises of functions that can be used to develop graphical shapes, animations, etc.

In this appendix, we will start with graphics programming using C.

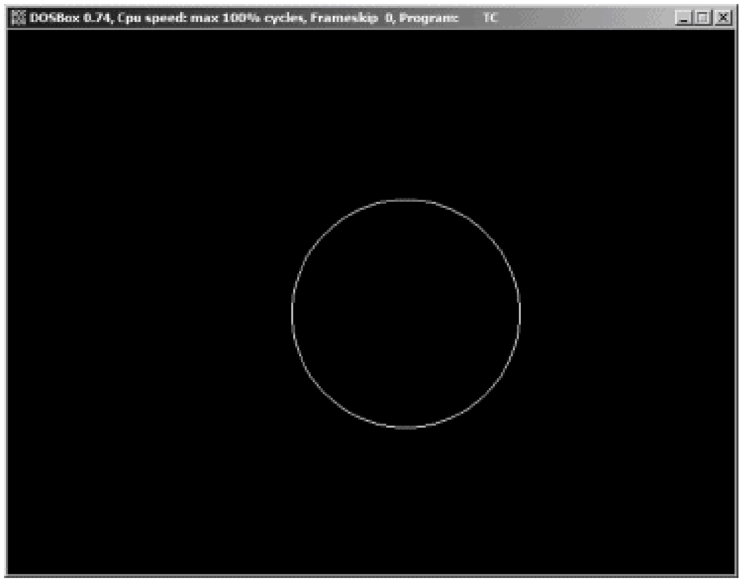
SAMPLE PROGRAM 1: DRAWING A CIRCLE

Consider a simple graphics program given in Figure VI.1.

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

void main()
{
   int gd-DETECT, gm;
   initgraph(&gd, &gm, 'C:\Turbo\TC\BGI' );

   circle(350,250,100);

   getch();
   closegraph();
}
```

Fig. VI.1 A program to draw a circle

The program when executed produces the following output:



Let us analyze the program instructions to see how the above output was produced. The program begins with file inclusion directives for including the following two header files:

- **graphics.h** : It comprises of graphics library functions which are used for drawing graphics on the output screen.
- **conio.h** : It comprises of functions for performing console input/output operations.

The main function begins with the following instructions for initializing the graphics mode:

```
int gd=DETECT, gm;
initgraph(&gd, &gm, "C:\Turbo\TC\BGI" );
```

Here, *gd* and *gm* are declared as integer variables representing graphics driver and graphics mode respectively. *gd* is assigned *DETECT* value, which is a macro defined in the *graphics.h* header file. The *initgraph* function is subsequently called to initialize the graphics i.e. choose an appropriate graphics driver and set the maximum screen resolution possible. The arguments of *initgraph* function include the following:

- Address of *gd* variable
- Address of *gm* variable

- Directory path of BGI files (the path may vary as per the location of C compiler)

Once the graphics driver and mode are set, the environment is ready for you to draw graphical shapes using functions in the graphics library. In the above program, *circle* function was called to draw a circle on the output screen. The first two arguments of the *circle* function are the x and y coordinates representing the center of the circle. The third argument represents the radius of the circle.

Once the graphical shape has been drawn, it is important to restore the graphics system. The *closegraph* function does just that by closing the graphics mode and restoring it to the mode it was when *initgraph* function was called.

> **Note** *initgraph* and *closegraph* functions are used in all programs where graphical shapes are drawn by calling the graphics library functions.

GRAPHICS LIBRARY FUNCTIONS

Table VI.1 lists some of the key graphics functions included in header file *graphics.h* .

Table VI.1 Graphics functions

Function	Purpose	Format
arc	Draws an arc	*void arc(int x, int y, int stangle, int endangle, int radius);* – *x, y* are the coordinates of the center of the arc – *stangle* and *endangle* are the respective starting and ending angles of the arc – *radius* is the radius of the arc
bar	Draws a 2-dimensional, filled-in, rectangular bar	*void bar(int left, int top, int right, int bottom);* – *left* is the X coordinate of the top left corner of the bar – *top* is the Y coordinate of the top left corner of the bar – *right* is the X coordinate of the bottom right corner of the bar – *bottom* is the Y coordinate of the bottom right corner of the bar
line	Draws a straight line	*void line(int x1, int y1, int x2, int y2);* – *x1, y1* are the coordinates of the starting point of the line – *x2, y2* are the coordinates of the ending point of the line
rectangle	Draws a rectangle	void rectangle(int left, int top, int right, int bottom); – *left* is the X coordinate of the top left corner of the rectangle – *top* is the Y coordinate of the top left corner of the rectangle – *right* is the X coordinate of the bottom right corner of the rectangle – *bottom* is the Y coordinate of the bottom right corner of the rectangle
getcolor	Returns integer code of current drawing color	*int getcolor();*
setcolor	Changes the current drawing color to the specified color code	*void setcolor(int color);* – *color* is the integer value representing color code
getbkcolor	Returns integer code of current background color	*int getbkcolor();*
setbkcolor	Changes the current background color to the specified color code	*void setcolor(int color);* – *color* is the integer value representing color code
getx	Returns the X coordinate of current cursor position	*int getx();*
gety	Returns the Y coordinate of current cursor position	*int gety();*
getmaxx	Returns the maximum X coordinate value for the current graphics mode and driver	*int getmaxx();*
getmaxy	Returns the maximum Y coordinate value for the current graphics mode and driver	*int getmaxy();*
getpixel	Returns the color code of the pixel present at specific coordinate location	*int getpixel(int x, int y);* – *x, y* are the coordinate values
putpixel	Sets the color code of the pixel present at specific coordinate	*int putpixel(int x, int y, int color);* – *x, y* are the coordinate values

	location	– *color* is the color code for the pixel

EXAMPLE PROGRAMS

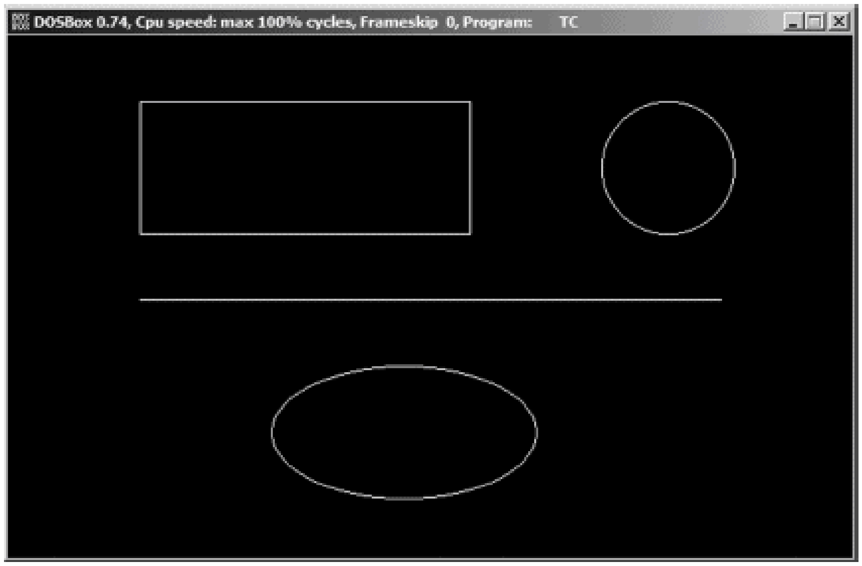
Program for Drawing Basic Shapes

Figure VI.2 shows the program for drawing basic shapes such as line, rectangle, circle, etc.

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

void main()
{
   int gd=DETECT, gm;
   initgraph(&gd, &gm, "C:\Turbo\TC\BGI" );

   circle(500,100,50);
   rectangle(100,50,350,150);
   ellipse(300, 300,0,360, 100,50);
   line(100,200,540,200);

   getch();
   closegraph();
}
```

Fig. VI.2 Program for drawing basic shapes

The program when executed produces the following output:



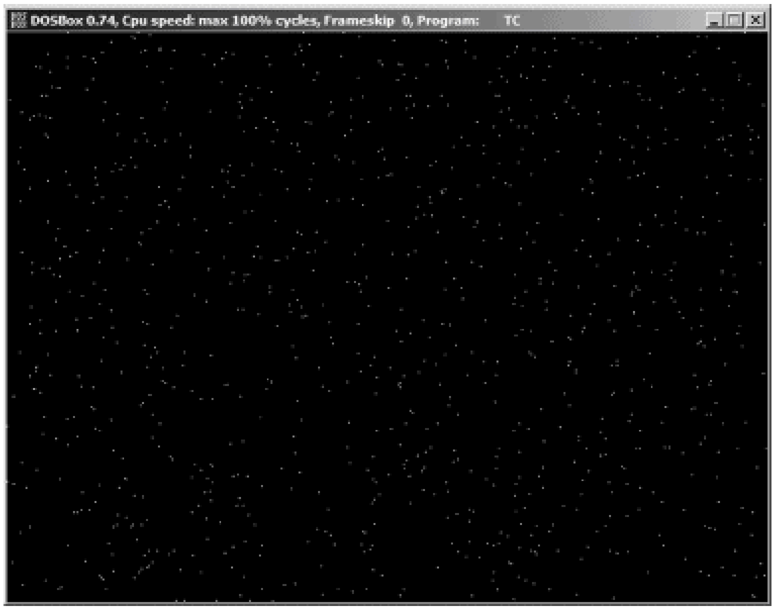
Program for Generating Graphics Effects

Figure VI.3 shows the program for depicting space filled with stars. The program uses *random()* function (part of *stdlib.h* header file) for generating random numbers.

```
#include<graphics.h>
#include<conio.h>
#include<stdlib.h>

void main()
{
   int x,y;
   int i=0;
   int gd=DETECT, gm;
   initgraph(&gd, &gm, "C:\Turbo\TC\BGI" );

   while(i<1000)
   {
   x=rand()%644;
   y=rand()%479;
   putpixel(x,y,15);
   setcolor(random(16));
   i++;
   }

   getch();
   closegraph();
}
```

Fig. VI.3 Program for depicting space filled with stars

The program when executed produces the following output:



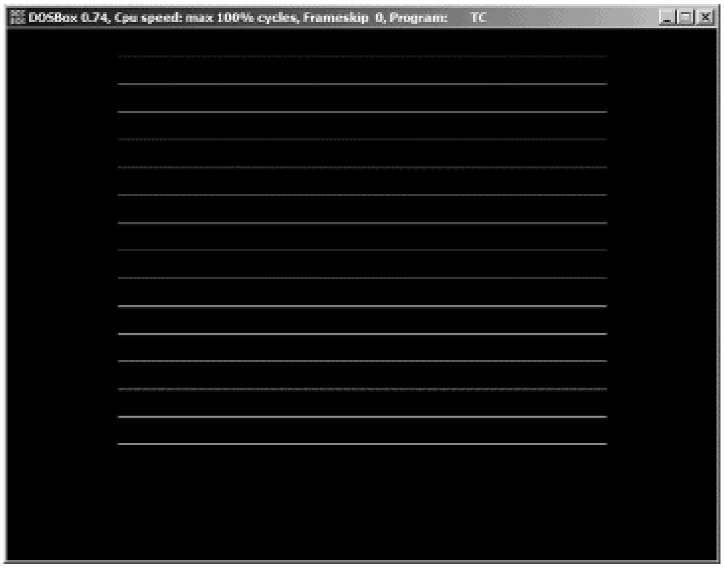
Program for Drawing Lines of Different Colors

Figure VI.4 shows the program for drawing straight lines in different colors.

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

void main()
{
   int gd=DETECT, gm;
   int i,x;
   initgraph(&gd, &gm, "");

   x=0;
   for(i=0;i<=15;i++)
   {
   setcolor(i);
   line(100,x,540,x);
   x=x+25;
   }

   getch();
   closegraph();
}
```

Fig. VI.4 Program for drawing lines in different colors

The program when executed produces the following output:



Bibliography

Barkakati, N., *Microsoft C Bible* , 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-Weslary, 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.

Schildt, H., C *: The Complete Reference* , McGraw-Hill, 2000.

Tim Grady, M., *Turbo C! Programming Principles and Practices* , McGraw-Hill 1989.

WIS Staff, *C user's Handbook* , Addison-Wesely, 1984.

Wortman, L. A., and Sidebottom, T.O., *The C Programming Tutor* , Prentice-Hall, 1984.

Index

Symbols

A

B

E

F

J

L

M

N

O

P

Q

R

S

T

U

V

W