Basic Computer Engineering

(RGPV 2011)

BE 205

Bachelor of Engineering B.E. (Common to all Disciplines)

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*
- *Programming in ANSI C*, 5e
- *Programming in Java*, 4e
- *Object-Oriented Programming with C++,* 5e
- *Programming in BASIC*, 3e
- *Programming in C#,* 3e
- *Numerical Methods*
- *Reliability Engineering*

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

BASIC COMPUTER ENGINEERING

(RGPV 2011)

BE 205

Bachelor of Engineering B.E. (Common to all Disciplines)

E Balagurusamy

Chairman
EBG Foundation
Coimbatore

Tata McGraw Hill Education Private Limited
NEW DELHI

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

Published by the Tata McGraw Hill Education Private Limited,
7 West Patel Nagar, New Delhi 110 008.

Basic Computer Engineering, (For RGPV), 2e

This edition can be exported from India only by the publishers,
Tata McGraw Hill Education Private Limited

ISBN-13: 978-0-07-13-2976-7
ISBN-10: 0-07-13-2976-5

Vice President and Managing Director—McGraw-Hill Education, Asia-Pacific Region: *Ajay Shukla*
Head—Higher Education Publishing and Marketing: *Vibha Mahajan*

Publishing Manager—SEM & Tech. Ed.: Shalini Jha
Asst Sponsoring Editor: *Tina Jajoriya*
Sr Editorial Researcher: *Manish Choudhary*
Executive—Editorial Services: *Sohini Mukherjee*
Sr Production Manager: *Satinder S. Baveja*
Proof Reader: *Yukti Sharma*

Sr Product Specialist—SEM & Tech. Ed.: *John Mathews*

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

Typeset at The Composers, 260, C.A. Apt., Paschim Vihar, New Delhi 110 063 and printed at Gopsons, A-2 & 3, Sector-64, Noida, U.P. 201 301

Cover Printer: Gopsons

RBLBCRAGRCYRC

The McGraw-Hill Companies

Contents

Preface

The developments in digital electronics and related technologies during the last few decades have ushered in the second Industrial Revolution, popularly referred to as the Information Revolution. Computer technology plays an ever-increasing role in this new revolution. Application of computers is all-pervasive in the life of every human today. A sound knowledge of how computers work and how they process data and information has, therefore, become indispensable for anyone who seeks employment not only in the area of IT but also in any other field.

Rightly so, many institutions and universities in India have introduced a subject covering the fundamentals of computers and programming in C++ at the undergraduate and diploma levels of arts, science and engineering disciplines. This book is designed for the students taking the first-year paper (BE 205) common to all B.E. courses in Rajiv Gandhi Proudyogiki Vishwavidyalaya(RGPV).

Why C++ Language?

Object-Oriented Programming (OOP) has become the preferred programming approach by most software industries, as it offers a powerful way to cope with the complexity of real-world problems. Among the OOP languages available today, C++ is by far the most widely used language.

Since its creation by Bjarne Stroustrup in the early 1980s, C++ has undergone many changes and improvements. The language was standardized in 1998 by the American National Standards Institute (ANSI) and the International Standards Organization (ISO) by incorporating not only many new features but also the changes suggested by user groups. This edition confirms to the specifications of ANSI/ISO standards.

Highlights of the book

The book is for the programmer who wishes to know all about computer fundamentals, the C++ language and object-oriented programming. It explains in a simple and easy-to-understand style the what, why and how of object-oriented programming with C++.

The highlight of the edition is the inclusion of two programming projects in Appendix B: (1) Menu-Based Calculation System, and (2) Banking System that demonstrates how to integrate the various features of C++ in real-life applications. Solution of the latest RGPV question paper B.E. First Sem (Dec. 2010) is given at the end of the book.

Salient Features

- **Comprehensively in sync** with the latest syllabus of RGPV
- **Bottom-up approach** of explaining concepts
- **Precise theory** presented in lucid language
- **Codes with comments** are provided throughout the book to illustrate the use of various features of the language
- **Supplementary information and notes** that complement but stand apart from the text have been included in special boxes
- **Key Terms** and **Summary** are given at the end of each chapter for quick recap of the concepts
- **Review Questions** provide ample opportunities to test the conceptual understanding of features
- **Programming and Debugging Exercises** stimulate interest to practice programming applications
- **Fill in the Blanks** and **Multiple Choice Questions** serve as effective assessment techniques
- **Appendices** for Major and Minor Projects, Operator Precedence in C++ will give an insight on how to integrate the various features of C++ in real-life problems

Chapter Organisation

The book has been divided into 14 chapters. **Chapter 1** gives an overview of computers, their development, characteristics and evolution. Next, operating systems are introduced in **Chapter 2. Chapter 3** explains programming languages, their history, some popular programming languages and the methods to develop and run a program. The theory of object-oriented programming and C++ are introduced in **chapters 4 and 5** respectively.

Tokens, instructions and control structures are taken up in **Chapter 5,** while functions in C++, classes, objects, constructors and destructors are covered subsequently in **chapters 6 to 9**. Operator overloading and type conversions are presented in **Chapter 10. Chapter 11** deals with inheritance and extending classes. Managing console I/O operations forms the subject of discussion in **Chapter 12,** and database management system of **Chapter 13** respectively. Finally, computer networking is covered in **Chapter 14.**

Two **appendices** are present at the end of the book. The first appendix discusses C++ operator precedence. The second appendix contains various projects including a menu-based calculation system and a banking system.

The book provides numerous examples, illustrations and complete programs. The sample programs are meant to be both simple and educational. Wherever necessary, pictorial descriptions of concepts are included to improve clarity and facilitate better understanding.

The book also presents the concept of oriented approach and discusses briefly the important elements of object-oriented analysis and design of systems.

Web Supplement

The following additional information is available on the web at:
http://www.mhhe.com/balagurusamy/bce/rgpv11

- Solution to the *Debugging Exercises*
- Chapter-wise *self-test quiz* with answers
- Complete code with step-by-step description and user manual for *Payroll Management Systems* (*Major Project*) and *Hospital Management Systems* (*Minor Project*).
- Differences between ANSI C, C++ and ANSI/ISO C++

Feedback

I welcome any constructive criticism of the book and will be grateful for any appraisal by the readers. Feedback to improve the book will be highly appreciated.

— E Balagurusamy

Publisher's Note

Tata McGraw Hill Education looks forward to receiving from teachers and students their valuable views, comments and suggestions for improvements, all of which may be sent to tmh.corefeedback@gmail.com, mentioning the title and author's name in the subject line. Also, please feel free to report any piracy of the book spotted by you.

Roadmap to the Syllabus

RAJIV GANDHI PROUDYOGIKI VISWAVIDYALAYA, BHOPAL
BASIC COMPUTER ENGINEERING
(BE 205)

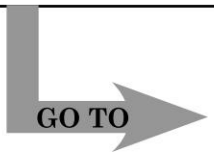
I year B.E.

L	T	P
3	1	2

UNIT 1

Computer: Definition, Classification, Organization, i.e., CPU, Register, Bus Architecture, Instruction Set, Memory and Storage Systems, I/O Devices, System and Application Software

Computing Ethics, Computer Application in e-Business, Bio-Informatics, Healthcare, Remote Sensing and GIS, Meteorology and Climatology, Computer Gaming, Multimedia and Animation



CHAPTER 1 Fundamentals of Computers

UNIT 2

***Operating System*:** Definition, Function, Types, Management of File, Process and Memory

***Programming Languages*:** Generations, Characteristics and Categorization

***Introduction to Programming*:** Procedure Oriented Programming vs Object Oriented Programming, OOPS Features and Merits

GO TO

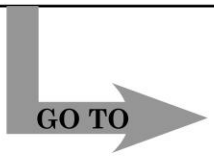
UNIT 3

***C++* :** Features Character, Tokens, Precedence and Associativity, Program Structure, Data Types, Variables, Operators, Expressions, Statements and Control Structures, I/O Operations, Array, Functions, Structures and Unions, Object and Classes, Constructors and Destructors, Overloading Functions and Operators, Derived Classes and Inheritance

GO TO

UNIT 4

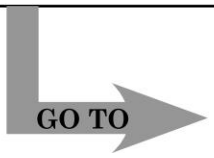
Database Management System: Introduction, File oriented approach and Database Approach, Data Models, Architecture of Database System, Data independence, Data Dictionary, DBA, Primary Key, Data Definition Language and Manipulation Languages



CHAPTER 13 Database Management Systems

UNIT 5

Computer Networking: Introduction, Goals, ISO-OSI Model, Functions of Different Layers, Internetworking Concepts, Devices, TCP/IP Model, Introduction to Internet, World Wide Web, Network Security and E-commerce



Chapter 14 Computer Networking

1

Fundamentals of Computers

Key Concepts

- Computer
- Computer Classification
- Computer Organization
- CPU
- Register
- Bus Architecture
- Instruction Set
- Memory and Storage Systems
- Input Devices
- System Software
- Application Software

1.1 Introduction

The term *computer* is derived from the word *compute*. A computer is an electronic device that takes data and instructions as an *input* from the user, processes data, and provides useful information known as *output*. This cycle of operation of a computer is known as the *input–process–output* cycle and is shown in Fig. 1.1. The electronic device is known as *hardware* and the set of instructions is known as *software*.

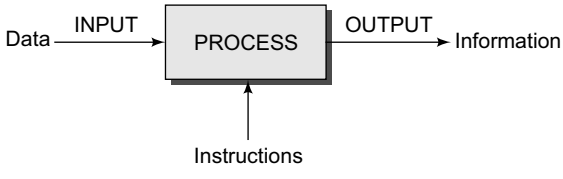


Fig. 1.1 ⇔ *Input–process–output concept*

The spurt of innovations and inventions in computer technology during the last few decades has led to the development of a variety of computers. They are so versatile that they have become indispensable to engineers, scientists, business executives, managers, administrators, accountants, teachers and students. They have strengthened man's powers in numerical computations and information processing.

Modern computers possess certain characteristics and abilities peculiar to them. They can:

(i) perform complex and repetitive calculations *rapidly* and *accurately,*
(ii) *store large amounts of data* and information for subsequent manipulations,
(iii) *hold a program of a model* which can be explored in many different ways,
(iv) *compare* items and make decisions,
(v) provide information to the user in many different forms,
(vi) *automatically correct* or modify the parameters of a system under control,
(vii) *draw and print graphs*,
(viii) *converse* with users *interactively*, and
(ix) receive and *display audio and video signals.*

These capabilities of computers have enabled us to use them for a variety of tasks. Application areas may broadly be classified into the following major categories.

1. Data processing (commercial use)
2. Numerical computing (scientific use)
3. Text (word) processing (office and educational use)
4. Message communication (e-mail)
5. Image processing (animation and industrial use)
6. Voice recognition (multimedia)

Engineers and scientists make use of the high-speed computing capability of computers to solve complex mathematical models and design problems. Many calculations that were previously beyond contemplation have now become possible. Many of the technological achievements such as landing on the moon would not have been possible without computers.

The areas of computer applications are too numerous to mention. Computers have become an integral part of man's everyday life. They continue to grow and open new horizons of discovery and application such as the electronic office, electronic commerce, and the home computer center.

The microelectronics revolution has placed enormous computational power within the reach of not only every organisation but also individual professionals and businessmen. However, it must be remembered that computers are machines created and managed by human beings. A computer has no brain of its own. Anything it does is the result of human instructions. It is an obedient slave which carries out the master's instructions as long as it can understand them, no matter whether they are right or wrong. A computer has no common sense.

1.2 Classification of Computers

Computers can be classified into several categories depending on their computing ability and processing speed. These include

- Microcomputer
- Minicomputer
- Mainframe computers
- Supercomputers

1.2.1 Microcomputers

A microcomputer is defined as a computer that has a microprocessor as its CPU. The microcomputer system can perform the following basic operations:

- **Inputting** — It is the process of entering data and instructions into the microcomputer system.
- **Storing** — It is the process of saving data and instructions in the memory of the microcomputer system, so that they can be used whenever required.
- **Processing** — It is the process of performing arithmetic or logical operations on data, where data can be converted into useful information. Various arithmetic operations include addition, subtraction, multiplication and division. Among logical operations, operations of comparisons like equal to, less than, greater than, etc., are prominent in use.
- **Outputting** — It provides the results to the user, which could be in the form of visual display and/or printed reports.
- **Controlling** — It helps in directing the sequence and manner in which all the above operations are performed.

1.2.2 Minicomputers

A minicomputer is a medium-sized computer that is more powerful than a microcomputer. An important distinction between a microcomputer and a minicomputer is that a minicomputer is usually designed to serve multiple users simultaneously. A system that supports multiple users is called a multiterminal, time-sharing system. Minicomputers are the popular computing systems among research and business organizations today. They are more expensive than microcomputers.

1.2.3 Mainframe Computers

Mainframe computers are those computers, which help in handling the information processing of various organizations like banks, insurance companies, hospitals and railways. Mainframe computers are placed on a central location and are connected to several user terminals, which can act as access stations and may be located in the same building. Mainframe computers are larger and expensive in comparison to the workstations.

1.2.4 Supercomputers

Supercomputers are the most powerful and expensive computers available at present. They are also the fastest computers available. Supercomputers are primarily used for complex scientific applications, which need a higher level of processing. Some of these applications include weather forecasting, climate research, molecular modeling used for chemical compounds, aeroplane simulations and nuclear fusion research.

In supercomputers, multiprocessing and parallel processing technologies are used to promptly solve complex problems. Here, the multiprocessor can enable the user to divide a complex problem into smaller problems. A supercomputer also supports multiprogramming where multiple users can access the computer simultaneously. Presently, some of the popular manufacturers of supercomputers are IBM, Silicon Graphics, Fujitsu, and Intel.

1.3 Computing Concepts

We can understand how a computer functions by analysing the fundamental computing concepts. The most elementary computing concepts include receiving input—known as *data*—from the user, manipulating the input according to the given set of instructions and delivering the output—known as *information*—to the user. Figure 1.2 shows the functioning of a computer based on these concepts.

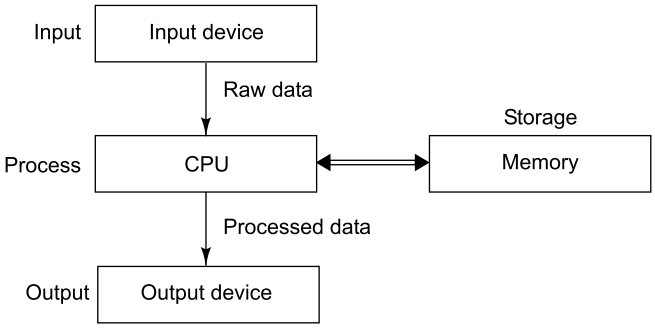


Fig. 1.2 ⇔ *The Input-Process-Output Cycle of a computer*

The various functions performed by the computer are briefly described below:

Accepting the raw data The first task to be performed by a computer is to accept the data from the user, with the help of an input device, such as mouse and keyboard. Mouse is used to enter the data through point-and-click operation while keyboard is used to enter the character data by typing the various keys. We need to enter the data into the computer so as to obtain the required result as output.

Processing the data The data is processed with the help of specific instructions known as *programs* after taking the input from the user. The manipulation of data is handled by the CPU of the computer. CPU is considered as the *brain of the computer* because it controls the execution of various instructions. The raw data entered by the user through input devices is processed by the CPU to generate meaningful information.

Storing the data The data is stored in the main memory of a computer in its processed form. The various external storage devices—such as hard disk and magnetic disk—can also be used for storing the processed data so that it can again be fetched later.

Delivering the output The processed data is delivered as useful information to the user with the help of output devices, such as printer and monitor.

1.4 Central Processing Unit

The function of any computer system revolves around a central component known as *central processing unit* (CPU). The CPU, which is popularly referred to as the "brain" of the computer, is responsible for processing the data inside the computer system. It is also responsible for controlling all other components of the system. Figure 1.3 shows a typical block diagram of the computer system, illustrating the arrangement of CPU with the input and output units as well as the memory of the computer system.

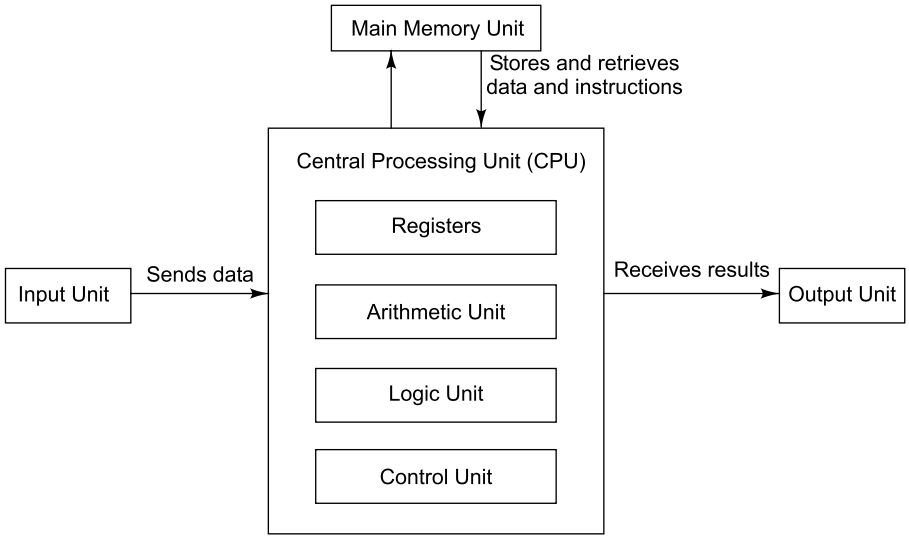


Fig. 1.3 ⇔ *The block diagram of a computer system*

The main operations of the CPU include four phases:

- Fetching instructions from the memory.
- Decoding the instructions to decide what operations to be preformed.
- Executing the instructions.
- Storing the results back in the memory.

This four-phase process is known as the *CPU cycle*, which is illustrated in Fig. 1.4.

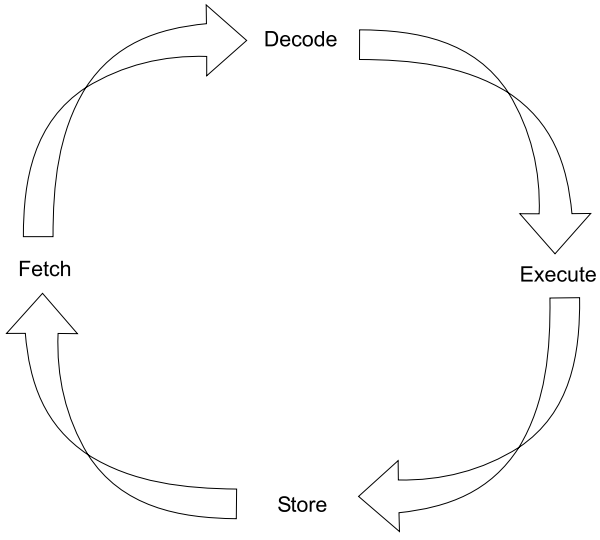


Fig. 1.4 ⇔ *The CPU cycle*

As shown in the Fig. 1.3, the central processing unit consists of the following subsystems:

- Arithmetic Unit (AU)
- Logic Unit (LU)
- Control Unit (CU)

The CPU makes use of the following memory subsystems for carrying out its processing operations:

- Main Memory Unit
- Cache Memory
- Registers

1.4.1 Arithmetic Unit

Arithmetic Unit (AU) is a part of the CPU that performs arithmetic operations on the data. The arithmetic operations can be addition, subtraction, multiplication or division. The multiplication and division operations are usually implemented by the AU as the repetitive process of addition and subtraction operations respectively. Some CPUs contain separate AUs for integer or fixed-point operations (integers) and real or floating-point operations (real/decimal). AU takes the input in the form of an instruction that contains an opcode, operands and the format code. The opcode specifies the operation to be performed and the operands specify the data on which operation is to be performed. The format code suggests the format of the operands, such as fixed-point or floating-point. The output of AU contains the result of the operation and the status of the result, whether it is final or not. The output is stored in a storage register by the AU. Register is a small storage area inside the CPU from where data is retrieved faster than any other storage area.

1.4.2 Logic Unit

Logic Unit (LU) is a part of the CPU that performs logical operations on the data. It performs 16 different types of logical operations. The various logical operations include greater than (>), less than (<), equal to (=), not equal to (≠), shift left, shift right, etc. LU makes use of various logic gates, such as AND, OR, NOR, etc. for performing the logical operations on the data.

1.4.3 Control Unit

Control Unit (CU) is an important component of CPU that controls the flow of data and information. It maintains the sequence of operations being performed by the CPU. It fetches an instruction from the storage area, decodes the instruction and transmits the corresponding signals to the AU or LU and the storage registers. CU guides the AU and LU about the operations that are to be performed and also suggests the I/O devices to which the data is to be communicated. CU uses a program counter register for retrieving the next instruction that is to be executed. It also uses a status register for handling conditions such as overflow of data.

1.4.4 Main Memory Unit

The *main memory* is referred to as the *internal memory* or *primary memory* of the computer. It is also known as Random Access Memory (RAM). It is a temporary storage medium that holds the data only for a short period of time. Once the computer is switched off, the data stored in the RAM gets erased. The memory space of RAM is limited and therefore all the files and instructions cannot be stored in it. These files and instructions are normally stored in a different location known as secondary storage and are copied from there to the RAM before execution. This technique is referred as *swapping*. The memory space available in RAM also affects the speed of a computer system. If the memory space is more, more number of instructions can be copied and executed at the same time. As a result, the computer system need not read the data from the secondary storage again and again, thus making the processing faster. The main memory is also responsible for holding intermediate data transferred between CPU and the I/O devices.

1.4.5 Cache Memory

Cache memory is a small, fast and expensive memory that stores the copies of data that needs to be accessed frequently from the main memory. The processor, before reading data from or writing data to the main memory, checks for the same data in the cache memory. If it finds the data in the cache memory the processor reads the data from or writes the data to the cache itself because its access time is much faster than the main memory. The cache memory is always placed between CPU and the main memory of the computer system, as shown in Fig. 1.5.

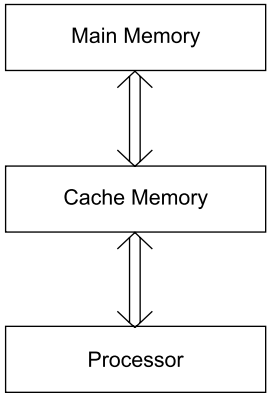


Fig. 1.5 ⇔ *The cache memory*

Figure 1.5 shows that the transfer of data between the processor and the cache memory is bidirectional. The availability of data in the cache is known as *cache hit*. The capability of a cache memory is measured on the basis of cache hit.

There are usually two types of cache memory found in the computer system:

- **Primary cache** It is also known as Level 1 (L1) cache or internal cache. The primary cache is located inside the CPU. It is smaller but fastest type of cache that provides a quick access to the frequently accessed data by the microprocessor.
- **Secondary cache** It is also known as Level 2 (L2) cache or external cache. The secondary cache is located outside the CPU. It is normally positioned on the motherboard of a computer. The secondary cache is larger but slower than the primary cache.

1.4.6 Registers

Central processing unit contains a few special purpose, temporary storage units known as *registers*. They are high-speed memory locations used for holding instructions, data and

intermediate results that are currently being processed. A processor can have different types of registers to hold different types of information. They include, among others:

- *Program Counter (PC)* to keep track of the next instruction to be executed.
- *Instruction Register (IR)* to hold instructions to be decoded by the control unit.
- *Memory Address Register (MAR)* to hold the address of the next location in the memory to be accessed.
- *Memory Buffer Register (MBR)* for storing data received from or sent to CPU.
- *Memory Data Register (MDR)* for storing operands and data.
- *Accumulator (ACC)* for storing the results produced by arithmetic and logic units.

Many computers employ additional registers for implementing various other requirements. The number and sizes of registers therefore vary from processor to processor. An effective implementation of registers can increase considerably the speed of the processor.

1.5 Internal Communications

CPU of the computer system communicates with the memory and the I/O devices in order to transfer data between them. However, the method of communication of the CPU with memory and I/O devices is different. The CPU may communicate with the memory either directly or through the cache memory. However, the communication between the CPU and I/O devices is usually implemented with the help of interfaces. Therefore, the internal communication of a processor in the computer system can be divided into two major categories:

- Processor to memory communication
- Processor to I/O devices communication

1.5.1 Processor to Memory Communication

The direct communication between the processor and memory of the computer system is implemented with the help of two registers, Memory Address Register (MAR) and Memory Buffer Register (MBR). Figure 1.6 shows the communication between the processor and the memory of the computer system.

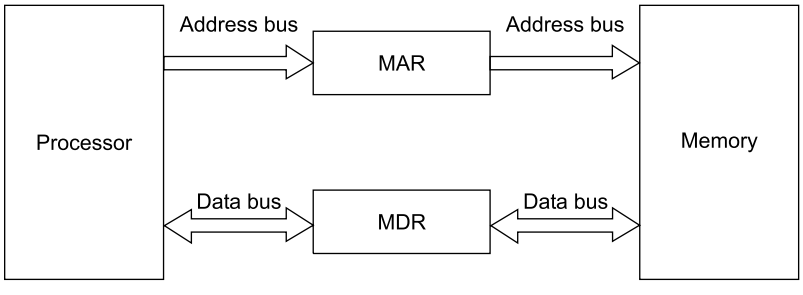


Fig. 1.6 ⇔ *Processor to memory communication*

The processor can interact with the memory of the computer system for reading data from the memory as well as for writing data on to the memory. The MAR and MBR registers play a very important role in implementing this type of communication. These registers are the special-purpose registers of the processor. MAR is used by the processor to keep track of the memory location where it needs to perform the reading or writing operation. This register actually holds the address of the memory location. On the other hand, the Memory Data Register (MDR) is used by the processor to store the data that needs to be transferred from/to the memory of the computer system. The reading and writing operations performed by the processor are called *memory read* and *memory write* operations.

Figure 1.7 illustrates the memory read operation performed by the processor of the computer system. The processor performs the following steps to read the data from the desired memory location:

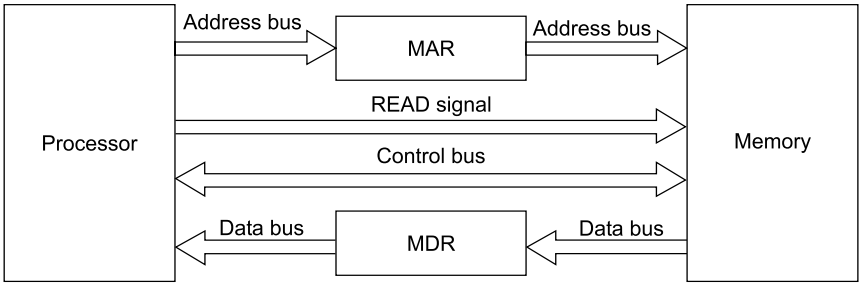


Fig. 1.7 ⇔ *Illustrating the memory read operation*

1. First, the processor loads the address of the memory location from where data is to be read into the MAR register, using the address bus.
2. After loading the address of the memory location, the processor issues the READ control signal through the control bus. The control bus is used to carry the commands issued by the processor, and the status signals are generated by the various devices in response to these commands.
3. After receiving the READ control signal, the memory loads the data into the MDR register from the location specified in the MAR register, using the data bus.
4. Finally, the data is transferred to the processor.

The memory write operation helps the processor to write the data at the desired memory location. Figure 1.8 illustrates the memory write operation performed by the processor of the computer system.

The processor performs the following steps for writing the data at the desired memory location in the computer system:

1. First, the processor loads the address of the memory location where data is to be written in the MAR register, using the address bus.
2. After loading the address of the memory location, the processor loads the desired data in the MDR register, using the data bus.
3. After this, the processor issues the WRITE control signal to the memory, using the control bus.

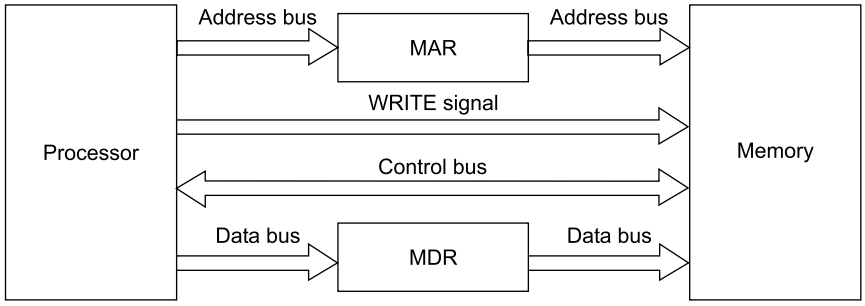


Fig. 1.8 ⇔ *Illustrating memory write operation*

4. Finally, the memory stores the data loaded in the MDR register at the desired memory location.

1.5.2 Processor to I/O Devices Communication

The communication between I/O devices and processor of the computer system is implemented using an *interface unit*. In a computer system, data is transferred from an input device to the processor and from the processor to an output device. Each input and output device in the computer system is provided with a controller, called *device controller*. The device controller is used to manage the working of various peripheral devices. The processor actually communicates with the device controllers of the various I/O devices for performing the I/O operations.

Figure 1.9 illustrates how the communication between the processor and the I/O devices of the computer system is implemented. The interface unit acts as an intermediary between the processor and the device controllers of various peripheral devices in the computer system. The basic function of the interface unit is to accept the control commands from the processor and interpret the commands so that they can be easily understood by the device controllers for carrying out necessary operations. Therefore, we can say that the interface unit is responsible for controlling the input and output operations between the processor and the I/O devices. The interface unit contains data register and status register. The data register is used to store the data to be transferred, either to the processor or to an output device. The status register is

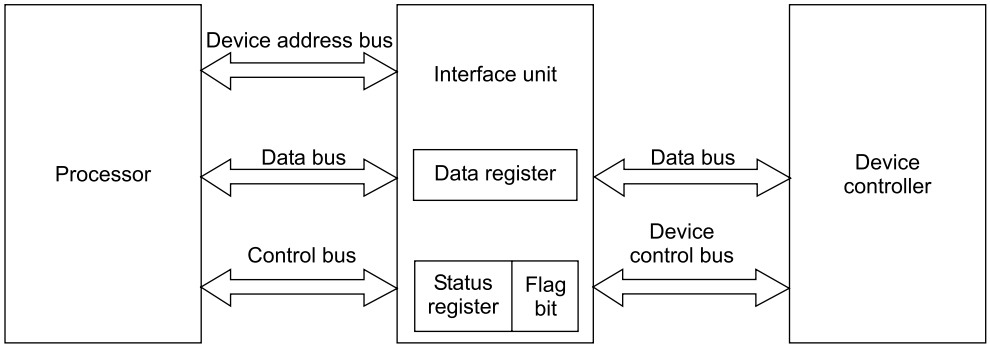


Fig. 1.9 ⇔ *Illustrating the communication process between the processor and I/O devices*

used to indicate the status of the data register, i.e., whether it is currently holding the data or not. If the data register is holding the data to be transferred, the flag bit of the status register is set to one. The processor to I/O devices communication involves two important operations, i.e., I/O read and I/O write. The I/O read operation helps the processor to read the data from an input device.

Figure 1.10 illustrates how the data is transferred from an input device to the processor of the computer system. The steps performed while transferring the data from an input device to the processor are:

1. The data to be transferred is placed on the data bus by the input device, which transfers single byte of data at a time.
2. The input device then issues the data valid signal through the device control bus to the data register, indicating that the data is available on the data bus.
3. When the data register of the interface unit accepts the data, it issues a data accepted signal through the device control bus as an acknowledgement to the input device, indicating that the data has been received. The input device then disables the data valid signal.
4. As the data register now holds the data, the F or the flag bit of the status register is set to 1.
5. The processor now issues an I/O read signal to the data register in the interface unit.
6. The data register then places the data on the data bus connected to the processor of the computer system. After receiving the data, the processor sends an appropriate acknowledgement signal to the input device, indicating that the data has been received.

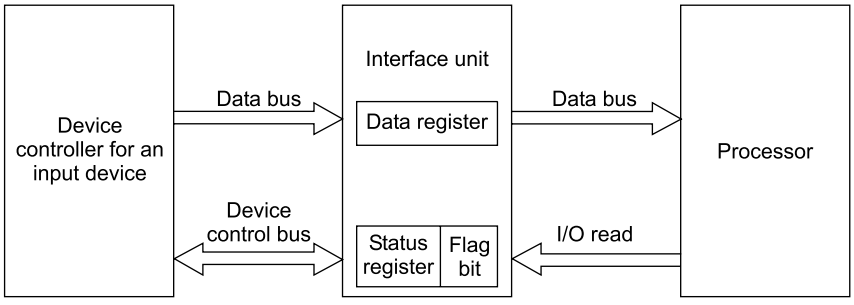


Fig. 1.10 ⇔ *Illustrating the I/O read operation*

The I/O write operation helps the processor to write the data to an output device. Figure 1.11 illustrates how the data is transferred from the processor to an output device. The steps performed while transferring the data from the processor to the output device are:

1. The processor places the data that needs to be transferred on the data bus connected to the data register of the interface unit.
2. The CPU also places the address of the output device on the device address bus.
3. After placing the address and data on the appropriate buses, CPU issues the I/O write signal, which writes the data on the data register. The flag bit in the interface unit is set to 1, indicating that the data register now holds the data.

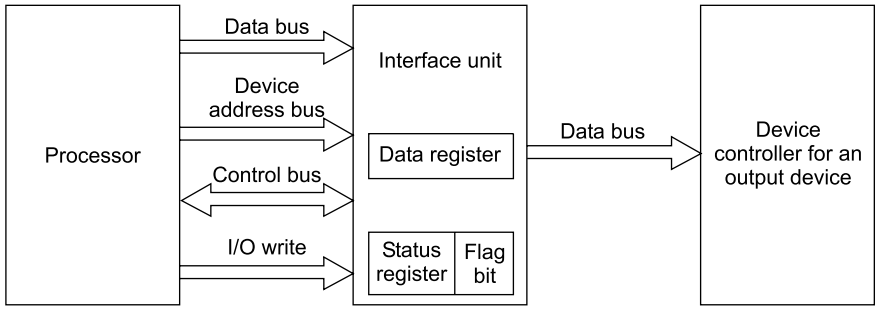


Fig. 1.11 ⇔ *Illustrating I/O write operation*

4. The data register of the interface unit issues a data accepted signal through the control bus to the processor, indicating that the data has been received.
5. The interface unit then places the data stored in the data register on to the data bus connected to the device controller of the output device.
6. The output device then receives the data and sends an acknowledgement signal to the processor of the computer system through the interface unit, indicating that the desired data has been received.

1.6 The Bus

A bus is a set of wires that is used to connect the different internal components of the computer system for the purpose of transferring data as well addresses amongst them. There may be several buses in a computer system. A bus can either be a serial bus or a parallel bus. In serial bus, only one bit of data is transferred at a time amongst the various hardware components. On the other hand, in parallel bus, several bits of data can be transferred at a time amongst the various hardware components. The speed of any type of bus is measured in terms of the number of bits transferred per second, between two components.

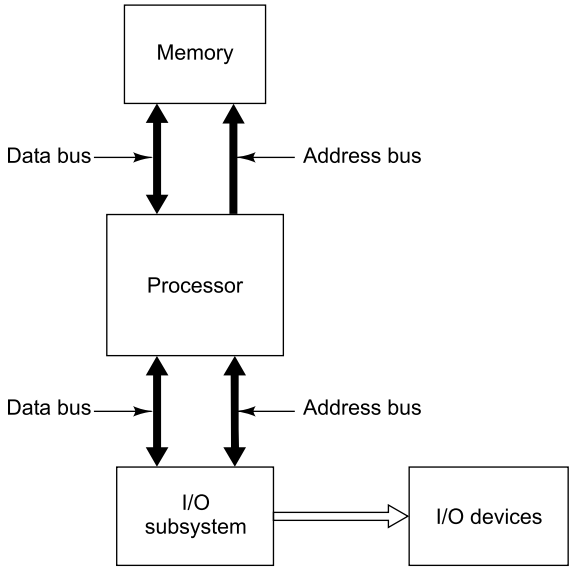


Fig. 1.12 ⇔ *Data and address buses*

Figure 1.12 shows a bus system used in a computer system. The figure depicts the two different types of buses according to the type of operations performed by them. These buses are *data bus* and the *address bus*. Apart from data and address bus, a third type of bus—known as *control bus*—also exists in the computer system. The control bus manages the transfer of data and addresses among various components by transferring appropriate control signals.

1.6.1 Data Bus

As the name suggests, the data bus in a computer system is used to transfer data amongst the different internal components. The speed of the data bus also affects the overall processing power of a computer system. Modern computer systems use 32-bit data buses for data transfer. This means that these buses are capable of transferring 32 bits of data at a time. Figure 1.13 shows the data bus implemented between the main memory and the processor of a computer system.

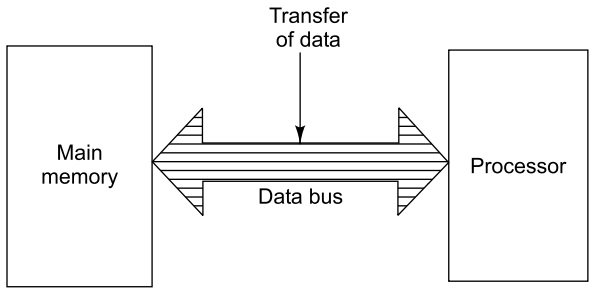


Fig. 1.13 ⇔ *The data bus*

The figure shows that a bidirectional data bus is implemented between the main memory and the processor of the computer system. The bidirectional data bus allows the transfer of data in both the directions. The data bus is generally bidirectional in nature in most computer systems.

1.6.2 Address Bus

The address bus is also known as memory bus. It transfers the memory addresses for read and write memory operations. It contains a number of address lines that determine the range of memory addresses that can be referenced using the address bus. For example, a 32-bit address bus can be used to reference 2^{32} memory locations. Like data bus, the address bus can also be a serial or a parallel bus. Figure 1.14 shows the address bus, used for transferring memory locations between processor and memory.

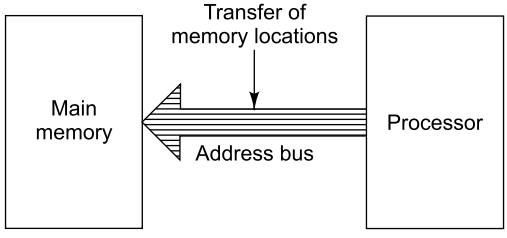


Fig. 1.14 ⇔ *An address bus*

The figure shows that the address bus between the main memory and the processor of a computer system is unidirectional. However, an address bus may also be bidirectional. For example, the address bus between the processor and the I/O system is bidirectional.

1.7 Instruction Set

An instruction set can be defined as a group of instructions that a processor can execute to perform different operations. On the basis of complexity and the number of instructions used, the instruction set can be classified as:

- Complex instruction set
- Reduced instruction set

1.7.1 Complex Instruction Set

The complex instruction set refers to the set of instructions that includes very complex and large number of instructions. The number of instructions in this set varies from 100 to 250. The instructions in this set are mostly memory-based instructions, which involve frequent references to the memory. The complex instruction set makes use of a large number of addressing modes because of the frequent references to registers as well as memory. The instructions in this instruction set have variable length instruction format, which is not limited to only 32-bits. The execution of the instructions takes a lot of time because the instructions are memory-based and accessing the memory is a time consuming process as compared to accessing the registers.

The computer, which makes use of complex instruction set, is called Complex Instruction Set Computer (CISC). The instruction set of CISC has a large number of instructions and for each instruction type, the computer requires a separate circuitry, which makes the CPU design more complicated.

Some of the advantages of CISC are as follows:

- There is no need to invent an instruction set for each new design. A new processor can use the instruction set of its predecessor.
- A program written in CISC requires less memory space, as the code is confined to less number of instructions.
- CISC makes the job of a compiler easier by facilitating the implementation of high-level language constructs.

Some of the disadvantages of a CISC are as follows:

- The inheritance of old instructions into new processors increases the complexity.
- Many CISC instructions are not frequently used.
- CISC commands are translated into a large number of lines of microcode, which makes the CPU processing slower.
- CISC systems have a complex hardware, so they require more time for designing.

1.7.2 Reduced Instruction Set

The reduced instruction set refers to a set of instructions that contains very few instructions ranging from 0 to 100. It comprises of only those instructions, that are frequently used by the processor for the execution of a program. These instructions are generally very simple to execute. The instructions used in this set are mostly register-based, which means that the execution of the instruction involves frequent references to the registers. The memory-based instructions, which involve frequent references to the memory locations, are very few in this instruction set. The memory-based instructions include only load and store instructions. The instructions in this instruction set have fixed length instruction format of 32 bits. An instruction format divides the bits of instruction into small groups called fields. Generally, an instruction has the following fields:

- **Opcode field.** It represents the operation to be performed by the instruction.
- **Operand field.** It represents the data on which the operation is to be performed, or the memory location or register where the data is stored.

- **Mode field.** It represents the method of fetching the operands stored at specified memory location or registers.

The computer, which makes use of reduced instruction set, is called Reduced Instruction Set Computer (RISC). As the instruction set of RISC has very few instructions, the design of hardware circuitry becomes easier and also the speed of processing increases. The speed of RISC processors is measured in MIPS (Millions of Instructions Per Second).

The comparison of RISC and CISC processors indicates that the RISC processors are always preferred over the CISC processors because of their compact size and small instruction set. The other advantages of the RISC processors over the CISC processors are as follows:

- In RISC processors, the instructions are executed by decoding, whereas in CISC processors, the instructions are executed by first translating them into equivalent microcode instructions. The conversion of instructions into microcode consumes a lot of space in the memory, thereby reducing the speed of execution.
- The RISC processors execute instructions in a single clock cycle, while the CISC processors require multiple clock cycles for the execution of an instruction.
- The hardware of the RISC processors is very simple and can be designed easily, as compared to the hardware of the CISC processors that is very complex, difficult to design and large in size.

The only disadvantage of RISC, in comparison to CISC, is that the number of instructions required to perform an operation is comparatively large.

1.8 Memory and Storage Systems

The memory unit of a computer is used to store data, instructions for processing data, intermediate results of processing and the final processed information. The memory units of a computer are classified as primary memory and secondary memory.

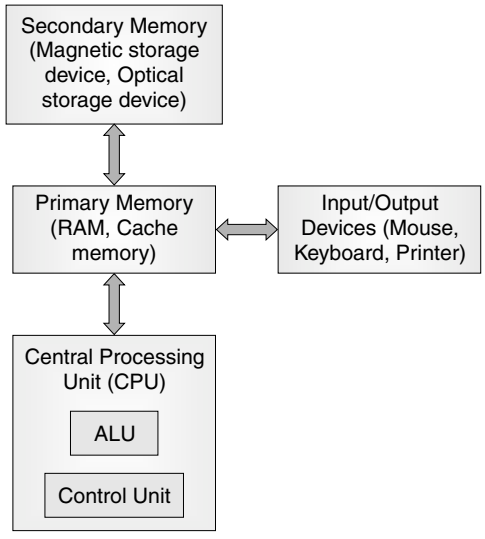


Fig. 1.15 ⇔ *Computer Memory Organization*

1.8.1 Primary Memory

The primary memory is available in the computer as a built-in unit of the computer. The primary memory is represented as a set of locations with each location occupying 8 bits. Each bit in the memory is identified by a unique address. The data is stored in the machine-understandable binary form in these memory locations. The commonly used primary memories are as follows:

- **ROM** — ROM represents Read Only Memory that stores data and instructions, even when the computer is turned off. It is the permanent memory of the computer where the contents cannot be modified by an end user. ROM is a chip that is inserted into the

motherboard. It is generally used to store the Basic Input/Output system (BIOS), which performs the Power On Self Test (POST).

- **RAM** — RAM is the read/write memory unit in which the information is retained only as long as there is a regular power supply. When the power supply is interrupted or switched off, the information stored in the RAM is lost. RAM is volatile memory that temporarily stores data and applications as long as they are in use. When the use of data or the application is over, the content in RAM is erased.
- **Cache memory** — Cache memory is used to store the data and the related application that was last processed by the CPU. When the processor performs processing, it first searches the cache memory and then the RAM, for an instruction. The cache memory can be either soldered into the motherboard or is available as a part of RAM.

1.8.2 Secondary Memory

Secondary memory represents the external storage devices that are connected to the computer. They provide a non-volatile memory source used to store information that is not in use currently. A storage device is either located in the CPU casing of the computer or is connected externally to the computer. The secondary storage devices can be classified as:

- **Magnetic storage device** — The magnetic storage devices store information that can be read, erased and rewritten a number of times. These include floppy disk, hard disk and magnetic tapes.
- **Optical storage device** — The optical storage devices are secondary storage devices that use laser beams to read the stored data. These include CD-ROM, rewritable compact disk (CD-RW), digital video disks with read only memory (DVD-ROM), etc.
- **Magneto-optical storage device** — The magneto-optical devices are generally used to store information, such as large programs, files and back up data. The end user can modify the information stored in magneto-optical storage devices multiple times. These devices provide higher storage capacity as they use laser beams and magnets for reading and writing data to the device.

1.9 Input Devices

Input devices can be connected to the computer system using cables. The most commonly used input devices among others are:

- Keyboard
- Mouse
- Scanner

1.9.1 Keyboard

A standard keyboard includes alphanumeric keys, function keys, modifier keys, cursor movement keys, spacebar, escape key, numeric keypad, and some special keys, such as Page Up, Page Down, Home, Insert, Delete and End. The alphanumeric keys include the number keys and the alphabet keys. The function keys are the keys that help perform a specific task such as searching a file or refreshing a Web page. The modifier keys such as Shift and Control

keys modify the casing style of a character or symbol. The cursor movement keys include up, down, left and right keys and are used to modify the direction of the cursor on the screen. The spacebar key shifts the cursor to the right by one position. The numeric keypad uses separate keypads for numbers and mathematical operators. A keyboard is shown in Fig. 1.16.

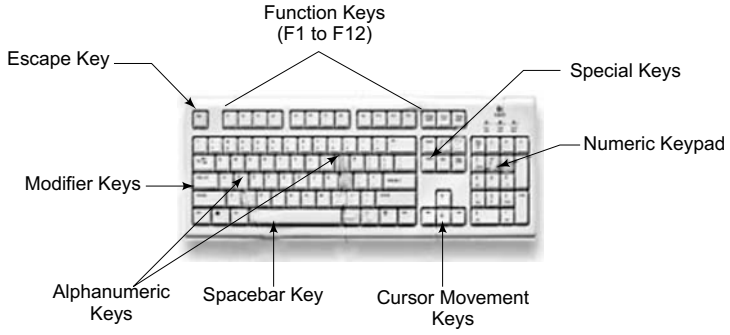


Fig. 1.16 ⇔ *Keyboard*

1.9.2 Mouse

The mouse allows the user to select elements on the screen, such as tools, icons, and buttons, by pointing and clicking them. We can also use a mouse to draw and paint on the screen of the computer system. The mouse is also known as a pointing device because it helps change the position of the pointer or cursor on the screen.

The mouse consists of two buttons, a wheel at the top and a ball at the bottom of the mouse. When the ball moves, the cursor on the screen moves in the direction in which the ball rotates. The left button of the mouse is used to select an element and the right button, when clicked, displays the special options such as **open** and **explore** and **shortcut** menus. The wheel is used to scroll down in a document or a Web page. A mouse is shown in Fig. 1.17.

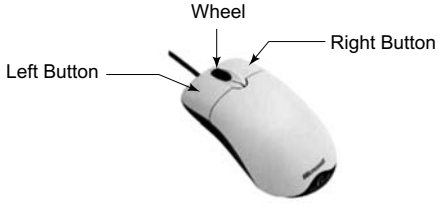


Fig. 1.17 ⇔ *Mouse*

1.9.3 Scanner

A scanner is an input device that converts documents and images as the digitized images understandable by the computer system. The digitized images can be produced as black and white images, gray images, or colored images. In case of colored images, an image is considered as a collection of dots with each dot representing a combination of red, green, and blue colors, varying in proportions. The proportions of red, green, and blue colors assigned to a dot are together called as *color description*. The scanner uses the color description of the dots to produce a digitized image. Figure 1.18 shows a scanner.

Fig. 1.18 ⇔ *Scanner*

There are the following types of scanners that can be used to produce digitized images:

- **Flatbed scanner** — It contains a scanner head that moves across a page from top to bottom to read the page and converts the image or text available on the page in digital form. The flatbed scanner is used to scan graphics, oversized documents, and pages from books.
- **Drum scanner** — In this type of scanner, a fixed scanner head is used and the image to be scanned is moved across the head. The drum scanners are used for scanning prepress materials.
- **Slide scanner** — It is a scanner that can scan photographic slides directly to produce files understandable by the computer.
- **Handheld scanner** — It is a scanner that is moved by the end user across the page to be scanned. This type of scanner is inexpensive and small in size.

1.10 Output Devices

The data, processed by the CPU, is made available to the end user by the output devices. The most commonly used output devices are:

- Monitor
- Printer
- Speaker
- Plotter

1.10.1 Monitor

A monitor is the most commonly used output device that produces visual displays generated by the computer. The monitor, also known as a screen, is connected as an external device using cables or connected either as a part of the CPU case. The monitor connected using cables, is connected to the video card placed on the expansion slot of the motherboard. The display device is used for visual presentation of textual and graphical information.

The monitors can be classified as cathode ray tube (CRT) monitors or liquid crystal display (LCD) monitors. The CRT monitors are large, occupy more space in the computer, whereas LCD monitors are thin, light weighted, and occupy lesser space. Both the monitors are available as monochrome, gray scale and color models. However, the quality of the visual display produced by the CRT is better than that produced by the LCD.

The inner side of the screen of the CRT contains the red, green, and blue phosphors. When a beam of electrons strike the screen, the beam strikes the red, green and blue phosphors on the screen and irradiates it to produce the image. The process repeats itself for a change in the image, thus refreshing the changing image. To change the color displayed by the monitor, the intensity of the beam striking the screen is varied. If the rate at which the screen gets refreshed is large, then the screen starts flickering, when the images are refreshed.

The LCD monitor is a thin display device that consists of a number of color or monochrome pixels arrayed in front of a light source or reflector. LCD monitors consume a very small amount of electric power.

A monitor can be characterized by its monitor size and resolution. The monitor size is the length of the screen that is measured diagonally. The resolution of the screen is expressed as the number of picture elements or pixels of the screen. The resolution of the monitor is also called the dot pitch. The monitor with a higher resolution produces a clearer image.

1.10.2 Printer

The printer is an output device that transfers the text displayed on the screen, onto paper sheets that can be used by the end user. The various types of printers used in the market are generally categorized as dot matrix printers, inkjet printers, and laser printers. Dot matrix printers are commonly used in low quality and high volume applications like invoice printing, cash registers, etc. However, inkjet printers are slower than dot matrix printers and generate high quality photographic prints. Since laser printers consist of microprocessor, ROM and RAM, they can produce high quality prints in quicker time without being connected to a computer.

The printer is an output device that is used to produce a hard copy of the electronic text displayed on the screen, in the form of paper sheets that can be used by the end user. The printer is an external device that is connected to the computer system using cables. The computer needs to convert the document that is to be printed to data that is understandable by the printer. The *printer driver software* or the *print driver software* is used to convert a document to a form understandable by the computer. When the computer components are upgraded, the upgraded printer driver software needs to be installed on the computer.

The performance of a printer is measured in terms of dots per inch *(DPI)* and *pages per minute (PPM)* produced by the printer. The greater the DPI parameter of a printer, the better is the quality of the output generated by it. The higher PPM represents higher efficiency of the printer. Printers can be classified based on the technology they use to print the text and images:

- **Dot matrix printers** — Dot matrix printers are impact printers that use perforated sheet to print the text. The process to print a text involves striking a pin against a ribbon to produce its impression on the paper. As the striking motion of the pins help in making carbon copies of a text, dot matrix printers are used to produce multiple copies of a print out.
- **Inkjet printers** — Inkjet printers are slower than dot matrix printers and are used to generate high quality photographic prints. Inkjet printers are not impact printers. The ink cartridges are attached to the printer head that moves horizontally, from left to right. The print out is developed as the ink of the cartridges is sprayed onto the paper. The ink in the inkjet is heated to create a bubble. The bubble bursts out at high pressure, emitting a jet of the ink on the paper thus producing images.
- **Laser printers** — The laser printer may or may not be connected to a computer, to generate an output. These printers consist of a microprocessor, ROM and RAM, which can be used to store the textual information. The printer uses a cylindrical drum, a toner and the laser beam. The toner stores the ink that is used in generating the output. The fonts used for printing in a laser printer are stored in the ROM or in the cartridges that are attached to the printer. The laser printers are available as gray scale, black and white or color models. To print high quality pages that are graphic intensive, laser printers use the PageMaker software.

1.10.3 Speaker

The speaker is an electromechanical transducer that converts an electrical signal into sound. They are attached to a computer as output devices, to provide audio output, such as warning sounds and Internet audios. We can have built-in speakers or attached speakers in a computer to warn end users with error audio messages and alerts. The audio drivers need to be installed in the computer to produce the audio output. The sound card being used in the computer system decides the quality of audio that we listen using music CDs or over the Internet. The computer speakers vary widely in terms of quality and price. The sophisticated computer speakers may have a subwoofer unit, to enhance bass output.

1.10.4 Plotter

The plotter is another commonly used output device that is connected to a computer to print large documents, such as engineering or constructional drawings. Plotters use multiple ink pens or inkjets with color cartridges for printing. A computer transmits binary signals to all the print heads of the plotter. Each binary signal contains the coordinates of where a print head needs to be positioned for printing. Plotters are classified on the basis of their performance, as follows:

- **Drum plotter** — They are used to draw perfect circles and other graphic images. They use a drawing arm to draw the image. The drum plotter moves the paper back and forth through a roller and the drawing arm moves across the paper.
- **Flat-bed plotter** — A flat bed plotter has a flat drawing surface and the two drawing arms that move across the paper sheet, drawing an image. The plotter has a low speed of printing and is large in size.
- **Inkjet plotter** — Spray nozzles are used to generate images by spraying droplets of ink onto the paper. However, the spray nozzles can get clogged and require regular cleaning, thus resulting in a high maintenance cost.
- **Electrostatic plotter** — As compared to other plotters, an electrostatic plotter produces quality print with highest speed. It uses charged electric wires and special dielectric paper for drawing. The electric wires are supplied with high voltage that attracts the ink in the toner and fuses it with the dielectric paper.

1.11 Types of Software

In the field of computer science, software is defined as a computer program, which includes logical instructions used for performing a particular task on a computer system using hardware components. The following are the two major categories of software under which different types of computer programs can be classified:

- System software
- Application software

Figure 1.19 shows the relationship among hardware, software, and user.

The figure shows a layered architecture, which represents different components of a computer such as hardware, system software, application software, and user in a hierarchical manner.

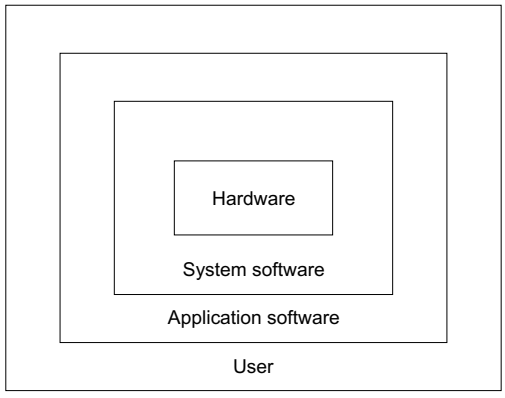


Fig. 1.19 ⇔ *Relationship among hardware, software, and user*

1.11.1 System Software

System software refers to a computer program that manages and controls hardware components of a computer system. In other words, the system software is responsible for handling the functioning of the computer hardware. The system software is also responsible for the proper functioning of the application software on a computer system. The system software includes general programs, which are written to provide an environment for developing new application software using programming languages. In computer science, there are several types of system software, such as operating systems and utility programs. The operating system is the primary system software, which controls the hardware and software resources of a computer system. It also performs various operations, such as memory allocation, instruction processing, and file management. The most commonly used operating systems are MS DOS, MS Windows, and UNIX. The following are the various functions of system software:

- Process management
- Memory management
- Secondary storage management
- I/O system management
- File management

1.11.2 Application Software

Application software is a computer program that is executed on the system software. It is designed and developed for performing specific tasks and is also known as end-user program. Application software is unable to run without the system software, such as operating system and utility programs. It includes several applications, such as word-processing and spreadsheet. The word-processing application helps in creating and editing a document. Using this application software, we can also format and print the document. For word-processing, many applications are available, such as WordStar, WordPerfect, and Open-source. The most commonly used word-processing application is MS Word, which is a part of the MS Office suite. Spreadsheet application helps in creating a customized ledger, which has number of columns

and rows for entering the data values. The most commonly used spreadsheet application is MS Excel, which is also a part of MS Office suite. It helps in storing and maintaining a database in a structural manner.

1.12 Computer Ethics

Ethics is a set of moral guidelines or principles that help ascertain what is right or wrong in a particular situation. When such principles are applied to computers and their related usage it is referred as computer ethics. Computers along with the advent of Internet have opened up a plethora of options for the users. There is no set pattern to what all tasks a user can or will perform while working on computers. Both professional and personal usage of computers may require the users to make certain decisions based on their ethical judgement; thus, it is important to be aware of the right conduct while working on computers and the Internet.

With growing use of computers in almost every field, the awareness towards computer ethics has also increased. Now, most of us are aware of what comprises a computer crime or an online fraud. In fact, most of the ethical guidelines have also been adopted into relevant laws both nationally as well as internationally. Some of the ethical issues concerning the use of computers are explained below.

1.12.1 Hacking

It is the unauthorized access to somebody else's computer system with malicious intentions. Hacking may be done to carry out illegal tasks, such as information theft, online theft of funds, privacy breach, etc. A number of security measures are adopted to prevent any attempt to computer hacking. Additionally, the computer users must be made aware of the fact that accessing a restricted system without appropriate rights or permissions is absolutely illegal and unethical.

Of late, a popular term called ethical hacking has come to the fore. It is used by professional hackers to expose the loopholes in the security framework of a computer system. The owners of the system are later made aware of these gaps so that appropriate mitigating actions can be taken.

1.12.2 Breach of Privacy

While blatant breaches of privacy like illegal access to somebody else's e-mail or social networking account come under the purview of hacking, there are certain finer privacy issues that are quite debatable. For instance, the use of cookies or spywares to log personal details of the Internet users do not obviously go well with some users but the providers argue that these details are used only for the same users' convenience. For example, if a user browses for cookery books on an online books portal, then he/she may get happily surprised to see the new add-ons in the cookery category prominently displayed on the Web site's main page on his/her subsequent visits to the portal. However, certain users may object to such a situation where their personal information is being gathered without their consent. There are appropriate laws in place now to make sure that such activities are carried out within the legal and ethical purview.

1.12.3 Intellectual Property Theft

Intellectual property comprises of all those intangible entities that originate from somebody's intellect or creativity. For instance, digital content, songs, images, etc. are all referred as intellectual property. The use of computers with Internet has made it extremely difficult to check intellectual property thefts. For instance, it is very easy for a person to copy the content written by somebody else and publish it under his own credentials at some other platform. Thus, ethical and legal grounds are a must to check the theft of intellectual property. The owners of intellectual property are advised to clearly tag the property with Intellectual Property Rights statement that mentions who the owners of the property are and how it can be accessed and utilized by the other users.

1.12.4 Ethical Standards

Several organizations and professional bodies have come up with a set of ethical standards or code of ethics for computer professionals. Some of these organizations are:

- Association for Computing Machinery
- British Computer Society
- Uniform Computer Information Transactions Act (UCITA)

The objective of all sets of ethical standards is same, i.e., to make the individuals aware of the right ethical behaviour while working with computers.

1.13 Application of Computers

The use of computers has grown in leaps and bounds in the last two decades. There is virtually no field untouched by computers, either directly or indirectly. For instance, e-mail, online funds transfer, e-ticketing, e-commerce, etc. are all automated alternatives to otherwise manual tasks. The use of computers has become so much widespread that it is difficult to imagine even a single day without them. Here, we'll focus on some of the specific areas where computers are of great importance and usage.

1.13.1 E-business

E-business or electronic business refers to the use of information and communication technologies (ICT) for conducting business. ICT uses computer, networking and telecommunication systems for the purpose of communication. To understand the concept of e-business, let us take the example of buyer-supplier relationship. By using a computer equipped with Internet, a buyer can receive quotations from the supplier through e-mail. Further, the buyer can raise a purchase order towards the supplier on its vendor management portal. The supplier, on the other hand can raise the corresponding invoices on the same portal once the delivery of the ordered materials is done. All these e-business activities greatly reduce the amount of paper work as well as the related overheads. In addition to the buyer-supplier scenario, we can also refer other activities such as online banking, online shopping, online stock trading, etc., as examples of e-business activities.

1.13.2 Bioinformatics

Bioinformatics is the field of science which applies computer-based tools and technologies on biological research and development. It primarily involves collection and storage of biological and genetic data on which statistical techniques are applied to arrive at the required solution.

Bioinformatics has revolutionized the way research and development activities are carried out in the field of molecular biology. Some of the key research-oriented application areas of bioinformatics are DNA mapping, medicinal research, 3-D modelling, pattern recognition, computational genomics, etc. The field of bioinformatics is continuously evolving with the discovery of advanced tools and techniques for addressing challenges in the field of molecular biology.

1.13.3 Healthcare

Nowadays, computers are being used to cater to several different aspects of healthcare. The use of computers is evident right from the beginning when a patient approaches a healthcare facility. The healthcare staff logs the patient's details in an organized manner in the computer system. The same computer is used for finding and allocating a vacant bed to the patient, if required. If the patient had visited the healthcare facility earlier then the physician can check history of treatments that the patient has already undergone. This helps the physician in effective diagnosis of the patient's ailment. Further, if the doctor wishes to seek advice on a complicated case then he/she can do that instantly by using ICT technologies.

The most significant use of computers within healthcare has been its amalgamation with medical equipments. A majority of medical equipments are now computer based, thus enabling accurate capture of data in digital form. Further, devices like CT scanner help the physicians to view a 3-D imagery of body organs, thus facilitating effective diagnosis.

1.13.4 Remote Sensing and GIS

Remote sensing is the technique of acquiring information about a subject (material or spatial) without coming in direct contact with it. Since, there is no direct contact involved, wireless devices are used for performing remote sensing tasks. Such devices are typically real-time systems that continuously gather and store data related to the subject under observation. A RADAR system can be considered as a good example of a remote sensing device that measures the time delay between sending and receiving of signals to detect information related to objects. Computers are used to collect, manage and analyze such data collected using remote sensing techniques. Marine navigation and air traffic management are some of the typical examples where computers are used along with remote sensing mechanisms.

Geographical Information System or GIS is a system that gathers location-specific data and presents it in various meaningful forms. It is basically a computer-based information system that captures and stores location-specific data against different parameters. Relationships are then drawn from these data elements and presented in a suitable format. For example, a utility GIS system fitted in a car may help a person while travelling at an unknown location with the help of real-time maps. Further, the user may additionally be fed vital information around the current location, such as the closest filling station, automobile service station, restaurant, etc. Some of the application areas of GIS systems are environmental research, disaster management, demographic studies, and so on.

1.13.5 Meteorology and Climatology

Meteorology is the study of atmosphere and the related weather conditions over short time intervals aimed at making routine weather forecasts. Climatology, on the other hand is the study of weather conditions over a long period of time (in years) so as to explore the climate of a region in totality. Both these processes use computers for collection, storage and processing of data. Climatology in particular requires extensive data analysis to study the weather trends over long periods of time. The results of the analysis are displayed using graphics and animations so that accurate forecasts can be predicted by the weather experts.

1.13.6 Computer Gaming

Computers are widely used for playing games that are similar to video or console-based games. A computer must possess graphics and animations support for ensuring rich gaming experience to the users. The computer gaming industry has evolved tremendously over the past few decades. A number of software companies are dedicatedly involved in developing computer games that are sold to the users under the licensing arrangement.

In the last decade or so, the concept of online games has also come to the fore. Now, users do not need to install games on their personal computers. They can simply log on to the provider's Web site and play games in real-time. The gamers are represented on the Web through their virtual identities. In certain cases like virtual reality, these virtual entities behave just like their real-world counterparts. Although, computer-based games are meant for the sole purpose of entertainment, they have certain flip sides as well. For instance, excessive use of computer games at times makes the users game-addicts, thus hampering their daily routine lives.

1.13.7 Multimedia and Animation

Multimedia is a system that represents information through various media components such as music, video, sound, text and images. Animation is an instance of multimedia that represents images or art works in motion. A cartoon film is a very good example of an animation. Multimedia and animations are actively used in the entertainment industry to add special effects to a video. Similarly, they are also used for the purpose of training and development. For instance, a flight simulator that creates a virtual flying environment for the trainee pilot is an example of a high-end animation system. A number of development tools such as Flash, Captivate, etc., can be used by designers to build a multimedia and animation system.

SUMMARY

⇔ In simple terms, a computer is defined as an electronic machine that takes input from the user, processes the given input and generates output in the form of useful information. However, in actuality the power and capability of computers go way beyond this simple definition. There are hardly any areas that have not been impacted by computers in some way or the other. In fact, it is now hard to imagine a life without the direct or indirect usage of computers.

- Based on operating principles, applications and size and capability, the computers are classified into different categories, such as analog computer, digital computer, mini computer, mainframe, etc. Each of these categorizations helps us to explore a different facet of computers in terms of their functioning and performance.
- Any type of computer system possesses four key constituents i.e., an input system, output system, CPU and memory. While input and output systems are essential for user interaction, CPU and memory are used for data processing and storage. The primary memory of a computer system stores the data temporarily till the time a program is executed while the secondary memory stores the data permanently for later usage. Bus is another important hardware component that enables the transfer of data to and from CPU and memory and CPU and input/output system.
- Even though hardware constitutes the backbone of a computer system, it the software that brings hardware into operation. It acts as an interface between the users and the underlying hardware system. Computer software are broadly classified into two categories, system software and application software. All system management programs (such as operating system, device driver, etc.) and system development programs (such as compilers, IDEs, etc) come under the category of system software. On the other hand, all standard application programs (such as word processor, spreadsheet, etc.) and unique application programs (such as inventory management system, payroll system, etc.) come under the application software category.

Key Terms

- Computer
- Data
- Program
- Information
- Analog Computer
- Digital computer
- Hybrid computer
- Microcomputer
- Minicomputer
- Super computer
- Mainframe computer
- Hardware
- Software
- AU
- LU
- CU
- Opcode
- Operand
- Main memory
- Cache memory
- Primary cache
- Secondary cache
- Registers
- PC
- IR
- MAR

- MBR
- MDR
- Accumulator
- Bus
- Data bus
- Address bus
- Control bus
- Complex instruction set
- Reduced instruction set
- CISC
- RISC
- Input devices
- Output devices
- Memory
- Scanner
- Motherboard
- RAM
- Printer
- Speaker
- Plotter
- Application software
- System software

Review Questions

1.1 *What are the different components of a computer? Explain, each of them.*

1.2 *Discuss briefly the various characteristics of a computer.*

1.3 *How are computers classified? Explain briefly.*

1.4 *Draw the block diagram of a microcomputer.*

1.5 *Differentiate between hardware and software of a computer?*

1.7 *Draw the block diagram of a computer system and explain its main components.*

1.8 *How does the control unit assist the CPU in carrying out its operations?*

1.9 *What do you understand by CPU cycle? What are the main operations accomplished using the CPU cycle?*

1.10 *Explain the concept of cache memory with diagram. What are the different types of cache memory found in a computer system?*

1.11 *List the different types of CPU registers.*

1.12 *What do you understand by internal communications? List the major categories of internal communications.*

1.13 *Explain the importance of MAR and MDR in accomplishing processor to memory communication.*

1.14 *What is the difference between memory read and memory write operations? List the different steps involved in memory read and memory write operations.*

1.15 *Explain the concept of processor to I/O devices communication. What is the role of interface unit in implementing such type of communication in a computer system?*

1.16 *Explain how data is transferred from processor to an output device within a computer system.*

1.17 *Explain the importance of a bus in the computer system. What are the different types of buses usually found in the computer system?*

1.18 *What is the function of a control bus in a computer system?*

1.19 *Explain the concept of instruction set. Which component of a computer system is mostly associated with instruction set?*

1.20 *What are the different factors considered for classifying an instruction set?*

1.21 *Explain in detail the different categories of instruction sets.*

1.22 *What do you understand by CISC? List the advantages and disadvantages of CISC.*

1.23 *What do you understand by RISC? List the advantages of RISC processors.*

1.24 *What are input devices? Briefly explain some popular input devices.*

1.25 *What is the purpose of an output device? Explain various types of output devices.*

1.66 *List, with examples, five important application areas of computers today.*

Fill in the Blanks

1.1 The central processing unit of the computer system is popularly known as _____ of the computer system.

1.2 _____ unit is responsible for performing all the arithmetic operations in the computer system.

1.3 The _____ register keeps the track of the next instruction to be executed.

1.4 _____ is called the main-memory of the computer.

1.5 The access time of _____ memory is faster as compared to that of main memory.

1.6 The group of wires used to connect the components of CPU to transfer the data is called _____.

1.7 The time taken by the CPU to fetch an instruction and execute it is called _____.

1.8 The group of instructions that a processor can execute is called _____.

1.9 _____ bus is used to transfer data from the memory.

1.10 The instruction set can be classified as _____ and _____.

1.11 The _____ field represents the operation to be performed by the instruction.

1.12 A _____ is an electronic machine that takes input from the user and stores and processes the given input to generate the output in the form of useful information to the user.

1.13 The raw details that need to be processed to generate some useful information is known as _____.

1.14 The set of instructions that can be executed by the computer is known as _____.

1.15 _____ is the processor of the computer that is responsible for controlling and executing the various instructions.

1.16 _____ is a screen, which displays the information in visual form, after receiving the video signals from the computer.

1.17 A _____ is the fastest type of computer that can perform complex operations at a very high speed.

1.18 The term _____ refers to the programs and instructions that help the computer in carrying out their processing.

1.19 The programs, which are designed to perform a specific task for the user, are known as _____.

1.20 The programs, which are designed to control the different operations of the computer, are known as _____.

1.21 The instruction set of _____ has a few instructions compared to that of _____.

1.22 The direct communication between processor and memory of the computer system is implemented with the help of two registers, _____ and _____.

1.23 The two important operations performed while communicating with the memory of the computer system are _____ and _____.

1.24 The flag bit of the status register is _____, when the register holds the data.

1.25 The computer, which makes use of reduced instruction set, is called _____.

1.26 The speed of the RISC processors is measured in _____.

1.27 _____ bus transfers the memory addresses for reading or writing the data.

Multiple Choice Questions

1.1 Which component of the computer is known as the brain of computer?

A. Monitor B. CPU C. Memory D. None of the above

1.2 Which of the following is an input device?

A. Printer B. Monitor C. Mouse D. None of the above

1.3 Which of the following is a characteristic of the modern digital computer?

A. High speed B. Large storage capacity

C. Greater accuracy D. All of the above

1.4 On what basis computers can be classified?

A. Operating principles B. Applications

C. Size and capability D. All of the above

1.5 Which of the following unit is a part of the CPU?

A. ALU B. CU C. Memory unit D. All of the above

1.6 Which of the following is known as a midrange computer?

A. Microcomputer
B. Mini computer
C. Mainframe computer
D. Super computer

1.7 The programs and instructions that help the computer in carrying out their processing are known as?

A. Hardware B. Software C. Data D. None of the above

1.8 What does CPU stand for?

A. Center processing unit
B. Central processing unit
C. Central programming unit
D. Computer processing unit

1.9 Which one of the following is not an internal component of CPU?

A. Arithmetic unit B. Logic unit C. Interface unit D. Control unit

1.10 What is the main function of CPU in a computer system?

A. Storing the data
B. Programming the computer
C. Transferring the data to an output device
D. Processing the data

1.11 Which of the following memory locations are first referred by the CPU while searching for data?

A. Main memory
B. Cache Memory
C. ROM
D. Secondary memory

1.12 Cache memory is used to transfer data between:

A. Main memory and secondary memory
B. Processor and an input device
C. Main memory and processor
D. Processor and an output device

1.13 Which one of the following cache memory is also known as internal cache?

A. L2 cache B. L1 cache C. L3 cache D. L4 cache

1.14 Which one of the following hardware components is normally used to accommodate secondary cache?

A. Motherboard
B. Processor
C. RAM
D. Any secondary storage device

1.15 Which one of the following statement is not true about L1 cache?

A. L1 cache is a type of cache memory.
B. L1 cache stores the data from the main memory.
C. L1 cache is an expensive type of memory.
D. L1 cache is usually slower than L2 cache.

1.16 Which one of the following register is not a CPU register?

A. Memory control register B. Memory data register

C. Memory buffer register D. Instruction register

1.17 What is the purpose of memory address register?

A. Stores the address of the next location in the main memory

B. Stores the address of the next location in the secondary memory

C. Stores the address of the next location in the cache memory

D. Stores the address of an output device to which the data is to be sent

1.18 Which of the following registers is used to indicate whether the data register holds the data to be transferred or not?

A. Status register B. MAR C. MBR D. MDR

1.19 Which of the following two registers are used by the CPU to transfer the data between processor and memory?

A. MDR and IR B. PC and IR C. IR and MAR D. MAR and MDR

1.20 Which of the following is not a type of bus used in the computer system?

A. Data bus B. Address bus

C. Information bus D. Control bus

1.21 Which of the following is a characteristic of the CISC processor?

A. Number of instructions varying between 100 and 250

B. Large number of addressing modes

C. Variable length instruction format

D. All of the above

1.22 What is the fixed length of the instruction format used in RISC processors?

A. 30-bits B. 16-bits C. 32-bits D. 24-bits

1.23 Which of the following is the most powerful and expensive computer at present?

(a) Minicomputer (b) Mainframe computer

(c) Supercomputer (d) Microcomputer

1.24 Which of the following is used to perform computations on the entered data?

(a) Memory (b) Processor (c) Input device (d) Output device

1.25 Which of the following is not an input device?

(a) Plotter (b) Scanner (c) Keyboard (d) Mouse

1.26 Which of the following is not an output device?

(a) Plotter (b) Scanner (c) Printer (d) Speaker

1.27 Which of the following is used as a primary memory of the computer?

(a) Magnetic storage device (b) RAM

(c) Optical storage device (d) Magneto-optical storage device

1.28 Which of the following is used as a secondary memory of the computer?

(a) Magnetic storage device (b) RAM

(c) Cache memory (d) ROM

1.29 Which of the following is defined as a computer program for performing a particular task on the computer system?

(a) Hardware (b) Software (c) Processor (d) Memory

2

Operating Systems

Key Concepts

- Operating System
- History
- Function of Operating System
- Processs Management
- Memory Management
- File Management
- Batch Processing Operating Systems
- Multi-User Operating Systems
- Multitasking Operating Systems
- Real-Time Operating Systems
- Multiprocessor Operating Systems
- Embedded Operatings System
- MS-DOS
- UNIX
- Windows

2.1 Introduction

An operating system (OS) is a software that makes the computer hardware to work. While the hardware provides 'raw computer power', the OS is responsible for making the computer power useful for the users. As discussed in the previous chapter, the OS is the main component of system software and therefore must be loaded and activated before we can accomplish any other task.

The operating system provides an interface for users to communicate with the computer. It also manages the use of hardware resources and enables proper implementation of application programs. In short, the operating system is the master control program of a computer. Figure 2.1 shows the different roles performed by an operating system. The main functions include:

- Operates CPU of the computer.
- Controls input/output devices that provide the interface between the user and the computer.

- Handles the working of application programs with the hardware and other software systems.
- Manages the storage and retrieval of information using storage devices such as disks.

Every computer, irrespective of its size and application, needs an operating system to make it functional and useful. Operating systems are usually prewritten by the manufacturers and supplied with the hardware and are rarely developed in-house owing to its technical complexity. There are many operating systems developed during the last few decades but the popular among them are MS-DOS, Windows 2000, Windows XP, Windows Server 2003, UNIX and Linux.

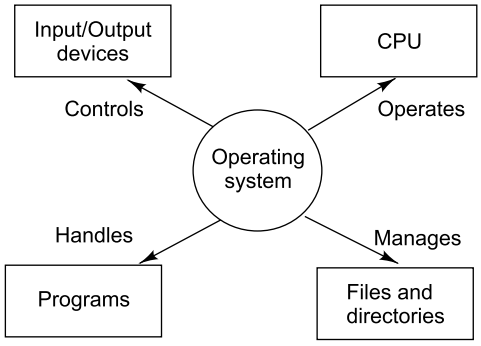


Fig. 2.1 ⇔ *The roles of an operating system*

In this chapter we shall discuss in detail the various functions of operating systems, different types of operating systems and their services, and the types of user interfaces available.

2.2 History of Operating Systems

A series of developments in the computer architecture led to the evolution of the operating system during the later half of the 20th century. During the 1940's, there was no operating system and assembly language was used to develop programs that could directly interact with the hardware. The computer systems during this period were mainly used by the researchers, who were both the programmers as well as the end users of the computer system.

During the 1950s, more number of people started using the computer systems. This led to a repetition of tasks as everyone started developing their own programs and device drivers. Different people created device drivers for the same input and output devices. To avoid this repetition of tasks, various batch processing operating systems such as BKY, CAL and Chios were developed during this period. FORTRAN Monitor System, General Motors Operating System and Input Output System are the other operating systems developed in the 1950s. The operating systems developed during this period were capable of performing only a single task at a time.

During the 1960s, multi-tasking operating systems were developed. These operating systems ensured better utilisation of resources by allowing the multiple tasks to be performed simultaneously. They also allowed multiple programs to remain in memory at the same time. Central Processing Unit (CPU) executed multiple processes at a single time and also handled the hardware devices attached to the computer system. These operating systems used the concepts of spooling and time-sharing model for achieving the multi-tasking functionality. The various operating systems developed during the 1960s include Admiral, Basic Executive System, Input Output Control System and SABRE (Semi-Automatic Business Related Environment).

During the 1970s, a major breakthrough was achieved in the development of operating system with the introduction of UNIX by AT&T Bell Labs. UNIX supported a multi-user

environment where multiple users could work on a computer system. The core functionality of UNIX resided in a kernel that was responsible for performing file, memory and process management. UNIX also came bundled with utility programs for performing specific tasks. The other operating systems that were introduced in the 1970s include DOS/VS, OS/VS1 and OpenVMS.

During the 1980s, some key operating systems were developed including MS-DOS, HP-UX and Macintosh. MS-DOS was developed by Microsoft and could be installed on desktop Personal Computers (PCs), such as Intel 80x86 PCs. HP-UX was similar to UNIX and was developed by Hewlett Packard. This operating system could be installed on the HP PA RISC computer systems. Macintosh was developed by Apple computers and could be installed on the desktop PCs such as Motorola 680x0. MS DOS and Macintosh became quite popular in the 1980's and are still in use.

A number of operating systems were developed during the 1990s including Windows 95, Windows 98, Windows NT, FreeBSD and OS/2. Windows 95, Windows 98 and Windows NT were GUI based operating systems developed by Microsoft. FreeBSD was similar to UNIX and was available free of cost. OS/2 was introduced by IBM and could be installed on Intel/AMD Pentium and Intel 80x86 based computer systems. The decade of 1990 revolutionised the way of computing through robust GUI-based operating systems and fast processing devices.

The first decade of 21st century has seen the development of operating systems such as MAC OS X, Windows 2000, Windows Server 2003, Windows ME and Windows XP. With the advent of Internet, security has been the prime focus of the operating systems of this era.

2.3 Functions of Operating Systems

The main function of an operating system is to manage the resources such as memory and files of a computer system. The operating system also resolves the conflicts that arise when two users or programs request the same resource at the same time. Therefore, the operating system is also called the resource manager of a computer system. The currently used operating systems such as Windows 2000, Windows Server 2003 and Linux also support networking that allows the sharing of files and resources such as printer and scanner. The following are some of the important functions of an operating system:

- **Process management** It manages the processes running in a computer system. A process is basically a program that is being currently run by a user on a computer system. For example, a word processor application program such as Microsoft Word runs as a process in a computer system.
- **Memory management** It manages the memory resources of a computer system. There are various memory resources of a computer system including primary memory or Random Access Memory (RAM) and secondary memory like hard disk and Compact Disk (CD). All the programs are loaded in the main memory before their execution. It is the function of the operating system to determine how much memory should be provided to each process.
- **File management** It manages the files and directories of a computer system. A file can be defined as a collection of information or data that is stored in the memory of a

computer system. Every file has a unique name associated with it. The organisation of files and directories in a computer system is referred as file system. An operating system allows us to create, modify, save, or delete files in a computer system.

- **Device management** This function of operating system deals with the management of peripheral devices, such as printer, mouse and keyboard attached to a computer system. An operating system interacts with the hardware devices through specific device drivers. The primary task of an operating system is to manage the input/output operations performed by the end users.
- **Security management** It ensures security for a computer system from various threats such as virus attacks and unauthorised access. An operating system uses various techniques, such as authentication, authorisation, cryptography, etc. for ensuring security of a computer system.

Figure 2.2 depicts the various functions of an operating system.

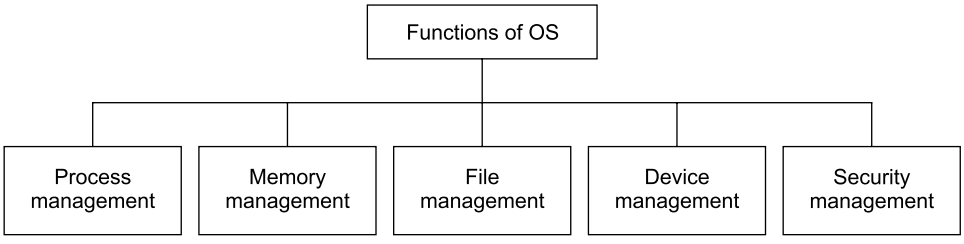


Fig. 2.2 ⇔ *The functions of an operating system*

2.4 Process Management

Process management involves the execution of various tasks such as creation of processes, scheduling of processes, management of deadlocks and termination of processes. When a process runs in a computer system, a number of resources such as memory and CPU of the computer system are utilised. It is the responsibility of an operating system to manage the running processes by performing tasks such as resource allocation and process scheduling. The operating system also has to synchronise the different processes effectively in order to ensure consistency of shared data.

Generally, only a single process is allowed to access the CPU for its execution at a particular instant of time. When one process is being processed by the CPU, the other processes have to wait until the execution of that particular process is complete. After the CPU completes the execution of a process, the resources being utilised by that process are made free and the execution of the next process is initiated. All the processes that are waiting to be executed are said to be in a queue. In some cases, a computer system supports parallel processing allowing a number of processes to be executed simultaneously.

A process consists of a set of instructions to be executed called *process code*. A process is also associated with some data that is to be processed. The resources that a process requires for its execution are called *process components*. There is also a state associated with a process at

a particular instant of time called *process state*. Similar to these concepts, there are a number of concepts associated with the process management function of an operating system. Some of these key concepts are:

- Process state
- Process Control Block (PCB)
- Process operations
- Process scheduling
- Process synchronisation
- Interprocess communication
- Deadlock

2.4.1 Process State

A process state can be defined as the condition of a process at a particular instant of time. There are basically seven states of a process:

- **New** It specifies the time when a process is created.
- **Ready** It specifies the time when a process is loaded into the memory and is ready for execution.
- **Waiting** It specifies the time when a process waits for the allocation of CPU time and other resources for its execution.
- **Executing** It is the time when a process is being executed by the CPU.
- **Blocked** It specifies the time when a process is waiting for an event like I/O operation to complete.
- **Suspended** It specifies the time when a process is ready for execution but has not been placed in the ready queue by the operating system.
- **Terminated** It specifies the time when a process is terminated and the resources being utilised by the process are made free.

Figure 2.3 illustrates the various process states.

The Fig. 2.3 shows that a process is initially in the new state when it is created. After the process has been created, the state of the process changes from new to ready state where the process is loaded into the memory. The state of the process changes from ready to the waiting state when the process is loaded into the memory. The process state changes from waiting to the executing state after the CPU time and other resources are allocated to it and the process starts running. After the process has executed successfully, it is terminated and its state changes to terminated.

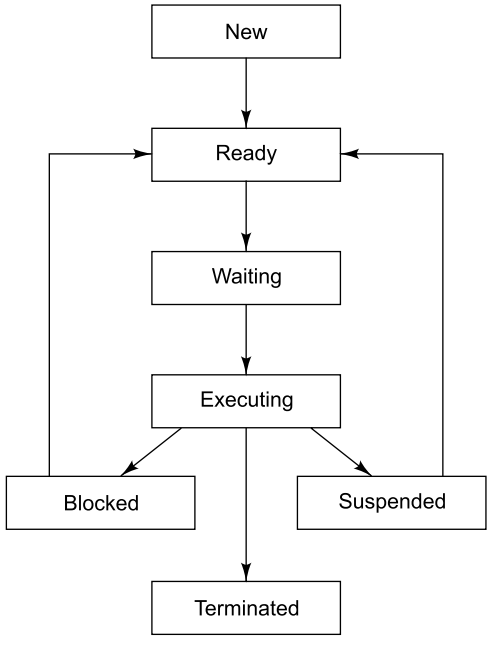


Fig. 2.3 ⇔ *The different states of a process*

2.4.2 Process Control Block (PCB)

PCB is a data structure associated with a process that provides complete information about the process. PCB is important in a multiprogramming environment as it captures information pertaining to a number of processes running simultaneously. PCB comprises of the following:

- **Process id** It is an identification number that uniquely identifies a process.
- **Process state** It refers to the state of a process such as ready and executing.
- **Program counter** It points to the address of the next instruction to be executed in a process.
- **Register information** It comprises of the various registers, such as index and stack that are associated with a process.
- **Scheduling information** It specifies the priority information pertaining to a process that is required for process scheduling.
- **Memory related information** This section of the PCB comprises of page and segment tables. It also stores the data contained in base and limit registers.
- **Accounting information** This section of the PCB stores the details relate to CPU utilisation and execution time of a process.
- **Status information related to I/O** This section of the PCB stores the details pertaining to resource utilisation and the files opened during process execution.

The operating system maintains a table called *process table*, which stores the PCBs related to all the processes. Figure 2.4 shows the structure of PCB.

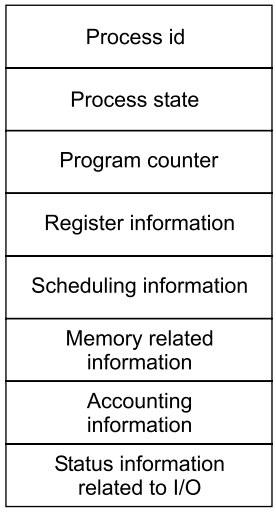


Fig. 2.4 ⇔ *The structure of a PCB*

2.4.3 Process Operations

The process operations carried out by an operating system are primarily of two types, *process creation* and *process termination*. Process creation is the task of creating a new process. There are different situations in which a new process is created. A new process can be created during the time of initialisation of operating system or when system calls such as create-process and fork( ) are initiated by other processes. The process, which creates a new process using system calls, is called *parent process* while the new process that is created is called *child process*. The child processes can further create new processes using system calls. A new process can also be created by an operating system based on the request received from the user. Figure 2.5 shows the hierarchical structure of multiple processes running in a computer system.

The process creation operation is very common in a running computer system because corresponding to every task that is performed there is a process associated with it. For instance, a new process is created every time a user logs on to a computer system, an application program such as MS Word is initiated, or when a document is printed.

Process termination is an operation in which a process is terminated after it has executed its last instruction. When a process is terminated, the resources that were being utilised by

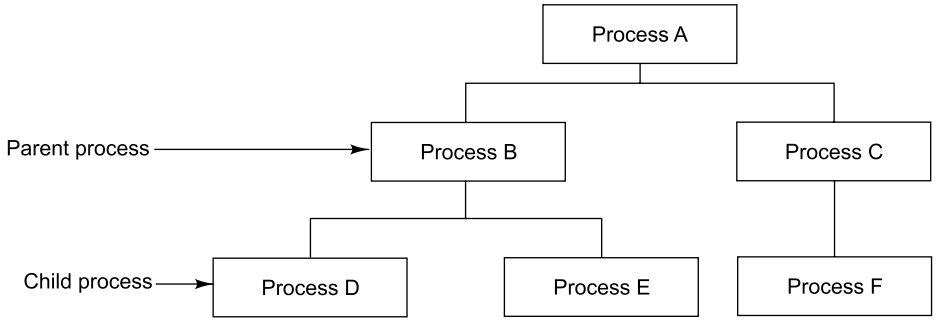


Fig. 2.5 ⇔ *The hierarchical structure of processes*

the process are released by the operating system. When a child process terminates, it sends the status information back to the parent process before terminating. The child process can also be terminated by the parent process if the task performed by the child process is no longer needed. In addition, when a parent process terminates, it has to terminate the child process as well because a child process cannot run when its parent process has been terminated.

The termination of a process when all its instructions have been executed successfully is called *normal termination*. However, there are instances when a process terminates due to some error. This is called *abnormal termination* of a process.

2.4.4 Process Scheduling

Process scheduling is the task performed by an operating system for deciding the priority in which the processes, which are in ready and waiting states, are allocated the CPU time for their execution. Process scheduling is very important in multiprogramming and multitasking operating systems where multiple processes are executed simultaneously. Process scheduling ensures maximum utilisation of CPU because a process is always running at a specific instant of time. At first, the processes that are to be executed are placed in a queue called *job queue*. The processes, which are present in the main memory and are ready for CPU allocation, are placed in a queue called *ready queue*. If a process is waiting for an I/O device then that process is placed in a queue called *device queue*.

An operating system uses a program called *scheduler* for deciding the priority in which a process is allocated the CPU time. Scheduler is of three types:

- **Long term scheduler** It selects the processes that are to be placed in the ready queue. The long term scheduler basically decides the priority in which processes must be placed in the main memory.
- **Mid term schedule**r It places the blocked or suspended processes in the secondary memory of a computer system. The task of moving a process from the main memory to the secondary memory is referred as swapping out. The task of moving back a swapped-out process from the secondary memory to the main memory is referred as swapping in. The swapping of processes is performed to ensure the best utilisation of main memory.
- **Short term scheduler** It decides the priority in which processes in the ready queue are allocated the CPU time for their execution. The short term scheduler is also referred as CPU scheduler.

An operating system uses two types of scheduling policies for process execution, preemptive and non preemptive. In the preemptive scheduling policy, a low priority process has to suspend its execution if a high priority process is waiting in the queue for its execution. However in the non preemptive scheduling policy, processes are executed in first come first serve basis, which means the next process is executed only when currently running process finishes its execution. The selection of the next process, however, may be done considering the associated priorities. Operating systems perform the task of assigning priorities to processes based on certain scheduling algorithms. Some of the key scheduling algorithms are:

- **First Come First Served (FCFS) scheduling** In this scheduling algorithm, the first process in a queue is processed first.
- **Shortest Job First (SJF) scheduling** In this scheduling algorithm, the process that requires shortest CPU time is executed first.
- **Priority scheduling** In this scheduling algorithm, a priority is assigned to all the processes and the process with highest priority is executed first. Priority assignment of processes is done on the basis of internal factors such as CPU and memory requirements or external factors such as user's choice. The priority scheduling algorithm can support either preemptive or non-preemptive scheduling policy.
- **Round Robin (RR) scheduling** In this scheduling algorithm, a process is allocated the CPU for a specific time period called time slice or time quantum, which is normally of 10 to 100 milliseconds. If a process completes its execution within this time slice then it is removed from the queue otherwise it has to wait until the next time slice.

2.4.5 Process Synchronisation

Process synchronisation is the task of synchronising the execution of processes in such a manner that no two processes have access to the same shared data or resource. When multiple processes are concurrently running then they may attempt to gain access to the same shared data or resource. This can lead to inconsistency in the shared data as the changes made by one process in the shared data may not be reflected when another process accesses the same shared data. In order to avoid such inconsistency of data, it is important that the processes are synchronised with each other.

One of the important concepts related to process synchronisation is that of critical section problem. Each process contains a set of code called critical section through which a specific task, such as changing the value of a global variable and writing certain data to a file, is performed. To ensure that only a single process enters its critical section at a specific instant of time, the processes need to coordinate with other by sending requests for entering the critical section. When a process is in its critical section no other process is allowed to enter the critical section.

Peterson's solution is one of the solutions to critical section problem involving two processes. Peterson's solution states that when one process is executing its critical section then the other process executes the rest of the code and vice versa. This ensures that only one process is in the critical section at a particular instant of time.

Locking is another solution to critical section problem in which a process acquires a lock before entering its critical section. When the process finishes executing its critical section,

it releases the lock. The lock is then available for any other process that wants to enter the critical section. The locking mechanism also ensures that only one process is in the critical section at a particular period of time.

Another solution to the critical section problem is that of *Semaphore*. It is basically a synchronisation tool in which the value of an integer variable called semaphore is retrieved and set using wait and signal operations. Based on the value of the Semaphore variable, a process is allowed to enter its critical section.

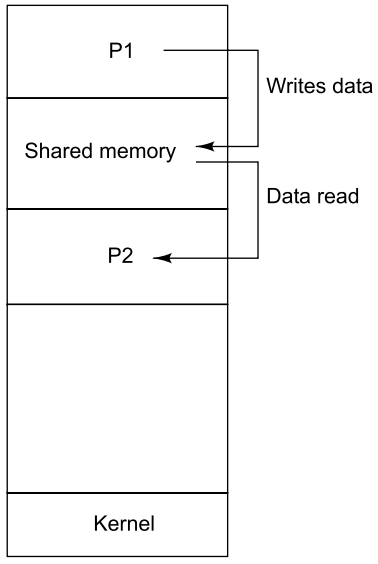


Fig. 2. 6 ⇔ *The shared memory method of interprocess communication*

2.4.6 Interprocess Communication

Interprocess communication is the method of communication between processes through which processes interact with each other for gaining access to shared data and resources. There are two methods of interprocess communication, *shared memory* and *message passing*.

In the shared memory method, a part of memory is shared between the processes. A process can write the data that it wants to share with other processes in to the memory. Similarly, another process can read the data that has been written by another process. Figure 2.6 shows the shared memory method of interprocess communication.

In Fig. 2.6, P1 and P2 represent the two processes. P1 writes the data that it needs to share with P2in the shared memory. P2 then reads the data written by P1 from the shared memory.

In the message passing method, a process sends a message to another process for communication. This method allows the sharing of data between processes in the form of messages. Figure 2.7 shows the message passing method of interprocess communication.

In Fig. 2.7, P1 sends the shared data in the form of a message to the kernel and then the kernel sends the message sent by P1 to P2.

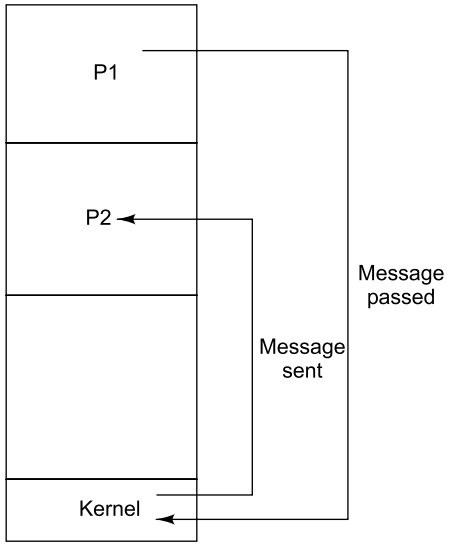


Fig. 2.7 ⇔ *The message passing method for the interprocess communication*

2.4.7 Deadlock

Deadlock is a condition that occurs when multiple processes wait for each other to free up resources and as a result all the processes remain halted. Let us

understand the concept of deadlock with the help of an example. Suppose there are two processes P1 and P2 running in a computer system. P1 requests for a resource, such as printer that is being utilised by the P2 process. As a result, the P1 process has to wait till the time P2 completes its processing and frees the resource. At the same time, the P2 process requests for a resource, such as shared data that has been locked by the process P1. Thus, both the processes end up waiting for each other to free up the required resources. This situation is called a deadlock.

The following are some of the reasons due to which a deadlock situation may arise.

- **Mutual exclusion** In mutual exclusion, processes are not allowed to share resources with each other. This means that if one process has control over a resource, then that resource cannot be used by another process until the first process releases the resource.
- **Hold and wait** In this condition, a process takes control of a resource and waits for some other resource or activity to complete.
- **No preemption** In this condition, a process is not allowed to force some other process to release a resource.

There are a number of methods through which the deadlock condition can be avoided. Some of these methods are:

- **Ignore deadlock** In this method, it is assumed that a deadlock would never occur. There is a good chance that a deadlock may not occur in a computer system for a long period of time. As a result, the ignore deadlock method can be useful in some cases.
- **Detect and recover from deadlock** In this method, the deadlock is first detected using allocation/request graph. This graph represents the allocation of resources to different processes. After a deadlock has been detected, a number of methods can be used to recover from the deadlock. One way is preemption in which a resource held by one process is provided to another process. The second way is rollback in which the operating system keeps a record of the process states and makes a process roll back to its previous state; thus eliminating the deadlock situation. The third way is to kill one or more processes to overcome the deadlock situation.
- **Avoid deadlock** In this method, a process requesting a resource is allocated the resource only if there is no possibility of deadlock occurrence.

2.5 Memory Management

Memory management function of an operating system helps in allocating the main memory space to the processes and their data at the time of their execution. Along with the allocation of memory space, memory management also perform the following activities:

- Upgrading the performance of the computer system
- Enabling the execution of multiple processes at the same time
- Sharing the same memory space among different processes

Memory management is one of the most important functions of operating system because it directly affects the execution time of a process. The execution time of a process depends on the availability of data in the main memory. Therefore, an operating system must perform

the memory management in such a manner that the essential data is always present in the main memory. An effective memory management system ensures accuracy, availability and consistency of the data imported from the secondary memory to the main memory.

An effective memory management system must ensure the following:

- **Correct relocation of data** The data should be relocated to and from the main memory in such a manner that the currently running processes are not affected. For example, if two processes are sharing a piece of data then the memory management system must relocate this data only after ensuring that the two processes are no longer referencing the data.
- **Protection of data from illegal change** The data present in the main memory should be protected against unauthorised access or modifications. The memory management system should ensure that a process is able to access only that data for which it has the requisite access and it should be prohibited from accessing data of other processes.
- **Provision to share the information** An ideal memory management system must facilitate sharing of data among multiple processes.
- **Utilisation of small free spaces** A memory management system should be able to apply appropriate defragmentation techniques in order to utilise small chunks of scattered vacant spaces in the main memory.

Segmentation, paging and swapping are the three key memory management techniques used by an operating system.

2.5.1 Segmentation

Segmentation refers to the technique of dividing the physical memory space into multiple blocks. Each block has a specific length and is known as a segment. Each segment has a starting address called the base address. The length of a segment determines the available memory spaces in the segment. Figure 2.8 shows the organisation of segments in a memory unit.

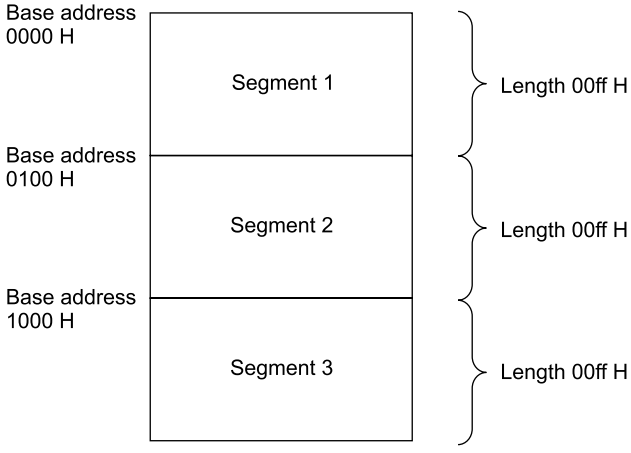


Fig. 2.8 ⇔ *Memory unit having segments*

The location of data values stored in a segment can be determined by the distance of the actual position of data value from the base address of the segment. The distance between the actual position of data and the base address of segment is known as displacement or offset value. In other words, whenever it is required to obtain data from the segmented memory then the actual address of the data is calculated by adding the base address of the segment and with offset value. The base address of the segment and the offset value is specified in a program instruction itself. Figure 2.9 shows how the actual position of an operand in a segment is obtained by adding the base address and the offset value.

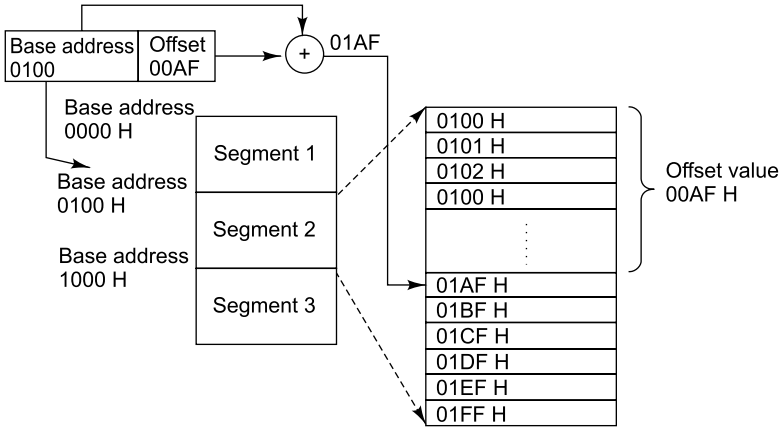


Fig. 2.9 ⇔ *Obtaining the actual address of data*

2.5.2 Paging

Paging is a technique in which the main memory of the computer system is organised in the form of equal sized blocks called pages. In this technique, the addresses of the occupied pages of the physical memory are stored in a table, which is known as page table.

Paging enables the operating system to obtain data from the physical memory location without specifying its lengthy memory address in the instruction. In this technique, a virtual address is used to map the physical address of the data. The length of the virtual address is specified in the instruction and is smaller than the physical address of the data. It consists of two different numbers, first number is the address of a page called virtual page in the page table and second number is the offset value of the actual data in the page. Figure 2.10 shows how the virtual address is used to obtain the physical address of an occupied page of physical memory using a page table.

2.5.3 Swapping

Swapping is a technique used by an operating system for efficient management of memory space of a computer system. Swapping involves performing two tasks called swapping in and swapping out. The task of placing the pages or blocks of data from hard disk to the main memory is called swapping in. On the other hand, the task of removing pages or blocks of data

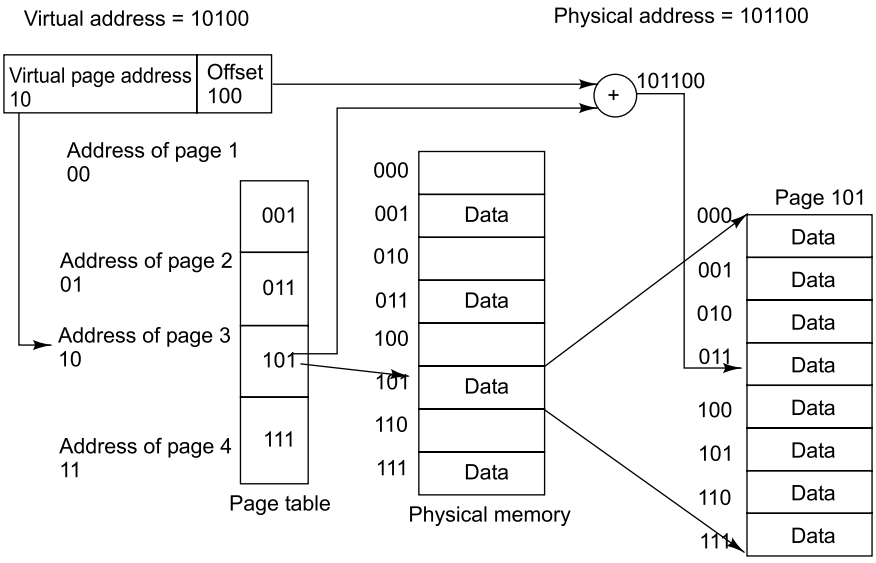


Fig. 2.10 ⇔ *Obtaining data from a page using the paging technique*

from main memory to hard disk is called swapping out. The swapping technique is useful when a large program has to be executed or some operations have to be performed on a large file.

The main memory in a computer system is limited. Therefore, to run a large program or to perform some operation on a large file, the operating system swaps in certain pages or blocks of data from the hard disk. To make space for these pages or blocks of data in the main memory, the operating system swaps out the pages or blocks of data that are no longer required in the main memory. The operating system places the swapped out pages or blocks of data in a swap file. A swap file is the space in the hard disk that is used as an extension to the main memory by the operating system. Figure 2.11 shows the technique of swapping used by the operating system for memory management.

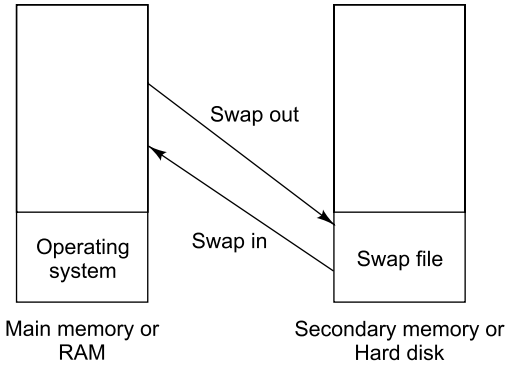


Fig. 2.11 ⇔ *Swapping of pages*

2.6 File Management

File management is defined as the process of manipulating files in a computer system. A file is a collection of specific information stored in the memory of the computer system. File management includes the process of creating, modifying and deleting the files. The following are some of the tasks performed by the file management function of operating system:

- It helps in creating new files and placing them at a specific location.
- It helps in easily and quickly locating the files in the computer system.
- It makes the process of sharing the files among different users easy.
- It helps store the files in separate folders known as directories that ensure better organisation of data.
- It helps modify the content as well as the name of the file as per the user's requirement.

Figure 2.12 shows the general hierarchy of file storage in an operating system.

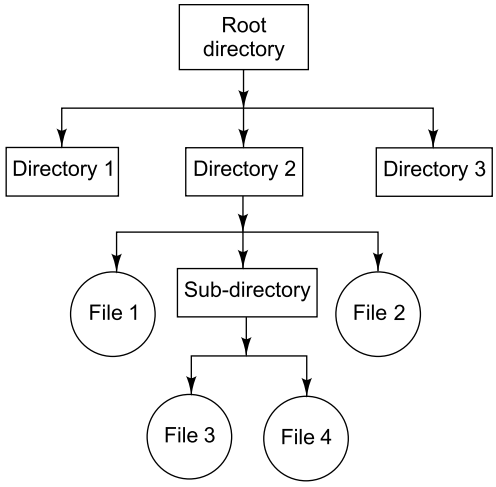


Fig. 2.12 ⇔ *The general hierarchy of file storage in an operating system*

In Fig. 2.12, the root directory is present at the highest level in the hierarchical structure. It includes all the subdirectories in which the files are stored. Subdirectory is a directory present inside another directory in the file storage system. The directory based storage system ensures better organisation of files in the memory of the computer system.

The file management function of OS is based on the following concepts:

- **File attributes** It specifies the characteristics, such as type and location that completely describe a file.
- **File operations** It specifies the tasks that can be performed on a file such as opening and closing a file.
- **File access permissions** It specifies the access permissions related to a file such as read and write.
- **File systems** It specifies the logical method of file storage in a computer system. Some of the commonly used file systems include FAT and NTFS.

2.6.1 File Attributes

File attributes are the properties associated with a file that specify different information related to a file. The following are some of the key file attributes:

- **Name** It specifies the name of a file given by the user at the time of saving it.
- **File type** It specifies the type of a file such as a Word document or an Excel worksheet.
- **Location** It specifies the location of a file where it is stored in the memory.
- **Size** It specifies the size of the file in bytes.
- **Date and time** It specifies the date and time when the file was created, last modified and last accessed.
- **Read-only** It specifies that the file can be opened only for reading purpose.
- **Hidden** If this attribute of a file is selected, then the file is hidden from the user.
- **Archive** If this attribute of a file is selected, then the back up of a file is created.

2.6.2 File Operations

File operations are the various tasks that are performed on files. A user can perform these operations by using the commands provided by the operating system. The following are some of the typical file operations:

- **Creating** It helps in creating a new file at the specified location in a computer system. The new file could be a Word document, an image file or an Excel worksheet.
- **Saving** It helps in saving the content written in a file at some specified location. The file can be saved by giving it a name of our choice.
- **Opening** It helps in viewing the contents of an existing file.
- **Modifying** It helps in changing the existing content or adding new to an existing file.
- **Closing** It helps in closing an already open file.
- **Renaming** It helps in changing the name of an existing file.
- **Deleting** It helps in removing a file from the memory of the computer system.

2.6.3 File Access Permissions

File access permissions help specify the manner in which a user can access a file. These are the access rights that allow us to read, write or execute a file. The following are some of the typical file access permissions:

- **Read** It allows a user to only read the content of an existing file.
- **Write** It allows a user to only modify the content of an existing file.
- **Execute** It allows a user to run an existing file stored in the computer system.

2.6.4 File Systems

File systems are used by an operating system to store and organise the various files and their information on a hard disk. The following are the two different file systems that are used to organise files in a computer system:

- **File Allocation Table (FAT)** It is a method used for organising the files and folders in the form of a table, which is known as FAT. This type of system is used for disks that are smaller in size and contain simple folders. The different types of FAT systems are FAT12, FAT16 and FAT32.
- **New Technology File System (NTFS)** This file system is specifically designed for large hard disks for performing basic file operations, such as reading, writing, modifying, saving, etc., quickly and efficiently. NTFS overcomes the drawbacks of the FAT system.

2.7 Types of Operating Systems

Many different types of operating systems have evolved till date. As the computers have improved in terms of speed, reliability, and cost so have the operating systems in terms of their capabilities. The operating systems supported by first generation computers were not very powerful. They were only designed and developed to cater the needs of a single user at a

time. Also, the users of these operating systems were capable of performing only one task at a time. However, there has been a tremendous amount of improvement in operating systems in the recent years. The modern-day operating systems allow multiple users to carry out multiple tasks simultaneously. Based on their capabilities and the types of applications supported, the operating systems can be divided into the following six major categories:

- Batch processing operating systems
- Multi-user operating systems
- Multitasking operating systems
- Real-time operating systems
- Multiprocessor operating systems
- Embedded operating systems

2.7.1 Batch Processing Operating Systems

The batch processing operating systems are capable of executing only one job at a time. The jobs or the programs submitted by different users are grouped into batches and one batch of jobs is provided as input to the computer system at a time. The jobs in the batch are processed on the first-come-first-serve basis. After getting an appropriate command from the operator, the batch processing operating system starts executing the jobs one-by-one. The execution of a particular job generally involves three major activities, which are reading the job from the input device, executing the job by the system and printing the calculated result on to the output device. After the execution of one job is complete, the operating system automatically fetches the next job from the batch without any human intervention.

The following are some of the advantages of batch processing operating systems:

- The computer systems employing the batch processing operating systems were very efficient computer systems of their times because the idle time for these systems was very small.
- These operating systems facilitated the execution of jobs in an organised manner.

The following are some of the disadvantages of batch processing operating systems:

- The jobs are processed only in the order in which they are placed in a batch and not as per their priority.
- The debugging of a program at execution time is not possible in these operating systems.
- The executing jobs may enter an infinite loop, as each job is not associated with a proper timer.

2.7.2 Multi-user Operating Systems

The multi-user operating systems enable multiple users to use the resources of a computer system at the same time. In other words, a multi-user operating system allows a number of users to work simultaneously on the same computer system. These types of operating systems are specially designed for the multi-user systems. A multi-user system is usually implemented by following the multi-terminal configuration. In this type of configuration, a single powerful computer system is connected to multiple terminals though serial ports. This computer system is responsible for processing the different requests generated by the various terminals at

the same time. The devices connected with the various terminals are keyboard, mouse, and monitor. The central computer system is equipped with a fast processor and a memory of large capacity for catering to the multiple requests of the end users. Examples of multi-user operating system include Unix, Linux, Windows 2000 and VM-386

The following are some of the advantages of the multi-user operating systems:

- It allows the resources of the computer system to be utilised in an efficient manner.
- It enhances the overall productivity of the various users by providing simultaneous access to the various computer resources.

The following are the disadvantages of the multi-user operating systems:

- The configuration of the computer system employing multi-user operating system is complex and hence, is difficult to handle and maintain.
- This type of system may result in an inconsistent data if the activities of one user are not protected from another user.
- This type of operating system is required to have robust security mechanisms.

2.7.3 Multitasking Operating Systems

The multitasking operating systems allow a user to carry out multiple tasks at the same time on a single computer system. The multitasking operating systems are also known as by several other names, such as multiprocessing, multiprogramming, concurrent or process scheduling operating systems. The first multitasking operating systems evolved during 1960s. The number of tasks or processes that can be processed simultaneously in this type of operating system depends upon various factors, such as the speed of the CPU, the capacity of the memory, and the size of the programs.

In this type of operating system, the different processes are executed simultaneously by implementing the concept of time slicing. According to this concept, a regular slice of CPU time is provided to each of the processes running in the computer system. Multitasking operating systems can be of two different types, which are preemptive multitasking operating systems and cooperative multitasking operating systems. In preemptive multitasking operating system, slices of CPU time are allocated to the various processes on some priority basis. These priorities are assigned to the various processes in such a manner that the overall efficiency of the system is maintained. In cooperative multitasking operating system, it strongly depends upon the processes whether or not to relinquish CPU control for other running processes. Examples of multitasking operating system include Unix, Linux, Windows 2000, and Windows XP.

The following are some of advantages of multitasking operating systems:

- It helps in increasing the overall performance of the computer system.
- It helps in increasing the overall productivity of the user by performing a number of tasks at the same time.

The following are some of the disadvantages of multitasking operating systems:

- A large amount of memory is required to execute several programs at the same time.
- Some mechanism needs to be implemented to ensure that the activities of one process do not interfere with the activities of another process.

2.7.4 Real-time Operating Systems

The real-time operating systems are similar to multitasking operating systems in their functioning. However, these operating systems are specially designed and developed for handling real-time applications or embedded applications. The real time applications are those critical applications that are required to be executed within a specific period of time. Therefore, time is the major constraint for these applications. The different examples of real-time applications are industrial robots, spacecrafts, industrial control applications and scientific research equipments.

The real-time operating systems can be of two different types, hard real-time operating system, and soft real-time operating system. In the hard real-time operating system, it is necessary to perform a task in the specified amount of time, i.e., within the given deadline. On the other hand, in the soft real-time operating system, a task can be performed even after its allocated time has elapsed.

The following are some of the examples of real-time operating system:

- MTOS
- Lynx
- RTX

The following are some of the advantages of the real-time operating systems:

- It is easy to design and develop and execute real-time applications under real-time operating system as compared to other types of operating systems.
- The real-time operating systems are usually more compact as compared to other operating systems. Thus, these systems require less memory space.

The following are some of the disadvantages of real-time operating systems:

- It is primarily focused on optimising the execution time of an application and thus, it sometimes overlooks some of the other critical factors related to the overall efficiency of the computer system.
- It is only used for providing some dedicated functionality, and thus, cannot be used as a general-purpose operating system.

2.7.5 Multiprocessor Operating Systems

The multiprocessor operating system allows the use of multiple CPUs in a computer system for executing multiple processes at the same time. By using more than one CPU, the processes are executed in a faster manner as compared to the computer systems performing multiprocessing with a single CPU.

The following are some of the examples of the multiprocessor operating system:

- Linux
- Unix
- Windows 2000

The following are some of advantages of multiprocessor operating systems:

- It helps in improving the overall performance and throughput of the computer system.

- It helps in increasing the reliability of the computer system. If one CPU of the computer system fails, the other CPU takes control and executes the currently running process.

The following are some of disadvantages of the multiprocessor operating systems:

- The cost of the computer systems employing multiprocessor operating systems is very high.
- A large amount of memory is required for running and executing several user programs.

2.7.6 Embedded Operating Systems

The embedded operating systems are somewhat similar to real-time operating systems. The embedded operating system is installed on an embedded computer system, which is primarily used for performing computational tasks in electronic devices. These operating systems provide limited functionality that is required for the corresponding embedded computer system. The other common functions that a usual operating system supports are not found in these operating systems.

The following are some of the examples of embedded operating systems:

- Palm OS
- Windows CE

The following are some of the advantages of embedded operating systems:

- These operating systems allow the implementation of embedded systems in an efficient manner.
- The computer system with embedded operating system is easy to use and maintain.

The following are some of the disadvantages of embedded operating systems:

- It is only possible to perform some specific operations with these operating systems.
- These operating systems cannot be used in frequently changing environments.

2.8 Popular Operating Systems

To date, many operating systems have been developed that suit different requirements of the users. Some of these operating systems became quite popular while others did not do well. The following are some of the popular operating systems:

- MS-DOS
- UNIX
- Windows

2.8.1 MS-DOS

MS-DOS was developed and introduced by Microsoft in 1981. It is a single-user and single-tasking operating system developed for personal computers. MS-DOS was specifically designed for the family of Intel 8086 microprocessors. This operating system provides a command line user interface, which means that a user needs to type a command at the command line for performing a specific task. The CLI of MS-DOS is more commonly known as DOS prompt. The

user interface of MS-DOS is very simple to use but not very user-friendly because of its non-graphical nature. The user has to issue a command to carry out even a simple task.

The command prompt of MS-DOS only allows the execution of the files with the extensions: .COM (Command files), .BAT (Batch files) and .EXE (Executable file). The structure of MS-DOS comprises the following programs:

- **IO.SYS** It is an important hidden and read only system file of MS-DOS that is used to start the computer system. It is also responsible for the efficient management and allocation of the hardware resources through the use of appropriate device drivers.
- **MSDOS.SYS** It is another hidden and read only system file that is executed immediately after the execution of IO.SYS file is finished. MSDOS.SYS acts as the kernel of MS-DOS. It is responsible for managing the memory, processors and the input/output devices of the computer system.
- **CONFIG.SYS** It is a system file that is used to configure various hardware components of the computer system so that they can be used by the various applications.
- **COMMAND.COM** It is the command interpreter that is used to read and interpret the various commands issued by the users.
- **AUTOEXEC.BAT** It is a batch file consisting of a list of commands that is executed automatically as the computer system starts up.

2.8.2 UNIX

UNIX is an operating system that allows several users to perform a number of tasks simultaneously. The first version of UNIX was introduced during the 1970s. However, since then, it is in constant development phase for further improving its functionality. UNIX operating system provides a GUI that enables its users to work in a more convenient environment. UNIX is most suitable for the computers that are connected to a Local Area Network (LAN) for performing scientific and business related operations. It can also be implemented on personal computers. The following are the core components of the UNIX operating system:

- **Kernel** It is the central part of the UNIX operating system that manages and controls the communication between the various hardware and software components of the computer system. The other major functions performed by the kernel are process management, memory management and device management.
- **Shell** It is the user interface of the UNIX operating system that acts as an intermediary between the user and the kernel of the operating system. Shell is the only program in UNIX operating system that takes the commands issued by the users and interprets them in an efficient manner to produce the desired result.
- **Files and processes** The UNIX operating system arranges everything in terms of files and processes. The directory in this operating system is also considered as a file that is used to house other files within it. The process is usually a program executed under the UNIX operating system. Several processes can be executed simultaneously in this operating system and are identified by a unique Process Identifier (PID) assigned to them.

Figure 2.13 shows the directory structure of UNIX operating system.

The UNIX operating system supports hierarchical directory structure in the form of a tree for arranging different files in the computer system. The root of the tree is always denoted by

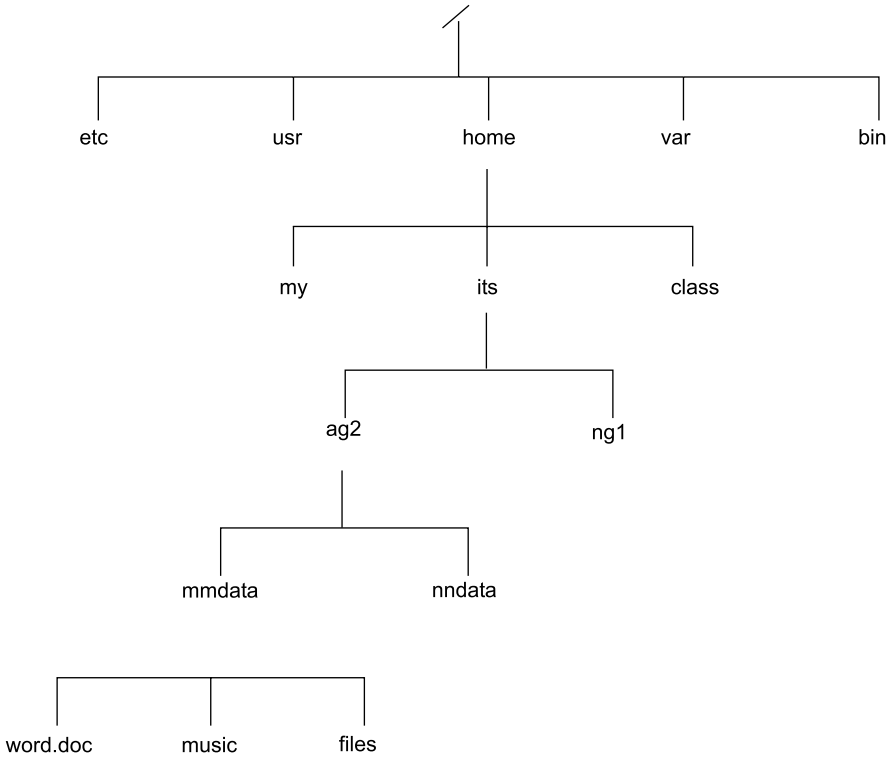


Fig. 2.13 ⇔ *UNIX directory structure*

slash (/). The current working directory of the user is denoted by home. There can be several home directories corresponding to the different users of the UNIX operating system. All the files and directories under the home directory belong to a particular user. The path of any file or directory in UNIX operating system always starts with the root (/). For example, the full path of the file word.doc can be represented as /home/its/ag2/mmdata/word.doc.

The following are some of the significant features of UNIX operating system:

- It allows multiple users to work simultaneously.
- It allows the execution of several programs and processes at the same time to ensure efficient utilisation of the processor.
- It implements the concept of virtual memory in an efficient manner. This feature enables the UNIX operating system to execute a program whose size is larger than the main memory of the computer system.

2.8.3 Windows

Microsoft has provided many operating systems to cater the needs of different users. Microsoft is a well known name in the development of operating system as well as various software applications. Initially, Microsoft introduced Windows 1.x, Windows 2.x and Windows 386 operating systems.

However, these operating systems lacked certain desirable features, such as networking and interactive user interface. Microsoft continued to work towards developing an operating system that met the desirable features of users and came up with a new operating system in the year 1993, which was known as Windows NT 3.1. This operating system was specially designed for the advanced users performing various business and scientific operations. After the release of Windows NT 3.1, several other operating systems were introduced by Microsoft in the successive years with their own unique features. Table 2.1 lists some of other important Windows operating system introduced by Microsoft with their release dates and significant features.

Table 2.1 *Microsoft Windows operating system*

Name of operating system	*Date of release*	*Significant features*
Windows 95	August, 1995	✹ 32-bit file system ✹ Multitasking ✹ Object Linking and Embedding (OLE) ✹ Plug and play ✹ Optimised memory management
Windows 98	June, 1998	✹ 32-bit Data Link Control (DLC) protocol ✹ Improved GUI ✹ Improved online communication through various tools, such as outlook express, personal web server and web publishing wizard ✹ Multiple display support ✹ Windows update
Windows 2000	February, 2000	✹ More reliable against application failures ✹ Improved Windows explorer ✹ Secure file system using encryption ✹ Microsoft Management Console (MMC) ✹ Improved maintenance operations
Windows ME	September, 2000	✹ System restoration against failure ✹ Universal plug and play ✹ Automatic updates ✹ Image preview
Windows XP	October, 2001	✹ Attractive desktop and user interface ✹ System restore ✹ Windows firewall ✹ Files and settings transfer wizard
Windows Server 2003	April, 2003	✹ Enhanced Internet Information Services (IIS) ✹ Enhanced Microsoft Message Queuing (MSMQ) ✹ Enhanced active directory support ✹ Watchdog timer
Windows Vista	November, 2006	✹ Multilingual user interface ✹ Enhanced search engine ✹ Enhanced Internet explorer ✹ Enhanced Windows media player ✹ Enhanced Windows update ✹ Windows system assessment tool

SUMMARY

- Operating system is a system software installed on a computer system that performs several key tasks, such as process management, memory management, device management, file management, etc. An operating system also secures the computer system from various threats such as virus and unauthorised access. There are different types of operating system available such as multi-user, batch processing and embedded. The multi-user operating system allows multiple users to use the computer system simultaneously. The batch processing operating system processes the jobs in groups called batches. The embedded operating system is installed on an embedded computer system, which is primarily used for performing computational tasks in electronic devices.
- An end user interacts with an operating system through a user interface, which is of two types, GUI or CLI. A GUI interface allows the end users to issue commands through point-and-click operations while a CLI interface allows the end users to issue commands only by typing them at the command prompt. Windows, UNIX and MS-DOS are some of the most popular operating systems. Windows is a GUI based operating system, while MS-DOS is a CLI based operating system.

Key Terms

- Operating system
- Process management
- Process state
- Process Control Block
- Process scheduling
- Process synchronisation
- Interprocess communication
- Deadlock
- Memory management
- File management
- Batch processing operating system
- Multi-user operating system
- Multitasking operating system
- Real-time operating system
- Multiprocessor operating system
- Embedded operating system
- Kernel
- Shell

Review Questions

2.1 *What is an operating system? Explain briefly with the help of examples.*

2.2 *Briefly state the history of operating system.*

2.3 *Briefly explain the various functions of an operating system.*

2.4 *What is a process state? Explain the various states of a process with the help of a figure.*

2.5 *What is a PCB and what information is contained in it?*

2.6 *Explain the process creation and process termination operations related to a process.*

2.7 *Briefly explain the term process scheduling.*

2.8 *Explain the three types of schedulers that help in process scheduling.*

2.9 *Which scheduling algorithms are used in process scheduling? Explain each one of them briefly.*

2.10 *Briefly explain the task of process synchronisation performed by an operating system.*

2.11 *What is interprocess communication? Explain the two methods used for interprocess communication.*

2.12 *What is a deadlock? Explain briefly the methods that can be used to handle this condition.*

2.13 *What is memory management? Explain briefly.*

2.14 *Briefly explain the function of file management performed by an operating system.*

2.15 *Briefly explain any three types of operating system.*

2.16 *State the programs that are a part of the MS-DOS structure.*

2.17 *Explain the core components of UNIX operating system.*

2.18 *Briefly explain why Windows operating system is one of the most popular operating systems.*

Fill in the Blanks

2.1 _____ is a system software that allows the users to interact with the hardware and other resources of a computer system.

2.2 In _____ interface, users type the commands pertaining to the tasks that they want to perform.

2.3 In _____ interface, commands are given by means of point-and-click operations performed using a pointing device, such as mouse.

2.4 The process of managing the files and directories contained in a computer system is known as _____.

2.5 The _____ state specifies the time when a process is ready for execution but has not been placed in the ready queue by the operating system.

2.6 _____ is a data structure associated with a process that provides complete information about the process.

2.7 The operating system maintains a table called _____, which stores the PCBs related to all the processes.

2.8 The process, which creates a new process using the system call, is called _____.

2.9 _____ is the task performed by an operating system for deciding the priority in which the processes, which are in ready and waiting states, are allocated the CPU time for their execution.

2.10 An operating system uses a program called _____ for deciding the priority in which a process is allocated the CPU time.

2.11 In _____ scheduling algorithm, a process is allocated the CPU for a specific time period called time slice or time quantum.

2.12 _____ is the task of synchronising the execution of processes in such a manner that no two processes have access to the same shared data or resource.

2.13 _____ is a condition that occurs when multiple processes wait for each other to free up resources and as a result all the processes remain halted.

2.14 The _____ function of an operating system helps in allocating the main memory space to the processes and their data at the time of their execution.

2.15 Paging is a technique in which the main memory of the computer system is organized in the form of equal sized blocks called _____.

2.16 In _____ operating system, jobs are grouped into groups called batches and assigned to the computer system with the help of a card reader.

2.17 In _____ operating system, multiple users can make use of computer system's resources simultaneously.

2.18 _____ is the central part of the UNIX operating system that manages and controls the communication between the various hardware and software components.

Multiple Choice Questions

2.1 Which of the following program is essential for the functioning of a computer system?

A. MS Word
B. Operating system
C. MS Excel
D. System software

2.2 Which of the following operating systems makes use of CLI?

A. MS-DOS
B. Windows 2000
C. Windows Server 2003
D. None of the above

2.3 Which of the following functions is provided by an operating system?

A. Process management
B. Security management
C. File management
D. All of the above

2.4 Which of the following provides complete information related to a process?

A. Process state
B. Process scheduling
C. Process communication
D. PCB

2.5 Which of the following is an operation related to a process?

A. Process creation
B. Process execution
C. Process completion
D. None of the above

2.6 Which of the following is responsible for ascertaining the order in which processes are executed?

A. Scheduler
B. Process schedule manager
C. Operating system scheduler
D. Process Scheduler

2.7 Which one of the following scheduling algorithm is the simplest algorithm for the scheduling of processes?

A. FCFS scheduling algorithm
B. RR scheduling algorithm
C. Priority scheduling algorithm
D. SJF scheduling algorithm

2.8 Which of the following methods is used for interprocess communication?

A. Shared cache memory
B. Shared region
C. Message passing
D. Cache memory

2.9 Which one of the following methods is used for handling of deadlocks?

A. Detect and recover deadlock
B. Mutual exclusion
C. No preemption
D. All of the above

2.10 Which one of the following is a file operation?

A. Opening a file
B. Manipulating a file
C. Resising a file
D. All of the above

2.11 Which one of the following types of the operating systems allows multiple users to work simultaneously?

A. Multitasking operating system
B. Multi-user operating system
C. Multiprocessor operating system
D. None of the above

2.12 Which of the following is a part of MS-DOS?

A. DOS.SYS
B. CONFIGURATION.SYS
C. EXEC.BAT
D. COMMAND.COM

2.13 Which of the following is the core component of UNIX?

A. Command shell
B. Kernel
C. Directories and programs
D. None of the above

3

Programming Languages

Key Concepts

- Programming Languages
- History
- Generations
- Characteristics of a Good Programming Language
- Categorization
- Popular High Level Languages
- Fortran
- LISP
- COBOL
- BASIC
- PASCAL
- C
- C++
- JAVA
- Python
- C#
- Factors affecting choice of a language
- Developing a Program
- Running a Program

3.1 Introduction

Computers can perform a variety of tasks. However, they cannot perform any of them on their own. As we know, computers have no commonsense and they cannot think. They need clear-cut instructions to tell them *what to do, how to do and when to do.* A set of instructions to carry out these functions is called a *computer program*.

The communication between two parties, whether they are machines or human beings, always needs a common language or terminology. The language used in the communication of instructions to a computer is known as *computer language* or *programming language*. There are many different types of languages available today. A computer program can be written using any of the programming languages depending upon the task to be performed and the knowledge of the person developing the program. The process of writing instructions using a computer language is known as *programming or coding*. The person who writes such instructions is referred as a *programmer*.

We know that natural languages such as English, Hindi or Tamil have a set of characters and use some rules known as *grammar* in framing sentences and statements. Similarly, set of characters and rules known as *syntax* that must be adhered to by the programmers while developing computer programs.

Although, during the initial years of computer programming. All the instructions were written in the machine language, a large number of different type of programming languages have been developed during the last six decades. Each one of them has its own unique features and specific applications. In this chapter, we shall discuss briefly the various types of programming languages, their evolution and characteristics and how they are used to solve a problem using a computer.

3.2 History of Programming Languages

The history of programming languages is interlinked with the evolution of computer systems. As the computer systems became smaller, faster and cheaper with time, the programming languages also became more and more user friendly. Ada Augusta Lovelace, a companion of Charles Babbage, was considered as the *first computer programmer* in the history of programming languages. In the year 1843, Ada Augusta Lovelace wrote a set of instructions to program the *analytical engine* designed by Charles Babbage. This computer program was used to transform the data entered by the users into binary form before being processed by the computer system. This program increased the efficiency and the productivity of the analytical engine by automating various mathematical tasks. Later, in the year 1946, Konrad Zuse, a German engineer, developed a programming language known as **Plankalkul**. It was developed to target the various scientific, business and engineering needs of the users. It was considered as the first complete programming language that supported various programming constructs and the concept of data structures as well. The various programming constructs were implemented in this programming language with the help of Boolean algebra.

During the 1940s, *machine languages* were developed to program the computer system. The machine languages which used binary codes 0s and 1s to represent instructions were regarded as *low-level programming languages*. The instructions written in the machine language could be executed directly by the CPU of the computer system. These languages were hardware dependent languages. Therefore, it was not possible to run a program developed for one computer system in another computer system. This is because of the fact that the internal architecture of one computer system may be different from that of another. The development of programs in machine languages was not an easy task for the programmers. One was required to have thorough knowledge of the internal architecture of the computer system before developing a program in machine language.

During the 1950s, *assembly language*, which is another low-level programming language, was developed to program the computer systems. The assembly language used the concept of mnemonics to write the instructions of a computer program. Mnemonics refer to the symbolic names that are used to replace the machine language code. The programmers enjoyed working with this programming language because it was easy to develop a program in the assembly language as compared to the machine language. However, unlike machine language programs, assembly language programs could not be directly executed by the CPU of the computer system and required some a software program to convert these programs into machine understandable form.

During the period between 1950 and 1960, many high-level programming languages were developed to cater to the needs of the users of various disciplines, such as business, science and engineering. In 1951, Grace Hopper, an American computer scientist, started working towards designing a compiler called **A-0**, and in the year 1957, developed a high-level programming language known as **MATH-MATIC**. In 1952, another programming system known as **AUTOCODE** was developed by Alick E. Glennie. Grace Hopper was considered as the first person who had put some serious efforts towards the development of a high-level programming language.

In the year 1957, another popular high-level programming language known as **FORTRAN** (FORmula TRANslation) was developed. During its era, it was the only high-level programming language that became hugely popular among its users. FORTRAN was developed by John Backus and his team at International Business Machines (IBM). FORTAN was best suited for solving problems related to scientific and numerical analysis field. Another high-level programming language known as ALGOL (Algorithm Language) was developed in the year 1958. Some other high-level languages that evolved during this era were LISt Processing (LISP) in 1958, Common Business Oriented Language (COBOL) in 1959 and ALGOL 60 in 1960.

In the next decade, from 1960 to 1970, more high-level programming languages evolved. In the year 1964, the Beginners All-Purpose Symbolic Instruction Code (BASIC) was designed by John G. Kemeny and Thomos E. Kurtz at Dartmouth college. It was a general-purpose programming language that was very simple to use. In the same year, another powerful high-level programming language, **PL/1** with many rich programming features such as complex data type and methods, was designed for developing engineering and business applications. PL/I was considered to have the best features of its ancestor programming languages: COBOL, FORTRAN, and ALGOL 60. The other programming languages that evolved during this era were **Simula I, Simula 67, Algol 68 and APL**.

The period between 1970 and 1980 was actually the golden era for the development high-level programming languages. This period saw the birth of many general-purpose and powerful high-level programming languages. In the early 1970s, a procedural programming language, **Pascal** was developed by Niklaus Wirth. This programming language was provided with strong data structures and pointers, which helped in utilizing the memory of the computer system in an efficient manner. In the year 1972, Dennis Ritchie developed a powerful procedural and block structured programming language known as C. C is still very popular among the users for developing system as well as application software. In 1974, IBM developed Structured Query Language (SQL) that was used for performing various operations on the databases, such as creating, retrieving, deleting and updating. Apart from these programming languages, some

other high-level programming languages that evolved during this era were **Forth, Smalltalk**, and **Prolog**.

During the next decade, from 1980 to 1990, the focus of development of high-level programming languages shifted towards enhancing the performance and design methodology. The languages of this period used modular approach for designing large-scale applications. The modular approach of program design can be regarded as a design methodology, which divides the whole system into smaller parts that could be developed independently. This approach of designing software applications is still employed by modern programming languages. Some of the high-level programming languages that evolved during this era include **Ada, C++, Perl** and **Eiffel**.

The high-level programming languages developed and designed in the 1990s are considered as the fifth generation programming languages. During this period, Internet technology evolved tremendously. Therefore, the basic purpose of the programming languages of this period was to develop web-based applications. However, these languages could also be used for the development of desktop applications. The important high-level programming languages of this era are **Java**, **VB** and **C#**. Most of the programming languages of this era employed object-oriented programming paradigm for designing and developing robust and reliable software applications.

Table 3.1 summarises the history of development of programming languages:

Table 3.1 *The evolution of programming languages*

Period of employment	*Programming language*	*Characteristics*
1940s	Machine language	✹ Machine dependent ✹ Faster execution ✹ Difficult to use and understand ✹ More prone to errors
1950s	Assembly language	✹ Machine dependent ✹ Faster execution ✹ More prone to errors ✹ Relatively simple to use
1950–1970	FORTRAN, LISP, COBOL, ALGOL 60, BASIC, APL	✹ High-level languages ✹ Easy to develop and understand programs ✹ Less prone to errors
1970–1990	C, C++, Forth, Prolog, Smalltalk, Ada, Perl, SQL	✹ Very high-level languages ✹ Easier to learn ✹ Highly portable
1990s	Java, HTML, VB, PHP, XML, C #	✹ Internet-based languages ✹ Object-oriented languages ✹ More efficient ✹ Reliable and robust

3.3 Generations of Programming Languages

Programming languages have been developed over the years in a phased manner. Each phase of development has made the programming languages more user-friendly, easier to use and more *powerful*. Each phase of improvement made in the development of the programming languages can be referred as a *generation*. The programming languages, in terms of their performance, reliability and robustness can be grouped into five different generations.

- First generation languages (1GL)
- Second generation languages (2GL)
- Third generation languages (3GL)
- Fourth generation languages (4GL)
- Fifth generation languages (5GL)

3.3.1 First Generation: Machine Languages

The first generation programming languages are also called *low-level programming language* because they were used to program the computer systems at a very low level of abstraction, i.e., at the machine-level. The machine language, also referred as the native language of the computer system, is the first generation programming language. In the machine language, a programmer can issue the instructions to the computer system in the binary form only. Therefore, machine language programming only deals with two numbers, 0 and 1. The machine language programs are entered into the computer system by setting the appropriate switches available in the front panel system. These switches are actually the devices used to alter the course of the flow of electric current. The enable state of the switch represents the binary value, 1 and the disable state of the switch represents the binary value, 0. The programs written in the machine language are directly executed by the CPU of the computer system and therefore, unlike the modern programming languages, there is no need of using a translator in a machine language. Figure 3.1 shows the typical instruction format of the machine language instruction.

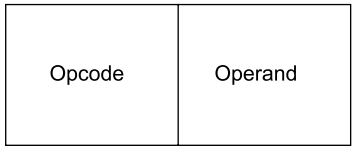


Fig. 3.1 ⇔ *Machine instruction format*

As seen in the figure, the instruction in the machine language is made up of two parts only, opcode and operand. The opcode part of the machine language instruction specifies the operation to be performed by the computer system and the operand part of the machine language instruction specifies the data on which the operation is to be performed. However, the instruction format of any instruction in the machine language strongly depends upon the CPU architecture.

The advantages of the first generation programming languages are:

- They are translation free and can be directly executed by the computers.
- The programs written in these languages are executed very speedily and efficiently by the CPU of the computer system.
- The programs written in these languages utilise the memory in an efficient manner because it is possible to keep track of each bit of data.

There are many disadvantages of using the first generation programming languages. They include:

- It is very difficult to develop a program in the machine language.
- The programs developed in these languages cannot be understood very easily by a person, who has not actually developed these programs.
- The programs written in these languages are so prone to frequent errors that they are very difficult to maintain.
- The errors in the programs developed in these languages cannot be detected and corrected easily.
- A programmer has to write a large number of instructions for executing even a simple task in these languages. Therefore, we can say that these languages result in poor productivity while developing programs.
- The programs developed in these languages are hardware dependent and thus they are non-portable.

Due to these limitations, machine languages are very rarely used for developing application programs.

3.3.2 Second Generation: Assembly Languages

Like the first generation programming languages, the second generation programming languages also belong to the category of *low-level programming languages*. The second generation programming languages comprise of assembly languages that use the concept of mnemonics for writing programs. Similar to the machine language, the programmer of assembly language needs to have internal knowledge of the CPU registers and the instructions set before developing a program. In the assembly language, symbolic names are used to represent the opcode and the operand part of the instruction. For example, to move the contents of the CPU register, **a1** to another CPU register, **b1** the following assembly language instruction can be used:

```
mov b1, a1
```

The above code shows the use of symbolic name, **mov** in an assembly language instruction. The symbolic name, **mov** instructs the processor to transfer the data from one register to another. Using this symbolic name, a value can also be moved to a particular CPU register.

The use of symbolic names made these languages little bit user-friendly as compared to the first generation programming languages. However, the second generation languages were still machine-dependent. Therefore, one was required to have adequate knowledge of the internal architecture of the computer system while developing programs in these languages.

Unlike the machine language programs, the programs written in the assembly language cannot be directly executed by the CPU of the computer system because they are not written in the binary form. As a result, some mechanism is needed to convert the assembly language programs into the machine understandable form. A software program called *assembler* is used to accomplish this purpose. An assembler is a *translator program* that converts the assembly language program into the machine language instructions. Figure 3.2 shows the role of an assembler in executing an assembly language program.

An assembler acts as an intermediary between the assembly language program and the machine language program. It takes a program written in the assembly language as input and generates the corresponding machine language instructions as output.

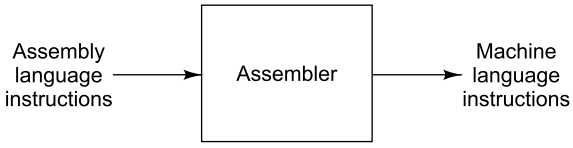


Fig. 3.2 ⇔ *Functioning of an assembler*

The following are some of the advantages of second generation programming languages:

- It is easy to develop, understand and modify the programs developed in these languages as compared to those developed in the first generation programming languages.
- The programs written in these languages are less prone to errors, and therefore can be maintained with great ease.
- The detection and correction of errors is relatively easy in these languages in comparison to the first generation programming languages.

The following are some of the disadvantages of the second generation programming languages:

- The programs developed in these languages are not executed as quickly as the programs developed in the machine language. This is because of the fact that the computer system needs to convert these programs into machine language before executing them.
- The programs developed in these languages are not portable as these languages are machine dependent.
- The programmer of these languages needs to have thorough knowledge of the internal architecture of the CPU for developing a program.
- The assembly language programs, like the machine language programs, still result in poor productivity.

3.3.3 Third Generation: High-level Languages

The third generation programming languages were designed to overcome the various limitations of the first and second generation programming languages. The languages of the third and later generations are considered as *high-level programming languages* because they enable the programmer to concentrate only on the logic of the program without concerning about the internal architecture of the computer system. In other words, we can also say that these languages are machine independent languages.

The third generation programming languages are also quite user-friendly because they relieve the programmer from the burden of remembering operation codes and instruction sets while writing a program. The instructions used in the third and later generations of languages can be specified in English-like sentences, which are easy to comprehend for a programmer. The programming paradigm employed by most of the third generation programming languages was *procedural programming*, which is also known as *imperative programming*. In the procedural programming paradigm, a program is divided into a large number of procedures, also known as *subroutines*. Each procedure contains a set of instructions for performing a specific task. A procedure can be called by the other procedures while a program is being executed by the computer system.

The third generation programming languages were considered as domain-specific programming languages because they were designed to develop software applications for a specific field. For example, the third generation programming language, COBOL, was designed to solve a large number of problems related to the business field only.

Unlike the assembly language, the programs developed in the third and the later generation of programming languages were not directly executed by the CPU of the computer system. These programs require translator programs for converting them into machine language. There are two types of translator programs, namely, *compiler* and *interpreter*. Figure 3.3 shows the translation of a program developed in the high-level programming language into the machine language program.

Fig. 3.3 ⇔ *Functioning of a compiler and an interpreter*

A program written in any high-level language can be converted by the compiler or the interpreter into the machine-level instructions. Both the translator programs, compiler and interpreter, are used for the same purpose except for one point of difference. The compiler translates the whole program into the machine language program before executing any of the instructions. If there are any errors, the compiler generates error messages which are displayed on the screen. All errors must be rectified before compiling again. On the other hand, the interpreter executes each statement immediately after translating it into the machine language instruction. Therefore, the interpreter performs the translation as well as the execution of the instructions simultaneously. If any error is encountered, the execution is halted after displaying the error message.

The following are some of the popular third generation programming languages:

- FORTRAN
- ALGOL
- BASIC
- COBOL
- C/C++

The following are some of the advantages of the third generation programming languages:

- It is easy to develop, learn and understand the programs.
- The programs developed in these languages are highly portable as compared to the programs developed in the first and second generation programming languages. Hence, we can also say that the third generation programming languages are machine independent programming languages.
- The programs written in these languages can be developed in very less time as compared to the first and second generation programming languages. This is because of the fact that the third generation programming languages are quite user-friendly and provide necessary inbuilt tools required for developing an application.
- As the programs written in these languages are less prone to errors, they are easy to maintain.

- The third generation programming languages provide more enhanced documentation and debugging techniques as compared to the first and the second generation programming languages.

The following are some of the disadvantages of the third generation programming languages:

- As compared to the assembly and the machine language programs, the programs written in the third generation programming languages are executed slowly by the computer system.
- The memory requirement of the programs written in these programming languages is more as compared to the programs developed using the assembly and machine languages.

3.3.4 Fourth Generation: Very High-level Languages

The languages of this generation were considered as very high-level programming languages. The process of developing software using the third generation programming languages required a lot of time and effort that affected the productivity of a programmer. Moreover, most of the third generation programming languages were domain-specific. The fourth generation programming languages were designed and developed to reduce the time, cost and effort needed to develop different types of software applications. Most of the fourth generation programming languages were general-purpose programming languages. This means that most of the fourth generation programming languages could be used to develop software applications related to any domain. During this generation, the concept of Database Management System (DBMS) also evolved tremendously. Therefore, most of the fourth generation programming languages had database related features for working with databases. These languages have simple, English-like syntax rules. Since 4GLs are non-procedural languages, they are easier to use and therefore more user-friendly. We need to specify WHAT is required rather than specifying How to do it.

The following are some of the fourth generation programming languages:

- PowerBuilder
- SQL
- XBase++
- CSS
- ColdFusion

Apart from being machine independent, the following are some of the other important advantages of the fourth generation programming languages:

- The fourth generation programming languages are easier to learn and use as compared to the third generation programming languages.
- These programming languages require less time, cost and effort to develop different types of software applications.
- These programming languages allow the efficient use of data by implementing various database concepts.
- As compared to the third generation programming languages, these languages required less number of instructions for performing a specific task.

- The programs developed in these languages are highly portable as compared to the programs developed in the languages of other generations.

The following are some of the disadvantages of the fourth generation programming languages:

- As compared to the programs developed in the programming languages of previous generations, the programs developed in the 4GLs are executed at a slower speed by the CPU.
- As compared to the third generation programming languages, the programs developed in these programming languages require more space in the memory of the computer system.

3.3.5 Fifth Generation: Artificial Intelligence Languages

The programming languages of this generation mainly focus on constraint programming. The constraint programming, which is somewhat similar to declarative programming, is a programming paradigm in which the programmer only needs to specify the solution to be found within the constraints rather than specifying the method or algorithm of finding the desired solution. The major fields in which the fifth generation programming languages are employed are Artificial Intelligence (AI) and Artificial Neural Network (ANN). AI is the branch of computer science in which the computer system is programmed to have human intelligence characteristics. It helps make computer system so intelligent that it can take decisions on its own while solving various complicated problems. On the other hand, ANN refers to a network that is used to imitate the working of a human brain. ANN is widely used in voice recognition systems, image recognition systems and industrial robotics.

The following are some of the fifth generation programming languages:

- Mercury
- Prolog
- OPS5

The following are two important advantages of fifth generation programming languages:

- The fifth generation languages allow the users to communicate with the computer system in a simple and an easy manner. Programmers can use normal English words while interacting with the computer system.
- These languages can be used to query the databases in a fast and efficient manner.

3.4 Characteristics of a Good Programming Language

The popularity of any programming language depends upon the useful features that it provides to its users. A large number of programming languages are in existence around the world but not all of them are popular. The following are some of the important characteristics of a good programming language:

- The language must allow the programmer to write simple, clear and concise programs.
- The language must be simple to use so that a programmer can learn it without any explicit training.

- The glossary used in the language should be very close to the one used in human languages.
- The function library used in the language should be well documented so that the necessary information about a function can be easily obtained while developing an application.
- The various programming constructs supported by the language must match well with the application area it is being designed for.
- The language must allow the programmer to focus only on the design and the implementation of the different programming concepts without requiring the programmer to be well acquainted with the background details of the concepts being used.
- The programs developed in the language must make efficient use of memory as well as other computer resources.
- The language must provide necessary tools for development, testing, debugging and maintenance of a program. All these tools must be incorporated into a single environment known as Integrated Development Environment (IDE), which enables the programmer to use them easily.
- The language must be platform independent, i.e., the programs developed using the programming language can run on any computer system.
- The Graphical User Interface (GUI) of the language must be attractive, user-friendly and self explanatory.
- The language must be object-oriented in nature so as to provide various features such as inheritance, information hiding, and dynamic binding to its programmers.
- The language must be consistent in terms of both syntax and semantics.

3.5 Categorisation of High-level Languages

The high-level languages can be categorised into different types on the basis of the application areas in which they are employed, as well as the different design paradigms supported by them. Figure 3.4 shows the different types of high-level languages categorised on the basis of application areas and design paradigms.

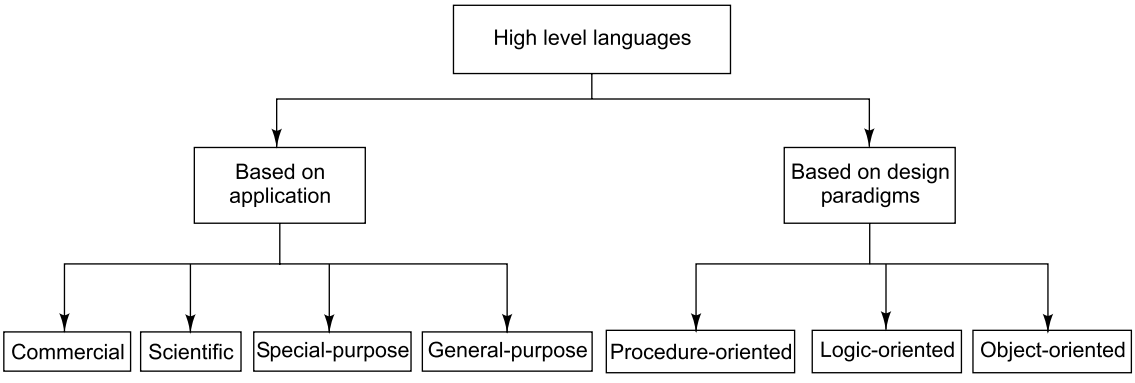


Fig. 3.4 ⇔ *Types of high-level languages*

The figure clearly shows that the high-level programming languages are designed for use in a number of areas. Each high-level language is designed by keeping its target application area in mind. Some of the high-level languages are best suited for business domain, while others are apt in scientific domain only.

The high-level programming languages can also be categorised on the basis of the various programming paradigms supported by them. The programming paradigm refers to the approach employed by the programming languages for solving the different types of problems.

3.5.1 Categorisation Based on Application

On the basis of application area, the high-level programming languages can be divided into the following types:

- **Commercial languages.** These programming languages are dedicated to the commercial domain and are specially designed for solving business-related problems. These languages can be used in organisations for processing and handling the data related to payroll, accounts payable and tax handling applications. COBOL is the best example of the commercial-based high-level programming language employed in the business domain. This language was developed with strong file handling capabilities and support for business arithmetic operations. Another example of business-oriented programming language is Programming Language for Business (PL/B), which was developed by Datapoint during the 1970s.
- **Scientific languages.** These programming languages are dedicated to the scientific domain and are specially designed for solving different scientific and mathematical problems. These languages can be used to develop programs for performing complex calculations during scientific research. FORTRAN is the best example of the scientific-based high-level programming language. This language is capable of performing various numerical and scientific calculations.
- **Special-purpose languages.** These programming languages are specially designed for performing some dedicated functions. For example, SQL is a high-level language specially designed to interact with the database programs only. Therefore, we can say that the special-purpose high-level programming languages are designed to support a particular domain area only.
- **General-purpose languages.** These programming languages are used for developing different types of software applications regardless of their application area. The various examples of general-purpose high-level programming languages are BASIC, C, C++ and Java.

3.5.2 Categorisation Based on Design Paradigm

On the basis of design paradigm, the high-level programming languages can be categorised into the following types:

- **Procedure-oriented languages.** These programming languages are also called *imperative programming languages*. In these languages, a program is written as a sequence of procedures. Each procedure contains a series of instructions for performing

a specific task. Each procedure can be called by the other procedures during the program execution. In this type of programming paradigm, a code once written in the form of a procedure can be used any number of times in the program by only specifying the corresponding procedure name. This approach also makes the program structure relatively very simple to follow as compared to the other programming paradigms. We can also say that the major emphasis of these languages is on the procedures and not on the data. Therefore, the procedure-oriented languages allow the data to move freely around the system. The various examples of procedure-oriented languages are FORTRAN, ALGOL, C, BASIC and Ada.

- **Logic-oriented languages.** These languages use *logic programming paradigm* as the design approach for solving various computational problems. In this programming paradigm, predicate logic is used to describe the nature of a problem by defining relationships between rules and facts. Prolog is the best example of the logic-oriented programming language.
- **Object-oriented languages.** These languages use *object-oriented programming paradigm* as the design approach for solving a given problem. In this programming paradigm, a problem is divided into a number of objects, which can interact by passing messages to each other. The other features included in the object-oriented languages are encapsulation, polymorphism, inheritance and modularity. C++, JAVA and C# are the examples of object-oriented programming language.

3.6 Popular High-level Languages

Today, a large number of high-level programming languages are available for developing different types of software applications. However, only few of these programming languages are popular among programmers. The following are some of the popular high-level programming languages used around the world:

- FORTRAN
- LISP
- COBOL
- BASIC
- PASCAL
- C
- C++
- Java
- Python
- C#

3.6.1 FORTRAN

FORTRAN is the most dominant high-level programming language employed in the science and engineering domain. As mentioned earlier, FORTRAN was initially developed by a team led by John Backus at IBM in the 1950s. Since then several new versions of FORTRAN have evolved. They include FORTRAN II, FORTRAN IV, FORTRAN 77 and FORTRAN 90.

FORTRAN 90, which is approved by the International Organisation for Standardisation, is more portable, reliable and efficient as compared to its earlier versions. The following are some of the important applications areas where FORTRAN can be employed:

- Finding solutions to partial differential equations
- Predicting weather
- Solving problems related to fluid mechanics
- Solving problems related to physics and chemistry

Some of the most significant characteristics of FORTRAN are enumerated as under:

- It is easier to learn as compared to the other scientific high-level languages.
- It has a powerful built-in library containing some useful functions, which are helpful in performing complex mathematical computations.
- It enables the programmers to create well-structured and well-documented programs.
- The internal computations in this language are performed rapidly and efficiently.
- The programs written in this language can be easily understood by other programmers, who have not actually developed the programs.

3.6.2 LISP

LISP (List Processing) was developed by John McCarthy in the year 1958 as a functional programming language for handling data structures known as *lists*. LISP is now extensively used for research in the field of artificial intelligence (AI). Some of the versions of LISP are Standared LISP, MACLISP, Inter LISP, Zeta LISP and Common LISP.

Some good features of LISP are:

- Easy to write and use.
- Recursion, that is a programming calling itself, is possible.
- Supports grabage collection.
- Supports interactive computing.
- Most suitable for AI applications.

Some negative aspects of LISP are:

- Poor reliability.
- Poor readability of programs.
- Not a general-purpose language.

3.6.3 COBOL

COBOL is a high-level programming language developed in the year 1959 by COnference on DAta SYstems Languages (CODASYL) committee. This language was specially designed and developed for the business domain. Apart from the business domain, COBOL can also be used to develop the programs for the various other applications. However, this language cannot be employed for developing various system software such as operating systems, device drivers etc. COBOL has gone through a number of improvement phases since its inception and, as a result, several new versions of COBOL have evolved. The most significant versions of COBOL,

which are standardised by American National Standards Institute (ANSI), are COBOL-68, COBOL-74 and COBOL-85.

Some of the most significant characteristics of COBOL are enumerated as follows:

- The applications developed in this language are simple, portable and easily maintainable.
- It has several built-in functions to automate the various tasks in business domain.
- It can handle and process a large amount of data at a time and in a very efficient manner.
- As compared to the other business-oriented high-level programming languages, the applications can be developed rapidly.
- It does not implement the concept of pointers, user-defined data types, and user-defined functions and hence is simple to use.

3.6.4 BASIC

BASIC (Beginner's All-purpose Symbolic Instruction Code) was developed by John Kemeny and Thomas Kurty at Dartmaith Collge, USA in the year 1964. The language was extensively used for microcomputers and home computers during 1970s and 1980s. BASIC was standardized by ANSI in 1978 and became popular among business and scientific users alike. BASIC continues to be widely used because it can be learned quickly.

During the last four decades, different versions of BASIC have appeared. These include Altair BASIC, MBASIC, GWBASIC, Quick BASIC, Turbo BASIC and Visual BASIC. Microsoft's Visual BASIC adds object-oriented features and a graphical user interface to the standard BASIC.

Main features of BASIC are:

- It is the first interpreted language.
- It is a general-purpose language.
- It is easy to learn as it uses common English words.

3.6.5 PASCAL

PASCAL is one of the oldest high-level programming languages developed by Niklaus Wirth in the year 1970. It was the most efficient and productive language of its time. The programming paradigm employed by PASCAL is procedural programming. A number of different versions of PASCAL have evolved since 1970 that help in developing the programs for various applications such as research projects, computer games and embedded systems. Some of the versions of PASCAL include USCD PASCAL, Turbo PASCAL, Vector PASCAL and Morfik PASCAL. This programming language was also used for the development of various operating systems such as Apple Lisa and Mac.

Some of the most significant characteristics of PASCAL are enumerated as under:

- It is simple and easy to learn as compared to the other high-level programming languages of its time.
- It enables the programmers to develop well-structured and modular programs that are easy to maintain and modify.
- The data in this language is stored and processed efficiently with the help of strong data structures.

- It enables the programmer to create the data types according to their requirements that are also referred as user-defined data types.
- The PASCAL compiler has strong type checking capability that prevents the occurrence of data type mismatch errors in a program.

3.6.6 C

C is a general-purpose high-level programming language developed by Dennis Ritchie and Brain Kernighan at Bell Telephone Laboratories, the USA in the year 1972. C is a well-known high-level programming language that is used for developing the application as well as system programs. It is also block-structured and procedural, which means that the code developed in C can be easily understood and maintained. C is the most favourite language of system programmers because of its several key characterises, that are hardly found in other high-level programming languages. The first major system program developed in C was the UNIX operating system. C is also regarded as a middle-level language because it contains the low-level as well as the high-level language features.

Some of the most significant characteristics of C are:

- C is machine and operating system independent language. Therefore, the programs developed in C are highly portable as compared to the programs developed in the other high-level programming languages.
- It is a highly efficient programming language because the programs developed in this language are executed very rapidly by the CPU of the computer system. Also, the memory requirement for the storage and the processing of C programs is comparatively less. Therefore, C is considered to be equivalent to assembly language in terms of efficiency.
- It can be used to develop a variety of applications; hence, it is considered to be quite flexible.
- It allows the programmer to define and use their own data types.
- C allows the use of pointers that allows the programmers to work with the memory of the computer system in an efficient manner.

3.6.7 C++

C++ is a general-purpose, object-oriented programming language developed by Bjarne Stroustrup at Bell Labs in the year 1979. Initially, Bjarne Stroustrup named his new language as C with classes because this new language was the extended version of the existing programming language, C. Later, this new language was renamed as C++. It is also regarded as the superset of the C language because it retains many of its salient features. In addition to having the significant features of C, C++ was also expanded to include several object-oriented programming features, such as classes, virtual functions, operator overloading, inheritance and templates.

Some of the most significant characteristics of C++ are as follows:

- It uses the concept of objects and classes for developing programs.
- The code developed in this language can be reused in a very efficient and productive manner.

- Like C, C++ is also a machine and operating system independent language. Therefore, the programs developed in this language are highly portable.
- It is a highly efficient language in terms of the CPU cycles and memory required for executing different programs.
- The number of instructions required to accomplish a particular task in C++ is relatively lesser as compared to some of the other high-level programming languages.
- It follows the modular approach of developing the programs for the different types of applications. Therefore, the programs developed in C++ can be understood and maintained easily.
- C++ is highly compatible with its ancestor language, i.e., C because a program developed in C can be executed under the C++ compiler with almost no change in the code.

3.6.8 JAVA

JAVA is an object-oriented programming language introduced by Sun Microsystems in the year 1995. It was originally developed in the year 1991 by James Gosling and his team. The syntax and the semantics of JAVA are somewhat similar to C++. However, it is regarded as more powerful than C++ and the other high-level programming languages. In the current scenario, JAVA is the most dominant object-oriented programming language for developing web-based applications. Apart from the web-based applications, JAVA can also be employed to develop other types of applications, such as desktop applications and embedded systems applications.

JAVA is a highly platform independent language because it uses the concept of *just-in-time* compilation. In this type of compilation, the JAVA programs are not directly compiled into the native machine code. Instead, an intermediate machine code called *bytecode* is generated by the JAVA compiler that can be interpreted on any platform with the help of a program known as *JAVA interpreter*.

Some of the most significant characteristics of JAVA are enumerated as under:

- It is a highly object-oriented and platform independent language.
- The programs written in this language are compiled and interpreted in two different phases.
- The programs written in this language are more robust and reliable.
- It is more secure as compared to the other high-level programming languages because it does not allow the programmer to access the memory directly.
- It assists the programmers in managing the memory automatically with a feature called garbage collection.
- It also implements the concept of dynamic binding and threading in a better and efficient manner as compared to other object-oriented languages.

3.6.9 Python

Python is a high-level and object-oriented programming language developed by Guido Van Rossum in the year 1991. It is a general-purpose programming language that can be used to develop software for a variety of applications. Python is also regarded as the successor

language of ABC programming language. ABC was a general-purpose programming language developed by a team of three scientists, leo Geurts, Lambert Meertens, and Steven Pemberton. Several versions of Python have been evolved since 1991. Some of the versions of Python are Python 0.9, Python 1.0, Python 1.2, Python 1.4, Python 1.6 and Python 2.0.

Python has a strong built-in library for performing various types of computations. This built-in library also makes Python simple and easy to learn. Python is an interpreted language and its interpreter as well as other standard libraries are freely available on the Internet. The programs developed in this language can be run on different platforms and under different operating systems. Hence, Python is regarded as platform independent language.

Some of the salient features of Python are:

- It is an interpreted and object-oriented programming language.
- It implements the concept of exception handling and dynamic binding better than the other languages of its time.
- The syntax and the semantics of this language are quite clear and concise.
- It is a platform independent language.

3.6.10 C#

C#, pronounced as "C-sharp" is a new object-oriented programming language developed by Microsoft late in the 1990s. It combines the power of C++ with the programming ease of Visual BASIC. C# is directly descended from C++ and contains features similar to those of JAVA.

C# was specially designed to work with Microsoft's .NET platform launched in 2000. This platform offers a new software-development model that allows applications developed in different languages to communicate with each other. C# includes several modern programming features that include:

- concise, lean and modern language
- object-oriented visual programming
- component-oriented language
- multimedia (audio, animation and video) support
- very good exception handling
- suitable for Web-based applications
- language interoperability
- more type safe than C++.

As C# has been built upon widely used languages such as C and C++, it is easy to learn. Using the Integrated Development Environment (IDE), it is very easy and much faster to develop and test C# programs.

3.7 Factors Affecting the Choice of a Language

A large number of programming languages are available for developing programs for different types of applications. To develop software for a specific application, one needs to carefully choose a programming language so as to ensure that the programs can be developed easily and

efficiently in a specific period of time. There are certain factors that must be considered by a programmer while choosing a programming language for software development. These factors are described as follows:

- **Purpose** It specifies the objective for which a program is being developed. If a commercial application is to be developed, some business-oriented programming language such as COBOL is preferred. Similarly, if some scientific application is to be developed, then it is best to use some scientific-oriented language such as FORTRAN. The programs related to the AI field can be developed efficiently in the LISP or Prolog programming languages. Some object-oriented language should be preferred for developing web-based applications. A middle-level language such as C should be chosen for developing system programs.
- **Programmer's experience** If more than one programming language is available for developing the same application, then a programmer should choose a language as per his comfort level. Generally, the programmer should go for the language in which he has more experience. For this, the programmer can also compromise with the power of the programming language.
- **Ease of development and maintenance** The programmer should always prefer the language in which programs can be easily developed and maintained. Generally, the object-oriented languages are preferred over the procedural-oriented programming languages because the code developed in these languages can be reused and maintained with great ease.
- **Performance and efficiency** These are the two important factors, which need to be considered while selecting a programming language for software development. The language in which programs can be developed and executed rapidly should always be preferred. In addition, the languages, which require less amount of memory for the storage of programs, should be chosen.
- **Availability of IDE** The language with an IDE (Integrated Development Environment) of well-supported development, debugging and compilation tools should be preferred. A powerful IDE helps in increasing the productivity of a programmer.
- **Error checking and diagnosis** These two factors involve finding the errors and their causes in a program. A programmer must choose a programming language, which contains efficient error handling features. For example, JAVA provides an efficient error handling mechanism of try/catch block. The try/catch block in JAVA programs can be used to handle the unexpected errors that may occur during the execution of a program. Error checking and diagnosis is very important for developing quality and error free programs. A programming language with efficient and robust error detection and correction mechanism eases the task of code development and testing.

3.8 Developing a Program

Developing a program refers to the process of writing the source code for the required application by following the syntax and the semantics of chosen programming language. Syntax and semantics are the set of rules that a programmer needs to adhere while developing a program.

Before actually developing a program, the aim and the logic of the program should be very clear to the programmer. Therefore, the first stage in the development of a program is to carry out a detailed study of the program objectives. The objectives make the programmer aware of the purpose for which the program is being developed. After ascertaining the program objectives, the programmer needs to list down the set of steps to be followed for program development. This set of program development steps is called algorithm. The programmer may also use a graphical model known as flowchart to represent the steps defined in the program algorithm.

After the logic of the program has been developed either by an algorithm or a flowchart, the next step is to choose a programming language for actual development of the program code. There are a number of factors that should be taken into consideration while selecting the target programming language, such as performance and efficiency of the programming language, programmer's prior experience with the language, etc.

A programming language is typically bundled together with an IDE containing the necessary tools for developing, editing, running and debugging a computer program. For instance, Turbo C++ is provided with a strong and powerful IDE to develop, compile, debug and execute the programs.

Figure 3.5 shows the Turbo C++ IDE.

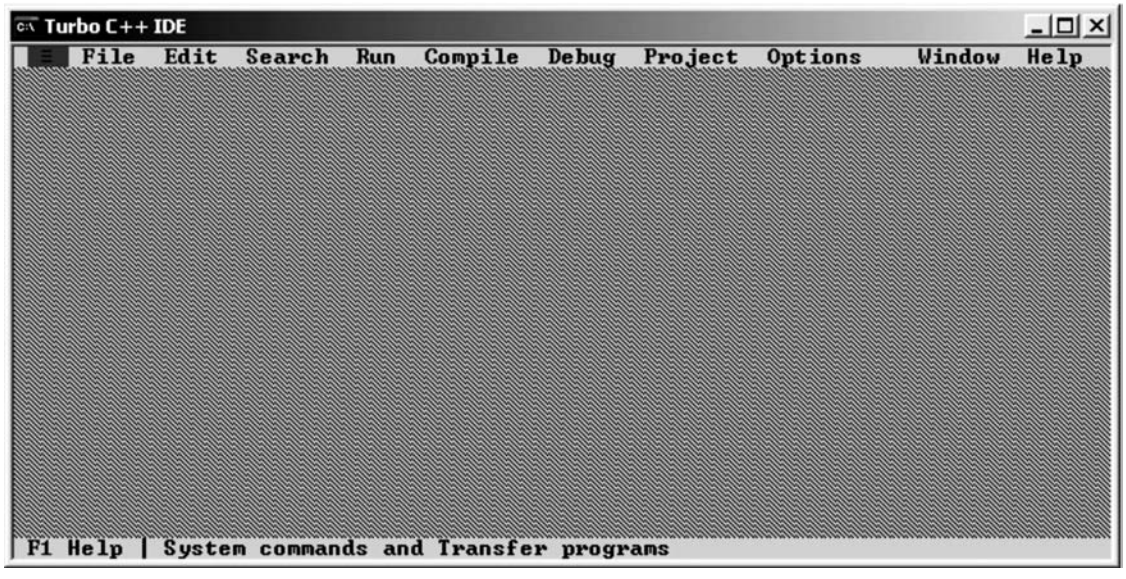


Fig. 3.5 ⇔ *The Turbo C++ IDE*

Suppose we are required to develop a program for calculating the percentage of marks of two subjects for a student and display the output. The first step in the development of a program for this problem is the preparation of an algorithm, as shown below:

```
Step 1 - Input the marks for first subject. (mark1)
Step 2 - Input the marks for second subject. (mark2)
Step 3 - Calculate the percentage.
         percentage = (mark1 + mark2)/200*100
Step 4 - If percentage > 40
Step 5 - Display Pass
Step 6 - Else
Step 7 - Display Fail
```

Figure 3.6 shows the flowchart for the above algorithm.

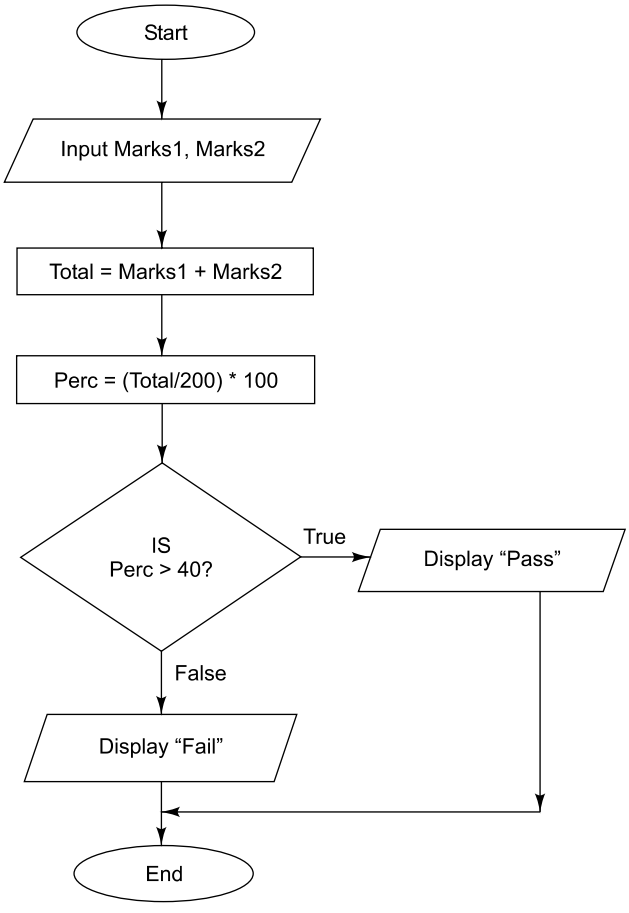


Fig. 3.6 ⇔ *Flowchart for calculating the percentage of marks and displaying the result*

After developing the algorithm and flowchart, the actual development of the program can be started in the source code editor of C++. The following code shows the C++ program for calculating the percentage of marks in two different subjects for a student.

```
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
  float mark1,mark2;
  float percentage;

  cout << "Enter marks of first subject: ";
  cin >> mark1;

  cout << "Enter marks of second subject: ";
  cin >> mark2;

  percentage =((mark1+mark2)/200)*100;

  if (percentage>40)
    cout << "\nThe student is passed";
  else
    cout <<"\nThe student is failed";

  getch();
  return 0;
}
```

Figure 3.7 shows the program code in the source code editor of Turbo C++ IDE.

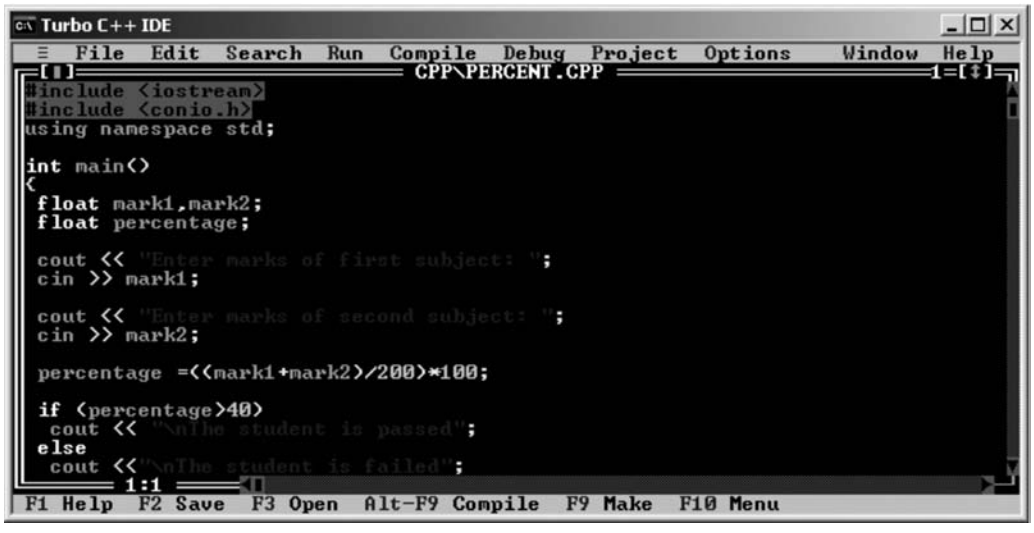


Fig. 3.7 ⇔ *Developing a program in the source code editor of C++ language*

3.9 Running a Program

After developing the program code, the next step is to compile the program. Program compilation helps identify any syntactical errors in the program code. If there are no syntax errors in the source code, then the compiler generates the target object code. It is the machine language code that the processor of the computer system can understand and execute.

Once the corresponding object code or the executable file is built by the compiler, the program can be run in order to check the logical correctness of the program and generate the desired output. The logical errors also called semantic errors might cause the program to generate undesired results. Programming languages provide various mechanisms such as exception handling for handling these logical errors. If the output generated by the program corresponding to the given inputs matches with the desired result, then the purpose of developing the program is solved. Otherwise, the logic of the program should be checked again to obtain the correct solution for the given problem

Figure 3.8 shows the output of the program for displaying student's result.

Fig. 3.8 ⇔ *Running a program*

The above figure shows the output generated by running the C++ program. We can run the program in Turbo C++ IDE by either selecting Run → Run or by pressing the Alt and F9 keys simultaneously.

SUMMARY

⇔ Programming languages help perform a specific task by providing a set of instructions to the computer system. Programming languages have evolved a lot over the years. The

evolution of the programming languages is divided in five different generations. The first generation programming languages comprise of machine language, which is a fast and efficient programming language but not very easy to use. The second generation programming languages comprise of assembly language. The first and second generation languages are also known as low level languages. The third, fourth and fifth generation of languages comprise of various high-level languages. The high-level languages are machine independent languages in which a program can be developed with great ease. Some popular high-level languages are FORTRAN, BASIC, PASCAL, C, C++, and Java.

⇔ There are various high-level languages in existence that can be used for developing programs. Therefore, a proper high-level language should be chosen by considering certain factors such as the purpose of the program to be developed, the experience of the programmer, performance and efficiency of the language, and the IDE of the programming language. After selecting the right programming language, a program can be developed and run in the IDE of the selected programming language to obtain the desired result.

Key Terms

- Programming languages
- Low-level programming languages
- High-level programming languages
- Machine language
- Assembly language
- Assembler
- Interpreter
- Compiler
- Opcode
- Operand
- Program development

Review Questions

3.1 *What do you understand by programming languages?*

3.2 *What is the difference between low-level and high-level programming languages? Which one of these is considered more user-friendly and why?*

3.3 *Why are the programs developed in low-level programming languages considered as efficient programs? Is it not possible to develop efficient programs in high-level programming languages?*

3.4 *Explain in detail the evolution of the programming languages.*

3.5 *What do you understand by generations of programming languages?*

3.6 *Why the second generation programming languages are considered as more user-friendly than the first generation programming languages?*

3.7 *Explain in detail the machine instruction format.*

3.8 *What is the function of an assembler in assembly language? Does it affect the efficiency of the programs developed in the assembly language?*

3.9 *What is the difference between a compiler and an interpreter?*

3.10 *What are the different advantages and disadvantages of third generation programming languages?*

3.11 *What do you understand by constraint programming? List out some of the applications in which constraint programming is employed.*

3.12 *What are the different characteristics of a good programming language?*

3.13 *Explain the difference between different programming paradigms used in programming languages.*

3.14 *List out some popular high-level languages and explain in detail any three of them.*

3.15 *Why is C++ considered as a superset of C?*

3.16 *Do the programming languages, C and C++, use the same programming paradigm?*

3.17 *What are the different factors that affect the choice of a language for software development?*

3.18 *What are the different points that should be remembered before developing a program?*

3.19 *What is the difference between compiling and running of a program? Do these two processes generate the same output?*

2.20 *Explain the terms—source code, syntax, semantics and IDE.*

Fill in the Blanks

3.1 Programming languages are used to provide _____ to the computer system.

3.2 The person who uses the programming languages to communicate with the computer system is referred as _____.

3.3 _____ was considered as the first computer programmer in the history of programming.

3.4 The _____ of the front panel of the ancient computer systems were used to enter the machine language programs into the computer system.

3.5 The two important parts of the machine instruction are _____ and _____.

3.6 The opcode part of the machine instruction specifies the _____ to be performed by the computer system.

3.7 The operand part of the machine instruction specifies the _____ on which the operation is to be performed by the computer system.

3.8 The second generation programming languages used the concept of _____ for writing computer programs.

3.9 An assembler is a software program that converts the assembly language program into _____ language instructions.

3.10 The programming paradigm employed by most of the third generation programming languages was _____.

3.11 _____ and _____ are the software programs that can be used to convert the high-level programs into machine language programs.

3.12 _____ is a good example of scientific-oriented high-level programming language.

3.13 _____ is a high-level language specially designed to interact with database programs.

3.14 C language was developed by _____ in the year _____.

3.15 C++ uses the _____ programming paradigm for developing programs for various applications.

3.16 Java is best suited for developing _____ applications.

3.17 A program is developed by following the _____ and _____ of the programming language.

3.18 A program is compiled for finding the _____ errors in the program.

3.19 The object code is the _____ code that a processor of the computer system understands and executes.

3.20 The running of a program may reveal the _____ errors in the source code.

Multiple Choice Questions

1. What is a programming language?

A. It is the language that instructs the computer system to perform a certain action.

B. It is used to change the configuration of the computer system.

C. It is the language for managing computer hardware.

D. None of the above.

2. The person who uses the programming languages to develop programs is usually known as:

A. Hardware engineer B. Programmer

C. Analyst D. All of the above

3. Who was considered as the first computer programmer in the history of programming languages?

A. Charles Babbage B. Ada Augusta Lovelace

C. Konrad Zuse D. John Backus

4. Which of the following is a low-level programming language?

A. FORTRAN B. Ada C. C D. Machine language

5. Machine language programs are very efficient because

A. They are directly executed by the CPU

B. The are very easy to develop

C. Their object code is very small in size

D. None of the above

6 What is a machine language?

A. It is an object-oriented language. B. It is a language of 0s and 1s.

C. It is a high-level language. D. All of the above

7. Which of the following domain is best suited for FORTRAN?

A. Engineering B. Medical C. Education D. Business

8. Who developed the SQL language?

A. Microsoft B. Dell C. Google D. IBM

9. Who was the inventor of C programming language?

A. Lary Wall B. Roussel C. Dennis Ritchie D. Bjarne Stroustrup

10. Which of the following is a part of the machine instruction?

A. Data B. Mnemonics C. Opcode D. Address

11. Which of the following languages is considered as the second generation language?

A. Machine language B. Assembly language

C. Ada D. BASIC

12. Which of the following programs is used to convert the assembly programs into machine instructions?

A. Compiler B. Interpreter C. Assembler D. SQL

13. In which generation were database languages developed?

A. 2GL B. 3GL C. 4GL D. 5GL

14. What is the full form of FORTRAN?

A. Form transaction B. Formulation transcription

C. Formula transition D. Formula translation

15. Which one of the following is not a programming paradigm?

A. Procedure-oriented B. Logic-oriented

C. Data-oriented D. Object-oriented

16. Who was the developer of PASCAL?

A. Dannis Ritchie B. Niklaus Wirth

C. John Backus D. James Gosling

17. Which of the following operating systems was written in C?

A. UNIX B. Windows C. Macintosh D. None of the above

18. Which of the following paradigm is employed by C++?

A. Procedural B. Object-oriented

C. Logic-oriented D. None of the above

19. Which of the following errors are detected by the compiler?

A. Syntax errors B. Logical errors

C. Semantics errors D. Data errors

4

Introduction to Programming

Key Concepts

- Procedure-oriented programming
- Object-oriented programming
- Objects
- Classes
- Data abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

4.1 Programming Environment

A programming environment comprises all those components that facilitate the development of a program. These components are largely divided under two categories— programming tools and Application Programming Interfaces (APIs). They are regarded as the building blocks of any programming environment.

An API can be defined as a collection of data structures, classes, protocols, and pre-defined functions stored in the form of libraries. These libraries are included in the software packages of the programming languages like C, C++, etc. An API makes the development task easier for the programmers, as in-built API components are used again and again, ensuring reusability.

The software application which is used for the development, maintenance and debugging of a software program is known as programming tool. A good programming tool ensures that the programming activities are performed in an efficient manner. The following are some of the main categories of programming tools:

- **Integrated Development Environment (IDE)**: It is the most commonly used tool that offers an integrated environment to the programmers for software development. It

contains almost all the components for software development such as compiler, editor, debugger, etc.

- **Debugging tool**: It is a specialized tool that helps the programmer to detect and remove bugs or errors from a program.
- **Memory usage tool**: As the name suggests, memory usage tool helps the programmer to manage the memory resources in an efficient manner.

4.2 Introduction to the Design and Implementation of Correct, Efficient and Maintainable Programs

The design and development of a correct, efficient, and maintainable program depends on the approach followed by the programmer. A programmer should follow standard methodologies throughout the life cycle of program development. The entire program development process is divided into a number of phases, with each phase serving a definite purpose. Also, the output of one phase acts as an input for the next phase. Let us now understand these standard set of phases in the program development process:

1. **Analysis phase** As the name suggests, the first phase of program development involves analyzing the problem in order to ascertain the objectives that the program is supposed to meet. All the identified requirements are documented so as to avoid any doubts or uncertainties pertaining to the functionality of the program. This phase also emphasizes on determining the input and output values of the program.
2. **Designing phase** This phase involves making the plan of action before actually starting the development work. The plan is made on the basis of the program specifications identified in the previous phase. Different programs require different designing patterns depending on the program specifications. Thus, this phase helps in framing the core structure of the program. In addition, the designing phase has an added advantage of modularity. It basically helps to break the program into small modules or chunks. This breaking of the large program into smaller chunks results in the development of a well-organized program. Furthermore, it gives the programmer liberty of planning and creating the algorithm for each module separately.
3. **Development phase** This phase involves writing the instructions or code for the program on the basis of the design document created in the previous phase. The choice of the programming language in which the program will be developed is made on the basis of the type of program. For example, if it is a system program, then it is better to choose C++ language instead of Visual Basic (VB), which is more suited for applications programming.
4. **Implementation and Testing** In this stage, the developed program is implemented in its target environment and its key parameters are closely observed in order to ensure that the program runs correctly. Apart from ensuring the correct functioning of the program this phase primarily focuses on identifying the hidden bugs in the program. No matter how many preventive measures are taken in the development phase there is always the possibility of prevalence of hidden bugs in a program. Thus, to identify such bugs a program needs to be tested using large number of varied input values. Once identified the bugs are removed with the help of software patches.

4.3 A Look at Procedure-Oriented Programming

Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as *procedure-oriented programming (POP)*. In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on functions. A typical program structure for procedural programming is shown in Fig. 4.1. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem.

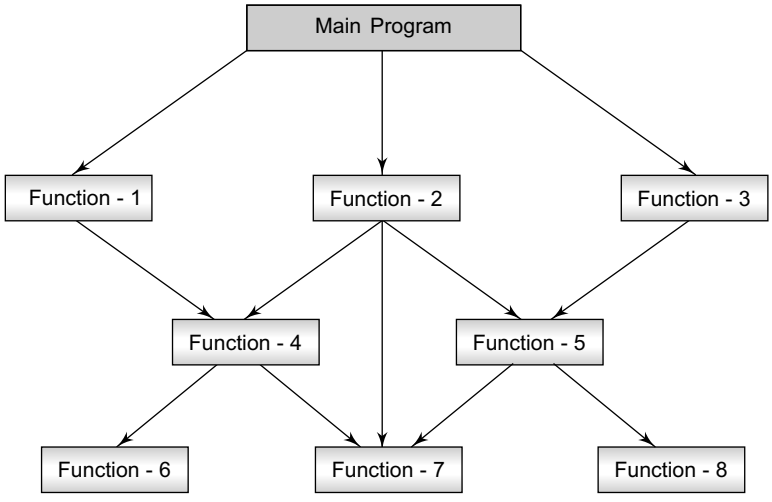


Fig. 4.1 ⇔ *Typical structure of procedure-oriented programs*

Procedure-oriented programming basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as functions. We normally use a flowchart to organize these actions and represent the flow of control from one action to another. While we concentrate on the development of functions, very little attention is given to the data that are being used by various functions. What happens to the data? How are they affected by the functions that work on them?

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. Figure 4.2 shows the relationship of data and functions in a procedure-oriented program.

Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.

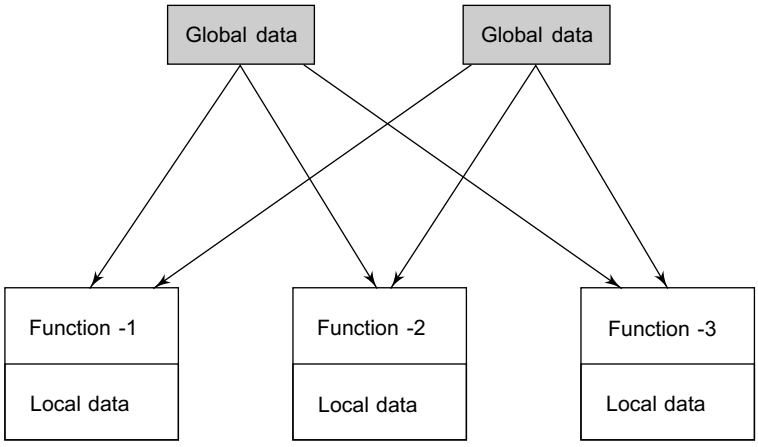


Fig. 4.2 ⇔ *Relationship of data and functions in procedural programming*

Another serious drawback with the procedural approach is that it does not model real world problems very well. This is because functions are action-oriented and do not really correspond to the elements of the problem.

Some characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs *top-down* approach in program design.

4.4 Object-Oriented Programming Paradigm

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. The organization of data and functions in object-oriented programs is shown in Fig. 4.3. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.

Some of the striking features of object-oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.

- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- *Follows bottom-up* approach in program design.

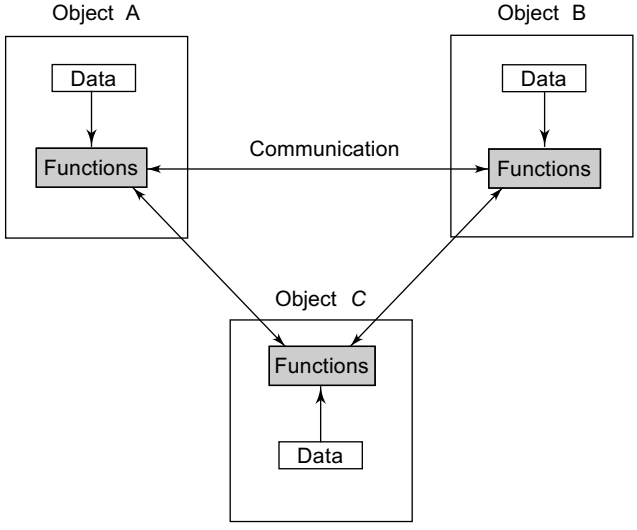


Fig. 4.3 ⇔ *Organization of data and functions in OOP*

Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people. It is therefore important to have a working definition of object-oriented programming before we proceed further. We define "object-oriented programming as an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand." Thus, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

4.5 Object-Oriented Programming Features

It is necessary to understand some of the features used extensively in object-oriented programming. These include:

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance

- Polymorphism
- Dynamic binding
- Message passing

We shall discuss these features in some detail in this section.

4.5.1 Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in C.

When a program is executed, the objects interact by sending messages to one another. For example, if "customer"and "account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data, and code to manipulate the data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted, and the type of response returned by the objects. Although different authors represent them differently, Fig. 4.4 shows two notations that are popularly used in object-oriented analysis and design.

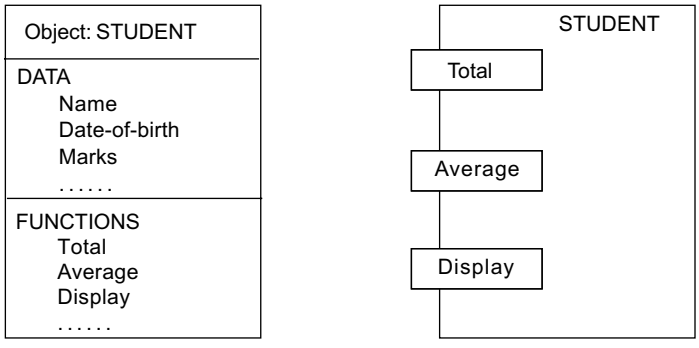


Fig. 4.4 ⇔ *Two ways of representing an object*

4.5.2 Classes

We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language. The syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement

```
fruit mango;
```

will create an object mango belonging to the class **fruit**.

4.5.3 Data Abstraction and Encapsulation

The wrapping up of data and functions into a single unit (called class) is known as *encapsulation*. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called *data hiding* or *information hiding*.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called data members because they hold information. The functions that operate on these data are sometimes called *methods* or *member functions*.

Since the classes use the concept of data abstraction, they are known as *Abstract Data Types* (ADT).

4.5.4 Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. For example, the bird 'robin' is a part of the class 'flying bird' which is again a part of the class 'bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in Fig. 4.5.

In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side-effects into the rest of the classes.

Note that each sub-class defines only those features that are unique to it. Without the use of classification, each class would have to explicitly include all of its features.

4.5.5 Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The

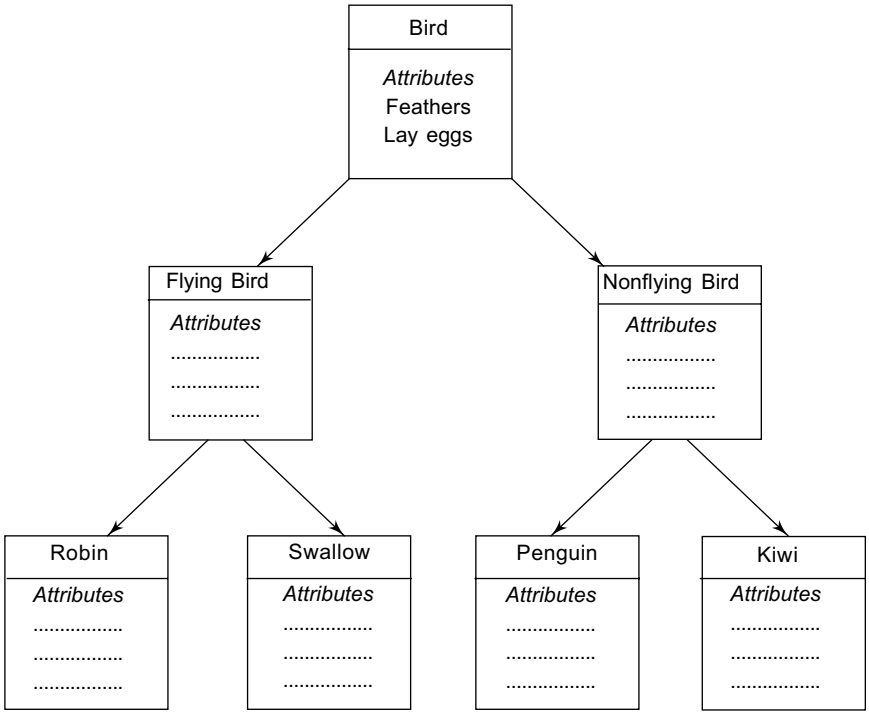


Fig. 4.5 ⇔ *Property inheritance*

process of making an operator to exhibit different behaviours in different instances is known as *operator overloading*.

Figure 4.6 illustrates that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context. Using a single function name to perform different types of tasks is known as *function overloading*.

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

4.5.6 Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. *Dynamic binding* (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

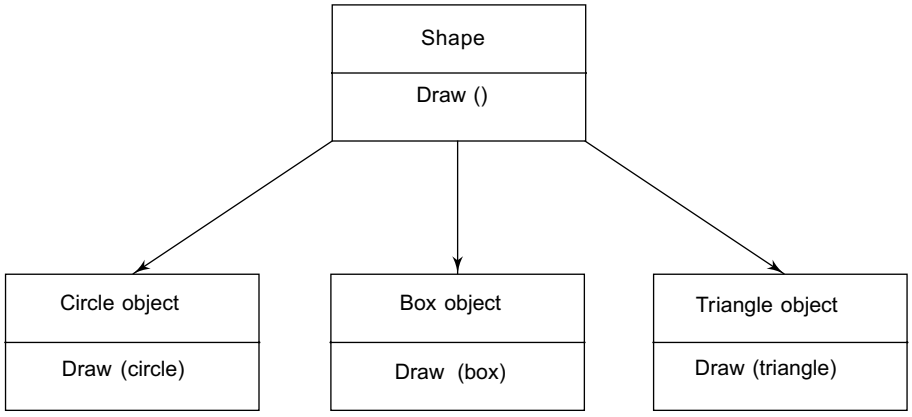


Fig. 4.6 ⇔ *Polymorphism*

Consider the procedure "draw" in Fig. 4.6. By inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

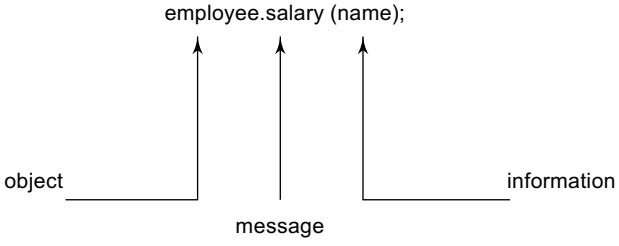
4.5.7 Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behaviour,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result. *Message passing* involves specifying the name of the object, the name of the function (message) and the information to be sent. Example:



Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

4.6 Merits of OOP

OOP offers several merits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer. There are a number of issues that need to be tackled to reap some of the benefits stated above. For instance, object libraries must be available for reuse. The technology is still developing and current products may be superseded quickly. Strict controls and protocols need to be developed if reuse is not to be compromised.

Developing a software that is easy to use makes it hard to build. It is hoped that the object-oriented programming tools would help manage this problem.

4.7 Applications of OOP

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as windows. Hundreds of windowing systems have been developed, using the OOP techniques.

Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of applications because it can simplify a complex problem. The promising areas for application of OOP include:

- Real-time systems
- Simulation and modeling
- Object-oriented databases
- Hypertext, hypermedia and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

The richness of OOP environment has enabled the software industry to improve not only the quality of software systems but also its productivity. Object-oriented technology is certainly changing the way the software engineers think, analyze, design and implement systems.

SUMMARY

- The most popular phase till recently was procedure-oriented programming (POP).
- POP employs *top-down* programming approach where a problem is viewed as a sequence of tasks to be performed. A number of functions are written to implement these tasks.
- POP has two major drawbacks, viz. (1) data move freely around the program and are therefore vulnerable to changes caused by any function in the program, and (2) it does not model very well the real-world problems.
- Object-oriented programming (OOP) was invented to overcome the drawbacks of the POP. It employs the *bottom-up* programming approach. It treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it in a data structure called **class**. This feature is called **data** *encapsulation*.
- In OOP, a problem is considered as a collection of a number of entities called **objects**. Objects are instances of classes.
- Insulation of data from direct access by the program is called *data hiding*.
- *Data abstraction* refers to putting together essential features without including background details.
- *Inheritance* is the process by which objects of one class acquire properties of objects of another class.
- *Polymorphism* means one name, multiple forms. It allows us to have more than one function with the same name in a program. It also allows overloading of operators so that an operation can exhibit different behaviours in different instances.

- *Dynamic binding* means that the code associated with a given procedure is not known until the time of the call at run-time.
- *Message passing* involves specifying the name of the object, the name of the function (message) and the information to be sent.
- Object-oriented technology offers several benefits over the conventional programming methods—the most important one being the reusability.
- Applications of OOP technology has gained importance in almost all areas of computing including real-time business systems.
- There are a number of languages that support object-oriented programming paradigm. Popular among them are C++, Smalltalk and Java. C++ has become an industry standard language today.

Key Terms

- assembly language
- bottom-up programming
- C++
- classes
- concurrency
- data abstraction
- data encapsulation
- data hiding
- data members
- dynamic binding
- early binding
- flowcharts
- function overloading
- functions
- garbage collection
- global data
- hierarchical classification
- inheritance
- late binding
- local data
- machine language
- member functions
- message passing
- methods
- modular programming
- multiple inheritance
- object libraries
- object-based programming
- Objective C
- object-oriented programming
- objects
- operator overloading
- persistence
- polymorphism
- procedure-oriented programming
- reusability
- Simula
- structured programming
- top-down programming

Review Questions

4.1 *What do you think are the major issues facing the software industry today?*

4.2 *Briefly discuss the software evolution during the period 1950 – 1990.*

4.3 *What is procedure-oriented programming? What are its main characteristics?*

4.4 *Discuss an approach to the development of procedure-oriented programs.*

4.5 *Describe how data are shared by functions in a procedure-oriented program.*

4.6 *What is object-oriented programming? How is it different from the procedure-oriented programming?*

4.7 *How are data and functions organized in an object-oriented program?*

4.8 *What are the unique advantages of an object-oriented programming paradigm?*

4.9 *Distinguish between the following terms:*
- (a) *Objects and classes*
- (b) *Data abstraction and data encapsulation*
- (c) *Inheritance and polymorphism*
- (d) *Dynamic binding and message passing*

4.10 *What kinds of things can become objects in OOP?*

4.11 *Describe inheritance as applied to OOP.*

4.12 *What do you mean by dynamic binding? How is it useful in OOP?*

4.13 *How does object-oriented approach differ from object-based approach?*

4.14 *List a few areas of application of OOP technology.*

4.15 *State whether the following statements are TRUE or FALSE.*
- (a) *In procedure-oriented programming, all data are shared by all functions.*
- (b) *The main emphasis of procedure-oriented programming is on algorithms rather than on data.*
- (c) *One of the striking features of object-oriented programming is the division of programs into objects that represent real-world entities.*
- (d) *Wrapping up of data of different types into a single unit is known as encapsulation.*
- (e) *One problem with OOP is that once a class is created it can never be changed.*
- (f) *Inheritance means the ability to reuse the data values of one object by*
- (g) *Polymorphism is extensively used in implementing inheritance.*
- (h) *Object-oriented programs are executed much faster than conventional programs.*
- (i) *Object-oriented systems can scale up better from small to large.*
- (j) *Object-oriented approach cannot be used to create databases.*

5

Beginning with C++

Key Concepts

- C with classes
- C++ features
- Main function
- C++ comments
- Output operator
- Input operator
- Header file
- Return statement
- Namespace
- Variables
- Cascading of operators
- C++ program structure
- Client-server model
- Source file creation

5.1 What is C++?

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented (incremented) version of C.

During the early 1990's the language underwent a number of improvements and changes. In November 1997, the ANSI/ISO standards committee standardised these changes and added several new features to the language specifications.

C++ is a superset of C. Most of what we know about C applies to C++ also. Therefore, almost all C programs are also C++ programs. However, there are a few minor differences that will prevent a C program to run under C++ compiler. We shall see these differences later as and when they are encountered.

The most important facilities that C++ adds on to C are classes, inheritance, function overloading, and operator overloading. These features enable creating of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

The object-oriented features in C++ allow programmers to build large programs with clarity, extensibility and ease of maintenance, incorporating the spirit and efficiency of C. The addition of new features has transformed C from a language that currently facilitates top-down, structured design, to one that provides bottom-up, object-oriented design.

5.2 Applications of C++

C++ is a versatile language for handling very large programs. It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real-life application systems.

- Since C++ allows us to create hierarchy-related objects, we can build-special object-oriented libraries which can be used later by many programmers.
- While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get close to the machine-level details.
- C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.
- It is expected that C++ will replace C as a general-purpose language in the near future.

5.3 A Simple C++ Program

Let us begin with a simple example of a C++ program that prints a string on the screen.

PRINTING A STRING

```
#include <iostream>   // include header file
using namespace std;
int main()
{
     cout << "C++ is better than C.\n";  // C++ statement
     return 0;
}                // End of example
```

PROGRAM 5.1

This simple program demonstrates several C++ features.

5.3.1 Program Features

Like C, the C++ program is a collection of functions. The above example contains only one function, **main()**. As usual, execution begins at main(). Every C++ program must have a **main()**. C++ is a free-form language. With a few exceptions, the compiler ignores carriage returns and white spaces. Like C, the C++ statements terminate with semicolons.

5.3.2 Comments

C++ introduces a new comment symbol // (double slash). Comments start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line, and whatever follows till the end of the line is ignored. Note that there is no closing symbol.

The double slash comment is basically a single line comment. Multiline comments can be written as follows:

```
//  This is an example of
//  C++ program to illustrate
//  Some of its features
```

The C comment symbols /*, */ are still valid and are more suitable for multiline comments. The following comment is allowed:

```
/*    This is an example of
      C++ program to illustrate
      some of its features
*/
```

We can use either or both styles in our programs. Since this is a book on C++, we will use only the C++ style. However, remember that we can not insert a // style comment within the text of a program line. For example, the double slash comment cannot be used in the manner as shown below:

```
for(j=0; j<n; /* loops n times */ j++)
```

5.3.3 Output Operator

The only statement in Program 5.1 is an output statement. The statement

```
cout << "C++ is better than C.";
```

causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, cout and <<. The identifier cout (pronounced as 'C out') is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices. We shall later discuss streams in detail.

The operator << is called the *insertion or put to operator*. It inserts (or sends) the contents of the variable on its right to the object on its left (Fig. 5.1).

The object **cout** has a simple interface. If string represents a string variable, then the following statement will display its contents:

```
cout  <<  string;
```

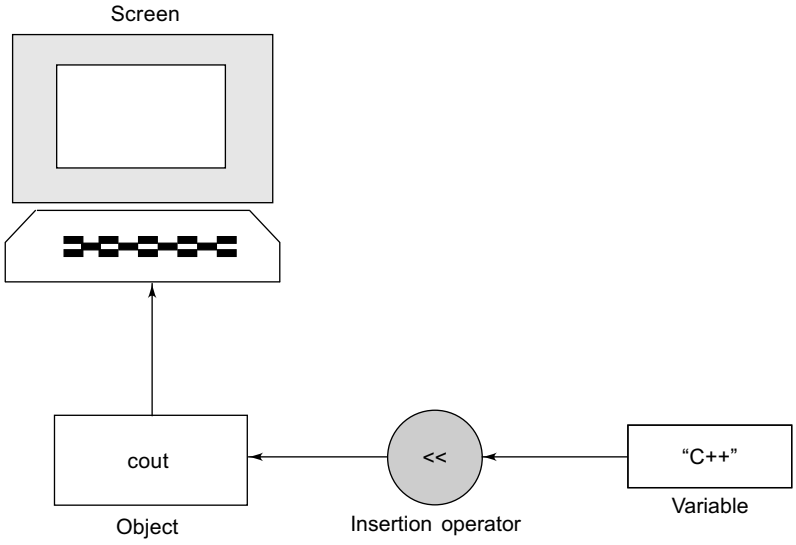


Fig. 5.1 ⇔ *Output using insertion operator*

You may recall that the operator << is the bit-wise left-shift operator and it can still be used for this purpose. This is an example of how one operator can be used for different purposes, depending on the context. This concept is known as *operator overloading*, an important aspect of polymorphism. Operator overloading is discussed in detail in Chapter 10.

It is important to note that we can still use printf() for displaying an output. C++ accepts this notation. However, we will use cout << to maintain the spirit of C++.

5.3.4 The iostream File

We have used the following #include directive in the program:

```
#include <iostream>
```

This directive causes the preprocessor to add the contents of the iostream file to the program. It contains declarations for the identifier **cout** and the operator <<. Some old versions of C++ use a header file called iostream.h. This is one of the changes introduced by ANSI C++. (We should use iostream.h if the compiler does not support ANSI C++ features.)

The header file **iostream** should be included at the beginning of all programs that use input/output statements. Note that the naming conventions for header files may vary. Some implementations use **iostream.hpp**; yet others **iostream.hxx**. We must include appropriate header files depending on the contents of the program and implementation.

Tables 5.1 and 5.2 provide lists of C++ standard library header files that may be needed in C++ programs. The header files with .h extension are "old style" files which should be used with old compilers. Table 5.1 also gives the version of these files that should be used with the ANSI standard compilers.

Table 5.1 *Commonly used old-style header files*

Header file	*Contents and purpose*	*New version*
<assert.h>	Contains macros and information for adding diagnostics that aid program debugging	<cassert>
<ctype.h>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.	<cctype>
<float.h>	Contains the floating-point size limits of the system.	<cfloat>
<limits.h>	Contains the integral size limits of the system.	<climits>
<math.h>	Contains function prototypes for math library functions.	<cmath>
<stdio.h>	Contains function prototypes for the standard input/output library functions and information used by them.	<cstdio>
<stdlib.h>	Contains function prototypes for conversion of numbers to text, text to numbers, memory allocation, random numbers, and various other utility functions.	<cstdlib>
<string.h>	Contains function prototypes for C-style string processing functions.	<cstring>
<time.h>	Contains function prototypes and types for manipulating the time and date.	
<iostream.h>	Contains function prototypes for the standard input and standard output functions.	<iostream>
<iomanip.h>	Contains function prototypes for the stream manipulators that enable formatting of streams of data.	<iomanip>
<fstream.h>	Contains function prototypes for functions that perform input from files on disk and output to files on disk.	<fstream>

Table 5.2 *New header files included in ANSI C++*

Header file	*Contents and purpose*
<utility>	Contains classes and functions that are used by many standard library header files.
<vector>, <list>, <deque> <queue>, <set>, <map>, <stack>, <bitset>	The header files contain classes that implement the standard library containers. Containers store data during a program's execution.
<functional>	Contains classes and functions used by algorithms of the standard library.
<memory>	Contains classes and functions used by the standard library to allocate memory to the standard library containers.
<iterator>	Contains classes for manipulating data in the standard library containers.

(*Contd*)

Table 5.2 *(Contd)*

Header file	*Contents and purpose*
<algorithm>	Contains functions for manipulating data in the standard library containers.
<exception>, <stdexcept>	These header files contain classes that are used for exception handling.
<string>	Contains the definition of class string from the standard library.
<sstream>	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory.
<locale>	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.)
<limits>	Contains a class for defining the numerical data type limits on each computer platform.
<typeinfo>	Contains classes for run-time type identification (determining data types at execution time).

5.3.5 Namespace

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifiers defined in the **namespace** scope we must include the using directive, like

```
using namespace std;
```

Here, **std** is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in **std** to the current global scope. **using** and **namespace** are the new keywords of C++.

5.3.6 Return Type of main( )

In C++, main() returns an integer type value to the operating system. Therefore, every main() in C++ should end with a return(0) statement; otherwise a warning or an error might occur. Since **main()** returns an integer type value, return type for main() is explicitly specified as **int**. Note that the default return type for all functions in C++ is **int**. The following **main** without type and return will run with a warning:

```
main()
{
     ......
     ......
}
```

5.4 More C++ Statements

Let us consider a slightly more complex C++ program. Assume that we would like to read two numbers from the keyboard and display their average on the screen. C++ statements to accomplish this is shown in Program 5.2.

AVERAGE OF TWO NUMBERS

```
#include <iostream>

using namespace std;

int main()
{
     float number1, number2,
          sum, average;

     cout << "Enter two numbers: "; // prompt
     cin  >> number1;    // Reads numbers
     cin  >> number2;    // from keyboard

     sum = number1 + number2;
     average = sum/2;

     cout <<  "Sum = " << sum << "\n";
     cout << "Average = " << average << "\n";

     return 0;
}
```

PROGRAM 5.2

The output of Program 5.2 is:

```
Enter two numbers:  6.5  7.5
Sum = 14
Average = 7
```

5.4.1 Variables

The program uses four variables number1, number2, sum, and average. They are declared as type float by the statement.

```
float number1, number2, sum, average;
```

All variables must be declared before they are used in the program.

5.4.2 Input Operator

The statement

```
cin >> number1;
```

is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier **cin** (pronounced 'C in') is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.

The operator >> is known as *extraction or get from* operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right (Fig. 5.2). This corresponds to the familiar scanf() operation. Like << , the operator >> can also be overloaded.

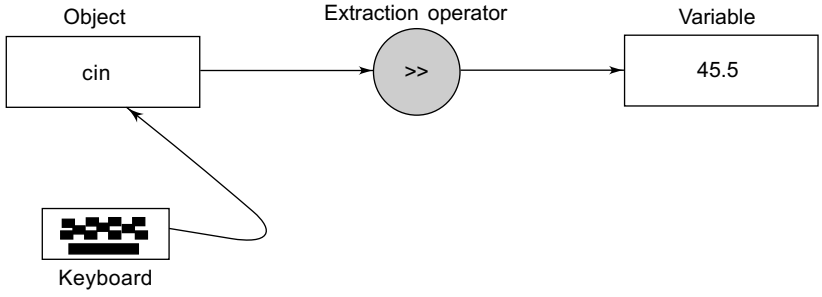


Fig. 5.2 ⇔ *Input using extraction operator*

5.4.3 Cascading of I/O Operators

We have used the *insertion operator* << repeatedly in the last two statements for printing results.

The statement

```
cout << "Sum = " << sum << "\n";
```

first sends the string "Sum =" to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of << in one statement is called *cascading*. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

```
cout << "Sum = " << sum << "\n"
     << "Average = " << average << "\n";
```

This is one statement but provides two lines of output. If you want only one line of output, the statement will be:

```
cout << "Sum = " << sum << ","
     << "Average = " << average << "\n";
```

The output will be:

```
Sum = 14, Average = 7
```

We can also cascade input operator >> as shown below:

```
cin >> number1 >> number2;
```

The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to number1 and 20 to number2.

5.5 An Example with Class

One of the major features of C++ is classes. They provide a method of binding together data and functions which operate on them. Classes are user-defined data types.

Program 5.3 shows the use of class in a C++ program.

USE OF CLASS

```
#include <iostream>
using namespace std;
class person
{
       char name[30];
       int  age;

       public:
           void getdata(void);
           void display(void);
};
void person :: getdata(void)
{
       cout << "Enter name: ";
       cin  >> name;
       cout << "Enter age: ";
       cin  >> age;
}
void person :: display(void)
{
     cout << "\nName: " << name;
     cout << "\nAge:  " << age;
}

int main()
{
     person p;
```

```
        p.getdata();
        p.display();

        return 0;
    }
```

PROGRAM 5.3

The output of Program 5.3 is:

```
Enter Name: Ravinder
Enter Age:  30

Name: Ravinder
Age:  30
```

note

cin can read only one word and therefore we cannot use names with blank spaces.

The program defines **person** as a new data of type class. The class person includes two basic data type items and two functions to operate on that data. These functions are called **member functions**. The main program uses **person** to declare variables of its type. As pointed out earlier, class variables are known as *objects*. Here, p is an object of type **person**. Class objects are used to invoke the functions defined in that class.

5.6 Structure of C++ Program

As it can be seen from the Program 5.3, a typical C++ program would contain four sections as shown in Fig. 5.3. These sections may be placed in separate code files and then compiled independently or jointly.

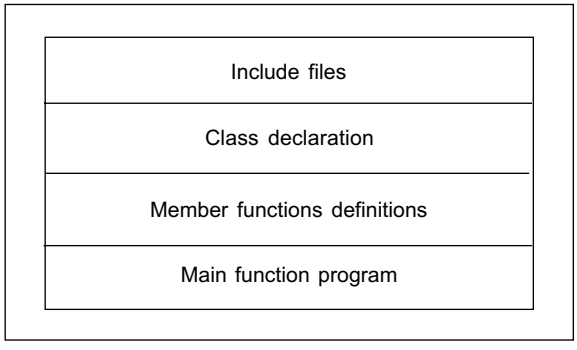


Fig. 5.3 ⇔ *Structure of a C++ program*

It is a common practice to organize a program into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file. This approach enables the programmer to separate the abstract specification of the interface (class definition) from the implementation details (member functions definition). Finally, the main program that uses the class is placed in a third file which "includes" the previous two files as well as any other files required.

This approach is based on the concept of client-server model as shown in Fig. 5.4. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

5.7 Creating the Source File

C++ programs can be created using any text editor. For example, on the UNIX, we can use vi or *ed* text editor for creating and editing the source code. On the DOS system, we can use *edlin* or any other editor available or a word processor system under non-document mode.

Some systems such as Turbo C++ provide an integrated environment for developing and editing programs. Appropriate manuals should be consulted for complete details.

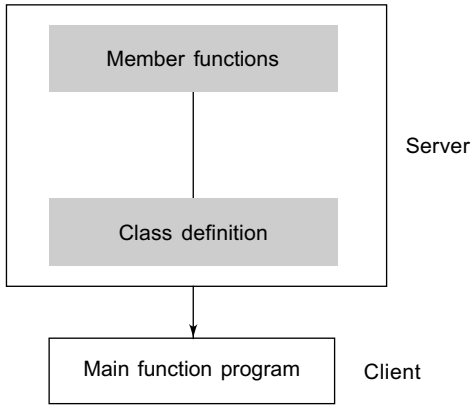


Fig. 5.4 ⇔ *The client-server model*

The file name should have a proper file extension to indicate that it is a C++ program file. C++ implementations use extensions such as .c, .C, .cc, .cpp and .cxx. Turbo C++ and Borland C++ use .c for C programs and .cpp (C plus plus) for C++ programs. Zortech C++ system uses .cxx while UNIX AT&T version uses .C (capital C) and .cc. The operating system manuals should be consulted to determine the proper file name extensions to be used.

SUMMARY

- ⇔ C++ is a superset of C language.
- ⇔ C++ adds a number of object-oriented features such as objects, inheritance, function overloading and operator overloading to C. These features enable building of programs with clarity, extensibility and ease of maintenance.
- ⇔ C++ can be used to build a variety of systems such as editors, compilers, databases, communication systems, and many more complex real-life application systems.
- ⇔ C++ supports interactive input and output features and introduces a new comment symbol // that can be used for single line comments. It also supports C-style comments.
- ⇔ Execution of all C++ programs begins at **main( )** function and ends at **return( )** statement. The header file **iostream** should be included at the beginning of all programs that use input/output operations.
- ⇔ All ANSI C++ programs must include **using namespace std** directive.
- ⇔ A typical C++ program would contain four basic sections, namely, include files section, class declaration section, member function section and main function section.
- ⇔ C++ programs can be created using any text editor.
- ⇔ Most compiler systems provide an integrated environment for developing and executing programs. Popular systems are UNIX AT&T C++, Turbo C++ and Microsoft Visual C++.

Key Terms

- #include
- **a.out**
- Borland C++
- cascading
- **cin**
- class
- client
- comments
- **cout**
- edlin
- extraction operator
- **float**
- free-form
- get from operator
- input operator
- insertion operator
- **int**
- iostream
- **iostream.h**
- keyboard
- **main()**
- member functions
- MS-DOS
- **namespace**
- object
- operating systems
- operator overloading
- output operator
- put to operator
- return ()
- screen
- server
- Simula67
- text editor
- Turbo C++
- Unix AT&T C++
- **using**
- Visual C++
- Windows
- Zortech C++

Review Questions

5.1 *State whether the following statements are TRUE or FALSE.*

(a) *Since C is a subset of C++, all C programs will run under C++ compilers.*

(b) *In C++, a function contained within a class is called a member function.*

(c) *Looking at one or two lines of code, we can easily recognize whether a program is written in C or C++.*

(d) I*n C++, it is very easy to add new features to the existing structure of an object.*

(e) *The concept of using one operator for different purposes is known as oerator overloading.*

(f) *The output function printf() cannot be used in C++ programs.*

5.2 *Why do we need the preprocessor directive #include <iostream> ?*

5.3 *How does a main() function in C++ differ from main() in C?*

5.4 *What do you think is the main advantage of the comment // in C++ as compared to the old C type comment?*

5.5 *Describe the major parts of a C++ program.*

Debugging Exercises

5.1 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
       int i = 0;
       i = i + 1;
       cout << i << " ";
       /*comment\*//i = i + 1;
       cout << i;
}
```

5.2 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
       short i=2500, j=3000;
       cout >> "i + j = " >> -(i+j);
}
```

5.3 What will happen when you run the following program?

```
#include <iostream.h>
void main()
{
       int i=10, j=5;
       int modResult=0;
       int divResult=0;

       modResult = i%j;
       cout << modResult << " ";

       divResult = i/modResult;
       cout << divResult;
}
```

5.4 Find errors, if any, in the following C++ statements.

(a) cout << "x=" x;
(b) m = 5; // n = 10; // s = m + n;
(c) cin >>x; >>y;
(d) cout << \n "Name:" << name;
(e) cout <<"Enter value:"; cin >> x;
(f) /*Addition*/ z = x + y;

Programming Exercises

5.1 *Write a program to display the following output using a single cout statement.*

Maths = 90
Physics = 77
Chemistry = 69

5.2 *Write a program to read two numbers from the keyboard and display the larger value on the screen.*

5.3 *Write a program to input an integer value from keyboard and display on screen "WELL DONE" that many times.*

5.4 *Write a program to read the values of a, b and c and display the value of x, where*

$$x = a / b - c$$

Test your program for the following values:

(a) *a = 250, b = 85, c = 25*
(b) *a = 300, b = 70, c = 70*

5.5 *Write a C++ program that will ask for a temperature in Fahrenheit and display it in Celsius.*

5.6 *Redo Exercise 5.5 using a class called* ***temp*** *and member functions.*

6

Tokens, Expressions and Control Structures

Key Concepts

- Tokens
- Keywords
- Identifiers
- Data types
- User-defined types
- Derived types
- Symbolic constants
- Declaration of variables
- Initialization
- Reference variables
- Type compatibility
- Scope resolution
- Dereferencing
- Memory management
- Formatting the output
- Type casting
- Constructing expressions
- Special assignment expressions
- Implicit conversion
- Operator overloading
- Control structures

6.1 Introduction

As mentioned earlier, C++ is a superset of C and therefore most constructs of C are legal in C++ with their meaning unchanged. However, there are some exceptions and additions. In this chapter, we shall discuss these exceptions and additions with respect to tokens and control structures.

6.2 Tokens

As we know, the smallest individual units in a program are known as tokens. C++ has the following tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators

A C++ program is written using these tokens, white spaces, and the syntax of the language. Most of the C++ tokens are basically similar to the C tokens with the exception of some additions and minor modifications.

6.3 Keywords

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements.

Table 6.1 gives the complete set of C++ keywords. Many of them are common to both C and C++. The ANSI C keywords are shown in boldface. Additional keywords have been added to the ANSI C keywords in order to enhance its features and make it an object-oriented language. ANSI C++ standards committee has added some more keywords to make the language more versatile. These are shown separately. Meaning and purpose of all C++ keywords are given in Appendix D.

Table 6.1 *C++ keywords*

asm	**double**	new	**switch**
auto	**else**	operator	template
break	**enum**	private	this
case	**extern**	protected	throw
catch	**float**	public	try
char	**for**	**register**	**typedef**
class	friend	**return**	**union**
const	**goto**	**short**	**unsigned**
continue	**if**	**signed**	virtual
default	inline	**sizeof**	**void**
delete	**int**	**static**	**volatile**
do	**long**	**struct**	**while**
Added by ANSI C++			
bool	export	reinterpret_cast	typename
const_cast	false	static_cast	using
dynamic_cast	mutable	true	wchar_t
explicit	namespace	typeid	

Note: The ANSI C keywords are shown in bold face.

6.4 Identifiers and Constants

Identifiers refer to the names of variables, functions, arrays, classes, etc. created by the programmer. They are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

- Only alphabetic characters, digits and underscores are permitted.
- The name cannot start with a digit.
- Uppercase and lowercase letters are distinct.
- A declared keyword cannot be used as a variable name.

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, ANSI C++ places no limit on its length and, therefore, all the characters in a name are significant.

Care should be exercised while naming a variable which is being shared by more than one file containing C and C++ programs. Some operating systems impose a restriction on the length of such a variable name.

Constants refer to fixed values that do not change during the execution of a program.

C++ supports several kinds of literal constants. They include integers, characters, floating point numbers and strings. Literal constant do not have memory locations. Examples:

```
123         // decimal integer
12.34       // floating point integer
037         // octal integer
0X2         // hexadecimal integer
"C++"       // string constant
'A'         // character constant
L'ab'       // wide-character constant
```

The **wchar_t** type is a wide-character literal introduced by ANSI C++ and is intended for character sets that cannot fit a character into a single byte. Wide-character literals begin with the letter L.

C++ also recognizes all the backslash character constants available in C.

note

C++ supports two types of string representation — the C-style character string and the string class type introduced with Standard C++. Although the use of the string class type is recommended, it is advisable to understand and use C-style strings in some situations.

6.5 Basic Data Types

Data types in C++ can be classified under various categories as shown in Fig. 6.1.

C++ compilers support all the built-in (also known as *basic* or *fundamental*) data types. With the exception of **void**, the basic data types may have several modifiers preceding them to serve

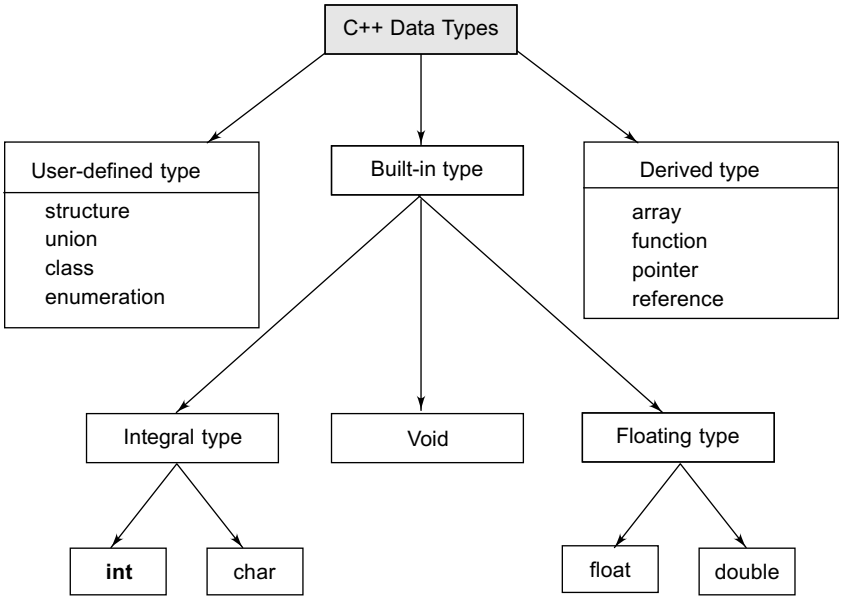


Fig. 6.1 *3 Hierarchy of C++ data types*

the needs of various situations. The modifiers **signed**, **unsigned**, **long**, and **short** may be applied to character and integer basic data types. However, the modifier **long** may also be applied to **double**. Data type representation is machine specific in C++. Table 6.2 lists all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine.

Table 6.2 *Size and range of C++ basic data types*

Type	*Bytes*	*Range*
char	1	–128 to 127
unsigned char	1	0 to 255
signed char	1	– 128 to 127
int	2	– 32768 to 32767
unsigned int	2	0 to 65535
signed int	2	– 31768 to 32767
short int	2	– 31768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	–32768 to 32767
long int	4	–2147483648 to 2147483647
signed long int	4	–2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E–38 to 3.4E+38
double	8	1.7E–308 to 1.7E+308
long double	10	3.4E–4932 to 1.1E+4932

ANSI C++ committee has added two more data types, **bool** and **wchar_t**.

The type **void** was introduced in ANSI C. Two normal uses of **void** are (1) to specify the return type of a function when it is not returning any value, and (2) to indicate an empty argument list to a function. Example:

```
void funct1(void);
```

Another interesting use of **void** is in the declaration of generic pointers. Example:

```
void *gp;      // gp becomes generic pointer
```

A generic pointer can be assigned a pointer value of any basic data type, but it may not be dereferenced. For example,

```
int *ip;       // int pointer
gp = ip;       // assign int pointer to void pointer
```

are valid statements. But, the statement,

```
*ip = *gp;
```

is illegal. It would not make sense to dereference a pointer to a **void** value.

Assigning any pointer type to a **void** pointer without using a cast is allowed in both C++ and ANSI C. In ANSI C, we can also assign a **void** pointer to a non-**void** pointer without using a cast to non-void pointer type. This is not allowed in C++. For example,

```
void *ptr1;
char *ptr2;
ptr2 = ptr1;
```

are all valid statements in ANSI C but not in C++. A **void** pointer cannot be directly assigned to other type pointers in C++. We need to use a cast operator as shown below:

```
ptr2 = (char *)ptr1;
```

6.6 User-Defined Data Types

6.6.1 Enumerated Data Type

An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The **enum** keyword (from C) automatically enumerates a list of words by assigning them values 0, 1, 2, and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an **enum** statement is similar to that of the **struct** statement. Examples:

```
enum shape{circle, square, triangle};
enum colour{red, blue, green, yellow};
enum position{off, on};
```

The enumerated data types differ slightly in C++ when compared with those in ANSI C. In C++, the tag names **shape, colour**, and **position** become new type names. By using these tag names, we can declare new variables. Examples:

```
shape ellipse;        // ellipse is of type shape
colour background;    // background is of type colour
```

ANSI C defines the types of **enums** to be **ints**. In C++, each enumerated data type retains its own separate type. This means that C++ does not permit an **int** value to be automatically converted to an **enum** value. Examples:

```
colour background = blue;          // allowed
colour background = 7;             // Error in C++
colour background = (colour) 7;    // OK
```

However, an enumerated value can be used in place of an **int** value.

```
int c = red;  // valid, colour type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can over-ride the default by explicitly assigning integer values to the enumerators. For example,

```
enum colour{red, blue=4, green=8};
enum colour{red=5, blue, green};
```

are valid definitions. In the first case, **red** is 0 by default. In the second case, **blue** is 6 and **green** is 7. Note that the subsequent initialized enumerators are larger by one than their predecessors.

C++ also permits the creation of anonymous **enums** (i.e., **enums** without tag names). Example:

```
enum{off, on};
```

Here, **off** is 0 and **on** is 1. These constants may be referenced in the same manner as regular constants. Examples:

```
int switch_1 = off;
int switch_2 = on;
```

In practice, enumeration is used to define symbolic constants for a **switch** statement. Example:

```
enum shape
{
  circle,
  rectangle,
  triangle
};

int main()
{
  cout << "Enter shape code:";
  int code;
  cin >> code;
  while(code >= circle && code <= triangle)
  {
        switch(code)
```

```
        {
                case circle:
                ......
                ......
                break;
                case rectangle:
                ......
                ......
                break;
                case triangle:
                ......
                ......
                break;
        }
        cout << "Enter shape code:";
        cin >> code;
    }
    cout << "BYE \n";

    return 0;
}
```

ANSI C permits an **enum** to be defined within a structure or a class, but the **enum** is globally visible. In C++, an **enum** defined within a class (or structure) is local to that class (or structure) only.

6.7 Structures and Unions

Standalone variables of primitive types are not sufficient enough to handle real world problems. It is often required to group logically related data items together. While arrays are used to group together similar type data elements, structures are used for grouping together elements with dissimilar types.

The general format of a structure definition is as follows:

```
struct        name
{
  data_type   member1;
  data_type   member2;
  . . .       . . .
  . . .       . . .
};
```

Let us take the example of a book, which has several attributes such as title, number of pages, price, etc. We can realize a book using structures as shown below:

```
struct book
{
  char title[25];
```

```
    charauthor[25];
    int  pages;
    float price;
};
struct book book1, book2, book3;
```

Here book1, book2 and book3 are declared as variables of the user-defined type book. We can access the member elements of a structure by using the dot (.) operator, as shown below:

```
book1.pages=550;
book2.price=225.75;
```

Unions are conceptually similar to structures as they allow us to group together dissimilar type elements inside a single unit. But there are significant differences between structures and unions as far as their implementation is concerned. The size of a structure type is equal to the sum of the sizes of individual member types. However, the size of a union is equal to the size of its largest member element. For instance, consider the following union declaration:

```
union result
{
   int  marks;
   char grade;
   float percent;
};
```

The union result will occupy four bytes in memory as its largest size member element is the floating type variable percent. However, if we had defined result as a structure then it would have occupied seven bytes in memory that is, the sum of the sizes of individual member elements. Thus, in case of unions the same memory space is used for representing different member elements. As a result, union members can only be manipulated exclusive of each other. In simple words, we can say that unions are memory-efficient alternatives of structures particularly in situations where it is not required to access the different member elements simultaneously.

In C++, structures and unions can be used just like they are used in C. However, there is an interesting aside to C++ structures that we will study in Chapter 11.

6.8 Derived Data Types

6.8.1 Arrays

The application of arrays in C++ is similar to that in C. The only exception is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant. For instance,

```
char string[3] = "xyz";
```

is valid in ANSI C. It assumes that the programmer intends to leave out the null character \0 in the definition. But in C++, the size should be one larger than the number of characters in the string.

```
char string[4] = "xyz";    // O.K. for C++
```

6.8.2 Functions

Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs. Many of these modifications and improvements were driven by the requirements of the object-oriented concept of C++. Some of these were introduced to make the C++ program more reliable and readable. All the features of C++ functions are discussed in Chapter 7.

6.8.3 Pointers

Pointers are declared and initialized as in C. Examples:

```
int *ip;       // int pointer
ip = &x;       // address of x assigned to ip
*ip = 10;      // 10 assigned to x through indirection
```

C++ adds the concept of constant pointer and pointer to a constant.

```
char * const ptr1 = "GOOD";  // constant pointer
```

We cannot modify the address that **ptr1** is initialized to.

```
int const * ptr2 = &m;  // pointer to a constant
```

ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents of what it points to cannot be changed.

We can also declare both the pointer and the variable as constants in the following way:

```
const char * const cp = "xyz";
```

This statement declares cp as a constant pointer to the string which has been declared a constant. In this case, neither the address assigned to the pointer cp nor the contents it points to can be changed.

Pointers are extensively used in C++ for memory management and achieving polymorphism.

6.9 Symbolic Constants

There are two ways of creating symbolic constants in C++:

- Using the qualifier **const**, and
- Defining a set of integer constants using **enum** keyword.

In C++, any value declared as **const** cannot be modified by the program in any way. However, there are some differences in implementation. In C++, we can use **const** in a constant expression, such as

```
const int size = 10;
char name[size];
```

This would be illegal in C. **const** allows us to create typed constants instead of having to use **#define** to create constants that have no type information.

As with **long** and **short**, if we use the **const** modifier alone, it defaults to **int**. For example,

```
const size = 10;
```

means

```
const int size = 10;
```

The *named constants* are just like variables except that their values cannot be changed.

C++ requires a **const** to be initialized. ANSI C does not require an initializer; if none is given, it initializes the **const** to 0.

The scoping of **const** values differs. A **const** in C++ defaults to the internal linkage and therefore it is local to the file where it is declared. In ANSI C, **const** values are global in nature. They are visible outside the file in which they are declared. However, they can be made local by declaring them as **static**. To give a **const** value an external linkage so that it can be referenced from another file, we must explicitly define it as an **extern** in C++. Example:

```
extern const total = 100;
```

Another method of naming integer constants is by enumeration as under;

```
enum {X,Y,Z};
```

This defines X, Y and Z as integer constants with values 0, 1, and 2 respectively. This is equivalent to:

```
const X = 0;
const Y = 1;
const Z = 2;
```

We can also assign values to X, Y, and Z explicitly. Example:

```
enum{X=100, Y=50, Z=200};
```

Such values can be any integer values. Enumerated data type has been discussed in detail in Section 6.6.

6.10 Type Compatibility

C++ is very strict with regard to type compatibility as compared to C. For instance, C++ defines **int, short int**, and **long int** as three different types. They must be cast when their values are assigned to one another. Similarly, **unsigned char, char**, and **signed char** are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility, or else, a cast must be applied. These restrictions in C++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

Another notable difference is the way **char** constants are stored. In C, they are stored as **ints,** and therefore,

```
sizeof ('x')
```

is equivalent to

```
sizeof(int)
```

in C. In C++, however, **char** is not promoted to the size of **int** and therefore

```
sizeof('x')
```

equals

```
sizeof(char)
```

6.11 Declaration of Variables

In C, all variables must be declared before they are used in executable statements. This is true with C++ as well. However, there is a significant difference between C and C++ with regard to the place of their declaration in the program. C requires all the variables to be defined at the beginning of a scope. When we read a C program, we usually come across a group of variable declarations at the beginning of each scope level. Their actual use appears elsewhere in the scope, sometimes far away from the place of declaration. Before using a variable, we should go back to the beginning of the program to see whether it has been declared and, if so, of what type.

C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use.

The example below illustrates this point.

```
int main()
{
       float x;                        // declaration
       float sum = 0;

       for(int i=1; i<5; i++)          // declaration
       {
         cin >> x;
         sum = sum +x;
       }
       float average;                  // declaration
       average = sum/(i-1);
       cout << average;

       return 0;
 }
```

The only disadvantage of this style of declaration is that we cannot see all the variables used in a scope at a glance.

6.12 Dynamic Initialization of Variables

In C, a variable must be initialized using a constant expression, and the C compiler would fix the initialization code at the time of compilation. C++, however, permits initialization of the variables at run time. This is referred to as *dynamic initialization*. In C++, a variable can be initialized at run time using expressions at the place of declaration. For example, the following are valid initialization statements:

```
.....
.....
int n = strlen(string);
.....
float area = 3.14159 * rad * rad;
```

Thus, both the declaration and the initialization of a variable can be done simultaneously at the place where the variable is used for the first time. The following two statements in the example of the previous section

```
float average; // declare where it is necessary
average = sum/i;
```

can be combined into a single statement:

```
float average = sum/i;   // initialize dynamically at run time
```

Dynamic initialization is extensively used in object-oriented programming. We can create exactly the type of object needed, using information that is known only at the run time.

6.13 Reference Variables

C++ introduces a new kind of variable known as the *reference* variable. A reference variable provides an *alias* (alternative name) for a previously defined variable. For example, if we make the variable **sum** a reference to the variable **total**, then **sum** and **total** can be used interchangeably to represent that variable. A reference variable is created as follows:

```
data-type & reference-name = variable-name
```

Example:

```
float total = 100;
float & sum = total;
```

total is a **float** type variable that has already been declared; **sum** is the alternative name declared to represent the variable **total**. Both the variables refer to the same data object in the memory. Now, the statements

```
cout << total;
```

and

```
cout << sum;
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both total and sum to 110. Likewise, the assignment

```
sum = 0;
```

will change the value of both the variables to zero.

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object which it names. It is important to note that the initialization of a reference variable is completely different from assignment to it.

C++ assigns additional meaning to the symbol **&**. Here, **&** is not an address operator. The notation **float &** means reference to **float**. Other examples are:

```
int n[10];
int & x = n[10];      // x is alias for n[10]
char & a = '\n';      // initialize reference to a literal
```

The variable **x** is an alternative to the array element **n[10]**. The variable **a** is initialized to the newline constant. This creates a reference to the otherwise unknown location where the newline constant **\n** is stored.

The following references are also allowed:

```
i.  int x;
    int *p = &x;
    int & m = *p;

ii. int & n = 50;
```

The first set of declarations causes **m** to refer to **x** which is pointed to by the pointer **p** and the statement in (ii) creates an **int** object with value 50 and name **n**.

A major application of reference variables is in passing arguments to functions. Consider the following:

```
void f(int & x)          // uses reference
{
  x = x+10;              // x is incremented; so also m
}
int main()
{
  int m = 10;
  f(m);                  // function call
  .....
  .....
}
```

When the function call **f(m)** is executed, the following initialization occurs:

```
int & x = m;
```

Thus x becomes an alias of m after executing the statement

```
f(m);
```

Such function calls are known as *call by reference*. This implementation is illustrated in Fig. 6.2. Since the variables **x** and **m** are aliases, when the function increments **x, m** is also incremented. The value of **m** becomes 20 after the function is executed. In traditional C, we accomplish this operation using pointers and dereferencing techniques.

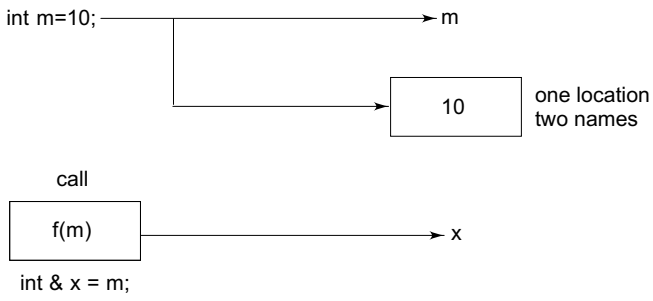


Fig. 6.2 ⇔ *Call by reference mechanism*

The call by reference mechanism is useful in object-oriented programming because it permits the manipulation of objects by reference, and eliminates the copying of object parameters back and forth. It is also important to note that references can be created not only for built-in data types but also for user-defined data types such as structures and classes. References work wonderfully well with these user-defined data types.

6.14 Operators in C++

C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators. We have already seen two such operators, namely, the insertion operator <<, and the extraction operator >>. Other new operators are:

```
::          Scope resolution operator
::*         Pointer-to-member declarator
->*         Pointer-to-member operator
.*          Pointer-to-member operator
delete      Memory release operator
endl        Line feed operator
new         Memory allocation operator
setw        Field width operator
```

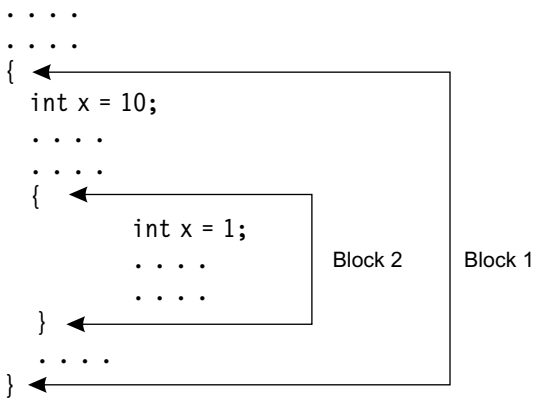
In addition, C++ also allows us to provide new definitions to some of the built-in operators. That is, we can give several meanings to an operator, depending upon the types of arguments used. This process is known as *operator overloading*.

6.15 Scope Resolution Operator

C++ is a block-structured language. Blocks and scopes can be used in constructing programs. Same variable name can be used to have different meanings in different blocks. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside a block is said to be local to that block. Consider the following segment of a program:

```
.....
.....
{
   int x = 10;
   .....
   .....
}
.....
.....
{
   int x = 1;
   .....
   .....
}
```

The two declarations of x refer to two different memory locations containing different values. Statements in the second block cannot refer to the variable x declared in the first block, and vice versa. Blocks in C++ are often nested. For example, the following style is common:



Block2 is contained in block1. Note that a declaration in an inner block *hides* a declaration of the same variable in an outer block and, therefore, each declaration of **x** causes it to refer to a different data object. Within the inner block, the variable **x** will refer to the data object declared therein.

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator :: called the *scope resolution operator*. This can be used to uncover a hidden variable. It takes the following form:

```
:: variable-name
```

This operator allows access to the global version of a variable. For example, ::count means the global version of the variable count (and not the local variable count declared in that block). Program 6.1 illustrates this feature.

SCOPE RESOLUTION OPERATOR

```
#include <iostream>
using namespace std;
int m = 10;      // global m

int main()
{
       int m = 20;    // m redeclared, local to main

       {
              int k = m;
              int m = 30;    // m declared again
                             // local to inner block

              cout <<  "we are in inner block \n";
              cout <<  "k = " << k << "\n";
              cout <<  "m = " << m << "\n";
              cout <<  "::m = " << ::m << "\n";
       }

       cout << "\nWe are in outer block \n";
       cout << "m = " << m << "\n";
       cout << "::m = " << ::m << "\n";

       return 0;
}
```

PROGRAM 6.1

The output of Program 6.1 would be:

```
We are in inner block
k = 20
m = 30
::m = 10

We are in outer block
m = 20
::m = 10
```

In the above program, the variable **m** is declared at three places, namely, outside the **main()** function, inside the **main()**, and inside the inner block.

note

It is to be noted **::m** will always refer to the global **m**. In the inner block, ::m refers to the value 10 and not 20.

A major application of the scope resolution operator is in the classes to identify the class to which a member function belongs. This will be dealt in detail later when the classes are introduced.

6.16 Member Dereferencing Operators

As you know, C++ permits us to define a class containing various types of data and functions as members. C++ also permits us to access the class members through pointers. In order to achieve this, C++ provides a set of three pointer-to-member operators. Table 6.3 shows these operators and their functions.

Table 6.3 *Member dereferencing operators*

Operator	*Function*
::*	To declare a pointer to a member of a class
*	To access a member using object name and a pointer to that member
–>*	To access a member using a pointer to the object and a pointer to that member

Further details on these operators will be meaningful only after we discuss classes, and therefore we defer the use of member dereferencing operators until then.

6.17 Memory Management Operators

C uses **malloc()** and **calloc()** functions to allocate memory dynamically at run time. Similarly, it uses the function **free()** to free dynamically allocated memory. We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. Although C++ supports these functions, it also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way. Since these operators manipulate memory on the free store, they are also known as *free store* operators.

An object can be created by using **new**, and destroyed by using **delete**, as and when required. A data object created inside a block with **new**, will remain in existence until it is explicitly destroyed by using **delete**. Thus, the lifetime of an object is directly under our control and is unrelated to the block structure of the program.

The **new** operator can be used to create objects of any type. It takes the following general form:

```
pointer-variable = new data-type;
```

Here, *pointer-variable* is a pointer of type *data-type*. The **new** operator allocates sufficient memory to hold a data object of type *data-type* and returns the address of the object. The *data-type* may be any valid data type. The *pointer-variable* holds the address of the memory space allocated. Examples:

```
p = new int;
q = new float;
```

where **p** is a pointer of type **int** and **q** is a pointer of type **float**. Here, **p** and **q** must have already been declared as pointers of appropriate types. Alternatively, we can combine the declaration of pointers and their assignments as follows:

```
int *p = new int;
float *q = new float;
```

Subsequently, the statements

```
*p = 25;
*q = 7.5;
```

assign 25 to the newly created **int** object and 7.5 to the **float** object.

We can also initialize the memory using the **new** operator. This is done as follows:

```
pointer-variable = new data-type(value);
```

Here, value specifies the initial value. Examples:

```
int *p = new int(25);
float *q = new float(7.5);
```

As mentioned earlier, **new** can be used to create a memory space for any data type including user-defined types such as arrays, structures and classes. The general form for a one-dimensional array is:

```
pointer-variable = new data-type[size];
```

Here, size specifies the number of elements in the array. For example, the statement

```
int *p = new int[10];
```

creates a memory space for an array of 10 integers. **p[0]** will refer to the first element, **p[1]** to the second element, and so on.

When creating multi-dimensional arrays with **new**, all the array sizes must be supplied.

```
array_ptr = new int[3][5][4];    // legal
array_ptr = new int[m][5][4];    // legal
array_ptr = new int[3][5][ ];    // illegal
array_ptr = new int[ ][5][4];    // illegal
```

The first dimension may be a variable whose value is supplied at runtime. All others must be constants.

The application of **new** to class objects will be discussed later in Chapter 9.

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

```
delete pointer-variable;
```

The *pointer-variable* is the pointer that points to a data object created with **new**. Examples:

```
delete p;
delete q;
```

If we want to free a dynamically allocated array, we must use the following form of **delete**:

```
delete [size] pointer-variable;
```

The *size* specifies the number of elements in the array to be freed. The problem with this form is that the programmer should remember the size of the array. Recent versions of C++ do not require the size to be specified. For example,

```
delete [ ]p;
```

will delete the entire array pointed to by **p**.

What happens if sufficient memory is not available for allocation? In such cases, like **malloc(), new** returns a null pointer. Therefore, it may be a good idea to check for the pointer produced by **new** before using it. It is done as follows:

```
.....
.....
p = new int;
if(!p)
{
  cout << "allocation failed \n";
}
.....
.....
```

The **new** operator offers the following advantages over the function **malloc()**.

1. It automatically computes the size of the data object. We need not use the operator **sizeof**.
2. It automatically returns the correct pointer type, so that there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. Like any other operator, **new** and **delete** can be overloaded.

6.18 Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw**.

The **endl** manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character "\n". For example, the statement

```
.....
.....
cout << "m = " << m << endl
     << "n = " << n << endl
     << "p = " << p << endl;
.....
.....
```

would cause three lines of output, one for each variable. If we assume the values of the variables as 2597, 14, and 175 respectively, the output will appear as follows:

```
m = 2597
n = 14
p = 175
```

It is important to note that this form is not the ideal output. It should rather appear as under:

```
m =  2597
n =    14
p =   175
```

Here, the numbers are *right-justified*. This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The **setw** manipulator does this job. It is used as follows:

```
cout << setw(5) << sum << endl;
```

The manipulator **setw(5)** specifies a field width 5 for printing the value of the variable sum. This value is right-justified within the field as shown below:

```
|  |  | 3 | 4 | 5 |
```

Program 6.2 illustrates the use of **endl** and **setw**.

USE OF MANIPULATORS

```
#include <iostream>
#include <iomanip>   // for setw

using namespace std;

int main()
{
     int Basic = 950, Allowance = 95, Total = 1045;

     cout << setw(10) << "Basic" << setw(10) << Basic << endl
```

```
            << setw(10) << "Allowance" << setw(10) << Allowance << endl
            << setw(10) << "Total" << setw(10) << Total << endl;

        return 0;
    }
```

PROGRAM 6.2

Output of this program is given below:

```
     Basic      950
Allowance       95
     Total     1045
```

note

Character strings are also printed right-justified.

We can also write our own manipulators as follows:

```
#include <iostream>
ostream & symbol(ostream & output)
{
  return output << "\tRs";
}
```

The **symbol** is the new manipulator which represents **Rs**. The identifier **symbol** can be used whenever we need to display the string **Rs**.

6.19 Type Cast Operator

C++ permits explicit type conversion of variables or expressions using the type cast operator.

Traditional C casts are augmented in C++ by a function-call notation as a syntactic alternative. The following two versions are equivalent:

```
(type-name) expression    // C notation
type-name (expression)    // C++ notation
```

Examples:

```
average = sum/(float)i;    // C notation
average = sum/float(i);    // C++ notation
```

A type-name behaves as if it is a function for converting values to a designated type. The function-call notation usually leads to simplest expressions. However, it can be used only if the type is an identifier. For example,

```
p = int * (q);
```

is illegal. In such cases, we must use C type notation.

```
p = (int *) q;
```

Alternatively, we can use **typedef** to create an identifier of the required type and use it in the functional notation.

```
typedef int * int_pt;
p = int_pt(q);
```

ANSI C++ adds the following new cast operators:

- const_cast
- static_cast
- dynamic_cast
- reinterpret_cast

6.20 Expressions and Their Types

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values. An expression may consist of one or more operands, and zero or more operators to produce a value. Expressions may be of the following seven types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

An expression may also use combinations of the above expressions. Such expressions are known as *compound expressions*.

6.20.1 Constant Expressions

Constant Expressions consist of only constant values. Examples:

```
15
20 + 5 / 2.0
'x'
```

6.20.2 Integral Expressions

Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions. Examples:

```
m
m * n - 5
```

```
m * 'x'
5 + int(2.0)
```

where **m** and **n** are integer variables.

6.20.3 Float Expressions

Float Expressions are those which, after all conversions, produce floating-point results. Examples:

```
x + y
x * y / 10
5 + float(10)
10.75
```

where **x** and **y** are floating-point variables.

6.20.4 Pointer Expressions

Pointer Expressions produce address values. Examples:

```
&m
ptr
ptr + 1
"xyz"
```

where **m** is a variable and **ptr** is a pointer.

6.20.5 Relational Expressions

Relational Expressions yield results of type **bool** which takes a value **true** or **false**. Examples:

```
x <= y
a+b == c+d
m+n > 100
```

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as *Boolean expressions*.

6.20.6 Logical Expressions

Logical Expressions combine two or more relational expressions and produces **bool** type results. Examples:

```
a>b  &&  x==10
x==10  ||  y==5
```

6.20.7 Bitwise Expressions

Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits. Examples:

```
x << 3    // Shift three bit position to left
y >> 1    // Shift one bit position to right
```

Shift operators are often used for multiplication and division by powers of two.

ANSI C++ has introduced what are termed as *operator keywords* that can be used as alternative representation for operator symbols.

6.21 Special Assignment Expressions

6.21.1 Chained Assignment

```
x = (y = 10);
     or
x = y = 10;
```

First 10 is assigned to y and then to x.

A chained statement cannot be used to initialize variables at the time of declaration. For instance, the statement

```
float a = b = 12.34;   // wrong
```

is illegal. This may be written as

```
float a=12.34, b=12.34   // correct
```

6.21.2 Embedded Assignment

```
x = (y = 50) + 10;
```

(y = 50) is an assignment expression known as embedded assignment. Here, the value 50 is assigned to y and then the result 50+10 = 60 is assigned to x. This statement is identical to

```
y = 50;
x = y + 10;
```

6.21.3 Compound Assignment

Like C, C++ supports a *compound assignment operator* which is a combination of the assignment operator with a binary arithmetic operator. For example, the simple assignment statement

```
x = x + 10;
```

may be written as

```
x += 10;
```

The operator += is known as *compound assignment operator* or *short-hand assignment operator*. The general form of the compound assignment operator is:

```
variable1 op= variable2;
```

where *op* is a binary arithmetic operator. This means that

```
variable1 = variable1 op variable2;
```

6.22 Implicit Conversions

We can mix data types in expressions. For example,

```
m = 5+2.75;
```

is a valid statement. Wherever data types are mixed in an expression, C++ performs the conversions automatically. This process is known as *implicit* or *automatic conversion*.

When the compiler encounters an expression, it divides the expressions into sub-expressions consisting of one operator and one or two operands. For a binary operator, if the operands type differ, the compiler converts one of them to match with the other, using the rule that the "smaller" type is converted to the "wider" type. For example, if one of the operand is an **int** and the other is a **float**, the **int** is converted into a **float** because a **float** is wider than an **int**. The "water-fall" model shown in **Fig. 6.3** illustrates this rule.

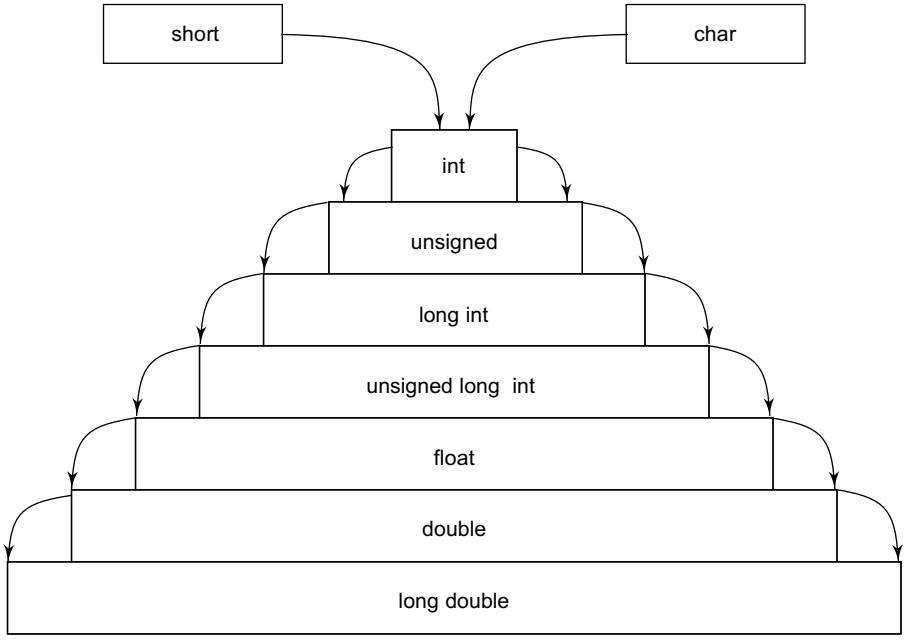


Fig. 6.3 ⇔ *Water-fall model of type conversion*

Whenever a **char** or **short int** appears in an expression, it is converted to an **int**. This is called *integral widening conversion*. The implicit conversion is applied only after completing all integral widening conversions.

Table 6.4 *Results of Mixed-mode Operations*

RHO / LHO	char	short	int	long	float	double	long double
char	int	int	int	long	float	double	long double
short	int	int	int	long	float	double	long double
int	int	int	int	long	float	double	long double
long	long	long	long	long	float	double	long double
float	float	float	float	float	float	double	long double
double	double	double	double	double	double	double	long double
long double	long double	long double	long double	long double	long double	long double	long double

RHO – Right-hand operand
LHO – Left-hand operand

6.23 Operator Overloading

As stated earlier, overloading means assigning different meanings to an operation, depending on the context. C++ permits overloading of operators, thus allowing us to assign multiple meanings to operators. The number and type of operands decide the nature of operation to follow.

The input/output operators << and >> are good examples of operator overloading. Although the built-in definition of the << operator is for shifting of bits, it is also used for displaying the values of various data types. This has been made possible by the header file *iostream* where a number of overloading definitions for << are included. Thus, the statement

```
cout << 75.86;
```

invokes the definition for displaying a **double** type value, and

```
cout << "well done";
```

invokes the definition for displaying a **char** value. However, none of these definitions in *iostream* affect the built-in meaning of the operator.

Similarly, we can define additional meanings to other C++ operators. For example, we can define + operator to add two structures or objects. Almost all C++ operators can be overloaded with a few exceptions such as the member-access operators (**.** and **.***), conditional operator (?:), scope resolution operator (::) and the size operator **(sizeof)**. Definitions for operator overloading are discussed in detail in Chapter 10.

6.24 Operator Precedence

Although C++ enables us to add multiple meanings to the operators, yet their association and precedence remain the same. For example, the multiplication operator will continue having higher precedence than the add operator. Table 6.5 gives the precedence and associativity of all the C++ operators. The groups are listed in the order of decreasing precedence. The labels *prefix* and *postfix* distinguish the uses of ++ and --. Also, the symbols +, –, *, and & are used as both unary and binary operators.

A complete list of ANSI C++ operators and their meanings, precedence, associativity and use are given in Appendix B.

Table 6.5 *Operator precedence and associativity*

Operator	*Associativity*
::	left to right
–> . ( ) [ ] postfix ++ postfix – –	left to right
prefix ++ prefix – – ~ ! unary + unary – unary * unary & (type) sizeof new delete	right to left
–> * *	left to right
* / %	left to right
+ –	left to right
<< >>	left to right
<< = >> =	left to right
= = !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
| |	left to right
?:	left to right
= * = / = % = + = =	right to left
<< = >> = & = ^= |=	left to right
, (comma)	

The unary operations assume higher precedence.

6.25 Control Structures

In C++, a large number of functions are used that pass messages, and process the data contained in objects. A function is set up to perform a task. When the task is complex, many different algorithms can be designed to achieve the same goal. Some are simple to comprehend, while others are not. Experience has also shown that the number of bugs that occur is related to the format of the program. The format should be such that it is easy to trace the flow of execution of statements. This would help not only in debugging but also in the review and maintenance of the program later. One method of achieving the objective of an accurate, error-resistant and maintainable code is to use one or any combination of the following three control structures:

1. Sequence structure (straight line)
2. Selection structure (branching)
3. Loop structure (iteration or repetition)

Figure 6.4 shows how these structures are implemented using *one-entry, one-exit* concept, a popular approach used in modular programming.

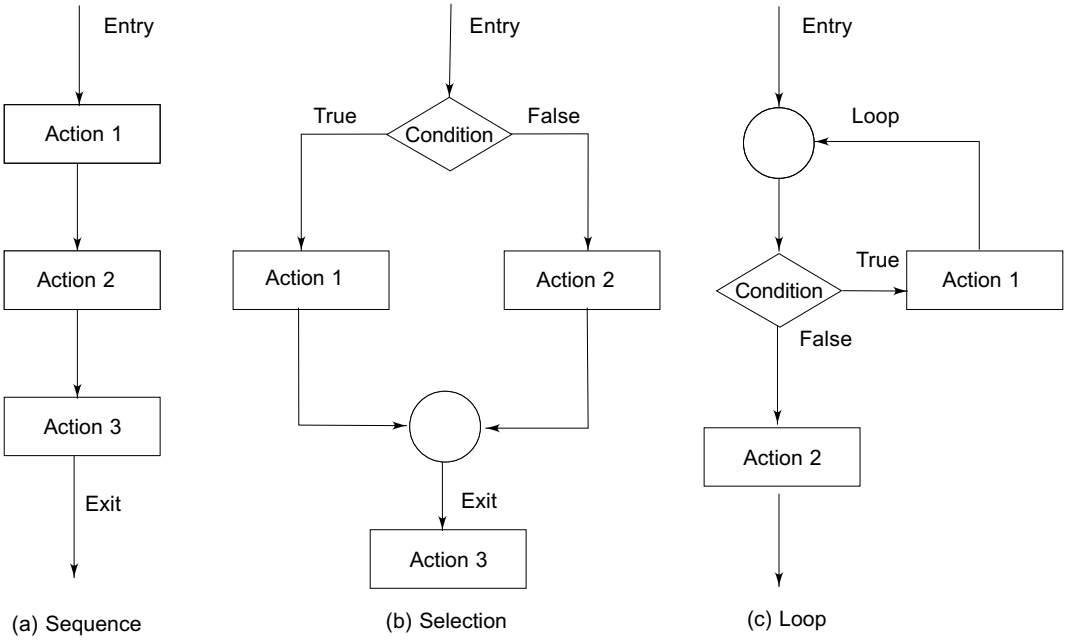


Fig. 6.4 ⇔ *Basic control structures*

It is important to understand that all program processing can be coded by using only these three logic structures. The approach of using one or more of these basic control constructs in programming is known as *structured programming*, an important technique in software engineering.

Using these three basic constructs, we may represent a function structure either in detail or in summary form as shown in Figs 6.5 (a), (b) and (c).

Like C, C++ also supports all the three basic control structures, and implements them using various control statements as shown in Fig. 6.6. This shows that C++ combines the power of structured programming with the object-oriented paradigm.

6.25.1 The if Statement

The **if** statement is implemented in two forms:

- Simple **if** statement
- **if...else** statement

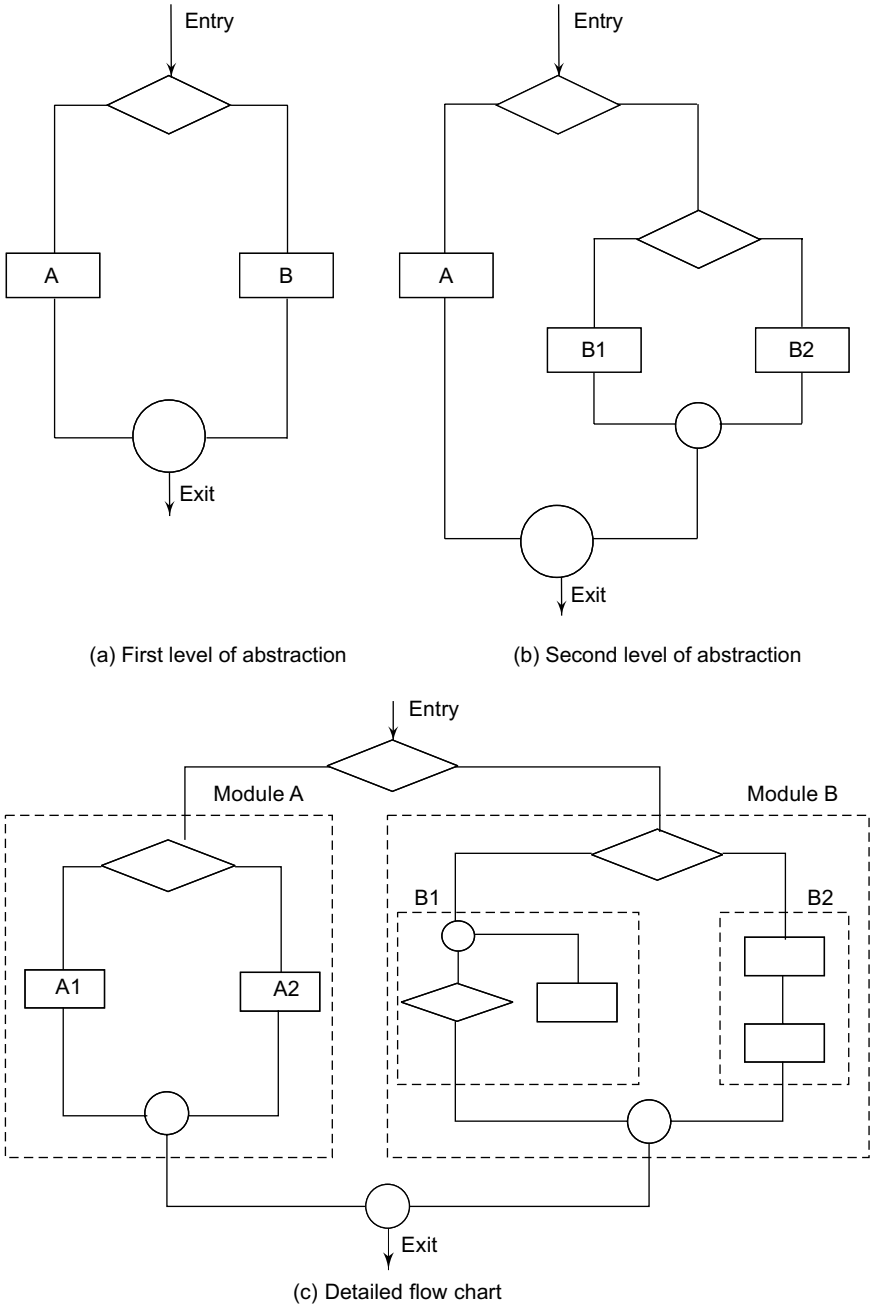


Fig. 6.5 ⇔ *Different levels of abstraction*

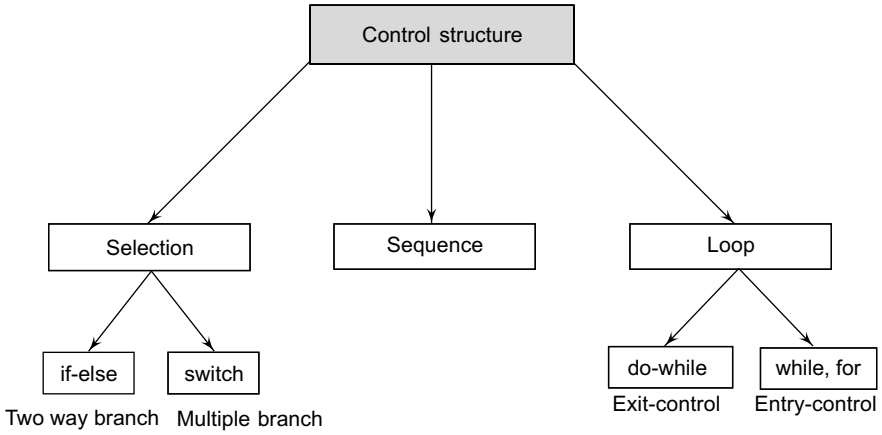


Fig. 6.6 ⇔ *C++ statements to implement in two forms*

Examples:

Form 1

```
if(expression is true)
{
   action1;
}
action2;
action3;
```

Form 2

```
if(expression is true)
{
   action1;
}
else
{
   action2;
}
action3;
```

6.25.2 The Switch Statement

This is a multiple-branching statement where, based on a condition, the control is transferred to one of the many possible points. This is implemented as follows:

```
switch(expression)
{
  case1:
```

```
    {
    action1;
    }
    case2:
    {
    action2;
    }
    case3:
    {
    action3;
    }
    default:
    {
    action4;
    }
}
action5;
```

6.25.3 The do-while statement

The **do-while** is an *exit-controlled* loop. Based on a condition,the control is transferred back to a particular point in the program. The syntax is as follows:

```
do
{
  action1;
}
while(condition is true);
action2;
```

6.25.4 The while Statement

This is also a loop structure, but is an *entry-controlled* one. The syntax is as follows:

```
while(condition is true)
{
  action1;
}
action2;
```

6.25.5 The for Statement

The **for** is an *entry-entrolled* loop and is used when an action is to be repeated for a predetermined number of times. The syntax is as follows:

```
for(initial value; test; increment)
{
  action1;
```

```
}
action2;
```

The syntax of the control statements in C++ is very much similar to that of C and therefore they are implemented as and when they are required.

SUMMARY

- C++ provides various types of tokens that include keywords, identifiers, constants, strings, and operators.
- Identifiers refer to the names of variables, functions, arrays, classes, etc.
- C++ provides an additional use of **void**, for declaration of generic pointers.
- The enumerated data types differ slightly in C++. The tag names of the enumerated data types become new type names. That is, we can declare new variables using these tag names.
- In C++, the size of character array should be one larger than the number of characters in the string.
- C++ adds the concept of constant pointer and pointer to constant. In case of constant pointer we can not modify the address that the pointer is initialized to. In case of pointer to a constant, contents of what it points to cannot be changed.
- Pointers are widely used in C++ for memory management and to achieve polymorphism.
- C++ provides a qualifier called **const** to declare named constants which are just like variables except that their values can not be changed. A **const** modifier defaults to an **int**.
- C++ is very strict regarding type checking of variables. It does not allow to equate variables of two different data types. The only way to break this rule is type casting.
- C++ allows us to declare a variable anywhere in the program, as also its initialization at run time, using the expressions at the place of declaration.
- A reference variable provides an alternative name for a previously defined variable. Both the variables refer to the same data object in the memory. Hence, change in the value of one will also be reflected in the value of the other variable.
- A reference variable must be initialized at the time of declaration, which establishes the correspondence between the reference and the data object that it names.
- A major application of the scope resolution (::) operator is in the classes to identify the class to which a member function belongs.
- In addition to **malloc(), calloc()** and **free()** functions, C++ also provides two unary operators, **new** and **delete** to perform the task of allocating and freeing the memory in a better and easier way.
- C++ also provides manipulators to format the data display. The most commonly used manipulators are **endl** and **setw**.

- C++ supports seven types of expressions. When data types are mixed in an expression, C++ performs the conversion automatically using certain rules.
- C++ also permits explicit type conversion of variables and expressions using the type cast operators.
- Like C, C++ also supports the three basic control structures namely, sequence, selection and loop, and implements them using various control statements such as, **if, if...else, switch, do..while, while** and **for**.

Key Terms

- array
- associativity
- automatic conversion
- backslash character
- bitwise expression
- **bool**
- boolean expression
- branching
- call by reference
- **calloc()**
- character constant
- chained assignment
- **class**
- compound assignment
- compound expression
- **const**
- constant
- constant expression
- control structure
- data types
- decimal integer
- declaration
- **delete**
- dereferencing
- derived-type
- **do...while**
- embedded assignment
- **endl**
- entry control
- enumeration
- exit control
- explicit conversion
- expression
- float expression
- floating point integers
- **for**
- formatting
- free store
- **free()**
- function
- hexadecimal integer
- identifier
- **if**
- **if...else**
- implicit conversion
- initialization
- integer constant
- integral expression
- integral widening
- iteration

- keyword
- literal
- logical expression
- loop
- loop structure
- **malloc()**
- manipulator
- memory
- named constant
- **new**
- octal integer
- operator
- operator keywords
- operator overloading
- operator precedence
- pointer
- pointer expression
- pointer variable
- reference
- reference variable
- relational expression
- repetition
- scope resolution
- selection
- selection structure
- sequence
- sequence structure
- **setw**
- short-hand assignment
- **sizeof()**
- straight line
- **string**
- string constant
- **struct**
- structure
- structured programming
- switch
- symbolic constant
- token
- type casting
- type compatibility
- **typedef**
- **union**
- user-defined type
- variable
- **void**
- water-fall model
- **wchar_t**
- **while**
- wide-character

Review Questions

6.1 *Enumerate the rules of naming variables in C++. How do they differ from ANSI C rules?*

6.2 *An **unsigned int** can be twice as large as the **signed int**. Explain how?*

6.3 *Why does C++ have type modifiers?*

6.4 *What are the applications of **void** data type in C++?*

6.5 *Can we assign a **void** pointer to an **int** type pointer? If not, why? How can we achieve this?*

6.6 *Describe, with examples, the uses of enumeration data types.*

6.7 *Describe the differences in the implementation of* ***enum*** *data type in ANSI C and C++.*

6.8 *Why is an array called a derived data type?*

6.9 *The size of a* ***char*** *array that is declared to store a string should be one larger than the number of characters in the string. Why?*

6.10 *The* ***const*** *was taken from C++ and incorporated in ANSI C, although quite differently. Explain.*

6.11 *How does a constant defined by* ***const*** *differ from the constant defined by the preprocessor statement* ***#define****?*

6.12 *In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?*

6.13 *What do you mean by dynamic initialization of a variable? Give an example.*

6.14 *What is a reference variable? What is its major use?*

6.15 *List at least four new operators added by C++ which aid OOP.*

6.16 *What is the application of the scope resolution operator :: in C++?*

6.17 *What are the advantages of using* ***new*** *operator as compared to the function* ***malloc()****?*

6.18 *Illustrate with an example, how the* ***setw*** *manipulator works.*

6.19 *How do the following statements differ?*

(a) *char * const p;*

(b) *char const *p;*

Debugging Exercises

6.1 What will happen when you execute the following code?

```
#include <iostream.h>
void main()
{
     int i=0;
     i=400*400/400;
     cout << i;
}
```

6.2 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
     int num[]={1,2,3,4,5,6};
     num[1]==[1]num  ?    cout<<"Success"     :     cout<<"Error";
}
```

6.3 Identify the errors in the following program.

```
#include <iostream.h>
void main()
{
  int i=5;
  while(i)
  {
       switch(i)
       {
       default:
       case 4:
       case 5:

       break;

       case 1:
       continue;

       case 2:
       case 3:
       break;

       }
       i–;
  }
}
```

6.4 Identify the error in the following program.

```
#include <iostream.h>
#define pi 3.14
int squareArea(int &);
int circleArea(int &);

void main()
{
     int a=10;
     cout << squareArea(a) << " ";
     cout << circleArea(a) << " ";
     cout << a << endl;
}

int squareArea(int &a)
```

```
{
     return a *== a;
}

int circleArea(int &r)
{
     return r = pi * r * r;
}
```

6.5 Identify the error in the following program.

```
#include <iostream.h>
#include <malloc.h>

char* allocateMemory();

void main()
{
     char* str;
     str = allocateMemory();
     cout << str;
     delete str;
     str = "";
     cout << str;
}

char* allocateMemory()
{
     str = "Memory allocation test. ";
     return str;
}
```

6.6 Find errors, if any, in the following C++ statements.

(a) long float x;
(b) char *cp = vp; // vp is a void pointer
(c) int code = three; // three is an enumerator
(d) int *p = new; // allocate memory with new
(e) enum (green, yellow, red);
(f) int const *p = total;
(g) const int array_size;
(h) for (i=1; int i<10; i++) cout << i << "\n";
(i) int & number = 100;
(j) float *p = new int [10];
(k) int public = 1000;
(l) char name[3] = "USA";

Programming Exercises

6.1 *Write a function using reference variables as arguments to swap the values of a pair of integers.*

6.2 *Write a function that creates a vector of user-given size* ***M*** *using* ***new*** *operator.*

6.3 *Write a program to print the following output using* ***for*** *loops.*

```
1
22
333
4444
55555
.....
```

6.4 *Write a program to evaluate the following investment equation*

$V = P(1 + r)^n$

and print the tables which would give the value of V for various combination of the following values of P, r and n:

P: 1000, 2000, 3000, ..., 10,000

r: 0.10, 0.11, 0.12, ..., 0.20

n: 1, 2, 3, ..., 10

(*Hint: P is the principal amount and V is the value of money at the end of n years. This equation can be recursively written as*

$V = P(1 + r)$

$P = V$

In other words, the value of money at the end of the first year becomes the principal amount for the next year, and so on.

6.5 *An election is contested by five 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.*

6.6 *A cricket team has the following table of batting figures for a series of test matches:*

Player's name	*Runs*	*Innings*	*Times not out*
Sachin	8430	230	18
Saurav	4200	130	9
Rahul	3350	105	11
.	.	.	.
.	.	.	.

Write a program to read the figures set out in the above form, to calculate the batting averages and to print out the complete table including the averages.

6.7 *Write programs to evaluate the following functions to 0.0001% accuracy.*

(a) $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots\cdots$

(b) $SUM = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \cdots\cdots$

(c) $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots\cdots$

6.8 *Write a program to print a table of values of the function*
$y = e^{-x}$
for x varying from 0 to 10 in steps of 0.1. The table should appear as follows.

TABLE FOR Y = EXP [–X]

X	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*
0.0									
1.0									
.									
.									
.									
9.0									

6.9 *Write a program to calculate the variance and standard deviation of N numbers.*

$$\text{Variance} = \frac{1}{x}\sum_{i=1}^{N}(x_1 - \bar{x})^2$$

$$\text{Standard Deviation} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

where $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_1$

6.10 *An electricity board charges the following rates to domestic users to discourage large consumption of energy:*

For the first 100 units — 60P per unit

For next 200 units — 80P per unit

Beyond 300 units — 90P per unit

All users are charged a minimum of Rs. 50.00. If the total amount is more than Rs. 300.00 then an additional surcharge of 15% is added.

Write a program to read the names of users and number of units consumed and print out the charges with names.

7

Functions in C++

Key Concepts

- Return types in main()
- Function prototyping
- Call by reference
- Call by value
- Return by reference
- Inline functions
- Default arguments
- Constant arguments
- Function overloading

7.1 Introduction

Functions play an important role in realizing the concept of encapsulation and data hiding in object oriented programming. They are the only means of accessing and manipulating member data elements hidden inside the private section of a class. In C++, there are a number of interesting concepts surrounding functions; first and foremost amongst them is the main function. No matter how big a program is in terms of the number of classes, it is the main function where the execution of the program begins. The main function instantiates the class objects and also contains the overall logic of the program.

Functions often operate on data values passed as arguments during function calls. These values are copied into new variables by the function before executing its other statements. The computed values are then returned back to the calling function so that the original variables can be updated. To avoid so much copying of data that takes place in routine function calls, C++ provides a brilliant feature called reference variables. These variables are simple aliases to the actual variables that means, both the variable names refer to the same location in memory. Thus, any change made to the reference variable by the function changes the original value stored in the memory.

Another C++ concept of great significance is the concept of function overloading. It allows us to write multiple function definitions with the same name. The only point of distinction

between these function definitions is in their number and type of arguments. The choice of the function to be executed is made by the compiler by analysing the type and number of arguments passed in the function call.

```
void show();   /* Function declaration */
main()
{
     .....
     show();        /* Function call */
     .....
}
void show()    /* Function definition */
{
     .....
     .....     /* Function body */
     .....
}
```

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered. C++ is no exception. Functions continue to be the building blocks of C++ programs.

This chapter will focus on all these concepts in more detail.

7.2 The Main Function

C does not specify any return type for the **main()** function which is the starting point for the execution of a program. The definition of **main()** would look like this:

```
main()
{
      // main program statements
}
```

This is perfectly valid because the **main()** in C does not return any value.

In C++, the **main()** returns a value of type **int** to the operating system. C++, therefore, explicitly defines **main()** as matching one of the following prototypes:

```
int main();
int main(int argc, char * argv[]);
```

The functions that have a return value should use the **return** statement for termination. The **main()** function in C++ is, therefore, defined as follows:

```
int main()
{
```

```
    ......
    ......

    return 0;
}
```

Since the return type of functions is **int** by default, the keyword **int** in the **main()** header is optional. Most C++ compilers will generate an error or warning if there is no **return** statement. Turbo C++ issues the warning

```
Function should return a value
```

and then proceeds to compile the program. It is good programming practice to actually return a value from **main()**.

Many operating systems test the return value (called *exit value*) to determine if there is any problem. The normal convention is that an exit value of zero means the program ran successfully, while a nonzero value means there was a problem. The explicit use of a **return(0)** statement will indicate that the program was successfully executed.

7.3 Function Prototyping

Function *prototyping* is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. With function prototyping, a *template* is always used when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself. These checks and controls did not exist in the conventional C functions.

While C++ makes the prototyping essential, ANSI C makes it optional, perhaps, to preserve the compatibility with classic C.

Function prototype is a *declaration statement* in the calling program and is of the following form:

```
type function-name (argument-list);
```

The *argument-list* contains the types and names of arguments that must be passed to the function.

Example:

```
float volume(int x, float y, float z);
```

Note that each argument variable must be declared independently inside the parentheses. That is, a combined declaration like

```
float volume(int x, float y, z);
```

is illegal.

In a function declaration, the names of the arguments are *dummy* variables and therefore, they are optional. That is, the form

```
float volume(int, float, float);
```

is acceptable at the place of declaration. At this stage, the compiler only checks for the type of arguments when the function is called.

In general, we can either include or exclude the variable names in the argument list of prototypes. The variable names in the prototype just act as placeholders and, therefore, if names are used, they don't have to match the names used in the *function call or function definition*.

In the function definition, names are required because the arguments must be referenced inside the function. Example:

```
float volume(int a,float b,float c)
{
     float v = a*b*c;
     .....
     .....
}
```

The function **volume()** can be invoked in a program as follows:

```
float cube1 = volume(b1,w1,h1);  // Function call
```

The variable **b1, w1**, and **h1** are known as the actual parameters which specify the dimensions of **cube1**. Their types (which have been declared earlier) should match with the types declared in the prototype. Remember, the calling statement should not include type names in the argument list.

We can also declare a function with an *empty argument list*, as in the following example:

```
void display( );
```

In C++, this means that the function does not pass any parameters. It is identical to the statement

```
void display(void);
```

However, in C, an empty parentheses implies any number of arguments. That is, we have foregone prototyping. A C++ function can also have an 'open' parameter list by the use of ellipses in the prototype as shown below:

```
void do_something(...);
```

7.4 Call by Reference

In traditional C, a function call passes arguments by value. The *called function* creates a new set of variables and copies the values of arguments into them. The function does not have access

to the actual variables in the calling program and can only work on the copies of values. This mechanism is fine if the function does not need to alter the values of the original variables in the calling program. But, there may arise situations where we would like to change the values of variables in the *calling program*. For example, in bubble sort, we compare two adjacent elements in the list and interchange their values if the first element is greater than the second. If a function is used for *bubble sort*, then it should be able to alter the values of variables in the calling function, which is not possible if the call-by-value method is used.

Provision of the *reference variables* in C++ permits us to pass parameters to the functions by reference. When we pass arguments by reference, the 'formal' arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data. Consider the following function:

```
void swap(int &a,int &b)     // a and b are reference variables
{
     int t = a;              // Dynamic initialization
     a = b;
     b = t;
}
```

Now, if **m** and **n** are two integer variables, then the function call

```
swap(m, n);
```

will exchange the values of **m** and **n** using their aliases (reference variables) **a** and **b**. Reference variables have been discussed in detail in Chapter 3. In traditional C, this is accomplished using *pointers* and *indirection* as follows:

```
void swap1(int *a, int *b)   /* Function definition */
{
      int t;
      t = *a;      /* assign the value at address a to t */
      *a = *b;     /* put the value at b into a  */
      *b = t;      /* put the value at t into b  */
}
```

This function can be called as follows:

```
swap1(&x, &y);   /* call by passing */
                 /* addresses of variables */
```

This approach is also acceptable in C++. Note that the call-by-reference method is neaterin its approach.

7.5 Return by Reference

A function can also return a reference. Consider the following function:

```
int & max(int &x,int &y)
```

```
{
     if (x > y)
       return x;
     else
       return y;
}
```

Since the return type of **max()** is **int &**, the function returns reference to **x** or **y** (and not the values). Then a function call such as **max(a, b)** will yield a reference to either **a** or **b** depending on their values. This means that this function call can appear on the left-hand side of an assignment statement. That is, the statement

```
max(a,b) = -1;
```

is legal and assigns –1 to **a** if it is larger, otherwise -1 to **b**.

7.6 Inline Functions

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as *macros*. Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.

C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called *inline function*. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code (something similar to macros expansion). The inline functions are defined as follows:

```
inline function-header
{
   function body
}
```

Example:

```
inline double cube(double a)
{
  return(a*a*a);
}
```

The above inline function can be invoked by statements like

```
c = cube(3.0);
d = cube(2.5+1.5);
```

On the execution of these statements, the values of c and d will be 27 and 64 respectively. If the arguments are expressions such as 2.5 + 1.5, the function passes the value of the expression, 4 in this case. This makes the inline feature far superior to macros.

It is easy to make a function inline. All we need to do is to prefix the keyword **inline** to the function definition. All inline functions must be defined before they are called.

We should exercise care before making a function **inline**. The speed benefits of **inline** functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of **inline** functions may be lost. In such cases, the use of normal functions will be more meaningful. Usually, the functions are made inline when they are small enough to be defined in one or two lines. Example:

```
inline double cube(double a) {return(a*a*a);}
```

Remember that the inline keyword merely sends a request, not a command, to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a **switch**, or a **goto** exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain **static** variables.
4. If **inline** functions are recursive.

note

Inline expansion makes a program run faster because the overhead of a function call and return is eliminated. However, it makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called. So, a trade-off becomes necessary.

Program 7.1 illustrates the use of inline functions.

INLINE FUNCTIONS

```
#include <iostream>

using namespace std;

     inline float mul(float x, float y)
     {
          return(x*y);
```

```
}

inline double div(double p, double q)
{
    return(p/q);
}

int main()
{
    float a = 12.345;
    float b = 9.82;

    cout << mul(a,b) << "\n";
    cout << div(a,b) << "\n";

    return 0;
}
```

PROGRAM 7.1

The output of program 4.1 would be

```
121.228
1.25713
```

7.7 Default Arguments

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a *default value* to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. Here is an example of a prototype (i.e. function declaration) with default values:

```
float amount(float principal,int period,float rate=0.15);
```

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument **rate**. A subsequent function call like

```
value = amount(5000,7);       // one argument missing
```

passes the value of 5000 to **principal** and 7 to **period** and then lets the function use default value of 0.15 for rate. The call

```
value = amount(5000,5,0.12);      // no missing argument
```

passes an explicit value of 0.12 to **rate**.

A default argument is checked for type at the time of declaration and evaluated at the time of call. One important point to note is that only the trailing arguments can have default values and therefore we must add defaults from *right to left*. We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

```
int mul(int i, int j=5, int k=10);       // legal
int mul(int i=5, int j);                 // illegal
int mul(int i=0, int j, int k=10);       // illegal
int mul(int i=2, int j=5, int k=10);     // legal
```

Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a particular period of deposit. It also provides a greater flexibility to the programmers. A function can be written with more parameters than are required for its most common application. Using default arguments, a programmer can use only those arguments that are meaningful to a particular situation. Program 7.2 illustrates the use of default arguments.

DEFAULT ARGUMENTS

```
#include <iostream>

using namespace std;

int main()
{
        float amount;

        float value(float p, int n, float r=0.15);    // prototype
        void printline(char ch='*', int len=40);   // prototype

        printline();           // uses default values for arguments

        amount = value(5000.00,5);          // default for 3rd argument

        cout << "\n       Final Value = " << amount << "\n\n";

        printline('=');       // use default value for 2nd argument

        return 0;
}
/*-------------------------------------------------------*/
float value(float p, int n, float r)
{
        int year = 1;
        float sum = p;
```

```
        while(year <= n)
        {
                sum = sum*(1+r);
                year = year+1;
        }
        return(sum);
}

void printline(char ch, int len)
{
        for(int i=1; i<=len; i++) printf("%c",ch);
        printf("\n");
}
```

PROGRAM 7.2

The output of Program 7.2 would be

```
************************************
     Final Value = 10056.8

====================================
```

Advantages of providing the default arguments are:

1. We can use default arguments to add new parameters to the existing functions.
2. Default arguments can be used to combine similar functions into one.

7.8 const Arguments

In C++, an argument to a function can be declared as const as shown below.

```
int strlen(const char *p);
int length(const string &s);
```

The qualifier **const** tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

7.9 Function Overloading

As stated earlier, *overloading* refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as *function polymorphism* in OOP.

Using the concept of function overloading; we can design a family of functions with one function name but with different argument lists. The function would perform different

operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded **add()** function handles different types of data as shown below:

```
// Declarations
int add(int a, int b);                    // prototype 1
int add(int a, int b, int c);             // prototype 2
double add(double x, double y);           // prototype 3
double add(int p, double q);              // prototype 4
double add(double p, int q);              // prototype 5

// Function calls
cout << add(5, 10);               // uses prototype 1
cout << add(15, 10.0);            // uses prototype 4
cout << add(12.5, 7.5);           // uses prototype 3
cout << add(5, 10, 15);           // uses prototype 2
cout << add(0.75, 5);             // uses prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

   ```
   char to int
   float to double
   ```

   to find a match.
3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

   ```
   long square(long n)
   double square(double x)
   ```

   A function call such as

   ```
   square(10)
   ```

   will cause an error because **int** argument can be converted to either **long** or **double**, thereby creating an ambiguous situation as to which version of **square()** should be used.
4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

Program 7.3 illustrates function overloading.

FUNCTION OVERLOADING

```
// Function volume() is overloaded three times

#include <iostream>

using namespace std;

// Declarations (prototypes)
int volume(int);
double volume(double, int);
long volume(long, int, int);

int main()
{
        cout << volume(10) << "\n";
        cout << volume(2.5,8) << "\n";
        cout << volume(100L,75,15) << "\n";

        return 0;
}

// Function definitions

int volume(int s)  // cube
{
        return(s*s*s);
}

double volume(double r, int h)   // cylinder
{
        return(3.14519*r*r*h);
}

long volume(long l, int b, int h) // rectangular box
{
        return(l*b*h);
}
```

PROGRAM 7.3

The output of Program 7.3 would be:

```
1000
157.26
112500
```

Overloading of the functions should be done with caution. We should not overload unrelated functions and should reserve function overloading for functions that perform closely related tasks. Sometimes, the default arguments may be used instead of overloading. This may reduce the number of functions to be defined.

Overloaded functions are extensively used for handling class objects. They will be illustrated later when the classes are discussed in the next chapter.

7.10 Friend and Virtual Functions

C++ introduces two new types of functions, namely, friend function and virtual function. They are basically introduced to handle some specific tasks related to class objects. Therefore, discussions on these functions have been reserved until after the class objects are discussed. The friend functions are discussed in Sec. 8.15 of the next chapter.

7.11 Math Library Functions

The standard C++ supports many math functions that can be used for performing certain commonly used calculations. Most frequently used math library functions are summarized in Table 11.1.

Table 7.1 *Commonly used math library functions*

Function	*Purposes*
ceil(x)	Rounds x to the smallest integer not less than x ceil(8.1) = 9.0 and ceil(-8.8) = –8.0
cos(x)	Trigonometric cosine of x (x in radians)
exp(x)	Exponential function e^x.
fabs(x)	Absolute value of x. If x>0 then abs(x) is x If x=0 then abs(x) is 0.0 If x<0 then abs(x) is –x
floor(x)	Rounds x to the largest integer not greater than x floor(8.2) = 8.0 and floor(–8.8 = –9.0
log(x)	Natural logarithm of x(base e)
log10(x)	Logarithm of x(base 10)
pow(x,y)	x raised to power y(x^y)
sin(x)	Trigonometric sine of x (x in radians)
sqrt(x)	Square root of x
tan(x)	Trigonometric tangent of x (x in radians)

note

The argument variables **x** and **y** are of type **double** and all the functions return the data type **double**.

To use the math library functions, we must include the header file **math.h** in conventional C++ and **cmath** in ANSI C++.

SUMMARY

- It is possible to reduce the size of program by calling and using functions at different places in the program.
- In C++ the main() returns a value of type **int** to the operating system. Since the return type of functions is **int** by default, the keyword **int** in the main() header is optional. Most C++ compilers issue a warning, if there is no return statement.
- Function prototyping gives the compiler the details about the functions such as the number and types of arguments and the type of return values.
- Reference variables in C++ permit us to pass parameters to the functions by reference. A function can also return a reference to a variable.
- When a function is declared inline the compiler replaces the function call with the respective function code. Normally, a small size function is made as **inline**.
- The compiler may ignore the inline declaration if the function declaration is too long or too complicated and hence compile the function as a normal function.
- C++ allows us to assign default values to the function parameters when the function is declared. In such a case we can call a function without specifying all its arguments. The defaults are always added from right to left.
- In C++, an argument to a function can be declared as **const**, indicating that the function should not modify the argument.
- C++ allows function overloading. That is, we can have more than one function with the same name in our program. The compiler matches the function call with the exact function code by checking the number and type of the arguments.
- C++ supports two new types of functions, namely **friend** functions and **virtual** functions.
- Many mathematical computations can be carried out using the library functions supported by the C++ standard library.

Key Terms

- actual arguments
- argument list
- bubble sort
- call by reference

- call by value
- called function
- calling program
- calling statement
- **cmath**
- **const** arguments
- declaration statement
- default arguments
- default values
- dummy variables
- ellipses
- empty argument list
- exit value
- formal arguments
- **friend** functions
- function call
- function definition
- function overloading
- function polymorphism
- function prototype
- indirection
- **inline**
- inline functions
- macros
- **main()**
- math library
- **math.h**
- overloading
- pointers
- polymorphism
- prototyping
- reference variable
- return by reference
- **return** statement
- return type
- **return()**
- template
- virtual functions

Review Questions

7.1 *State whether the following statements are TRUE or FALSE.*

(a) *A function argument is a value returned by the function to the calling program.*

(b) *When arguments are passed by value, the function works with the original arguments in the calling program.*

(c) *When a function returns a value, the entire function call can be assigned to a variable.*

(d) *A function can return a value by reference.*

(e) *When an argument is passed by reference, a temporary variable is created in the calling program to hold the argument value.*

(f) *It is not necessary to specify the variable name in the function prototype.*

7.2 *What are the advantages of function prototypes in C++?*

7.3 *Describe the different styles of writing prototypes.*

7.4 *Find errors, if any, in the following function prototypes.*

(a) *float average(x,y);*

(b) *int mul(int a,b);*
(c) *int display(...);*
(d) *void Vect(int? &V, int & size);*
(e) *void print(float data [], size = 20);*

7.5 *What is the main advantage of passing arguments by reference?*

7.6 *When will you make a function* ***inline****? Why?*

7.7 *How does an* ***inline*** *function differ from a preprocessor macro?*

7.8 *When do we need to use default arguments in a function?*

7.9 *What is the significance of an empty parenthesis in a function declaration?*

7.10 *What do you meant by overloading of a function? When do we use this concept?*

7.11 *Comment on the following function definitions:*

(a)
```
int *f( )
{
     int m = 1;
     ....
     ....
     return(&m);
}
```

(b)
```
double f( )
{
     ....
     ....
     return(1);
}
```

(c)
```
int & f()
{
     int n = 10;
     ....
     ....
     return(n);
}
```

Debugging Exercises

7.1 Identify the error in the following program.

```
#include <iostream.h>
int fun()
{
       return 1;
}
float fun()
```

```
{
       return 10.23;
void main()
{
            cout << (int)fun() << ' ';
            cout << (float)fun() << ' ';
}
```

7.2 Identify the error in the following program.

```
#include <iostream.h>
void display(const int const1=5)
{
     const int const2=5;
     int array1[const1];
     int array2[const2];
     for(int i=0; i<5; i++)
     {
          array1[i] = i;
          array2[i] = i*10;
          cout << array1[i] << ' ' << array2[i] << ' ' ;
     }
}

void main()
{
     display(5);
}
```

7.3 Identify the error in the following program.

```
#include <iostream.h>
int gValue=10;
void extra()
{
       cout << gValue << '  ';
}
void main()
{
       extra();
       {
              int gValue = 20;
              cout << gValue << ' ';
              cout << : gValue << ' ';
       }
}
```

7.4 Find errors, if any, in the following function definition for displaying a matrix: void display(int A[ ] [ ], int m, int n)

```
{
     for(i=0; i<m; i++)
       for(j=0; j<n; j++)
         cout << " " << A[i][j];
       cout << "\n";
}
```

Programming Exercises

7.1 *Write a function to read a matrix of size m* x *n from the keyboard.*

7.2 *Write a program to read a matrix of size m* x *n from the keyboard and display the same on the screen using functions.*

7.3 *Rewrite the program of Exercise 7.2 to make the row parameter of the matrix as a default argument.*

7.4 *The effect of a default argument can be alternatively achieved by overloading. Discuss with an example.*

7.5 *Write a macro that obtains the largest of three numbers.*

7.6 *Redo Exercise 7.5 using inline function. Test the function using a* ***main*** *program.*

7.7 *Write a function* ***power()*** *to raise a number* ***m*** *to a power* ***n****. The function takes a* ***double*** *value for* ***m*** *and* ***int*** *value for* ***n****, and returns the result correctly. Use a default value of 2 for n to make the function to calculate squares when this argument is omitted. Write a* ***main*** *that gets the values of* ***m*** *and* ***n*** *from the user to test the function.*

7.8 *Write a function that performs the same operation as that of Exercise 7.7 but takes an* ***int*** *value for* ***m****. Both the functions should have the same name. Write a* ***main*** *that calls both the functions. Use the concept of function overloading.*

8

Classes and Objects

Key Concepts

- Using structures
- Creating a class
- Defining member functions
- Creating objects
- Using objects
- Inline member functions
- Nested member functions
- Private member functions
- Arrays as class members
- Storage of objects
- Static data members
- Static member functions
- Using arrays of objects
- Passing objects as parameters
- Making functions friendly to classes
- Functions returning objects
- **const** member functions
- Pointers to members
- Using dereferencing operators
- Local classes

8.1 Introduction

The most important feature of C++ is the "class". Its significance is highlighted by the fact that Stroustrup initially gave the name "C with classes" to his new language. A class is an

extension of the idea of structure used in C. It is a new way of creating and implementing a user-defined data type. We shall discuss, in this chapter, the concept of class by first reviewing the traditional structures found in C and then the ways in which classes can be designed, implemented and applied.

8.2 Traditional C Structures

One of the unique features of the C language is structures. They provide a method for packing together data of different types. A structure is a convenient tool for handling a group of logically related data items. It is a user-defined data type with a *template* that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations. For example, consider the following declaration:

```
struct student
{
      char   name[20];
      int    roll_number;
      float total_marks;
};
```

The keyword **struct** declares **student** as a new data type that can hold three fields of different data types. These fields are known as *structure members* or *elements*. The identifier student, which is referred to as *structure name* or *structure tag*, can be used to create variables of type student. Example:

```
struct student A;  // C declaration
```

A is a variable of type student and has three member variables as defined by the template. Member variables can be accessed using the *dot* or *period operator* as follows:

```
strcpy(A.name, "John");
A.roll_number = 999;
A.total_marks = 595.5;
Final_total = A.total_marks + 5;
```

Structures can have arrays, pointers or structures as members.

8.2.1 Limitations of C Structure

The standard C does not allow the struct data type to be treated like built-in types. For example, consider the following structure:

```
struct complex
{
      float x;
      float y;
};
           struct complex c1, c2, c3;
```

The complex numbers c1, c2, and c3 can easily be assigned values using the dot operator, but we cannot add two complex numbers or subtract one from the other. For example,

```
c3 = c1 + c2;
```

is illegal in C.

Another important limitation of C structures is that they do not permit *data hiding*. Structure members can be directly accessed by the structure variables by any function anywhere in their scope. In other words, the structure members are public members.

8.2.2 Extensions to Structures

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main principles of OOP. *Inheritance*, a mechanism by which one type can inherit characteristics from other types, is also supported by C++.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as 'private' so that they cannot be accessed directly by the external functions.

In C++, the structure names are stand-alone and can be used like any other type names. In other words, the keyword struct can be omitted in the declaration of structure variables. For example, we can declare the student variable A as

```
student A;   // C++ declaration
```

Remember, this is an error in C.

C++ incorporates all these extensions in another user-defined type known as **class**. There is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since class is a specially introduced data type in C++, most of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions. Therefore, we will not discuss structures any further.

note

The only difference between a structure and a class in C++ is that, by default, the members of a class are *private*, while, by default, the members of a structure are *public*.

8.3 Specifying a Class

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new *abstract data type* that can be treated like any other built-in data type. Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is:

```
class class_name
{
        private:
          variable declarations;
          function declarations;
        public:
          variable declarations;
        function declaration;
};
```

The **class** declaration is similar to a **struct** declaration. The keyword **class** specifies, that what follows is an abstract data of type *class_name*. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called *class members*. They are usually grouped under two sections, namely, *private* and *public* to denote which of the members are *private* and which of them are *public*. The keywords **private** and **public** are known as visibility labels. Note that these keywords are followed by a colon.

The class members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The data hiding (using private declaration) is the key feature of object-oriented programming. The use of the keyword private is optional. By default, the members of a class are **private**. If both the labels are missing, then, by default, all the members are **private**. Such a class is completely hidden from the outside world and does not serve any purpose.

The variables declared inside the class are known as *data members* and the functions are known as *member functions*. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class. This is illustrated in Fig. 8.1. The binding of data and functions together into a single class-type variable is referred to as *encapsulation*.

8.3.1 A Simple Class Example

A typical class declaration would look like:

```
class item
{
     int number;                    // variables declaration
     float cost;                    // private by default
  public:
```

```
    void getdata(int a, float b);      // functions declaration
    void putdata(void);                // using prototype
};
  // ends with semicolon
```

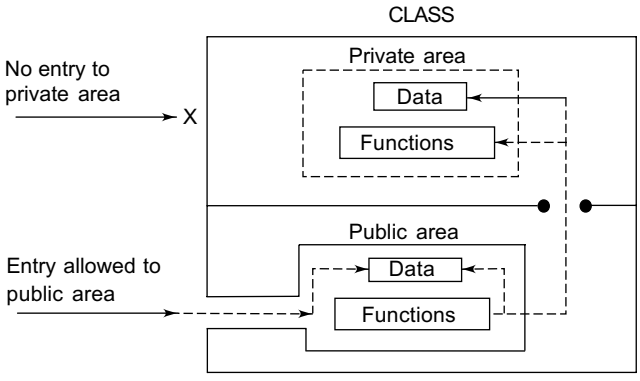


Fig. 8.1 ⇔ *Data hiding in classes*

We usually give a class some meaningful name, such as **item**. This name now becomes a new type identifier that can be used to declare *instances* of that class type. The class item contains two data members and two function members. The data members are private by default while both the functions are public by declaration. The function **getdata()** can be used to assign values to the member variables number and cost, and **putdata()** for displaying their values. These functions provide the only access to the data members from outside the class. This means that the data cannot be accessed by any function that is not a member of the class **item**. Note that the functions are declared, not defined. Actual function definitions will appear later in the program. The data members are usually declared as **private** and the member functions as **public**. Figure 8.2 shows two different notations used by the OOP analysts to represent a class.

8.3.2 Creating Objects

Remember that the declaration of **item** as shown above does not define any objects of **item** but only specifies *what* they will contain. Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable). For example,

```
item x;          // memory for x is created
```

creates a variable **x** of type **item**. In C++, the class variables are known as *objects*. Therefore, **x** is called an object of type **item**. We may also declare more than one object in one statement. Example:

```
item x, y, z;
```

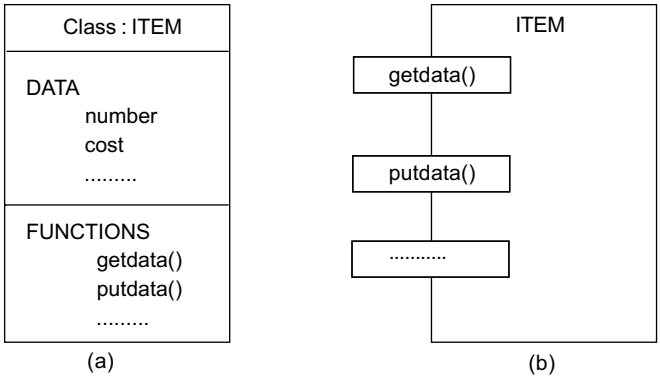


Fig. 8.2 ⇔ *Representation of a class*

The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that class specification, like a structure, provides only a *template* and does not create any memory space for the objects.

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. That is to say, the definition

```
class item
{
        .....
        .....
        .....
} x,y,z;
```

would create the objects **x, y** and **z** of type **item**. This practice is seldom followed because we would like to declare the objects close to the place where they are used and not at the time of class definition.

8.3.3 Accessing Class Members

As pointed out earlier, the private data of a class can be accessed only through the member functions of that class. The **main()** cannot contain statements that access **number** and **cost** directly. The following is the format for calling a member function:

```
object-name.function-name (actual-arguments);
```

For example, the function call statement

```
x.getdata(100,75.5);
```

is valid and assigns the value 100 to **number** and 75.5 to **cost** of the object **x** by implementing the **getdata()** function. The assignments occur in the actual function. Please refer Sec. 8.4 for further details.

Similarly, the statement

```
x.putdata();
```

would display the values of data members. Remember, a member function can be invoked only by using an object (of the same class). The statement like

```
getdata(100,75.5);
```

has no meaning. Similarly, the statement

```
x.number = 100;
```

is also illegal. Although **x** is an object of the type **item** to which **number** belongs, the number (declared private) can be accessed only through a member function and not by the object directly.

It may be recalled that objects communicate by sending and receiving messages. This is achieved through the member functions. For example,

```
x.putdata();
```

sends a message to the object **x** requesting it to display its contents.

A variable declared as public can be accessed by the objects directly. Example:

```
class xyz
{
     int x;
     int y;
  public:
     int z;
};
     .....
     .....
xyz p;
p.x = 0;       // error, x is private
p.z = 10       // OK, z is public
     .....
     .....
```

note

The use of data in this manner defeats the very idea of data hiding and therefore should be avoided.

8.4 Defining Member Functions

Member functions can be defined in two places:

- Outside the class definition.
- Inside the class definition.

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases.

However, there is a subtle difference in the way the function header is defined. Both these approaches are discussed in detail in this section.

8.4.1 Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body. Since C++ does not support the old version of function definition, the *ANSI prototype* form must be used for defining the function header.

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which **class** the function belongs to. The general form of a member function definition is:

```
return-type class-name :: function-name (argument declaration)
  {
        Function body
  }
```

The membership label class-name :: tells the compiler that the function *function-name* belongs to the class *class-name*. That is, the scope of the function is restricted to the *class-name* specified in the header line. The symbol :: is called the *scope resolution* operator.

For instance, consider the member functions **getdata()** and **putdata()** as discussed above. They may be coded as follows:

```
void item :: getdata(int a, float b)
{
     number = a;
     cost = b;
}
void item :: putdata(void)
{
     cout << "Number :" << number << "\n";
     cout << "Cost   :" << cost   << "\n";
}
```

Since these functions do not return any value, their return-type is void. Function arguments are declared using the ANSI prototype.

The member functions have some special characteristics that are often used in the program development. These characteristics are :

- Several different classes can use the same function name. The 'membership label' will resolve their scope.
- Member functions can access the private data of the class. A non-member function cannot do so. (However, an exception to this rule is a *friend* function discussed later.)
- A member function can call another member function directly, without using the dot operator.

8.4.2 Inside the Class Definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class. For example, we could define the item class as follows:

```
class item
{
    int number;
    float cost;
  public:
    void getdata(int a, float b);      // declaration
         // inline function
    void putdata(void)                 // definition inside the class
    {
         cout << number  <<  "\n";
         cout << cost    <<  "\n";
    }
};
```

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restrictions and limitations that apply to an **inline** function are also applicable here. Normally, only small functions are defined inside the class definition.

8.5 A C++ Program with Class

All the details discussed so far are implemented in Program 8.1.

CLASS IMPLEMENTATION

```
#include <iostream>
using namespace std;
class item
{
       int number;    // private by default
       float cost;    // private by default
   public:
       void getdata(int a, float b);      // prototype declaration,
                                          // to be defined
       // Function defined inside class
       void putdata(void)
       {
            cout << "number :" << number << "\n";
            cout << "cost   :" << cost   << "\n";
       }
};
```

```
//............ Member Function Definition ..............
void item :: getdata(int a, float b) // use membership label
{
       number = a;    // private variables
       cost = b;      // directly used
}
//..................... Main Program .....................

int main()
{
       item x;   // create object x

       cout <<  "\nobject x "  <<  "\n";

       x.getdata(100, 299.95);              // call member function
       x.putdata();                         // call member function

       item y;                              // create another object

       cout <<  "\nobject y"  << "\n";

       y.getdata(200, 175.50);
       y.putdata();

       return 0;
}
```

PROGRAM 8.1

This program features the class **item**. This class contains two private variables and two public functions. The member function **getdata()** which has been defined outside the class supplies values to both the variables. Note the use of statements such as

```
number = a;
```

in the function definition of **getdata()**. This shows that the member functions can have direct access to private data items.

The member function **putdata()** has been defined inside the class and therefore behaves like an **inline** function. This function displays the values of the private variables **number** and **cost**.

The program creates two objects, x and y in two different statements. This can be combined in one statement.

```
item x, y;  // creates a list of objects
```

Here is the output of Program 8.1:

```
object x
```

```
number :100
cost   :299.95
object y
number :200
cost   :175.5
```

For the sake of illustration we have shown one member function as **inline** and the other as an 'external' member function. Both can be defined as **inline** or external functions.

8.6 Making an Outside Function Inline

One of the objectives of OOP is to separate the details of implementation from the class definition. It is therefore good practice to define the member functions outside the class.

We can define a member function outside the class definition and still make it inline by just using the qualifier **inline** in the header line of function definition. Example:

```
class item
{
       .....
       .....
  public:
       void getdata(int a, float b);       // declaration
};
inline void item :: getdata(int a, float b) // definition
{
       number = a;
       cost = b;
}
```

8.7 Nesting of Member Functions

We just discussed that a member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as *nesting of member functions*. Program 8.2 illustrates this feature.

NESTING OF MEMBER FUNCTIONS

```
#include <iostream>
using namespace std;
class set
{
          int m, n;
  public:
```

```
        void input(void);
        void display(void);
        int largest(void);
};

int set :: largest(void)
{
        if(m >= n)
                return(m);
        else
                return(n);
}

void set :: input(void)
{
        cout << "Input values of m and n" << "\n";
        cin >> m >> n;
}

void set :: display(void)
{
        cout << "Largest value = "
             << largest() << "\n";      // calling member function
}

int main()
{
        set A;
        A.input();
        A.display();
        return 0;
  }
```

PROGRAM 8.2

The output of Program 8.2 would be:

```
Input values of m and n
25 18
Largest value = 25
```

8.8 Private Member Functions

Although it is normal practice to place all the data items in a private section and all the functions in public, some situations may require certain functions to be hidden (like private data) from the outside calls. Tasks such as deleting an account in a customer file, or providing increment to an

employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator. Consider a class as defined below:

```
class sample
{
     int m;
     void read(void);     // private member function
  public:
     void update(void);
     void write(void);
};
```

If **s1** is an object of **sample**, then

```
s1.read();     // won't work; objects cannot access
               // private members
```

is illegal. However, the function **read()** can be called by the function **update()** to update the value of **m**.

```
void sample :: update(void)
{
     read();     // simple call; no object used
}
```

8.9 Arrays within a Class

The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size=10;    // provides value for array size
class array
{
     int a[size];     // 'a' is int type array
  public:
     void setval(void);
     void display(void);
};
```

The array variable **a[ ]** declared as a private member of the class **array** can be used in the member functions, like any other array variable. We can perform any operations on it. For instance, in the above class definition, the member function **setval()** sets the values of elements of the array **a[ ]**, and **display()** function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

Let us consider a shopping list of items for which we place an order with a dealer every month. The list includes details such as the code number and price of each item. We would

like to perform operations such as adding an item to the list, deleting an item from the list and printing the total value of the order. Program 8.3 shows how these operations are implemented using a class with arrays as data members.

PROCESSING SHOPPING LIST

```
#include <iostream>

using namespace std;

const m=50;

class ITEMS

{
       int itemCode[m];
       float itemPrice[m];
       int count;
  public:
       void CNT(void){count = 0;}        // initializes count to 0
       void getitem(void);
       void displaySum(void);
       void remove(void);
       void displayItems(void);
};
//=========================================================
void ITEMS :: getitem(void)           // assign values to data
                                      // members of item
{
       cout << "Enter item code :";
       cin >> itemCode[count];

       cout << "Enter item cost :";
       cin >> itemPrice[count];
       count++;
}
void ITEMS :: displaySum(void)        // display total value of
                                      // all items
{
       float sum = 0;
for(int i=0; i<count; i++)
              sum = sum + itemPrice[i];

       cout << "\nTotal value :" << sum << "\n";
}
```

```
void ITEMS :: remove(void)          // delete a specified item
{
       int a;
       cout << "Enter item code :";
       cin >> a;

       for(int i=0; i<count; i++)
              if(itemCode[i] == a)
                     itemPrice[i] = 0;
}

void ITEMS :: displayItems(void)    // displaying items
{
       cout << "\nCode  Price\n";

       for(int i=0; i<count; i++)
       {
              cout <<"\n" << itemCode[i];
              cout <<"    " << itemPrice[i];
       }
       cout << "\n";
}
//=======================================================

int main()
{
       ITEMS order;
       order.CNT();
       int x;
       do            // do....while loop
       {
              cout << "\nYou can do the following;"
                   << "Enter appropriate number \n";
              cout << "\n1 : Add an item ";
              cout << "\n2 : Display total value";
              cout << "\n3 : Delete an item";
              cout << "\n4 : Display all items";
              cout << "\n5 : Quit";
              cout << "\n\nWhat is your option?";

              cin >> x;

              switch(x)
```

```
                {
                        case 1 : order.getitem(); break;
                        case 2 : order.displaySum(); break;
                        case 3 : order.remove(); break;
                        case 4 : order.displayItems(); break;
                        case 5 : break;
                        default : cout << "Error in input; try again\n";
                }

        }  while(x != 5);                // do...while ends

        return 0;
}
```

PROGRAM 8.3

The output of Program 8.3 would be:

```
You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?1
Enter item code :111
Enter item cost :100

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?1
Enter item code :222
Enter item cost :200

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit
```

```
What is your option?1
Enter item code :333
Enter item cost :300

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?2
Total value :600
You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?3
Enter item code :222
You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?4
Code      Price
111           100
222           0
333           300

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?5
```

note

The program uses two arrays, namely **itemCode**[ ] to hold the code number of items and **itemPrice**[ ] to hold the prices. A third data member **count** is used to keep a record of items in the list. The program uses a total of four functions to implement the operations to be performed on the list. The statement

```
const int m = 50;
```

defines the size of the array members.

The first function **CNT**( ) simply sets the variable **count** to zero. The second function **getitem**() gets the item code and the item price interactively and assigns them to the array members **itemCode[count]** and **itemPrice[count]**. Note that inside this function **count** is incremented after the assignment operation is over. The function **displaySum**( ) first evaluates the total value of the order and then prints the value. The fourth function **remove**() deletes a given item from the list. It uses the item code to locate it in the list and sets the price to zero indicating that the item is not 'active' in the list. Lastly, the function **displayItems**( ) displays all the items in the list.

The program implements all the tasks using a menu-based user interface.

8.10 Memory Allocation for Objects

We have stated that the memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects. This is shown in Fig. 8.3.

8.11 Static Data Members

A data member of a class can be qualified as static. The properties of a **static** member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are :

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects. Program 8.4 illustrates the use of a static data member.

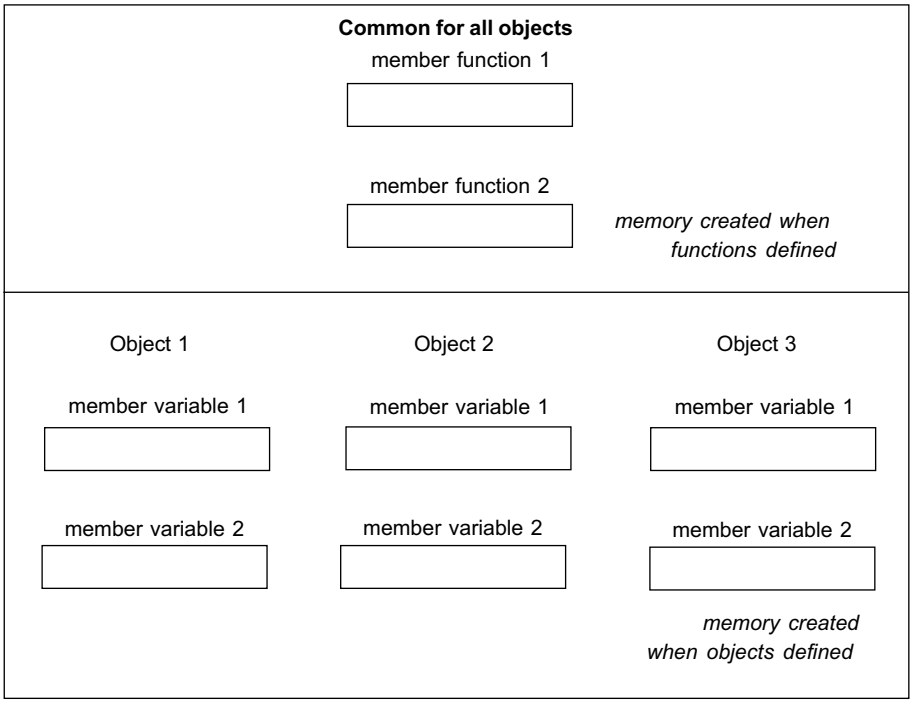


Fig. 8.3 ⇔ *Object of memory*

STATIC CLASS MEMBER

```
#include <iostream>

using namespace std;

class item
{
       static int count;
       int number;
  public:
       void getdata(int a)
       {
              number = a;
              count ++;
       }
       void getcount(void)
       {
              cout << "count: ";
              cout << count  << "\n";
```

```
        }
};

int item :: count;

int main()
{
     item a, b, c;          // count is initialized to zero
     a.getcount();          // display count
     b.getcount();
     c.getcount();

     a.getdata(100);        // getting data into object a
     b.getdata(200);        // getting data into object b
     c.getdata(300);        // getting data into object c

     cout << "After reading data" << "\n";

     a.getcount();          // display count
     b.getcount();
     c.getcount();
     return 0;
}
```

PROGRAM 8.4

The output of the Program 8.4 would be:

```
count: 0
count: 0
count: 0
After reading data
count: 3
count: 3
count: 3
```

note

Notice the following statement in the program:

```
int item :: count;      // definition of static data member
```

Note that the type and scope of each **static** member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are also known as *class variables*.

The **static** variable **count** is initialized to zero when the objects are created. The count is incremented whenever the data is read into an object. Since the data is read into objects three times, the variable count is incremented three times. Because there is only one copy of count shared by all the three objects, all the three output statements cause the value 3 to be displayed. Figure 8.4 shows how a static variable is used by the objects.

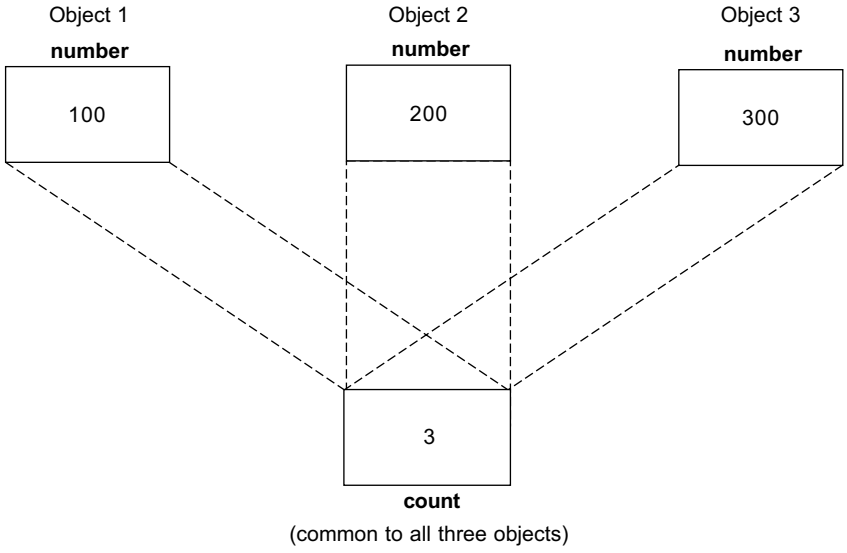


Fig. 8.4 ⇔ *Sharing of a static data member*

Static variables are like non-inline member functions as they are declared in a class declaration and defined in the source file. While defining a static variable, some initial value can also be assigned to the variable. For instance, the following definition gives count the initial value 10.

```
int item :: count = 10;
```

8.12 Static Member Functions

Like **static** member variable, we can also have **static** member functions. A member function that is declared **static** has the following properties:

- A **static** function can have access to only other static members (functions or variables) declared in the same class.
- A **static** member function can be called using the class name (instead of its objects) as follows:

```
class-name :: function-name;
```

Program 8.5 illustrates the implementation of these characteristics. The **static** function **showcount()** displays the number of objects created till that moment. A count of number of objects created is maintained by the **static** variable count.

The function **showcode()** displays the code number of each object.

STATIC MEMBER FUNCTION

```
#include <iostream>
using namespace std;
class test
{
     int code;
     static int count;              // static member variable
public:
     void setcode(void)
     {
     code = ++count;
     }
     void showcode(void)
     {
     cout << "object number: " << code << "\n";
     }
     static void showcount(void)    // static member function
     {
     cout << "count: " << count << "\n";
     }
};
int test :: count;
int main()
{
     test t1, t2;

     t1.setcode();
     t2.setcode();

     test :: showcount();           // accessing static function

     test t3;
     t3.setcode();

     test :: showcount();

     t1.showcode();
     t2.showcode();
     t3.showcode();

     return 0;
}
```

PROGRAM 8.5

Output of Program 8.5:

```
count: 2
count: 3
object number: 1
object number: 2
object number: 3
```

note

Note that the statement

```
code = ++count;
```

is executed whenever **setcode( )** function is invoked and the current value of **count** is assigned to **code**. Since each object has its own copy of **code**, the value contained in **code** represents a unique number of its object.

Remember, the following function definition will not work:

```
static void showcount()
{
     cout << code;    // code is not static
}
```

8.13 Arrays of Objects

We know that an array can be of any data type including **struct**. Similarly, we can also have arrays of variables that are of the type **class**. Such variables are called *arrays of objects*. Consider the following class definition:

```
class employee
{
          char name[30];
          float age;
    public:
          void getdata(void);
          void putdata(void);
};
```

The identifier **employee** is a user-defined data type and can be used to create objects that relate to different categories of the employees. Example:

```
employee manager[3];       // array of manager
employee foreman[15];      // array of foreman
employee worker[75];       // array of worker
```

The array **manager** contains three objects(managers), namely, **manager[0], manager[1]** and **manager[2]**, of type **employee** class. Similarly, the **foreman** array contains 15 objects (foremen) and the **worker** array contains 75 objects(workers).

Since an array of objects behaves like any other array, we can use the usual array-accessing methods to access individual elements, and then the dot member operator to access the member functions. For example, the statement

```
manager[i].putdata();
```

will display the data of the ith element of the array **manager**. That is, this statement requests the object **manager[i]** to invoke the member function **putdata()**.

An array of objects is stored inside the memory in the same way as a multi-dimensional array. The array manager is represented in Fig. 8.5. Note that only the space for data items of the objects is created. Member functions are stored separately and will be used by all the objects.

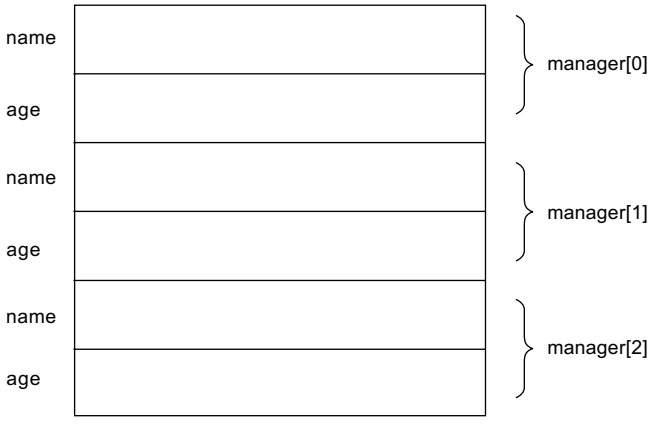


Fig. 8.5 ⇔ *Storage of data items of an object array*

Program 8.6 illustrates the use of object arrays.

ARRAYS OF OBJECTS

```
#include <iostream>

using namespace std;

class employee
{
       char  name[30];    // string as class member
       float age;
  public:
       void getdata(void);
       void putdata(void);
};
```

```
void employee :: getdata(void)
{
       cout << "Enter name: ";
       cin  >> name;
       cout << "Enter age:  ";
       cin  >> age;
}
void employee :: putdata(void)
{
       cout << "Name: " << name << "\n";
       cout << "Age:  " << age  << "\n";
}
const int size=3;
int main()
{
       employee manager[size];
       for(int i=0; i<size; i++)
       {
              cout << "\nDetails of manager" << i+1 << "\n";
              manager[i].getdata();
       }
       cout <<  "\n";
       for(i=0; i<size; i++)
       {
              cout << "\nManager" << i+1 << "\n";
              manager[i].putdata();
       }
       return 0;
}
```

PROGRAM 8.6

This being an interactive program, the input data and the program output are shown below:

Interactive input

```
Details of manager1
Enter name: xxx
Enter age:  45

Details of manager2
Enter name: yyy
Enter age:  37

Details of manager3
Enter name: zzz
Enter age:  50
```

```
Program output
     Manager1
     Name: xxx
     Age:  45

     Manager2
     Name: yyy
     Age:  37

     Manager3
     Name: zzz
     Age:  50
```

8.14 Objects as Function Arguments

Like any other data type, an object may be used as a function argument. This can be done in two ways:

- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the function.

The first method is called *pass-by-value*. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The second method is called *pass-by-reference*. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Program 8.7 illustrates the use of objects as function arguments. It performs the addition of time in the hour and minutes format.

OBJECTS AS ARGUMENTS

```
#include <iostream>

using namespace std;

class time
{
       int  hours;
       int  minutes;
  public:
       void gettime(int h, int m)
       {  hours = h; minutes = m; }
```

```
        void puttime(void)
        {
               cout << hours << " hours and ";
               cout << minutes << " minutes " << "\n";
        }
        void sum(time, time);  // declaration with objects as arguments
};
void time :: sum(time t1, time t2)    // t1, t2 are objects
{
        minutes = t1.minutes + t2.minutes;
        hours = minutes/60;
        minutes = minutes%60;
        hours = hours + t1.hours + t2.hours;
}
int main()
{
        time T1, T2, T3;

        T1.gettime(2,45);     // get T1
        T2.gettime(3,30);     // get T2

        T3.sum(T1,T2);        // T3=T1+T2

        cout << "T1 = "; T1.puttime();       // display T1
        cout << "T2 = "; T2.puttime();       // display T2
        cout << "T3 = "; T3.puttime(); // display T3

        return 0;
}
```

PROGRAM 8.7

The output of Program 8.7 would be:

```
T1 = 2 hours and 45 minutes
T2 = 3 hours and 30 minutes
T3 = 6 hours and 15 minutes
```

note

Since the member function **sum( )** is invoked by the object **T3**, with the objects **T1** and **T2** as arguments, it can directly access the hours and minutes variables of **T3**. But, the members of **T1** and **T2** can be accessed only by using the dot operator (like **T1.hours** and **T1.minutes**). Therefore, inside the function sum(), the variables **hours** and **minutes** refer to **T3**, **T1.hours** and **T1.minutes** refer to **T1**, and **T2.hours** and **T2.minutes** refer to **T2**.

Figure 8.6 illustrates how the members are accessed inside the function **sum()**.

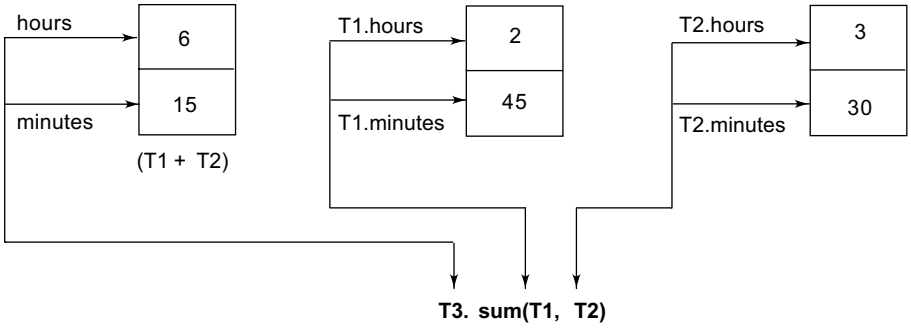


Fig. 8.6 ⇔ *Accessing members of objects with in a called function*

An object can also be passed as an argument to a non-member function. However, such functions can have access to the **public member** functions only through the objects passed as arguments to it. These functions cannot have access to the private data members.

8.15 Friendly Functions

We have been emphasizing throughout this chapter that the private members cannot be accessed from outside the class. That is, a non-member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. For example, consider a case where two classes, **manager** and **scientist**, have been defined. We would like to use a function **income_tax()** to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

To make an outside function "friendly" to a class, we have to simply declare this function as a **friend** of the class as shown below:

```
class ABC
{
        .....
        .....
  public:
        .....
        .....
        friend void xyz(void);  // declaration
};
```

The function declaration should be preceded by the keyword **friend**. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use

either the keyword **friend** or the scope operator ::. The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a **friend** in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as **friend**.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.(e.g. A.x).
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

The friend functions are often used in operator overloading which will be discussed later.

Program 8.8 illustrates the use of a friend function.

FRIEND FUNCTION

```
#include <iostream>

using namespace std;

class sample
{
   int a;
   int b;
  public:
   void setvalue() {a=25; b=40; }
   friend float mean(sample s);
};
float mean(sample s)
{
   return float(s.a + s.b)/2.0;
}

int main()
{
   sample X;      // object X
   X.setvalue();
   cout << "Mean value = " << mean(X) << "\n";

   return 0;
}
```

PROGRAM 8.8

The output of Program 8.8 would be:

```
Mean value = 32.5
```

note

The friend function accesses the class variables **a** and **b** by using the dot operator and the object passed to it. The function call **mean(X)** passes the object **X** by value to the friend function.

Member functions of one class can be **friend** functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

```
class X
{
     .....
     .....
     int fun1();        // member function of X
     .....
};

class Y
{
      .....
      .....
      friend int X :: fun1();    // fun1() of X
                                 // is friend of Y
      .....
};
```

The function **fun1()** is a member of **class X** and a **friend** of class **Y**.

We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class**. This can be specified as follows:

```
class Z
{
     .....
     friend class X;    // all member functions of X are
                        // friends to Z
};
```

Program 8.9 demonstrates how friend functions work as a bridge between the classes.

A FUNCTION FRIENDLY TO TWO CLASSES

```
#include <iostream>
```

```
using namespace std;

class ABC;    // Forward declaration
//-------------------------------------------------------//
class XYZ
{
       int x;
  public:
       void setvalue(int i) {x = i;}
       friend void max(XYZ, ABC);
};
//-------------------------------------------------------//
class ABC
{
       int a;
   public:
       void setvalue(int i) {a = i;}
       friend void max(XYZ, ABC);
};
//-------------------------------------------------------//
void max(XYZ m, ABC n)      // Definition of friend
{
       if(m.x >= n.a)
              cout << m.x;
       else
              cout << n.a;
}
//-------------------------------------------------------//
int main()
{
       ABC abc;
       abc.setvalue(10);
       XYZ xyz;
       xyz.setvalue(20);
       max(xyz, abc);

       return 0;
}
```

PROGRAM 8.9

The output of Program 8.9 would be:

```
20
```

note

The function max() has arguments from both **XYZ** and **ABC**. When the function max() is declared as a friend in **XYZ** for the first time, the compiler will not acknowledge the presence of ABC unless its name is declared in the beginning as

```
class ABC;
```

This is known as 'forward' declaration.

As pointed out earlier, a friend function can be called by reference. In this case, local copies of the objects are not made. Instead, a pointer to the address of the object is passed and the called function directly works on the actual object used in the call.

This method can be used to alter the values of the private members of a class. Remember, altering the values of private members is against the basic principles of data hiding. It should be used only when absolutely necessary.

Program 8.10 shows how to use a common friend function to exchange the private values of two classes. The function is called by reference.

SWAPPING PRIVATE DATA OF CLASSES

```
#include <iostream>

using namespace std;

class class_2;

class class_1
{
       int value1;
  public:
       void indata(int a) {value1 = a;}
       void display(void) {cout << value1 << "\n";}
       friend void exchange(class_1 &, class_2 &);
};

class class_2
{
       int value2;
  public:
       void indata(int a) {value2 = a;}
       void display(void) {cout << value2 << "\n";}
       friend void exchange(class_1 &, class_2 &);
};
```

```
void exchange(class_1 & x, class_2 & y)
{
        int temp = x.value1;
        x.value1 = y.value2;
        y.value2 = temp;
}

int main()
{
        class_1 C1;
        class_2 C2;

        C1.indata(100);
        C2.indata(200);

        cout << "Values before exchange" << "\n";
        C1.display();
        C2.display();

        exchange(C1, C2);        // swapping

        cout << "Values after exchange " << "\n";
        C1.display();
        C2.display();

        return 0;
}
```

PROGRAM 8.10

The objects x and y are aliases of **C1** and **C2** respectively. The statements

```
int temp = x.value1
x.value1 = y.value2;
y.value2 = temp;
```

directly modify the values of **value1** and **value2** declared in **class_1** and **class_2**.

Here is the output of Program 8.10:

```
Values before exchange
100
200
Values after exchange
200
100
```

8.16 Returning Objects

A function cannot only receive objects as arguments but also can return them. The example in Program 8.11 illustrates how an object can be created (within a function) and returned to another function

RETURNING OBJECTS

```
#include <iostream>

using namespace std;

class complex         // x + iy form
{
       float x;               // real part
       float y;               // imaginary part
  public:
       void input(float real, float imag)
       { x = real; y = imag; }
       friend complex sum(complex, complex);

       void show(complex);
};

complex sum(complex c1, complex c2)
{
       complex c3;      // objects c3 is created
       c3.x = c1.x + c2.x;
       c3.y = c1.y + c2.y;
       return(c3);      // returns object c3
}

void complex :: show(complex c)
{
       cout << c.x << " + j" << c.y << "\n";
}
int main()
{
       complex A, B, C;
       A.input(3.1, 5.65);
       B.input(2.75, 1.2);


       C = sum(A, B);       // C = A + B
```

```
        cout << "A = "; A.show(A);
        cout << "B = "; B.show(B);
        cout << "C = "; C.show(C);

        return 0;
}
```

PROGRAM 8.11

Upon execution, Program 8.11 would generate the following output:

```
A = 3.1 + j5.65
B = 2.75 + j1.2
C = 5.85 + j6.85
```

The program adds two complex numbers **A** and **B** to produce a third complex number **C** and displays all the three numbers.

8.17 const Member Functions

If a member function does not alter any data in the class, then we may declare it as a **const** member function as follows:

```
void mul(int, int) const;
double get_balance() const;
```

The qualifier **const** is appended to the function prototypes (in both declaration and definition). The compiler will generate an error message if such functions try to alter the data values.

8.18 Pointers to Members

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a "fully qualified" class member name. A class member pointer can be declared using the operator ::* with the class name. For example, given the class

```
class A
{
  private:
     int m;
  public:
     void show();
};
```

We can define a pointer to the member m as follows:

```
int A::* ip = &A :: m;
```

The **ip** pointer created thus acts like a class member in that it must be invoked with a class object. In the statement above, the phrase **A::*** means "pointer-to-member of **A** class". The phrase **&A::m** means the "address of the m member of A class".

Remember, the following statement is not valid:

```
int *ip = &m; // won't work
```

This is because **m** is not simply an **int** type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer **ip** can now be used to access the member **m** inside member functions (or friend functions). Let us assume that **a** is an object of **A** declared in a member function. We can access **m** using the pointer **ip** as follows:

```
cout << a.*ip;   // display
cout << a.m;    // same as above
```

Now, look at the following code:

```
ap = &a;              // ap is pointer to object a
cout << ap -> *ip;    // display m
cout << ap -> m; // same as above
```

The *dereferencing operator* ->* is used to access a member when we use pointers to both the object and the member. The *dereferencing* operator.* is used when the object itself is used with the member pointer. Note that ***ip** is used like a member name.

We can also design pointers to member functions which, then, can be invoked using the dereferencing operators in the **main** as shown below :

```
(object-name .* pointer-to-member function) (10);
(pointer-to-object ->* pointer-to-member function) (10)
```

The precedence of () is higher than that of .* and ->*, so the parentheses are necessary.

Program 8.12 illustrates the use of dereferencing operators to access the class members.

DEREFERENCING OPERATORS

```
#include <iostream>

using namespace std;

class M
{
          int x;
          int y;
  public:
          void set_xy(int a, int b)
```

```
        {
                x = a;
                y = b;
        }
        friend int sum(M m);
};
int sum(M m)
{
        int M ::* px = &M :: x;
        int M ::* py = &M :: y;
        M *pm = &m;
        int S = m.*px + pm->*py;
        return S;
}

int main()
{
        M n;
        void (M :: *pf)(int,int) = &M :: set_xy;
        (n.*pf)(10,20);
        cout << "SUM = " << sum(n) << "\n";

        M *op = &n;
        (op->*pf)(30,40);
        cout << "SUM = " << sum(n) << "\n";

        return 0;
}
```

PROGRAM 8.12

The output of Program 8.12 would be:

```
sum = 30
sum = 70
```

8.19 Local Classes

Classes can be defined and used inside a function or a block. Such classes are called local classes. Examples:

```
void test(int a)        // function
{
        .....
        .....
        class student       // local class
        {
```

```
            .....
            .....       // class definition
            .....
        };
        .....
        .....
        student s1(a);      // create student object
        .....               // use student object
    }
```

Local classes can use global variables (declared above the function) and static variables declared inside the function but cannot use automatic local variables. The global variables should be used with the scope operator (::).

There are some restrictions in constructing local classes. They cannot have static data members and member functions must be defined inside the local classes. Enclosing function cannot access the private members of a local class. However, we can achieve this by declaring the enclosing function as a friend.

SUMMARY

- ⇔ A class is an extension to the structure data type. A class can have both variables and functions as members.
- ⇔ By default, members of the class are private whereas that of structure are public.
- ⇔ Only the member functions can have access to the private data members and private functions. However the public members can be accessed from outside the class.
- ⇔ In C++, the class variables are called objects. With objects we can access the public members of a class using a dot operator.
- ⇔ We can define the member functions inside or outside the class. The difference between a member function and a normal function is that a member function uses a membership 'identity' label in the header to indicate the class to which it belongs.
- ⇔ The memory space for the objects is allocated when they are declared. Space for member variables is allocated separately for each object, but no separate space is allocated for member functions.
- ⇔ A data member of a class can be declared as a **static** and is normally used to maintain values common to the entire class.
- ⇔ The static member variables must be defined outside the class.
- ⇔ A static member function can have access to the static members declared in the same class and can be called using the class name.
- ⇔ C++ allows us to have arrays of objects.
- ⇔ We may use objects as function arguments.

- A function declared as a **friend** is not in the scope of the class to which it has been declared as friend. It has full access to the private members of the class.
- A function can also return an object.
- If a member function does not alter any data in the class, then we may declare it as a **const** member function. The keyword **const** is appended to the function prototype.
- It is also possible to define and use a class inside a function. Such a class is called a local class.

Key Terms

- abstract data type
- arrays of objects
- **class**
- class declaration
- class members
- class variables
- **const** member functions
- data hiding
- data members
- dereferencing operator
- dot operator
- elements
- encapsulation
- **friend** functions
- inheritance
- **inline** functions
- local class
- member functions
- nesting of member functions
- objects
- pass-by-reference
- pass-by-value
- period operator
- **private**
- prototype
- **public**
- scope operator
- scope resolution
- static data members
- static member functions
- static variables
- **struct**
- structure
- structure members
- structure name
- structure tag
- template

Review Questions

8.1 *How do structures in C and C++ differ?*

8.2 *What is a class? How does it accomplish data hiding?*

8.3 *How does a C++ structure differ from a C++ class?*

8.4 *What are objects? How are they created?*

8.5 *How is a member function of a class defined?*

8.6 *Can we use the same function name for a member function of a class and an outside function in the same program file? If yes, how are they distinguished? If no, give reasons.*

8.7 *Describe the mechanism of accessing data members and member functions in the following cases:*

(a) *Inside the **main** program.*

(b) *Inside a member function of the same class.*

(c) *Inside a member function of another class.*

8.8 *When do we declare a member of a class **static**?*

8.9 *What is a friend function? What are the merits and demerits of using friend functions?*

8.10 *State whether the following statements are TRUE or FALSE.*

(a) *Data items in a class must always be private.*

(b) *A function designed as private is accessible only to member functions of that class.*

(c) *A function designed as public can be accessed like any other ordinary functions.*

(d) *Member functions defined inside a class specifier become inline functions by default.*

(e) *Classes can bring together all aspects of an entity in one place.*

(f) *Class members are public by default.*

(g) *Friend functions have access to only public members of a class.*

(h) *An entire class can be made a friend of another class.*

(i) *Functions cannot return class objects.*

(j) *Data members can be initialized inside class specifier.*

Debugging Exercises

8.1 Identify the error in the following program.

```
#include <iostream.h>
struct Room
{
        int width;
        int length;

        void setValue(int w, int l)
        {
                width = w;
```

```
                length = l;
        }
};
void main()
{
        Room objRoom;
        objRoom.setValue(12, 1,4);
}
```

8.2 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
        int width, height;
        void setValue(int w, int h)
        {
             width = w;
             height = h;
        }
};
void main()
{
        Room objRoom;
        objRoom.width = 12;
}
```

8.3 Identify the error in the following program.

```
#include <iostream.h>
class Item
{
private:
        static int count;
public:
        Item()
        {
                count++;
        }
        int getCount()
        {
                return count;
        }
        int* getCountAddress()
        {
                return count;
                }
```

```
};
int Item::count = 0;

void main()
{
        Item objItem1;
        Item objItem2;

        cout << objItem1.getCount() << ' ';
        cout << objItem2.getCount() << ' ';

        cout << objItem1.getCountAddress() << ' ';
        cout << objItem2.getCountAddress() << ' ';
}
```

8.4 Identify the error in the following program.

```
#include <iostream.h>
class staticFunction
{
        static int count;
public:
        static void setCount()
        {
             count++;
        }
        void displayCount()
        {
             cout << count;
        }
};
int staticFunction::count = 10;
void main()
{
        staticFunction obj1;
        obj1.setCount(5);
        staticFunction::setCount();
        obj1.displayCount();
}
```

8.5 Identify the error in the following program.

```
#include <iostream.h>
class Length
{
        int feet;
        float inches;
public:
```

```
        Length()
        {
             feet = 5;
             inches = 6.0;
        }
        Length(int f, float in)
        {
             feet = f;
             inches=in;
        }
        Length addLength(Length l)
        {
             l.inches += this->inches;
             l.feet += this->feet;
             if(l.inches>12)
             {
                    l.inches-=12;
                    l.feet++;
             }
             return l;
        }
        int getFeet()
        {
             return feet;
        }
        float getInches()
        {
             return inches;
        }
};
void main()
{
        Length objLength1;
        Length objLength1(5, 6.5);
        objLength1 = objLength1.addLength(objLength2);
        cout << objLength1.getFeet() << ' ';
        cout << objLength1.getInches() << ' ';
}
```

8.6 Identify the error in the following program.

```
#include <iostream.h>
class Room;
void Area()
{
        int width, height;
        class Room
```

```
        {
             int width, height;
             public:
             void setValue(int w, int h)
             {
                  width = w;
                  height = h;
             }
             void displayValues()
             {
                  cout << (float)width << ' ' <<  (float)height;
             }
        };
        Room objRoom1;
        objRoom1.setValue(12, 8);
        objRoom1.displayValues();
}

void main()
{
        Area();
        Room objRoom2;
}
```

Programming Exercises

8.1 *Define a class to represent a bank account. Include the following members:*

Data members

1. *Name of the depositor*
2. *Account number*
3. *Type of account*
4. *Balance amount in the account*

Member functions

1. *To assign initial values*
2. *To deposit an amount*
3. *To withdraw an amount after checking the balance*
4. *To display name and balance*

Write a main program to test the program.

8.2 *Write a class to represent a vector (a series of float values). Include member functions to perform the following tasks:*

(a) *To create the 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, ...)*

Write a program to test your class.

8.3 *Modify the class and the program of Exercise 8.1 for handling 10 customers.*

8.4 *Modify the class and program of Exercise 8.2 such that the program would be able to add two vectors and display the resultant vector. (Note that we can pass objects as function arguments.)*

8.5 *Create two classes **DM** and **DB** which store the value of distances. **DM** stores distances in metres and centimetres and **DB** in feet and inches. Write a program that can read values for the class objects and add one object of **DM** with another object of **DB**.*

*Use a friend function to carry out the addition operation. The object that stores the results may be a **DM** object or **DB** object, depending on the units in which the results are required.*

The display should be in the format of feet and inches or metres and centimetres depending on the object on display.

9

Constructors and Destructors

Key Concepts

- Constructing objects
- Constructors
- Constructor overloading
- Default argument constructor
- Copy constructor
- Constructing matrix objects
- Automatic initialization
- Parameterized constructors
- Default constructor
- Dynamic initialization
- Dynamic constructor
- Destructors

9.1 Introduction

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as **putdata()** and **setvalue()** to provide initial values to the private member variables. For example, the following statement

```
A.input();
```

invokes the member function **input()**, which assigns the initial values to the data items of object **A**. Similarly, the statement

```
x.getdata(100,299.95);
```

passes the initial values as arguments to the function **getdata()**, where these values are assigned to the private variables of object **x**. All these 'function call' statements are used with the appropriate objects that have already been created. These functions cannot be used to initialize the member variables at the time of creation of their objects.

Providing the initial values as described above does not conform with the philosophy of C++ language. We stated earlier that one of the aims of C++ is to create user-defined data types such as **class**, that behave very similar to the built-in types. This means that we should

be able to initialize a **class** type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example,

```
int m = 20;
float x = 5.75;
```

are valid initialization statements for basic data types.

Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied. It is therefore clear that some more features of classes need to be explored that would enable us to initialize the objects when they are created and destroy them when their presence is no longer necessary.

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects. It also provides another member function called the *destructor* that destroys the objects when they are no longer required.

9.2 Constructors

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```
// class with a constructor

class integer
{
      int m, n;
   public:
      integer(void);      // constructor declared
      .....
      .....
};
integer :: integer(void)  // constructor defined
{
     m = 0; n = 0;
}
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

```
integer int1;             // object int1 created
```

not only creates the object **int1** of type **integer** but also initializes its data members **m** and **n** to zero. There is no need to write any statement to invoke the constructor function (as

we do with the normal member functions). If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately. This would be very inconvenient, if there are a large number of objects.

A constructor that accepts no parameters is called the *default constructor*. The default constructor for **class A is A::A()**. If no such constructor is defined, then the compiler supplies a default constructor. Therefore a statement such as

```
A a;
```

invokes the default constructor of the compiler to create the object **a**.

The constructor functions have some special characteristics. These are :

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual**.
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the operators **new** and **delete** when memory allocation is required.

Remember, when a constructor is declared for a class, initialization of the class objects becomes mandatory.

9.3 Parameterized Constructors

The constructor **integer()**, defined above, initializes the data members of all the objects to zero. However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called *parameterized constructors*.

The constructor **integer()** may be modified to take arguments as shown below:

```
class integer
{
        int m, n;
  public:
        integer(int x, int y);     // parameterized constructor
        .....
        .....
};
integer :: integer(int x, int y)
```

```
{
      m = x; n = y;
}
```

When a constructor has been parameterized, the object declaration statement such as

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

```
integer int1 = integer(0,100); // explicit call
```

This statement creates an integer object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100);       // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor. Program 9.1 demonstrates the passing of arguments to the constructor functions.

CLASS WITH CONSTRUCTORS

```
#include <iostream>

using namespace std;

class integer
{
      int m, n;
  public:
      integer(int, int);              // constructor declared

      void display(void)
      {
       cout << " m = " << m  << "\n";
       cout << " n = " << n  << "\n";
      }
};

integer :: integer(int x, int y)      // constructor defined
```

```
{
      m = x;  n = y;
}

int main()
{
     integer int1(0,100);              // constructor called implicitly

     integer int2 = integer(25, 75);   // constructor called explicitly

     cout << "\nOBJECT1" << "\n";
     int1.display();

     cout << "\nOBJECT2" << "\n";
     int2.display();

     return 0;
}
```

PROGRAM 9.1

Program 9.1 displays the following output:

```
OBJECT1
m = 0
n = 100
OBJECT2
m = 25
n = 75
```

The constructor functions can also be defined as **inline** functions. Example:

```
class integer
{
     int m, n;
  public:
     integer(int x, int y)   // Inline constructor
     {
          m = x; y = n;
     }
     .....
     .....
};
```

The parameters of a constructor can be of any type except that of the class to which it belongs. For example,

```
class A
{
```

```
        .....
        .....
    public:
      A(A);
};
```

is illegal.

However, a constructor can accept a *reference* to its own class as a parameter. Thus, the statement

```
Class A
{
     .....
     .....
  public:
    A(A&);
};
```

is valid. In such cases, the constructor is called the *copy constructor*.

9.4 Multiple Constructors in a Class

So far we have used two kinds of constructors. They are:

```
integer();            // No arguments
integer(int, int);    // Two arguments
```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In the second case, the function call passes the appropriate values from **main()**. C++ permits us to use both these constructors in the same class. For example, we could define a class as follows:

```
class integer
{
      int m, n;
   public:
      integer(){m=0; n=0;}        // constructor 1
      integer(int a, int b)
      {m = a; n = b;}             // constructor 2
      integer(integer & i)
      {m = i.m;  n = i.n;}        // constructor 3
};
```

This declares three constructors for an **integer** object. The first constructor receives no arguments, the second receives two **integer** arguments and the third receives one integer object as an argument. For example, the declaration

```
integer I1;
```

would automatically invoke the first constructor and set both **m** and **n** of **I1** to zero. The statement

```
integer I2(20,40);
```

would call the second constructor which will initialize the data members **m** and **n** of **I2** to 20 and 40 respectively. Finally, the statement

```
integer I3(I2);
```

would invoke the third constructor which copies the values of **I2** into **I3**. In other words, it sets the value of every data element of **I3** to the value of the corresponding data element of **I2**. As mentioned earlier, such a constructor is called the *copy constructor*. We learned in Chapter 7 that the process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor function is defined in a class, we say that the constructor is overloaded.

Program 9.2 shows the use of overloaded constructors.

OVERLOADED CONSTRUCTORS

```
#include <iostream>

using namespace std;

class complex
{
     float x, y;
  public:
     complex(){ }                    // constructor no arg
     complex(float a) {x = y = a;}   // constructor-one arg
     complex(float real, float imag)  // constructor-two args
     {x = real; y = imag;}

     friend complex sum(complex, complex);
     friend void show(complex);
};
complex sum(complex c1, complex c2) // friend
{
     complex c3;
     c3.x = c1.x + c2.x;
     c3.y = c1.y + c2.y;
     return(c3);
}
void show(complex c)         // friend
{
     cout << c.x << " + j" << c.y << "\n";
}
```

```
int main()
{
     complex A(2.7, 3.5);          // define & initialize
     complex B(1.6);               // define & initialize
     complex C;                    // define
     C = sum(A, B);                // sum() is a friend
     cout << "A = "; show(A);      // show() is also friend
     cout << "B = "; show(B);
     cout << "C = "; show(C);

// Another way to give initial values (second method)

     complex P,Q,R;                // define P, Q and R
     P = complex(2.5,3.9);         // initialize P
     Q = complex(1.6,2.5);         // initialize Q
     R = sum(P,Q);

     cout << "\n";
     cout << "P = "; show(P);
     cout << "Q = "; show(Q);
     cout << "R = "; show(R);

     return 0;
}
```

PROGRAM 9.2

The output of Program 9.2 would be:

```
A = 2.7 + j3.5
B = 1.6 + j1.6
C = 4.3 + j5.1

P = 2.5 + j3.9
Q = 1.6 + j2.5
R = 4.1 + j6.4
```

note

There are three constructors in the class **complex**. The first constructor, which takes no arguments, is used to create objects which are not initialized; the second, which takes one argument, is used to create objects and initialize them; and the third, which takes two arguments, is also used to create objects and initialize them to specific values. Note that the second method of initializing values looks better.

Let us look at the first constructor again.

```
complex(){ }
```

It contains the empty body and does not do anything. We just stated that this is used to create objects without any initial values. Remember, we have defined objects in the earlier examples without using such a constructor. Why do we need this constructor now? As pointed out earlier, C++ compiler has an *implicit constructor* which creates objects, even though it was not defined in the class.

This works fine as long as we do not use any other constructors in the class. However, once we define a constructor, we must also define the "do-nothing" implicit constructor. This constructor will not do anything and is defined just to satisfy the compiler.

9.5 Constructors with Default Arguments

It is possible to define constructors with default arguments. For example, the constructor complex() can be declared as follows:

```
complex(float real, float imag=0);
```

The default value of the argument **imag** is zero. Then, the statement

```
complex C(5.0);
```

assigns the value 5.0 to the **real** variable and 0.0 to **imag** (by default). However, the statement

```
complex C(2.0,3.0);
```

assigns 2.0 to **real** and 3.0 to **imag**. The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor **A::A()** and the default argument constructor **A::A(int = 0)**. The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

```
A a;
```

The ambiguity is whether to 'call' **A::A() or A::A(int = 0)**.

9.6 Dynamic Initialization of Objects

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment. Program 9.3 illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

DYNAMIC INITIALIZATION OF CONSTRUCTORS

```
// Long-term fixed deposit system

#include <iostream>

using namespace std;

class Fixed_deposit
{
       long int P_amount;      // Principal amount
       int    Years;           // Period of investment
       float  Rate;            // Interest rate
       float  R_value;         // Return value of amount
  public:
       Fixed_deposit(){ }
       Fixed_deposit(long int p, int y, float r=0.12);
       Fixed_deposit(long int p, int y, int r);
       void display(void);
};
Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
       P_amount = p;
       Years = y;
       Rate = r;
       R_value = P_amount;
       for(int i = 1; i <= y; i++)
       R_value = R_value * (1.0 + r);
}

Fixed_deposit :: Fixed_deposit(long int p, int y, int r)
{
       P_amount = p;
       Years = y;
       Rate = r;
       R_value = P_amount;

       for(int i=1; i<=y; i++)
            R_value = R_value*(1.0+float(r)/100);
}

void Fixed_deposit :: display(void)
{
     cout << "\n"
          << "Principal Amount = " << P_amount << "\n"
          << "Return Value     = " << R_value  << "\n";
}
```

```
int main()
{
     Fixed_deposit FD1, FD2, FD3;  // deposits created

     long int p;                   // principal amount

   int      y;               // investment period, years
   float    r;               // interest rate, decimal form
   int      R;               // interest rate, percent form

   cout << "Enter amount,period,interest rate(in percent)"<<"\n";
   cin >> p >> y >> R;
   FD1 = Fixed_deposit(p,y,R);

   cout << "Enter amount,period,interest rate(decimal form)" << "\n";
   cin >> p >> y >> r;
   FD2 = Fixed_deposit(p,y,r);

   cout << "Enter amount and period" << "\n";
   cin >> p >> y;
   FD3 = Fixed_deposit(p,y);

   cout << "\nDeposit 1";
   FD1.display();

   cout << "\nDeposit 2";
   FD2.display();

   cout << "\nDeposit 3";
   FD3.display();

   return 0;
}
```

PROGRAM 9.3

The output of Program 9.3 would be:

```
Enter amount,period,interest rate(in percent)
10000 3 18
Enter amount,period,interest rate(in decimal form)
10000 3 0.18
Enter amount and period
10000 3

Deposit 1
Principal Amount = 10000
```

```
Return Value     = 16430.3
Deposit 2
Principal Amount = 10000
Return Value     = 16430.3

Deposit 3
Principal Amount = 10000
Return Value     = 14049.3
```

The program uses three overloaded constructors. The parameter values to these constructors are provided at run time. The user can provide input in one of the following forms:

1. Amount, period and interest in decimal form.
2. Amount, period and interest in percent form.
3. Amount and period.

note

Since the constructors are overloaded with the appropriate parameters, the one that matches the input values is invoked. For example, the second constructor is invoked for the forms (1) and (3), and the third is invoked for the form (2). Note that, for form (3), the constructor with default argument is used. Since input to the third parameter is missing, it uses the default value for **r**.

9.7 Copy Constructor

We briefly mentioned about the copy constructor in Sec. 9.3. We used the copy constructor

```
integer(integer &i);
```

in Sec. 9.4 as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare and initialize an object from another object. For example, the statement

```
integer I2(I1);
```

would define the object I2 and at the same time initialize it to the values of I1. Another form of this statement is

```
integer I2 = I1;
```

The process of initializing through a copy constructor is known as *copy initialization*. Remember, the statement

```
I2 = I1;
```

will not invoke the copy constructor. However, if **I1** and **I2** are objects, this statement is legal and simply assigns the values of **I1** to **I2**, member-by-member. This is the task of the overloaded assignment operator(=). We shall see more about this later.

A copy constructor takes a reference to an object of the same class as itself as an argument. Let us consider a simple example of constructing and using a copy constructor as shown in Program 9.4.

COPY CONSTRUCTOR

```
#include <iostream>

using namespace std;

class code
{
     int id;
  public:
     code(){ }                    // constructor
     code(int a) { id = a;}       // constructor again
     code(code & x)               // copy constructor

     {
            id = x.id;            // copy in the value
     }
     void display(void)
     {
            cout << id;
     }
};
int main()
{
     code A(100);     // object A is created and initialized
     code B(A);       // copy constructor called
     code C = A;      // copy constructor called again

     code D; // D is created, not initialized
     D = A;           // copy constructor not called

     cout << "\n id of A: "; A.display();
     cout << "\n id of B: "; B.display();
     cout << "\n id of C: "; C.display();
     cout << "\n id of D: "; D.display();

     return 0;
}
```

PROGRAM 9.4

The output of Program 9.4 is shown below

```
id of A: 100
id of B: 100
id of C: 100
id of D: 100
```

note

A reference variable has been used as an argument to the copy constructor. We cannot pass the argument by value to a copy constructor.

When no copy constructor is defined, the compiler supplies its own copy constructor.

9.8 Dynamic Constructors

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator. Program 9.5 shows the use of new, in constructors that are used to construct strings in objects.

CONSTRUCTORS WITH NEW

```
#include <iostream>
#include <string>

using namespace std;

class String
{
      char *name;
      int  length;
  public:
      String()                      // constructor-1
      {
            length = 0;
            name = new char[length + 1];
      }

      String(char *s)               // constructor-2
      {
            length = strlen(s);
```

```
            name = new char[length + 1];      // one additional
                                              // character for \0
            strcpy(name, s);
            }

            void display(void)
            {cout << name << "\n";}
            void join(String &a, String &b);
};

void String :: join(String &a, String &b)
{
           length = a.length + b.length;
           delete name;
           name = new char[length+1];         // dynamic allocation

           strcpy(name, a.name);
           strcat(name, b.name);
};

int main()
{
        char *first = "Joseph ";
        String name1(first), name2("Louis "),name3("Lagrange"),s1,s2;

        s1.join(name1, name2);
        s2.join(s1, name3);
        name1.display();
        name2.display();
        name3.display();
        s1.display();
        s2.display();

        return 0;
}
```

PROGRAM 9.5

The output of Program 9.5 would be:

```
Joseph
Louis
Lagrange
Joseph Louis
Joseph Louis Lagrange
```

note

This Program uses two constructors. The first is an empty constructor that allows us to declare an array of strings. The second constructor initializes the **length** of the string, allocates necessary space for the string to be stored and creates the string itself. Note that one additional character space is allocated to hold the end-of-string character '\0'.

The member function **join( )** concatenates two strings. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string functions **strcpy( )** and **strcat( )**. Note that in the function **join( )**, **length** and **name** are members of the object that calls the function, while **a.length** and **a.name** are members of the argument object **a**. The **main( )** function program concatenates three strings into one string. The output is as shown below:

```
Joseph Louis Lagrange
```

9.9 Constructing Two-dimensional Arrays

We can construct matrix variables using the class type objects. The example in Program 9.6 illustrates how to construct a matrix of size m x n.

CONSTRUCTING MATRIX OBJECTS

```
#include <iostream>

using namespace std;

class matrix
{
        int **p;        // pointer to matrix
        int d1,d2;      // dimensions
   public:
        matrix(int x, int y);
        void get_element(int i, int j, int value)
        {p[i][j]=value;}
        int & put_element(int i, int j)
        {return p[i][j];}
};
matrix :: matrix(int x, int y)
{
        d1 = x;
        d2 = y;
        p = new int *[d1];              // creates an array pointer
        for(int i = 0; i < d1; i++)
        p[i] = new int[d2];             // creates space for each row
```

```
}

int main()
{
     int m, n;

     cout << "Enter size of matrix: ";
     cin  >> m >> n;
     matrix A(m,n); // matrix object A constructed

     cout << "Enter matrix elements row by row \n";
     int i, j, value;

     for(i = 0; i < m; i++)
            for(j = 0; j < n; j++)
            {
                   cin >> value;
                   A.get_element(i,j,value);
            }
   cout << "\n";
   cout << A.put_element(1,2);

   return 0;
};
```

PROGRAM 9.6

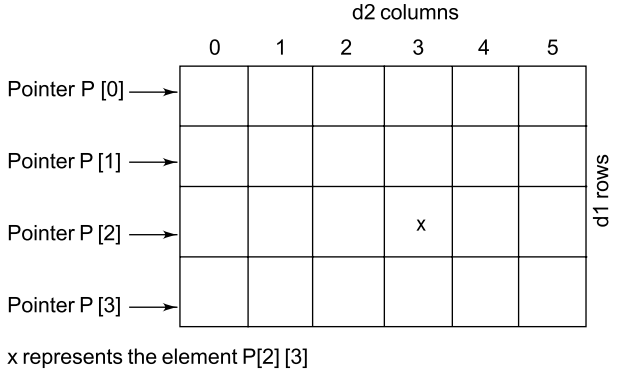
The output of a sample run of Program 9.6 is as follows.

```
Enter size of matrix: 3 4
Enter matrix elements row by row
11 12 13 14
15 16 17 18
19 20 21 22

17
```

17 is the value of the element (1,2).

The constructor first creates a vector pointer to an **int** of size **d1**. Then, it allocates, iteratively an **int** type vector of size **d2** pointed at by each element **p[i]**. Thus, space for the elements of a d1 × d2 matrix is allocated from free store as shown above.



9.10 const Objects

We may create and use constant objects using **const** keyword before object declaration. For example, we may create X as a constant object of the class **matrix** as follows:

```
const matrix X(m,n); // object X is constant
```

Any attempt to modify the values of **m** and **n** will generate compile-time error. Further, a constant object can call only **const** member functions. As we know, a **const** member is a function prototype or function definition where the keyword const appears after the function's signature.

Whenever **const** objects try to invoke non-**const** member functions, the compiler generates errors.

9.11 Destructors

A *destructor*, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example, the destructor for the class integer can be defined as shown below:

```
~integer(){ }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new** is used to allocate memory in the constructors, we should use **delete** to free that memory. For example, the destructor for the **matrix** class discussed above may be defined as follows:

```
matrix :: ~matrix()
{
     for(int i=0; i<d1; i++)
     delete p[i];
     delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

The example below illustrates that the destructor has been invoked implicitly by the compiler.

IMPLEMENTATION OF DESTRUCTORS

```
#include <iostream>

using namespace std;
```

```
int count = 0;

class alpha
{
  public:
    alpha()
    {
         count++;
         cout << "\nNo.of object created " << count;
    }

    ~alpha()
    {
         cout << "\nNo.of object destroyed " << count;
         count--;
    }
};

int main()
{
    cout << "\n\nENTER MAIN\n";

    alpha A1, A2, A3, A4;
    {
         cout << "\n\nENTER BLOCK1\n";
         alpha A5;
    }

    {
         cout << "\n\nENTER BLOCK2\n";
         alpha  A6;
    }
    cout << "\n\nRE-ENTER MAIN\n";

    return 0;
}
```

PROGRAM 9.7

The output of a sample run of Program 9.7 is shown below:

```
ENTER MAIN

No.of object created 1
No.of object created 2
```

```
No.of object created 3
No.of object created 4

ENTER BLOCK1

No.of object created 5
No.of object destroyed 5

ENTER BLOCK2

No.of object created 5
No.of object destroyed 5

RE-ENTER MAIN

No.of object destroyed 4
No.of object destroyed 3
No.of object destroyed 2
No.of object destroyed 1
```

note

As the objects are created and destroyed, they increase and decrease the count. Notice that after the first group of objects is created, **A5** is created, and then destroyed, **A6** is created, and then destroyed. Finally, the rest of the objects are also destroyed. When the closing brace of a scope is encountered, the destructors for each object in the scope are called. Note that the objects are destroyed in the reverse order of creation.

SUMMARY

- C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects.
- A constructor has the same name as that of a class.
- Constructors are normally used to initialize variables and to allocate memory.
- Similar to normal functions, constructors may be overloaded.
- When an object is created and initialized at the same time, a copy constructor gets called.
- We may make an object **const** if it does not modify any of its data values.
- C++ also provides another member function called the destructor that destroys the objects when they are no longer required.

Key Terms

- automatic initialization
- **Const**
- Constructor
- constructor overloading
- copy constructor
- copy initialization
- default argument
- default constructor
- **Delete**
- Destructor
- dynamic construction
- dynamic initialization
- explicit call
- implicit call
- implicit constructor
- initialization
- **new**
- parameterized constructor
- reference
- shorthand method
- **strcat()**
- **strcpy()**
- **strlen()**
- **virtual**

Review Questions

9.1 *What is a constructor? Is it mandatory to use constructors in a class?*

9.2 *How do we invoke a constructor function?*

9.3 *List some of the special properties of the constructor functions.*

9.4 *What is a parameterized constructor?*

9.5 *Can we have more than one constructors in a class? If yes, explain the need for such a situation.*

9.6 *What do you mean by dynamic initialization of objects? Why do we need to do this?*

9.7 *How is dynamic initialization of objects achieved?*

9.8 *Distinguish between the following two statements:*

```
time T2(T1);
time T2 = T1;
```

T1 and T2 are objects of **time** class.

9.9 *Describe the importance of destructors.*

9.10 *State whether the following statements are TRUE or FALSE.*

(a) *Constructors, like other member functions, can be declared anywhere in the class.*

(b) *Constructors do not return any values.*

(c) *A constructor that accepts no parameter is known as the default constructor.*

(d) *A class should have at least one constructor.*

(e) *Destructors never take any argument.*

Debugging Exercises

9.1 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
        int length;
        int width;
public:
        Room(int l, int w=0):
             width(w),
             length(l)
        {
        }
};
void main()
{
        Room objRoom1;
        Room objRoom2(12, 8);
}
```

9.2 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
        int length;
        int width;
public:
        Room()
        {
             length = 0;
             width = 0;
        }
        Room(int value=8)
        {
             length = width = 8;
        }
        void display()
        {
             cout << length << ' ' << width;
        }
};
```

```
void main()
{
        Room objRoom1;
        objRoom1.display();
}
```

9.3 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
        int width;
        int height;
        static int copyConsCount;
public:
        void Room()
        {
             width = 12;
             height = 8;
        }

        Room(Room& r)
        {
             width = r.width;
             height = r.height;
             copyConsCount++;
        }

        void dispCopyConsCount()
        {
             cout << copyConsCount;
        }
};

int Room::copyConsCount = 0;

void main()
{
        Room objRoom1;
        Room objRoom2(objRoom1);
        Room objRoom3 = objRoom1;
        Room objRoom4;
        objRoom4 = objRoom3;

        objRoom4.dispCopyConsCount();
}
```

9.4 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
        int width;
        int height;
        static int copyConsCount;
public:
        Room()
        {
             width = 12;
             height = 8;
        }

        Room(Room& r)
        {
             width = r.width;
             height = r.height;
             copyConsCount++;
        }

        void disCopyConsCount()
        {
             cout << copyConsCount;
        }
};

int Room::copyConsCount = 0;

void main()
{
        Room objRoom1;
        Room objRoom2 (objRoom1);
        Room objRoom3 = objRoom1;
        Room objRoom4;
        objRoom4 = objRoom3;

        objRoom4.dispCopyConsCount();
}
```

Programming Exercises

9.1 *Design constructors for the classes designed in Programming Exercises 8.1 through 8.5 of Chapter 8.*

9.2 *Define a class* ***String*** *that could work as a user-defined string type. Include constructors that will enable us to create an uninitialized string*

```
String s1; // string with length 0
```

and also to initialize an object with a string constant at the time of creation like

```
String s2("Well done!");
```

Include a function that adds two strings to make a third string. Note that the statement

```
s2 = s1;
```

will be perfectly reasonable expression to copy one string to another.

Write a complete program to test your class to see that it does the following tasks:

(a) *Creates uninitialized string objects.*
(b) *Creates objects with string constants.*
(c) *Concatenates two strings properly.*
(d) *Displays a desired string object.*

9.3 *A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise the message "Required copies not in stock" is displayed.*

Design a system using a class called ***books*** *with suitable member functions and constructors. Use* ***new*** *operator in constructors to allocate memory space required.*

9.4 *Improve the system design in Exercise 9.3 to incorporate the following features:*

(a) *The price of the books should be updated as and when required. Use a private member function to implement this.*
(b) *The stock value of each book should be automatically updated as soon as a transaction is completed.*
(c) *The number of successful and unsuccessful transactions should be recorded for the purpose of statistical analysis. Use* ***static*** *data members to keep count of transactions.*

9.5 *Modify the program of Exercise 9.4 to demonstrate the use of pointers to access the members.*

10

Operator Overloading and Type Conversions

Key Concepts

- Overloading
- Operator functions
- Overloading unary operators
- String manipulations
- Basic to class type
- Class to class type
- Operator overloading
- Overloading binary operators
- Using friends for overloading
- Type conversions
- Class to basic type
- Overloading rules

10.1 Introduction

Operator overloading is one of the many exciting features of C++ language. It is an important technique that has enhanced the power of extensibility of C++. We have stated more than once that C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as *operator overloading*.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can almost create a new language of our own by the creative use of the function and operator overloading techniques. We can overload (give additional meaning to) all the C++ operators except the following:

- Class member access operators (., .*).
- Scope resolution operator (::).

- Size operator **(sizeof)**.
- Conditional operator (?:).

The excluded operators are very few when compared to the large number of operators which qualify for the operator overloading definition.

Although the *semantics* of an operator can be extended, we cannot change its *syntax*, the grammatical rules that govern its use such as the number of operands, precedence and associativity. For example, the multiplication operator will enjoy higher precedence than the addition operator. Remember, when an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

10.2 Defining Operator Overloading

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called *operator function*, which describes the task. The general form of an operator function is:

```
return type classname :: operator op(arglist)
{
       Function body    // task defined
}
```

where *return type* is the type of value returned by the specified operation and *op* is the operator being overloaded. The *op* is preceded by the keyword **operator**. **operator** *op* is the function name.

Operator functions must be either member functions (non-static) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with **friend** functions. Arguments may be passed either by value or by reference. Operator functions are declared in the class using prototypes as follows:

```
vector operator+(vector);                    // vector addition
vector operator-();                          // unary minus
friend vector operator+(vector,vector);      // vector addition
friend vector operator-(vector);             // unary minus
vector operator-(vector &a);                 // subtraction
int operator==(vector);                      // comparison
friend int operator==(vector,vector)         // comparison
```

vector is a data type of **class** and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics)

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operaion.
2. Declare the operator function **operator** *op*() in the public part of the class. It may be either a member function or a **friend** function.
3. Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

```
op x or x op
```

for unary operators and

```
x op y
```

for binary operators. *op* x (or x *op*) would be interpreted as

```
operator op (x)
```

for **friend** functions. Similarly, the expression x op y would be interpreted as either

```
x.operator op (y)
```

in case of member functions, or

```
operator op (x,y)
```

in case of **friend** functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.

10.3 Overloading Unary Operators

Let us consider the unary minus operator. A minus operator when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an **int** or **float** variable. The unary minus when applied to an object should change the sign of each of its data items.

Program 10.1 shows how the unary minus operator is overloaded.

OVERLOADING UNARY MINUS

```
#include <iostream>
using namespace std;
class space
{
     int x;
     int y;
     int z;
public:
     void getdata(int a, int b, int c);
```

```
        void display(void);
        void operator-();       // overload unary minus
};
void space :: getdata(int a, int b, int c)
{
        x = a;
        y = b;
        z = c;
}
void space :: display(void)
{
        cout << x << " ";
        cout << y << " " ;
        cout << z << "\n";
}
void space :: operator-()
{
        x = -x;
        y = -y;
        z = -z;
}
int main()
{
        space S;
        S.getdata(10, -20, 30);
        cout << "S : ";
        S.display();
        -S;                          // activates operator-() function
        cout << "S : ";
        S.display();
        return 0;
}
```

PROGRAM 10.1

The Program 10.1 produces the following output:

```
S : 10 -20 30
S : -10 20 -30
```

note

The function **operator** – ( ) takes no argument. Then, what does this operator function do?. It changes the sign of data members of the object **S**. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

Remember, a statement like

```
S2 = -S1;
```

will not work because, the function **operator**–() does not return any value. It can work if the function is modified to return an object.

It is possible to overload a unary minus operator using a friend function as follows:

```
friend   void operator-(space &s);          // declaration
         void operator-(space &s)           // definition
  {
         s.x = –s.x;
         s.y = –s.y;
         s.z = –s.z;
  }
```

note

Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator-(). Therefore, the changes made inside the operator function will not reflect in the called object.

10.4 Overloading Binary Operators

We have just seen how to overload an unary operator. The same mechanism can be used to overload a binary operator. In Chapter 9, we illustrated, how to add two complex numbers using a friend function. A statement like

```
C = sum(A, B);      // functional notation.
```

was used. The functional notation can be replaced by a natural looking expression

```
C = A + B;          // arithmetic notation
```

by overloading the + operator using an operator+() function. The Program 10.2 illustrates how this is accomplished.

OVERLOADING + OPERATOR

```
#include <iostream>

using namespace std;

class complex
{
       float x;                             // real part
       float y;                             // imaginary part
```

```
  public:
       complex(){ }                          // constructor 1
       complex(float real, float imag)       // constructor 2
       { x = real; y = imag; }
       complex operator+(complex);
       void display(void);
};

complex complex :: operator+(complex c)
{
       complex temp;         // temporary
       temp.x = x + c.x;     // these are
       temp.y = y + c.y;     // float additions
       return(temp);
}

void complex :: display(void)
{
       cout << x << " + j" << y << "\n";
}

int main()
{
       complex C1, C2, C3;       // invokes constructor 1
       C1 = complex(2.5, 3.5);   // invokes constructor 2
       C2 = complex(1.6, 2.7);
       C3 = C1 + C2;

       cout << "C1 = "; C1.display();
       cout << "C2 = "; C2.display();
       cout << "C3 = "; C3.display();

       return 0;
}
```

PROGRAM 10.2

The output of Program 10.2 would be:

```
C1 = 2.5 + j3.5
C2 = 1.6 + j2.7
C3 = 4.1 + j6.2
```

note

Let us have a close look at the function **operator+( )** and see how the operator overloading is implemented.

```
complex complex :: operator+(complex c)
{
     complex temp;
     temp.x = x + c.x;
     temp.y = y + c.y;
     return(temp);
}
```

We should note the following features of this function:

1. It receives only one **complex** type argument explicitly.
2. It returns a **complex** type value.
3. It is a member function of **complex**.

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from? Now let us look at the statement that invokes this function:

```
C3 = C1 + C2;                // invokes operator+() function
```

We know that a member function can be invoked only by an object of the same class. Here, the object **C1** takes the responsibility of invoking the function and **C2** plays the role of an argument that is passed to the function. The above **invocation** statement is equivalent to

```
C3 = C1.operator+(C2);       // usual function call syntax
```

Therefore, in the **operator+()** function, the data members of **C1** are accessed directly and the data members of **C2** (that is passed as an argument) are accessed using the dot operator. Thus, both the objects are available for the function. For example, in the statement

```
temp.x = x + c.x;
```

c.x refers to the object **C2** and x refers to the object **C1. temp.x** is the real part of **temp** that has been created specially to hold the results of addition of **C1** and **C2**. The function returns the complex temp to be assigned to **C3**. Figure 10.1 shows how this is implemented.

As a rule, in overloading of binary operators, the *left-hand* operand is used to invoke the operator function and the *right-hand* operand is passed as an argument.

We can avoid the creation of the **temp** object by replacing the entire function body by the following statement:

```
return complex((x+c.x),(y+c.y));  // invokes constructor 2
```

What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object. Such an object is called a temporary object and goes out of space as soon as the contents are assigned to another object. Using *temporary objects* can make the code shorter, more efficient and better to read.

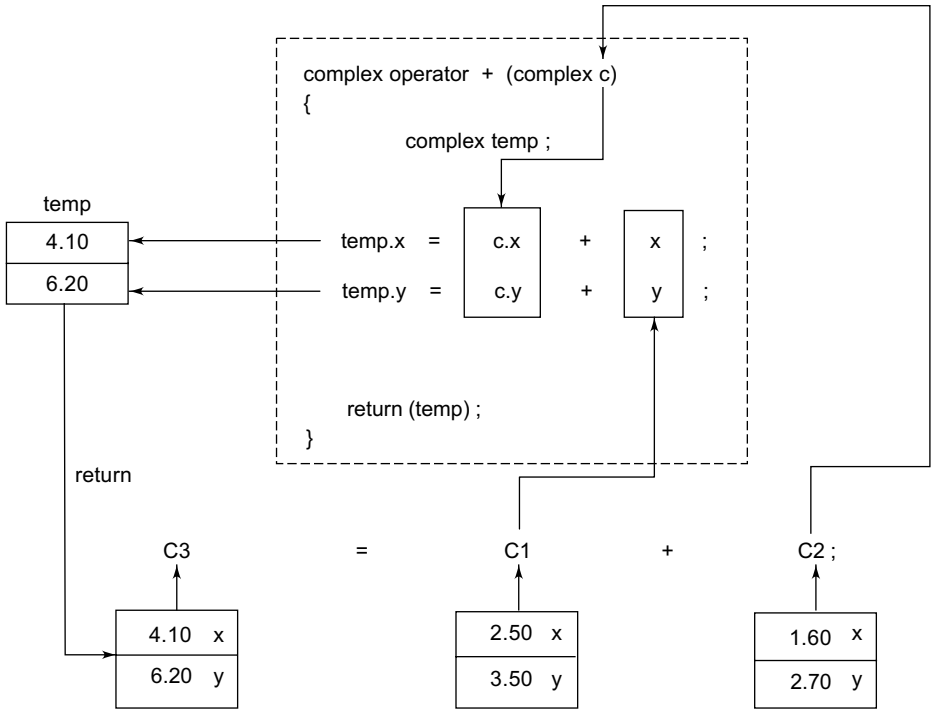


Fig. 10.1 ⇔ *Implementation of the overloded + operator*

10.5 Overloading Binary Operators Using Friends

As stated earlier, **friend** functions may be used in the place of member functions for overloading a binary operator, the only difference being that a **friend** function requires two arguments to be explicitly passed to it, while a member function requires only one.

The complex number program discussed in the previous section can be modified using a **friend** operator function as follows:

1. Replace the member function declaration by the **friend** function declaration.

```
friend complex operator+(complex, complex);
```

2. Redefine the operator function as follows:

```
complex operator+(complex a, complex b)
{
        return complex((a.x+b.x),(a.y+b.y));
}
```

In this case, the statement

```
C3 = C1 + C2;
```

is equivalent to

```
C3 = operator+(C1, C2);
```

In most cases, we will get the same results by the use of either a **friend** function or a member function. Why then an alternative is made available? There are certain situations where we would like to use a **friend** function rather than a member function. For instance, consider a situation where we need to use two different types of operands for a binary operator, say, one an object and another a built-in type data as shown below,

```
A = B + 2; (or A = B * 2;)
```

where **A** and **B** are objects of the same class. This will work for a member function but the statement

```
A = 2 + B; (or A = 2 * B)
```

will not work. This is because the left-hand operand which is responsible for invoking the member function should be an object of the same class. However **friend** function allows both approaches. How?

It may be recalled that an object need not be used to invoke a **friend** function but can be passed as an argument. Thus, we can use a friend function with a built-in type data as the *left-hand* operand and an object as the *right-hand* operand. Program 10.3 illustrates this, using scalar *multiplication* of a vector. It also shows how to overload the input and output operators >> and <<.

OVERLOADING OPERATORS USING FRIENDS

```
#include <iostream.h>
  const size = 3;
  class vector
  {
        int v[size];
   public:
        vector();                  // constructs null vector
        vector(int *x);            // constructs vector from array
        friend vector operator *(int a, vector b);      // friend 1
        friend vector operator *(vector b, int a);      // friend 2
        friend istream & operator >> (istream &, vector &);
        friend ostream & operator << (ostream &, vector &);
  };

  vector :: vector()
  {
        for(int i=0; i<size; i++)
              v[i] = 0;
  }
```

```
vector :: vector(int *x)
{
        for(int i=0; i<size; i++)
                v[i] = x[i];
}

vector operator *(int a, vector b)
{
        vector c;

        for(int i=0; i < size; i++)
                c.v[i] = a * b.v[i];
        return c;
}

        vector operator *(vector b, int a)
{
        vector c;

        for(int i=0; i<size; i++)
                c.v[i] = b.v[i] * a;
        return c;
}

istream & operator >> (istream &din, vector &b)

{
        for(int i=0; i<size; i++)
                din >> b.v[i];
        return(din);
}
ostream & operator <<  (ostream &dout, vector &b)
{
        dout << "(" << b.v [0];

        for(int i=1; i<size; i++)
                dout << ", " << b.v[i];
        dout << ")";
        return(dout);
}

int x[size] = {2,4,6};

int main()
{
```

```
        vector m;        // invokes constructor 1
        vector n = x;    // invokes constructor 2

        cout << "Enter elements of vector m " << "\n";
        cin >> m;        // invokes operator>>() function

        cout << "\n";
        cout << "m = " << m << "\n";   // invokes operator <<()

        vector p, q;

        p = 2 * m;       // invokes friend 1
        q = n * 2;       // invokes friend 2

        cout << "\n";
        cout << "p = " << p << "\n";        // invokes operator<<()
        cout << "q = " << q << "\n";

        return 0;
    }
```

PROGRAM 10.3

Shown below is the output of Program 10.3:

```
Enter elements of vector m
5 10 15

m = (5, 10, 15)
p = (10, 20, 30)
q = (4, 8, 12)
```

The program overloads the operator * two times, thus overloading the operator function operator*() itself. In both the cases, the functions are explicitly passed two arguments and they are invoked like any other overloaded function, based on the types of its arguments. This enables us to use both the forms of scalar multiplication such as

```
p = 2 * m;        // equivalent to p = operator*(2,m);
q = n * 2;        // equivalent to q = operator*(n,2);
```

The program and its output are largely self-explanatory. The first constructor

```
vector();
```

constructs a vector whose elements are all zero. Thus

```
vector m;
```

creates a vector m and initializes all its elements to 0. The second constructor

```
vector(int &x);
```

creates a vector and copies the elements pointed to by the pointer argument x into it. Therefore, the statements

```
int x[3] = {2, 4, 6};
vector n = x;
```

create *n* as a vector with components 2, 4, and 6.

note

We have used vector variables like **m** and **n** in input and output statements just like simple variables. This has been made possible by overloading the operators >> and << using the functions:

```
friend istream & operator>>(istream &, vector &);
friend ostream & operator<<(ostream &, vector &);
```

istream and **ostream** are classes defined in the **iostream.h** file which has been included in the program.

10.6 Manipulation of Strings Using Operators

ANSI C implements strings using character arrays, pointers and string functions. There are no operators for manipulating the strings. One of the main drawbacks of string manipulations in C is that whenever a string is to be copied, the programmer must first determine its length and allocate the required amount of memory.

Although these limitations exist in C++ as well, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers. (Recently, ANSI C++ committee has added a new class called **string** to the C++ class library that supports all kinds of string manipulations.

For example, we shall be able to use statements like

```
string3 = string1 + string2;
if(string1 >= string2) string = string1;
```

Strings can be defined as class objects which can be then manipulated like the built-in types. Since the strings vary greatly in size, we use *new* to allocate memory for each string and a pointer variable to point to the string array. Thus we must create string objects that can hold these two pieces of information, namely, length and location which are necessary for string manipulations. A typical string class will look as follows:

```
class string
{
        char *p;    // pointer to string
        int len;    // length of string
public:
        .....       // member functions
```

```
        .....       // to initialize and
        .....       // manipulate strings
};
```

We shall consider an example to illustrate the application of overloaded operators to strings. The example shown in Program 10.4 overloads two operators, + and <= just to show how they are implemented. This can be extended to cover other operators as well.

MATHEMATICAL OPERATIONS ON STRINGS

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

class string
{
        char *p;
        int len;
public:
        string() {len = 0; p = 0;}          // create null string
        string(const char * s);             // create string from arrays
        string(const string & s);           // copy constructor
        ~ string(){delete p;}               // destructor

        // + operator
        friend string operator+(const string &s, const string &t);

        // <= operator
        friend int operator<=(const string &s, const string &t);
        friend void show(const string s);
};
string :: string(const char *s)
{
        len = strlen(s);
        p   = new char[len+1];
        strcpy(p,s);
}

string :: string(const string & s)
{
        len = s.len;
        p   = new char[len+1];
        strcpy(p,s.p);
}

// overloading + operator
string operator+(const string &s, const string &t)
```

```
{
        string temp;
        temp.len = s.len + t.len;
        temp.p = new char[temp.len+1];
        strcpy(temp.p,s.p);
        strcat(temp.p,t.p);
        return(temp);
}
// overloading <= operator
int operator<=(const string &s, const string &t)
{
        int m = strlen(s.p);
        int n = strlen(t.p);

        if(m <= n) return(1);

        else return(0);
}
void show(const string s)
{
        cout << s.p;
}
int main()
{
        string s1 = "New ";
        string s2 = "York";
        string s3 = "Delhi";
        string t1,t2,t3;
        t1 = s1;
        t2 = s2;
        t3 = s1+s3;

        cout << "\nt1 = "; show(t1);
        cout << "\nt2 = "; show(t2);
        cout << "\n";
        cout << "\nt3 = "; show(t3);
        cout << "\n\n";

        if(t1 <= t3)
        {
                show(t1);
                cout << " smaller than ";
                show(t3);
                cout << "\n";
        }
```

```
        else
        {
                show(t3);
                cout << " smaller than ";
                show(t1);
                cout << "\n";
        }

        return 0;
}
```

PROGRAM 10.4

The following is the output of Program 10.4

```
t1 = New
t2 = York

t3 = New Delhi

New smaller than New Delhi
```

10.7 Rules for Overloading Operators

Although it looks simple to redefine the operators, there are certain restrictions and limitations in overloading them. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded. (See Table 10.1.)
6. We cannot use **friend** functions to overload certain operators. (See Table 10.2.) However, member functions can be used to overload them.
7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +, –, *, and / must explicitly return a value. They must not attempt to change their own arguments.

Table 10.1 *Operators that cannot be overloaded*

Sizeof	Size of operator
.	Membership operator
.*	Pointer-to-member operator
::	Scope resolution operator
?:	Conditional operator

Table 10.2 *Where a friend cannot be used*

=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
->	Class member access operator

10.8 Type Conversions

When constants and variables of different types are mixed in an expression, C applies automatic type conversion to the operands as per certain rules. Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left. For example, the statements

```
int m;
float x = 3.14159;
m = x;
```

convert **x** to an integer before its value is assigned to **m**. Thus, the fractional part is truncated. The type conversions are automatic as long as the data types involved are built-in types.

What happens when they are user-defined data types?

Consider the following statement that adds two objects and then assigns the result to a third object.

```
v3 = v1 + v2;    // v1, v2 and v3 are class type objects
```

When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. We have seen, in the case of class objects, that the values of all the data members of the right-hand object are simply copied into the corresponding members of the object on the left-hand. What if one of the operands is an object and the other is a built-in type variable? Or, what if they belong to two different classes?

Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types. We must, therefore, design the conversion routines by ourselves, if such operations are required.

Three types of situations might arise in the data conversion between uncompatible types:

1. Conversion from basic type to class type.

2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

We shall discuss all the three cases in detail.

10.8.1 Basic to Class Type

The conversion from basic type to class type is easy to accomplish. It may be recalled that the use of constructors was illustrated in a number of examples to initialize objects. For example, a constructor was used to build a vector object from an **int** type array. Similarly, we used another constructor to build a string type object from a **char*** type variable. These are all examples where constructors perform a *defacto* type conversion from the argument's type to the constructor's class type.

Consider the following constructor:

```
string :: string(char *a)
{
     length = strlen(a);
     P = new char[length+1];
     strcpy(P,a);
}
```

This constructor builds a **string** type object from a **char*** type variable **a**. The variables **length** and **p** are data members of the class **string**. Once this constructor has been defined in the string class, it can be used for conversion from **char*** type to string type. Example:

```
string s1, s2;
char* name1 = "IBM PC";
char* name2 = "Apple Computers";
s1 = string(name1);
s2 = name2;
```

The statement

```
s1 = string(name1);
```

first converts **name1** from **char*** type to **string** type and then assigns the string type values to the object **s1**. The statement

```
s2 = name2;
```

also does the same job by invoking the constructor implicitly.

Let us consider another example of converting an **int** type to a **class** type.

```
class time
{
    int hrs;
     int mins;
  public:
```

```
    ....
    ....
    time(int t)              // constructor
    {
        hours = t/60;        // t in minutes
        mins = t%60;
    }
};
```

The following conversion statements can be used in a function:

```
time T1;                     // object T1 created
int duration = 85;
T1 = duration;               // int to class type
```

After this conversion, the **hrs** member of **T1** will contain a value of 1 and **mins** member a value of 25, denoting 1 hours and 25 minutes.

note

The constructors used for the type conversion take a single argument whose type is to be converted.

In both the examples, the left-hand operand of = operator is always a **class** object. Therefore, we can also accomplish this conversion using an overloaded = operator.

10.8.2 Class to Basic Type

The constructors did a fine job in type conversion from a basic to class type. What about the conversion from a class to basic type? The constructor functions do not support this operation. Luckily, C++ allows us to define an overloaded *casting operator* that could be used to convert a class type data to a basic type. The general form of an overloaded casting operator function, usually referred to as a *conversion function*, is:

```
operator typename()
{
        .....
        ..... (Function statements)
        .....
}
```

This function converts a class type data to *typename*. For example, the **operator double()** converts a class object to type **double**, the **operator int()** converts a class type object to type int, and so on.

Consider the following conversion function:

```
vector :: operator double()
{
  double sum = 0;
```

```
    for(int i=0; i<size; i++)
           sum = sum + v[i] * v[i];
    return sqrt(sum);
}
```

This function converts a vector to the corresponding scalar magnitude. Recall that the magnitude of a vector is given by the square root of the sum of the squares of its components. The operator **double()** can be used as follows:

```
double length = double(V1);
    or
double length = V1;
```

where **V1** is an object of type **vector**. Both the statements have exactly the same effect. When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do the job.

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

Since it is a member function, it is invoked by the object and, therefore, the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

In the string example described in the previous section, we can do the conversion from string to **char*** as follows:

```
string :: operator char*()
{
       return(p);
}
```

10.8.3 One Class to Another Class Type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But there are situations where we would like to convert one class type data to another class type.

Example:

```
objX = objY;  // objects of different types
```

objX is an object of class **X** and **objY** is an object of class **Y**. The **class Y** type data is converted to the **class X** type data and the converted value is assigned to the **objX**. Since the conversion takes place from **class Y** to **class X, Y** is known as the *source* class and **X** is known as the *destination* class.

Such conversions between objects of different classes can be carried out by either a constructor or a conversion function. The compiler treats them the same way. Then, how do we

decide which form to use? It depends upon where we want the type-conversion function to be located in the source class or in the destination class.

We know that the casting operator function

```
operator typename()
```

converts the class object *of which it is a member* to *typename*. The *typename* may be a built-in type or a user-defined one (another class type). In the case of conversions between objects, *typename* refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used (i.e. source class). The conversion takes place in the source class and the result is given to the destination class object.

Now consider a single-argument constructor function which serves as an instruction for converting the *argument's type* to the class type of *which it is a member*. This implies that the argument belongs to the *source* class and is passed to the *destination* class for conversion. This makes it necessary that the conversion constructor be placed in the destination class. Figure 10.2 illustrates these two approaches.

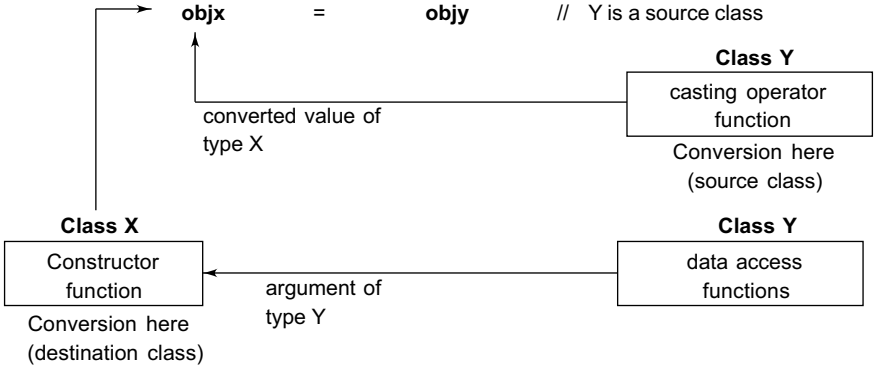


Fig. 10.2 ⇔ *Conversion between object*

Table 10.3 provides a summary of all the three conversions. It shows that the conversion from a class to any other type (or any other class) should make use of a casting operator in the source class. On the other hand, to perform the conversion from any other type/class to a class type, a constructor should be used in the destination class.

Table 10.3 *Type conversions*

Conversion required	***Conversion takes place in Source class***	***Destination class***
Basic → class	Not applicable	Constructor
Class → basic	Casting operator	Not applicable
Class → class	Casting operator	Constructor

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument. Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destination class.

10.8.4 A Data Conversion Example

Let us consider an example of an inventory of products in store. One way of recording the details of the products is to record their code number, total items in the stock and the cost of each item. Another approach is to just specify the item code and the value of the item in the stock. The example shown in Program 10.5 uses two classes and shows how to convert data of one type to another.

DATA CONVERSIONS

```
#include <iostream>

using namespace std;

class invent2                // destination class declared

class invent1                // source class
{
        int code;            // item code
        int items;           // no. of items
        float price;         // cost of each item
public:
  invent1(int a, int b, float c)
  {
        code =  a;
        items = b;
        price = c;
  }
  void putdata()
  {
              cout << "Code:  " << code  << "\n";
              cout << "Items: " << items << "\n";
              cout << "Value: " << price << "\n";
  }
  int getcode() {return code;}
  int getitems() {return items;}
  float getprice() {return price;}
  operator float() {return(items * price);}
```

```
    /* operator invent2()        // invent1 to invent2
    {
            invent2 temp;
            temp.code = code;
            temp.value = price * items;
            return temp;
    } */
};          // End of source class
class invent2    // destination class
{
            int code;
            float value;
  public:
            invent2()
            {
                    code = 0; value = 0;
            }
            invent2(int x, float y)       // constructor for
                                          // initialization
            {
                    code = x;
                    value = y;
            }
            void putdata()
            {
                    cout << "Code:  " << code  << "\n";
                    cout << "Value: " << value << "\n\n";
            }
            invent2(invent1 p)    // constructor for conversion
            {
                    code = p.getcode();
                    value =   p.getitems() * p.getprice();
            }
};          // End of destination class

int main()
{
            invent1 s1(100,5,140.0);
            invent2 d1;
            float total_value;

            /* invent1 To float  */
```

```
            total_value = s1;

            /* invent1 To invent2 */
            d1 = s1;

            cout << "Product details - invent1 type" << "\n";
            s1.putdata();

            cout << "\nStock value" << "\n";
            cout << "Value = " << total_value << "\n\n";

            cout << "Product details-invent2 type" << "\n";
            d1.putdata();

            return 0;
}
```

PROGRAM 10.5

Following is the output of Program 10.5:

```
Product details-invent1 type
Code:  100
Items: 5
Value: 140
Stock value
Value = 700
Product details-invent2 type
Code:  100
Value: 700
```

note

We have used the conversion function

```
operator float( )
```

in the class **invent1** to convert the **invent1** type data to a **float**. The constructor

```
invent2 (invent1)
```

is used in the class **invent2** to convert the **invent1** type data to the **invent2** type data.

Remember that we can also use the casting operator function

```
operator invent2()
```

in the class invent1 to convert **invent1** type to **invent2** type. However, it is important that we do not use both the constructor and the casting operator for the same type conversion, since this introduces an ambiguity as to how the conversion should be performed.

SUMMARY

- Operator overloading is one of the important features of C++ language. It is called compile time polymorphism.
- Using overloading feature we can add two user defined data types such as objects, with the same syntax, just as basic data types.
- We can overload almost all the C++ operators except the following:
  - class member access operators(., .*)
  - scope resolution operator (::)
  - size operator(sizeof)
  - conditional operator(?:)
- Operator overloading is done with the help of a special function, called operator function, which describes the special task to an operator.
- There are certain restrictions and limitations in overloading operators. Operator functions must either be member functions (non-static) or friend functions. The overloading operator must have at least one operand that is of user-defined type.
- The compiler does not support automatic type conversions for the user defined data types. We can use casting operator functions to achieve this.
- The casting operator function should satisfy the following conditions:
  - It must be a class member.
  - It must not specify a return type.
  - It must not have any arguments.

Key Terms

- arithmetic notation
- binary operators
- casting
- casting operator
- constructor
- conversion function
- destination class
- **friend**
- **friend** function
- functional notation
- manipulating strings
- operator
- operator function
- operator overloading
- scalar multiplication
- semantics

- **sizeof**
- source class
- syntax
- temporary object
- type conversion
- unary operators

Review Questions

10.1 *What is operator overloading?*

10.2 *Why is it necessary to overload an operator?*

10.3 *What is an operator function? Describe the syntax of an operator function.*

10.4 *How many arguments are required in the definition of an overloaded unary operator?*

10.5 *A class alpha has a constructor as follows:*

```
alpha(int a, double b);
```

Can we use this constructor to convert types?

10.6 *What is a conversion function How is it created Explain its syntax.*

10.7 *A friend function cannot be used to overload the assignment operator =. Explain why?*

10.8 *When is a friend function compulsory? Give an example.*

10.9 *We have two classes X and Y. If* ***a*** *is an object of X and* ***b*** *is an object of* ***Y*** *and we want to say* ***a*** *=* ***b****; What type of conversion routine should be used and where?*

10.10 *State whether the following statements are TRUE or FALSE.*

(a) *Using the operator overloading concept, we can change the meaning of an operator.*

(b) Operator overloading works when applied to class objects only.

(c) Friend functions cannot be used to overload operators.

(d) When using an overloaded binary operator, the left operand is implicitly passed to the member function.

(e) The overloaded operator must have at least one operand that is user-defined type.

(f) Operator functions never return a value.

(g) Through operator overloading, a class type data can be converted to a basic type data.

(h) A constructor can be used to convert a basic type to a class type data.

Debugging Exercises

10.1 Identify the error in the following program.

```
#include <iostream.h>
class Space
{
     int mCount;
```

```
public:
     Space()
     {
          mCount = 0;
     }

     Space operator ++()
     {
          mCount++;
          return Space(mCount);
     }
};

void main()
{
          Space objSpace;
          objSpace++;
}
```

10.2 Identify the error in the following program.

```
#include <iostream.h>
enum WeekDays
{
     mSunday,
     mMonday,
     mTuesday,
     mWednesday,
     mThursday,
     mFriday,
     mSaturday
};
bool op==(WeekDays& w1, WeekDays& w2)
{
     if(w1== mSunday && w2 == mSunday)
          return 1;
     else if(w1== mSunday && w2 == mSunday)
          return 1;
     else if(w1== mSunday && w2 == mSunday)
          return 1;
     else if(w1== mSunday && w2 == mSunday)
          return 1;
     else if(w1== mSunday && w2 == mSunday)
          return 1;
     else if(w1== mSunday && w2 == mSunday)
          return 1;
     else if(w1== mSunday && w2 == mSunday)
```

```
            return 1;
        else
            return 0;
    }
    void main()
    {
        WeekDays w1 = mSunday, w2 = mSunday;
        if(w1==w2)
            cout << "Same day";
        else
            cout << "Different day";
    }
```

10.3 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
    float mWidth;
    float mLength;
public:
    Room()
    {
    }
    Room(float w, float h)
        :mWidth(w), mLength(h)
    {
    }
    operator float()
    {
        return (float)mWidth * mLength;
    }

    float getWidth()
    {
    }

    float getLength()
    {
        return mLength;
    }
};

void main()
{
    Room objRoom1(2.5, 2.5);
    float fTotalArea;
```

```
    fTotalArea = objRoom1;
    cout << fTotalArea;
}
```

Programming Exercises

NOTE: *For all the exercises that follow, build a demonstration program to test your code.*

10.1 *Create a class* ***FLOAT*** *that contains one float data member. Overload all the four arithmetic operators so that they operate on the objects of* ***FLOAT****.*

10.2 *Design a class* ***Polar*** *which describes a point in the plane using polar coordinates* ***radius*** *and* ***angle****. A point in polar coordinates is shown in Fig. 10.3.*

Use the overloaded + operator to add two objects of Polar.

Note that we cannot add polar values of two points directly. This requires first the conversion of points into rectangular co-ordinates, then adding the corresponding rectangular co-ordinates and finally converting the result back into polar co-ordinates. You need to use the following trigonometric formulae:

```
x = r * cos(a);
y = r * sin(a);
a = atan(y/x);    // arc tangent
r = sqrt(x*x + y*y);
```

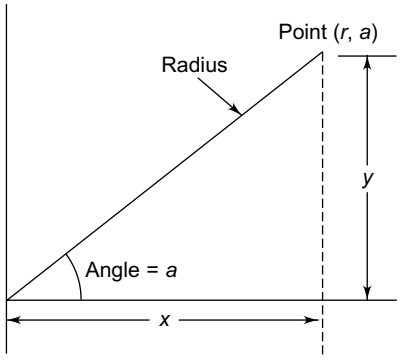


Fig. 10.3 ⇔ *Polar coordinates of a point*

10.3 *Create a class* ***MAT*** *of size* ***m*** *x* ***n****. Define all possible matrix operations for* ***MAT*** *type objects.*

10.4 *Define a class String. Use overloaded == operator to compare two strings.*

10.5 *Define two classes* ***Polar*** *and* ***Rectangle*** *to represent points in the polar and rectangle systems. Use conversion routines to convert from one system to the other.*

11

Derived Classes and Inheritance

Key Concepts

- Reusability
- Inheritance
- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hybrid inheritance
- Hierarchical inheritance
- Defining a derived class
- Inheritiing private members
- Virtual base class
- Direct base class
- Indirect base class
- Abstract class
- Defining derived class constructors
- Nesting of classes

11.1 Introduction

Reusability is yet another important feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not only save time and money but also reduce frustration and increase reliability. For instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Fortunately, C++ strongly supports the concept of *reusability*. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called *inheritance (or derivation)*. The old class is referred to as the *base class* and the new one is called the *derived class or subclass*.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one

class or from more than one level. A derived class with only one base class, is called *single inheritance* and one with several base classes is called *multiple inheritance*. On the other hand, the traits of one class may be inherited by more than one class. This process is known as *hierarchical inheritance*. The mechanism of deriving a class from another 'derived class' is known as *multilevel inheritance*. Figure 11.1 shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance. (Some authors show the arrow in opposite direction meaning "inherited from".)

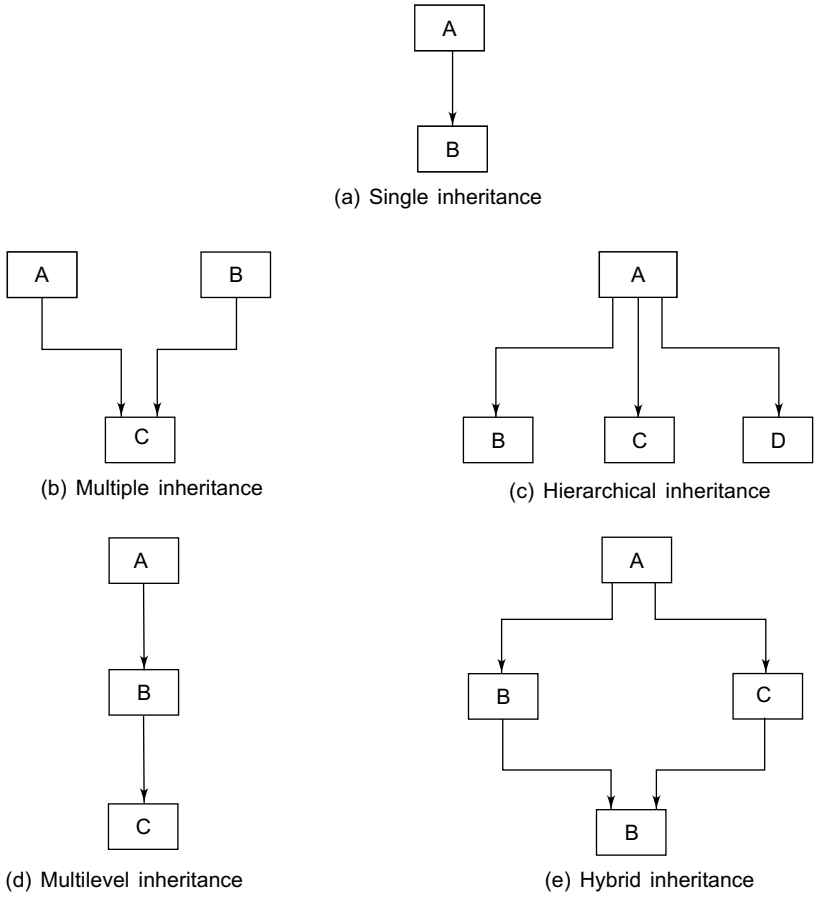


Fig. 11.1 ⇔ *Forms of inheritance*

11.2 Defining Derived Classes

A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:

```
class derived-class-name  : visibility-mode base-class-name
{
        .....//
        .....//   members of derived class
        .....//
};
```

The colon indicates that the *derived-class-name* is derived from the *base-class-name*. The *visibility-mode* is optional and, if present, may be either **private** or **public**. The default visibility-mode is **private**. Visibility mode specifies whether the features of the base class are *privately derived or publicly derived*.

Examples:

```
class ABC: private XYZ       // private derivation
{
      members of ABC
};

class ABC: public XYZ        // public derivation
{
      members of ABC
};

class ABC: XYZ              // private derivation by default
{
      members of ABC
};
```

When a base class is *privately inherited* by a derived class, 'public members' of the base class become 'private members' of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class. Remember, a public member of a class can be accessed by its own objects using the *dot operator*. The result is that no member of the base class is accessible to the objects of the derived class.

On the other hand, when the base class is *publicly inherited*, 'public members' of the base class become 'public members' of the derived class and therefore they are accessible to the objects of the derived class. In *both the cases, the private members are not inherited* and therefore, the private members of a base class will never become the members of its derived class.

In inheritance, some of the base class data elements and member functions are 'inherited' into the derived class. We can add our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

11.3 Single Inheritance

Let us consider a simple example to illustrate inheritance. Program 11.1 shows a base class B and a derived class D. The class B contains one private data member, one public data member, and three public member functions. The class D contains one private data member and two public member functions.

SINGLE INHERITANCE : PUBLIC

```
#include <iostream>

using namespace std;

class B
{
        int a;              // private; not inheritable
  public:
        int b;              // public; ready for inheritance
        void get_ab();
        int  get_a(void);
        void show_a(void);
};

class D : public B          // public derivation
{
        int c;
   public:
        void mul(void);
        void display(void);
};
//-------------------------------------------------------
void B :: get_ab(void)
{
        a = 5; b = 10;
}
int B :: get_a()
{
        return a;
}
void B :: show_a()
{
        cout << "a = " << a << "\n";
}
void D :: mul()
{
```

```
        c = b * get_a();
}
void D :: display()
{
        cout << "a = " << get_a() << "\n";
        cout << "b = " << b << "\n";
        cout << "c = " << c << "\n\n";
}
//-------------------------------------------------------
int main()
{
        D d;

        d.get_ab();
        d.mul();
        d.show_a();
        d.display();

        d.b = 20;
        d.mul();
        d.display();

        return 0;
}
```

PROGRAM 11.1

Given below is the output of Program 11.1:

```
a = 5
a = 5
b = 10
c = 50

a = 5
b = 20
c = 100
```

The class **D** is a public derivation of the base class **B**. Therefore, **D** inherits all the **public** members of **B** and retains their visibility. Thus a **public** member of the base class **B** is also a public member of the derived class **D**. The **private** members of **B** cannot be inherited by **D**. The class **D**, in effect, will have more members than what it contains at the time of declaration as shown in Fig. 11.2.

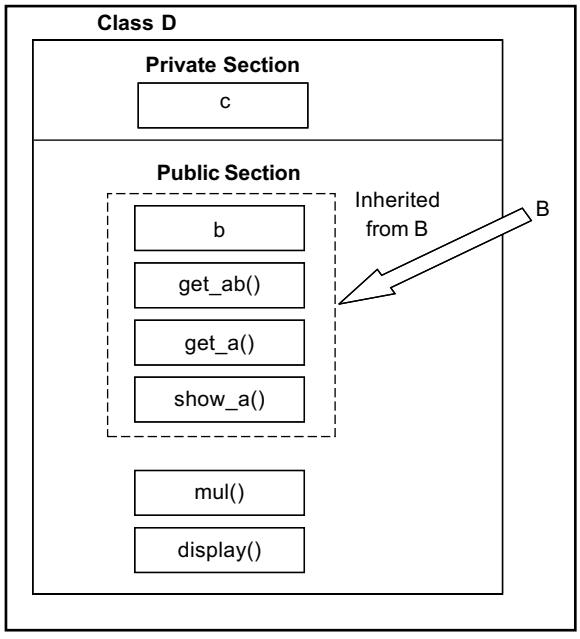


Fig. 11.2 ⇔ *Adding more members to a class (by public derivation)*

The program illustrates that the objects of class **D** have access to all the public members of **B**. Let us have a look at the functions **show_a()** and **mul()**:

```
void show_a()
{
     cout << "a = " << a << "\n";
}

void mul()
{
     c = b * get_a();      // c = b * a
}
```

Although the data member **a** is private in **B** and cannot be inherited, objects of **D** are able to access it through an inherited member function of **B**.

Let us now consider the case of private derivation.

```
class B
{
       int a;
  public:
       int b;
```

```
        void get_ab();
void get_a();
        void show_a();
};

class D : private B          // private derivation
{
        int c;
  public:
        void mul();
        void display();
};
```

The membership of the derived class **D** is shown in Fig. 11.3. In **private** derivation, the **public** members of the base class become **private** members of the derived class. Therefore, the objects of **D** can not have direct access to the public member functions of **B**.

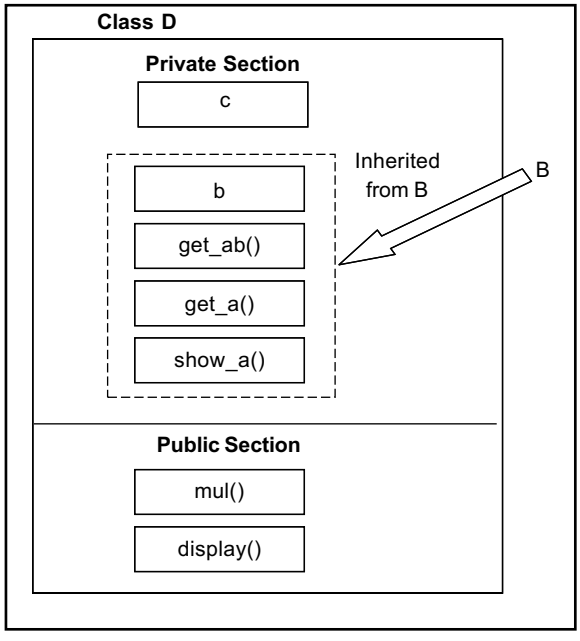


Fig. 11.3 ⇔ *Adding more members to a class (by private derivation)*

The statements such as

```
d.get_ab();      // get_ab() is private
d.get_a();       // so also get_a()
d.show_a();      // and show_a()
```

will not work. However, these functions can be used inside **mul()** and **display()** like the normal functions as shown below:

```
void mul()
{
     get_ab();
     c = b * get_a();
}

void display()
{
     show_a();       // outputs value of 'a'
     cout << "b = " << b << "\n"
          << "c = " << c << "\n\n";
}
```

Program 11.2 incorporates these modifications for private derivation. Please compare this with Program 11.1.

SINGLE INHERITANCE : PRIVATE

```
#include <iostream>

using namespace std;

class B
{
       int a;         // private; not inheritable
     public:
       int b;         // public; ready for inheritance
       void get_ab();
       int  get_a(void);
       void show_a(void);
};

class D : private B          // private derivation
{
      int c;
   public:
      void mul(void);
      void display(void);
};

//-------------------------------------------------------

void B :: get_ab(void)
```

```
{
     cout << "Enter values for a and b:";
     cin >> a >> b;
}

int B :: get_a()
{
     return a;
}

void B :: show_a()
{
      cout << "a = " << a << "\n";
}

void D :: mul()
{
      get_ab();
      c = b * get_a();    // 'a' cannot be used directly
}

void D :: display()
{
      show_a();           // outputs value of 'a'
      cout << "b = " << b << "\n"
           << "c = " << c << "\n\n";
}

//-------------------------------------------------------

int main()
{
     D d;

     // d.get_ab();  WON'T WORK
     d.mul();
     // d.show_a();  WON'T WORK
     d.display();
     // d.b = 20;     WON'T WORK; b has become private
     d.mul();
     d.display();

     return 0;
}
```

PROGRAM 11.2

The output of Program 11.2 would be:

```
Enter values for a and b:5 10
a = 5
b = 10
c = 50
Enter values for a and b:12 20
a = 12
b = 20
c = 240
```

Suppose a base class and a derived class define a function of the same name. What will happen when a derived class object invokes the function? In such cases, the derived class function supersedes the base class definition. The base class function will be called only if the derived class does not redefine the function.

11.4 Making a Private Member Inheritable

We have just seen how to increase the capabilities of an existing class without modifying it. We have also seen that a private member of a base class cannot be inherited and therefore it is not available for the derived class directly. What do we do if the **private** data needs to be inherited by a derived class? This can be accomplished by modifying the visibility limit of the **private** member by making it **public**. This would make it accessible to all the other functions of the program, thus taking away the advantage of data hiding.

C++ provides a third *visibility modifier*, **protected**, which serve a limited purpose in inheritance. A member declared as **protected** is accessible by the member functions within its class and any class *immediately* derived from it. It *cannot* be accessed by the functions outside these two classes. A class can now use all the three visibility modes as illustrated below:

```
class alpha
{
  private:              // optional
       .....            // visible to member functions
       .....            // within its class
  protected:
       .....            // visible to member functions
       .....            // of its own and derived class
  public:
       .....            // visible to all functions
       .....            // in the program
};
```

When a **protected** member is inherited in **public** mode, it becomes **protected** in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A **protected** member, inherited in the **private** mode derivation, becomes **private** in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance (since **private** members cannot be inherited). Figure 11.4 is the pictorial representation for the two levels of derivation.

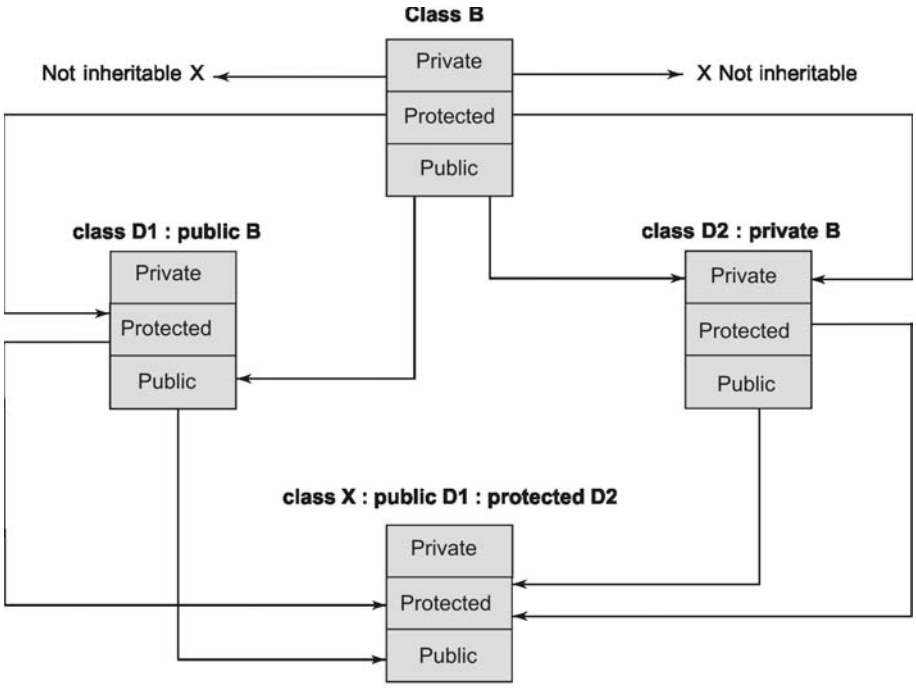


Fig. 11.4 ⇔ *Effect of inheritance on the visibility of members*

The keywords **private**, **protected**, and **public** may appear in any order and any number of times in the declaration of a class. For example,

```
class beta
{
  protected:
      .....
  public:
      .....
  private:
      .....
  public:
      .....
};
```

is a valid class definition.

However, the normal practice is to use them as follows:

```
class beta
{
     .....      // private by default
     .....
```

```
    protected:
        .....
    public:
        .....
}
```

It is also possible to inherit a base class in **protected** mode (known as *protected derivation*). In protected derivation, both the **public** and **protected** members of the base class become **protected** members of the derived class. Table 11.1 summarizes how the visibility of base class members undergoes modifications in all the three types of derivation.

Table 11.1 *Visibility of inherited members*

Base class visibility	*Derived class visibility*		
	Public derivation	***Private derivation***	***Protected derivation***
Private ⟶	Not inherited	Not inherited	Not inherited
Protected ⟶	Protected	Private	Protected
Public ⟶	Public	Private	Protected

Now let us review the access control to the **private** and **protected** members of a class. What are the various functions that can have access to these members? They could be:

1. A function that is a friend of the class.
2. A member function of a class that is a friend of the class.
3. A member function of a derived class.

While the friend functions and the member functions of a friend class can have direct access to both the **private** and **protected** data, the member functions of a derived class can directly access only the **protected** data. However, they can access the **private** data through the member functions of the base class. Figure 11.5 illustrates how the access control mechanism works in various situations. A simplified view of access control to the members of a class is shown in Fig. 11.6.

11.5 Multilevel Inheritance

It is not uncommon that a class is derived from another derived class as shown in Fig. 11.7. The class **A** serves as a base class for the derived class **B**, which in turn serves as a base class for the derived class **C**. The class **B** is known as *intermediate* base class since it provides a link for the inheritance between **A** and **C**. The chain **ABC** is known as *inheritance path*.

A derived class with multilevel inheritance is declared as follows:

```
class A{.....};                 // Base class
class B: public A {.....};      // B derived from A
class C: public B {.....};      // C derived from B
```

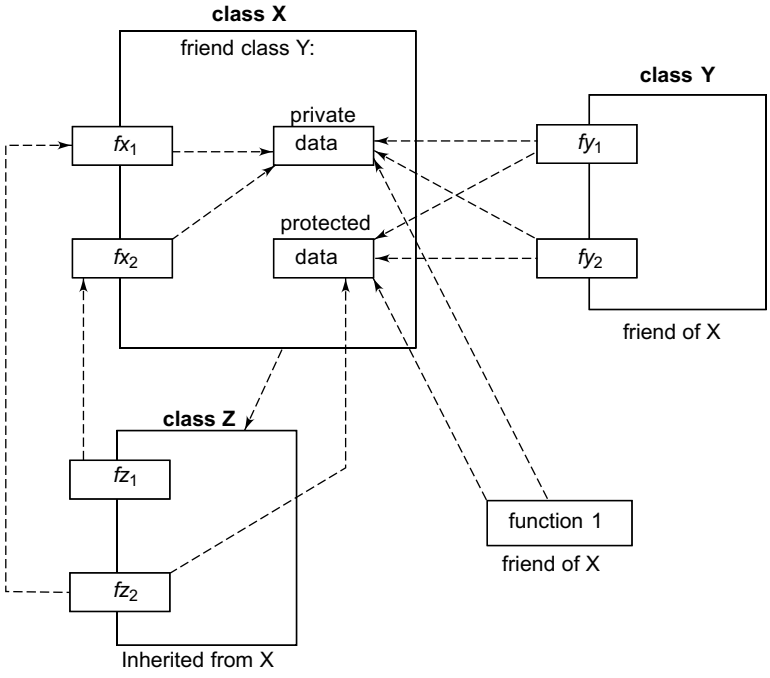


Fig. 11.5 ⇔ *Access mechanism in classes*

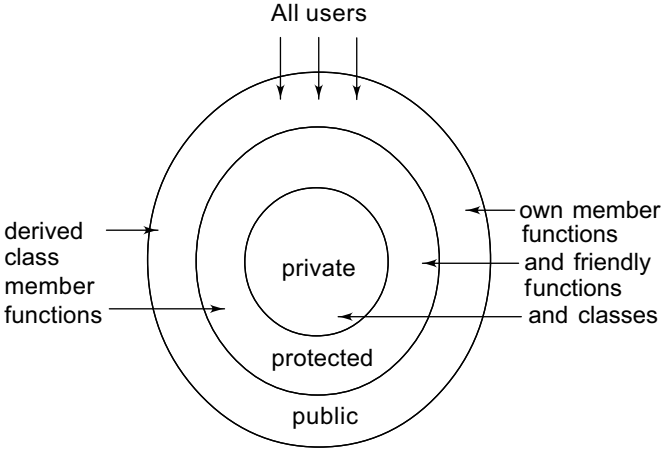


Fig. 11.6 ⇔ *A simple view of access control to the members of a class*

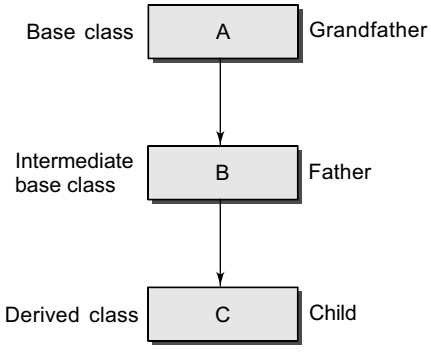


Fig. 11.7 ⇔ *Multilevel in heritance*

This process can be extended to any number of levels.

Let us consider a simple example. Assume that the test results of a batch of students are stored in three different classes. Class **student** stores the roll-number, class **test** stores the marks obtained in two subjects and class **result** contains the **total** marks obtained in the test. The class **result** can inherit the details of the marks obtained in the test and the roll-number of students through multilevel inheritance. Example:

```
class student
{
      protected:
      int roll_number;
   public:
      void get_number(int);
      void put_number(void);
};
void student :: get_number(int a)
{
    roll_number = a;
}
void student :: put_number()
{
       cout << "Roll Number: " << roll_number << "\n";
}

class test : public student    // First level derivation
{
  protected:
     float sub1;
     float sub2;
  public:
     void get_marks(float, float);
     void put_marks(void);
};
void test :: get_marks(float x, float y)
{
     sub1 = x;
     sub2 = y;
}
void test :: put_marks()
{
     cout << "Marks in SUB1 = " << sub1 << "\n";
     cout << "Marks in SUB2 = " << sub2 << "\n";
}
```

```
class result : public test     // Second level derivation
{
     float total;     // private by default
  public:
     void display(void);
};
```

The class **result**, after inheritance from 'grandfather' through 'father', would contain the following members:

```
private:
   float total;          // own member
protected:
   int roll_number;      // inherited from  student via test
   float sub1;           // inherited from test
   float sub2;           // inherited from test
public:
  void get_number(int);        // from student via test
  void put_number(void);       // from student via test
  void get_marks(float, float); // from test
  void put_marks(void);        // from test
  void display(void);          // own member
```

The inherited functions **put_number()** and **put_marks()** can be used in the definition of **display()** function:

```
void result :: display(void)
{
     total = sub1 + sub2;
     put_number();
     put_marks();
     cout << "Total = " << total << "\n";
}
```

Here is a simple **main()** program:

```
int main()
{
     result student1;                  // student1 created
     student1.get_number(111);
     student1.get_marks(75.0, 59.5);
     student1.display();

  return 0;
}
```

This will display the result of **student1**. The complete program is shown in Program 11.3.

MULTILEVEL INHERITANCE

```
#include <iostream>

using namespace std;

class student
{
  protected:
       int roll_number;
  public:
       void get_number(int);
       void put_number(void);
};

void student :: get_number(int a)
{
      roll_number = a;
}

void student :: put_number()
{
      cout << "Roll Number: " << roll_number << "\n";
}

class test : public student              // First level derivation
{
  protected:
     float sub1;
     float sub2;
  public:
        void get_marks(float, float);
        void put_marks(void);
};

void test :: get_marks(float x, float y)
{
     sub1 = x;
     sub2 = y;
}

void test :: put_marks()
{
       cout << "Marks in SUB1 = " << sub1 << "\n";
       cout << "Marks in SUB2 = " << sub2 << "\n";
}
```

```
class result : public test     // Second level derivation
{
      float total;             // private by default
  public:
      void display(void);
};

void result :: display(void)
{
       total = sub1 + sub2;
       put_number();
       put_marks();
       cout << "Total = " << total << "\n";
}

int main()
{
       result student1;        // student1 created

       student1.get_number(111);
       student1.get_marks(75.0, 59.5);

       student1.display();

   return 0;
}
```

PROGRAM 11.3

Program 11.3 displays the following output:

```
Roll Number: 111
Marks in SUB1 = 75
Marks in SUB2 = 59.5
Total = 134.5
```

11.6 Multiple Inheritance

A class can inherit the attributes of two or more classes as shown in Fig. 11.8. This is known as *multiple inheritance*. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligence of another.

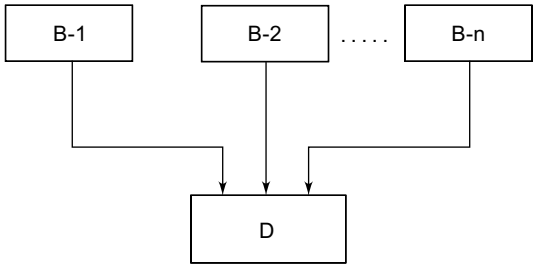


Fig. 11.8 ⇔ *Multiple inheritance*

The syntax of a derived class with multiple base classes is as follows:

```
class D: visibility B-1, visibility B-2 ...
{
      .....
      .....(Body of D)
      .....
};
```

where, *visibility* may be either **public** or **private**. The base classes are separated by commas.

Example:

```
class P : public M, public N
{
  public:
     void display(void);
};
```

Classes M and N have been specified as follows:

```
class M
{
  protected:
       int m;
  public:
       void get_m(int);
};
void M :: get_m(int x)
{
       m = x;
}
class N
{
  protected:
       int n;
  public:
       void get_n(int);
};
void N :: get_n(int y)
  {
       n = y;
  }
```

The derived class **P**, as declared above, would, in effect, contain all the members of **M** and **N** in addition to its own members as shown below:

```
class P
{
  protected:
      int m;                  // from M
      int n;                  // from N
  public:
      void get_m(int);        // from M
      void get_n(int);        // from N
      void display(void);     // own member
};
```

The member function display() can be defined as follows:

```
void P :: display(void)
{
     cout << "m = " << m << "\n";
     cout << "n = " << n << "\n";
     cout << "m*n =" << m*n << "\n";
};
```

The main() function which provides the user-interface may be written as follows:

```
main()
{
     P p;
     p.get_m(10);
     p.get_n(20);
     p.display();
}
```

Program 11.4 shows the entire code illustrating how all the three classes are implemented in multiple inheritance mode.

MULTIPLE INHERITANCE

```
#include <iostream>

using namespace std;

class M
{
  protected:
       int m;
```

```
  public:
       void get_m(int);
};

class N
{
  protected:
       int n;
  public:
       void get_n(int);
};

class P : public M, public N
{
  public:
       void display(void);
};

void M :: get_m(int x)
{
       m = x;
}

void N :: get_n(int y)
{
       n = y;
}

void P :: display(void)
{
       cout << "m = " << m << "\n";
       cout << "n = " << n << "\n";
       cout << "m*n = " << m*n << "\n";
}

int main()
{
       P p;

       p.get_m(10);
       p.get_n(20);
       p.display();

       return 0;
}
```

PROGRAM 11.4

The output of Program 11.4 would be:

```
m = 10
n = 20
m*n = 200
```

11.6.1 Ambiguity Resolution in Inheritance

Occasionally, we may face a problem in using the multiple inheritance, when a function with the same name appears in more than one base class. Consider the following two classes.

```
class M
{
  public:
    void display(void)
    {
        cout << "Class M\n";
    }
};

class N
{
  public:
       void display(void)
       {
            cout << "Class N\n";
       }
};
```

Which **display()** function is used by the derived class when we inherit these two classes? We can solve this problem by defining a named instance within the derived class, using the class resolution operator with the function as shown below:

```
class P : public M, public N
{
  public:
       void display(void)        // overrides display() of M and N
       {
            M :: display();
       }
};
```

We can now use the derived class as follows:

```
int main()
{
       P p;
       p.display();
}
```

Ambiguity may also arise in single inheritance applications. For instance, consider the following situation:

```
class A
{
  public:
    void display()
    {
          cout << "A\n";
    }
};
class B : public A
{
  public:
    void display()
    {
          cout << "B\n";
    }
};
```

In this case, the function in the derived class overrides the inherited function and, therefore, a simple call to **display()** by **B** type object will invoke function defined in **B** only. However, we may invoke the function defined in **A** by using the scope resolution operator to specify the class.

Example:

```
int main()
{
     B b;              // derived class object
     b.display();      // invokes display() in B
     b.A::display();   // invokes display() in A
     b.B::display();   // invokes display() in B

      return 0;
}
```

This will produce the following output:

```
B
A
B
```

11.7 Hierarchical Inheritance

We have discussed so far how inheritance can be used to modify a class when it did not satisfy the requirements of a particular problem on hand. Additional members are added through inheritance to extend the capabilities of a class. Another interesting application of inheritance

is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level.

As an example, Fig. 11.9 shows a hierarchical classification of students in a university. Another example could be the classification of accounts in a commercial bank as shown in Fig. 11.10. All the students have certain things in common and, similarly, all the accounts possess certain common features.

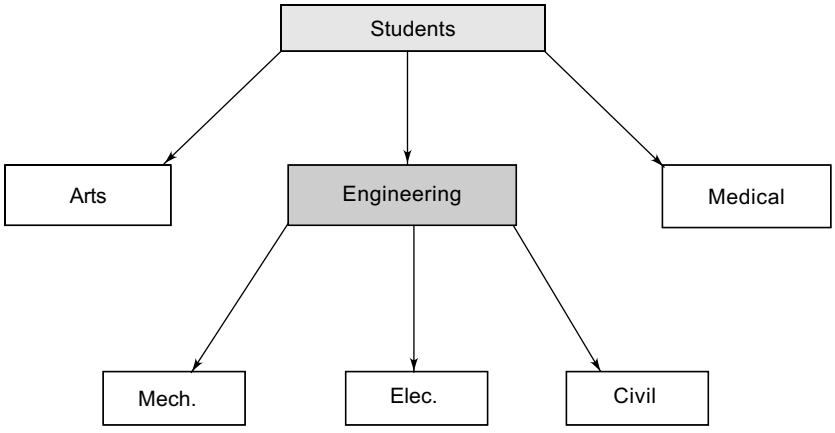


Fig. 11.9 ⇔ *Hierarchical classification of students*

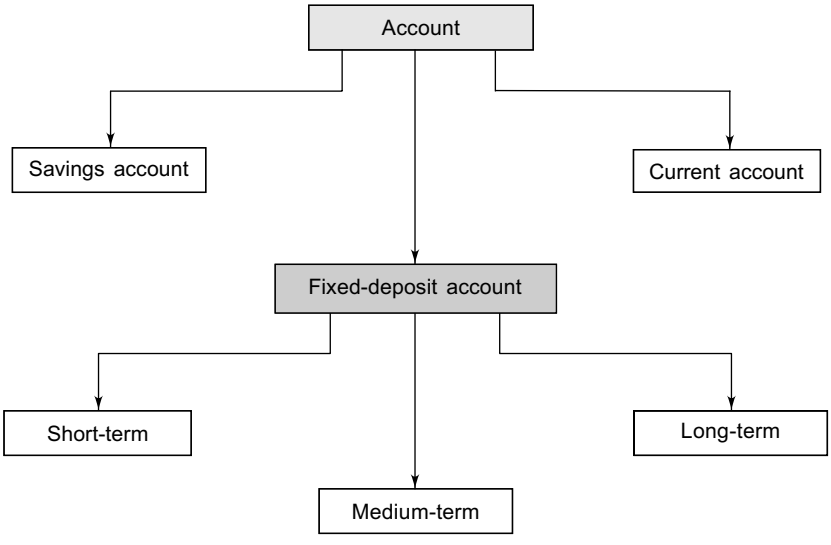


Fig. 11.10 ⇔ *Classification of bank accounts*

In C++, such problems can be easily converted into class hierarchies. The base class will include all the features that are common to the subclasses. A *subclass* can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.

11.8 Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program. For instance, consider the case of processing the student results discussed in Sec. 11.5. Assume that we have to give weightage for sports before finalising the results. The weightage for sports is stored in a separate class called **sports**. The new inheritance relationship between the various classes would be as shown in Fig. 11.11.

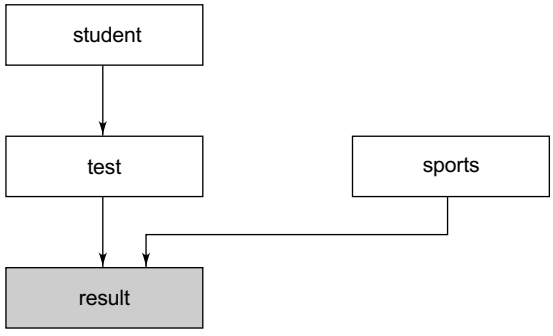


Fig. 11.11 ⇔ *Multilevel, multiple inheritance*

The **sports** class might look like:

```
class sports
{
  protected:
     float  score;
  public:
     void get_score(float);
     void put_score(void);
};
```

The result will have both the multilevel and multiple inheritances and its declaration would be as follows:

```
class result : public test, public sports
{
       .....
       .....
};
```

Where test itself is a derived class from student. That is

```
class test : public student
{
      .....
      .....
};
```

Program 11.5 illustrates the implementation of both multilevel and multiple inheritance.

HYBRID INHERITANCE

```
#include <iostream>

using namespace std;

class student
{
  protected:
    int  roll_number;
  public:
    void get_number(int a)
    {
         roll_number = a;
    }
    void put_number(void)
    {
        cout << "Roll No: " << roll_number << "\n";
    }
};

class test : public student
{
  protected:
    float part1, part2;
  public:
    void get_marks(float x, float y)
    {
        part1 = x;  part2 = y;
    }
    void put_marks(void)
    {
        cout << "Marks obtained: " << "\n"
             << "Part1 = " << part1 << "\n"
             << "Part2 = " << part2 << "\n";
    }
};

class sports
{
   protected:
       float score;
   public:
       void get_score(float s)
       {
```

```
            score = s;
        }
        void put_score(void)
        {
        cout << "Sports wt: " << score << "\n\n";
        }
};

class result : public test, public sports
{
        float total;
  public:
        void display(void);
};

void result :: display(void)
{
      total = part1 + part2 + score;

      put_number();
      put_marks();
      put_score();

      cout << "Total Score: " << total << "\n";
}

int main()
{
      result student_1;
      student_1.get_number(1234);
      student_1.get_marks(27.5, 33.0);
      student_1.get_score(6.0);
      student_1.display();

      return 0;
}
```

PROGRAM 11.5

Here is the output of Program 11.5:

```
Roll No: 1234
Marks obtained:
Part1 = 27.5
Part2 = 33
Sports wt: 6

Total Score: 66.5
```

11.9 Virtual Base Classes

We have just discussed a situation which would require the use of both the multiple and multilevel inheritance. Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. This is illustrated in Fig. 11.12. The 'child' has two *direct base classes* 'parent1' and 'parent2' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as *indirect base class*.

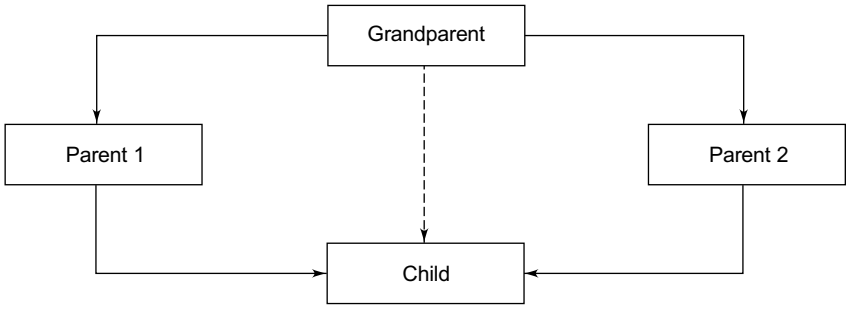


Fig. 11.12 ⇔ *Multipath inheritance*

Inheritance by the 'child' as shown in Fig. 11.12 might pose some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. This means, 'child' would have *duplicate* sets of the members inherited from 'grandparent'. This introduces ambiguity and should be avoided.

The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as *virtual base class* while declaring the direct or intermediate base classes as shown below:

```
class A                          // grandparent
{
        .....
        .....
};
class B1 : virtual public A      // parent1
{
        .....
        .....
};
class B2 : public virtual A      // parent2
{
        .....
        .....
};
```

```
class C : public B1, public B2    // child
{
        .....              // only one copy of A
        .....              // will be inherited
};
```

When a class is made a **virtual** base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.

note

The keywords **virtual** and **public** may be used in either order.

For example, consider again the student results processing system discussed in Sec. 11.8. Assume that the class **sports** derives the **roll_number** from the class **student**. Then, the inheritance relationship will be as shown in Fig. 11.13.

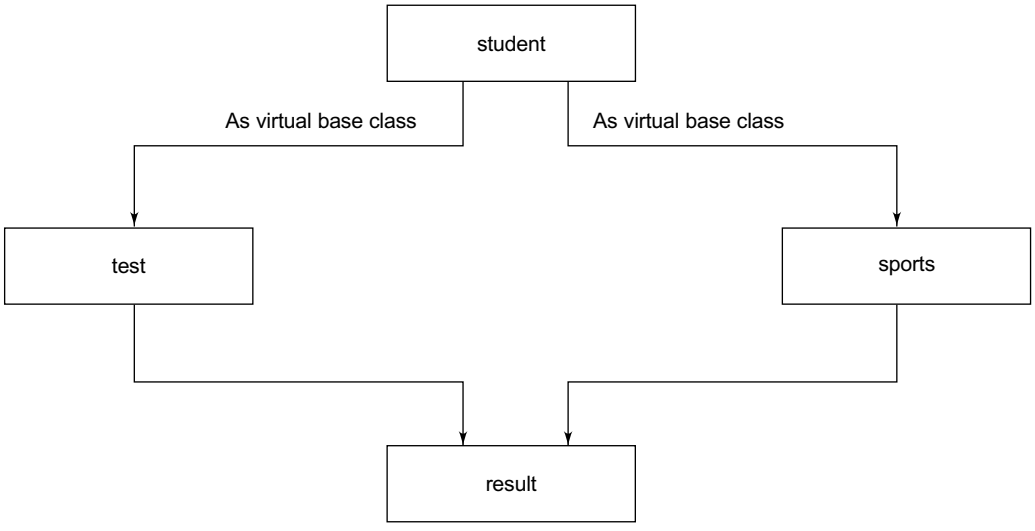


Fig. 11.13 ⇔ *Virtual base class*

A program to implement the concept of virtual base class is illustrated in Program 11.6.

VIRTUAL BASE CLASS

```
#include <iostream>

using namespace std;
```

```
class student
{
  protected:
    int roll_number;
  public:
    void get_number(int a)
    {
         roll_number = a;
  }
  void put_number(void)
  {
         cout << "Roll No: " << roll_number << "\n";
  }
};

class test : virtual public student
{
  protected:
    float part1, part2;
  public:
    void get_marks(float x, float y)
    {
         part1 = x;  part2 = y;
    }
    void put_marks(void)
    {
         cout << "Marks obtained: " << "\n"
              << "Part1 = " << part1 << "\n"
              << "Part2 = " << part2 << "\n";
    }
};

class sports : public virtual student
{
  protected:
    float score;
  public:
    void get_score(float s)
  {
         score = s;
  }
  void put_score(void)
  {
         cout << "Sports wt: " << score << "\n\n";
  }
```

```
};

class result : public test, public sports
{
    float total;
  public:
         void display(void);
};

void result :: display(void)
{
         total = part1 + part2 + score;

         put_number();
         put_marks();
         put_score();

         cout << "Total Score: " << total << "\n";
}

int main()
{
         result student_1;
         student_1.get_number(678);
         student_1.get_marks(30.5, 25.5);
         student_1.get_score(7.0);
         student_1.display();

         return 0;
}
```

PROGRAM 11.6

The output of Program 11.6 would be

```
Roll No: 678
Marks obtained:
Part1 = 30.5
Part2 = 25.5
Sport wt: 7

Total Score: 63
```

11.10 Abstract Classes

An *abstract class* is one that is not used to create objects. An abstract class is designed only to act as a base class (to be inherited by other classes). It is a design concept in program

development and provides a base upon which other classes may be built. In the previous example, the **student** class is an abstract class since it was not used to create any objects.

11.11 Constructors in Derived Classes

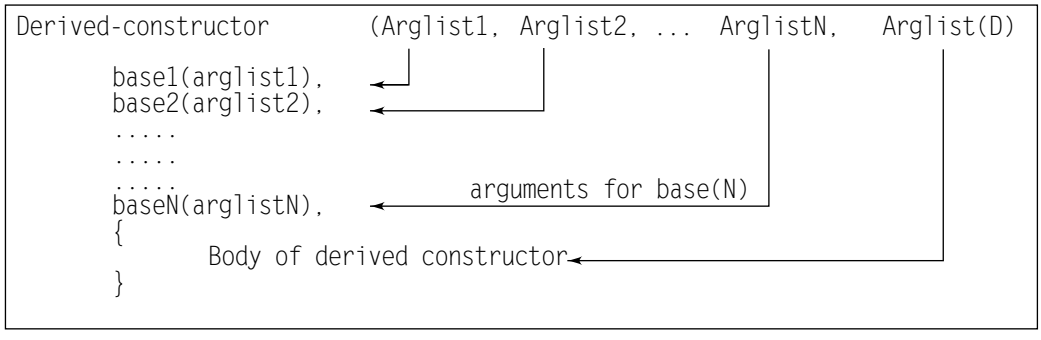
As we know, the constructors play an important role in initializing objects. We did not use them earlier in the derived classes for the sake of simplicity. One important thing to note here is that, as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is *mandatory* for the derived class to have a constructor and pass the arguments to the base class constructors. Remember, while applying inheritance we usually create objects using the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base classes are constructed *in the order in which they appear in the declaration of the derived class*. Similarly, in a multilevel inheritance, the constructors will be executed *in the order of inheritance*.

Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared. How are they passed to the base class constructors so that they can do their job? C++ supports a special argument passing mechanism for such situations.

The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor.

The general form of defining a derived constructor is:



The header line of *derived-constructor* function contains two parts separated by a colon(:). The first part provides the declaration of the arguments that are passed to the *derived-constructor* and the second part lists the function calls to the base constructors.

base1(arglist1), base2(arglist2) ... are function calls to base constructors **base1(), base2(),** ... and therefore *arglist1, arglist2,* ... etc. represent the actual parameters that are passed

to the base constructors. *Arglist1* through *ArglistN* are the argument declarations for base constructors *base1* through *baseN*. *ArglistD* provides the parameters that are necessary to initialize the members of the derived class.

Example:

```
D(int a1, int a2, float b1, float b2, int d1):
A(a1, a2),      /* call to constructor A */
B(b1, b2)      /* call to constructor B */
{
     d = d1;  // executes its own body
}
```

A(a1, a2) invokes the base constructor **A()** and B(b1, b2) invokes another base constructor **B()**. The constructor **D()** supplies the values for these four arguments. In addition, it has one argument of its own. The constructor **D()** has a total of five arguments. **D()** may be invoked as follows:

```
.....
D objD(5, 12, 2.5, 7.54, 30);
.....
```

These values are assigned to various parameters by the constructor **D()** as follows:

5 ⟶ a1
12 ⟶ a2
2.5 ⟶ b1
7.54 ⟶ b2
30 ⟶ d1

The constructors for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared. Any non-virtual bases are then constructed before the derived class constructor is executed. See Table 11.2.

Table 11.2 *Execution of base class constructors*

Method of inheritance	***Order of execution***

Class B: public A
{
};
	A( ) ; base constructor
	B( ) ; derived constructor

class A : public B, public C
{
};
	B( ) ; base(first)
	C( ) ; base(second)
	A( ) ; derived

class A : public B, virtual public C
{
};
	C( ) ; virtual base
	B( ) ; ordinary base
	A( ) ; derived

Program 11.7 illustrates how constructors are implemented when the classes are inherited.

CONSTRUCTORS IN DERIVED CLASS

```
#include <iostream>

using namespace std;

class alpha
{
    int x;
  public:
    alpha(int i)
    {
         x = i;
         cout << "alpha initialized \n";
    }
    void show_x(void)
    { cout << "x = " << x << "\n"; }
};

class beta
{
    float y;
  public:
    beta(float j)
    {
         y = j;
         cout << "beta initialized \n";
    }
    void show_y(void)
    { cout << "y = " << y << "\n"; }
};

class gamma: public beta, public alpha
{
    int m, n;
  public:
    gamma(int a, float b, int c, int d):
         alpha(a), beta(b)
    {
         m = c;
         n = d;
         cout << "gamma initialized \n";
    }
    void show_mn(void)
```

```
    {
        cout << "m = " << m << "\n"
             << "n = " << n << "\n";
    }
};

int main()
{
    gamma g(5, 10.75, 20, 30);
    cout << "\n";
    g.show_x();
    g.show_y();
    g.show_mn();

    return 0;
}
```

PROGRAM 11.7

The output of Program 11.7 would be:

```
beta initialized
alpha initialized
gamma initialized

x = 5
y = 10.75
m = 20
n = 30
```

note

beta is initialized first, although it appears second in the derived constructor. This is because it has been declared first in the derived class header line. Also, note that **alpha(a)** and bet**a(b)** are function calls. Therefore, the parameters should not include types.

C++ supports another method of initializing the class objects. This method uses what is known as initialization list in the constructor function. This takes the following form:

```
constructor (arglist) : intialization-section
{
      assignment-section
}
```

The *assignment-section* is nothing but the body of the constructor function and is used to assign initial values to its data members. The part immediately following the colon is known as the *initialization section*. We can use this section to provide initial values to the base constructors and also to initialize its own class members. This means that we can use

either of the sections to initialize the data members of the constructors class. The initialization section basically contains a list of initializations separated by commas. This list is known as *initialization list*. Consider a simple example:

```
class XYZ
{
     int a;
     int b;
  public:
     XYZ(int i, int j) : a(i), b(2 * j) { }
};

main()
{
     XYZ x(2, 3);
}
```

This program will initialize **a** to 2 and **b** to 6. Note how the data members are initialized, just by using the variable name followed by the initialization value enclosed in the parenthesis (like a function call). Any of the parameters of the argument list may be used as the initialization value and the items in the list may be in any order. For example, the constructor **XYZ** may also be written as:

```
XYZ(int i, int j) : b(i), a(i + j) { }
```

In this case, **a** will be initialized to 5 and **b** to 2. Remember, the data members are initialized in the order of declaration, independent of the order in the initialization list. This enables us to have statements such as

```
XYZ(int i, int j) : a(i), b(a * j) { }
```

Here **a** is initialized to 2 and **b** to 6. Remember, **a** which has been declared first is initialized first and then its value is used to initialize **b**. However, the following will not work:

```
XYZ(int i, int j) : b(i), a(b * j) { }
```

because the value of **b** is not available to **a** which is to be initialized first.

The following statements are also valid:

```
XYZ(int i, int j) : a(i) {b = j;}
XYZ(int i, int j) { a = i; b = j;}
```

We can omit either section, if it is not needed. Program 11.8 illustrates the use of initialization lists in the base and derived constructors.

INITIALIZATION LIST IN CONSTRUCTORS

```
#include <iostream>

using namespace std;
```

```
class alpha
{
       int x;
  public:
       alpha(int i)
       {
            x = i;
            cout << "\n alpha constructed";
       }

       void show_alpha(void)
       {
            cout << " x = " << x << "\n";
       }
};

class beta
{
       float p, q;
  public:
       beta(float a, float b): p(a), q(b+p)
       {
            cout << "\n beta constructed";
       }
       void show_beta(void)
       {
            cout << " p = " << p << "\n";
            cout << " q = " << q << "\n";
       }
};
class gamma : public beta, public alpha
{
       int u,v;
  public:
            gamma(int a, int b, float c):
            alpha(a*2), beta(c,c), u(a)
            { v = b; cout << "\n gamma constructed"; }

            void show_gamma(void)
            {
            cout << " u = " << u << "\n";
            cout << " v = " << v << "\n";
            }
```

```
};

int main()

{
        gamma g(2, 4, 2.5);

        cout << "\n\n Display member values " << "\n\n";

        g.show_alpha();
        g.show_beta();
        g.show_gamma();

        return 0;
};
```

PROGRAM 11.8

The output of Program 11.8 would be:

```
beta constructed
alpha constructed
gamma constructed

Display member values

x = 4
p = 2.5
q = 5
u = 2
v = 4
```

note

The argument list of the derived constructor **gamma** contains only three parameters **a**, **b** and **c** which are used to initialize the five data members contained in all the three classes.

11.12 Member Classes: Nesting of Classes

Inheritance is the mechanism of deriving certain properties of one class into another. We have seen in detail how this is implemented using the concept of derived classes. C++ supports yet another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is, a class can contain objects of other classes as its members as shown below:

```
class alpha {....};
class beta {....};
class gamma
{
     alpha a;          // a is an object of alpha class
     beta b;           // b is an object of beta class
     .....
};
```

All objects of **gamma** class will contain the objects **a** and **b**. This kind of relationship is called *containership* or *nesting*. Creation of an object that contains another object is very different than the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages. First, the member objects are created using their respective constructors and then the other 'ordinary' members are created. This means, constructors of all the member objects should be called before its own constructor body is executed. This is accomplished using an initialization list in the constructor of the nested class.

Example:

```
class gamma
{
       .....
       alpha a;     // a is object of alpha
       beta b;      // b is object of beta
  public:
       gamma(arglist): a(arglist1), b(arglist2)
       {
                // constructor body
       }
};
```

arglist is the list of arguments that is to be supplied when a **gamma** object is defined. These parameters are used for initializing the members of **gamma**. *arglist1* is the argument list for the constructor of **a** and *arglist2* is the argument list for the constructor of **b**. *arglist1* and *arglist2* may or may not use the arguments from *arglist*. Remember, **a**(*arglist1*) and **b**(*arglist2*) are function calls and therefore the arguments do not contain the data types. They are simply variables or constants.

Example:

```
gamma(int x, int y, float z) : a(x), b(x,z)
{
       Assignment section(for ordinary other members)
}
```

We can use as many member objects as are required in a class. For each member object we add a constructor call in the initializer list. The constructors of the member objects are called in the order in which they are declared in the nested class.

SUMMARY

- The mechanism of deriving a new class from an old class is called inheritance. Inheritance provides the concept of reusability. The C++ classes can be reused using inheritance.
- The derived class inherits some or all of the properties of the base class.
- A derived class with only one base class is called single inheritance.
- A class can inherit properties from more than one class which is known as multiple inheritance.
- A class can be derived from another derived class which is known as multilevel inheritance.
- When the properties of one class are inherited by more than one class, it is called hierarchical inheritance.
- A private member of a class cannot be inherited either in public mode or in private mode.
- A protected member inherited in public mode becomes protected, whereas inherited in private mode becomes private in the derived class.
- A public member inherited in public mode becomes public, whereas inherited in private mode becomes private in the derived class.
- The friend functions and the member functions of a friend class can directly access the private and protected data.
- The member functions of a derived class can directly access only the protected and public data. However, they can access the private data through the member functions of the base class.
- Multipath inheritance may lead to duplication of inherited members from a 'grandparent' base class. This may be avoided by making the common base class a virtual base class.
- In multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.
- In multilevel inheritance, the constructors are executed in the order of inheritance.
- A class can contain objects of other classes. This is known as containership or nesting.

Key Terms

- abstract class
- access control
- access mechanism
- ancestor class
- assignment section
- base class
- base constructor
- child class

- common base class
- containership
- derivation
- derived class
- derived constructor
- direct base class
- dot operator
- duplicate members
- father class
- **friend**
- grandfather class
- grandparent class
- hierarchical inheritance
- hybrid inheritance
- indirect base class
- inheritance
- inheritance path
- initialization list
- initialization section
- intermediate base
- member classes
- multilevel inheritance
- multiple inheritance
- nesting
- **private**
- private derivation
- private members
- privately derived
- **protected**
- protected members
- **public**
- public derivation
- public members
- publicly derived
- reusability
- single inheritance
- subclass
- virtual base class
- visibility mode
- visibility modifier

Review Questions

11.1 *What does inheritance mean in C++?*

11.2 *What are the different forms of inheritance? Give an example for each.*

11.3 *Describe the syntax of the single inheritance in C++.*

11.4 *We know that a private member of a base class is not inheritable. Is it anyway possible for the objects of a derived class to access the private members of the base class? If yes, how? Remember, the base class cannot be modified.*

11.5 *How do the properties of the following two derived classes differ?*

(a) *class D1: private B{//...};*

(b) *class D2: public B{//...};*

11.6 *When do we use the protected visibility specifier to a class member?*

11.7 *Describe the syntax of multiple inheritance. When do we use such an inheritance?*

11.8 *What are the implications of the following two definitions?*

(a) *class A: public B, public C{//....};*

(b) *class A: public C, public B{//....};*

11.9 *What is a virtual base class?*

11.10 *When do we make a class virtual?*

11.11 *What is an abstract class?*

11.12 *In what order are the class constructors called when a derived class object is created?*

11.13 *Class D is derived from class B. The class D does not contain any data members of its own. Does the class D require constructors? If yes, why?*

11.14 *What is containership? How does it differ from inheritance?*

11.15 *Describe how an object of a class that contains objects of other classes created?*

11.16 *State whether the following statements are TRUE or FALSE:*

(a) *Inheritance helps in making a general class into a more specific class.*

(b) *Inheritance aids data hiding.*

(c) *One of the advantages of inheritance is that it provides a conceptual framework.*

(d) *Inheritance facilitates the creation of class libraries.*

(e) *Defining a derived class requires some changes in the base class.*

(f) *A base class is never used to create objects.*

(g) *It is legal to have an object of one class as a member of another class.*

(h) *We can prevent the inheritance of all members of the base class by making base class virtual in the definition of the derived class.*

Debugging Exercises

11.1 Identify the error in the following program.

```
#include <iostream.h>
class Student {
        char* name;
        int rollNumber;
private:
        Student() {
                name = "AlanKay";
                rollNumber = 1025;
        }
        void setNumber(int no) {
                rollNumber = no;
        }
        int getRollNumber() {
                return rollNumber;
        }
};

class AnualTest: Student {
```

```
        int mark1, mark2;
public:
        AnualTest(int m1, int m2)
                :mark1(m1), mark2(m2) {
        }
        int getRollNumber() {
                return Student::getRollNumber();
        }
};

void main()
{
        AnualTest test1(92, 85);
        cout << test1.getRollNumber();
}
```

11.2 Identify the error in the following program.

```
#include <iostream.h>
class A
{
public:
        A()
        {
                cout << "A";
        }
};
class B: public A
{
public:
        B()
        {
                cout << "B";
        }
};
class C: public B
{
public:
        C()
        {
```

```
                cout << "C";
        }
};
class D
{
public:
        D()
        {
                cout << "D";
        }
};
class E: public C, public D
{
public:
        E()
        {
                cout << "D";
        }
};
class F: B, virtual E
{
public:
        F()
        {
                cout << "F";
        }
};
void main()
{
        F f;
}
```

11.3 Identify the error in the following program.

```
#include <iostream.h>
class A
{
        int i;
};

class AB: virtual A
{
```

```
        int j;
};
class AC: A, ABAC
{
        int k;
};
class ABAC: AB, AC
{
        int l;
};
void main()
{
        ABAC abac;
        cout << "sizeof ABAC:" << sizeof(abac);
}
```

11.4 Find errors in the following program. State reasons.

```
// Program test
#include <iostream.h>

class X
{
          private:
             int x1;
          protected:
             int x2;
          public:
             int x3;
};

class Y: public X
{
          public:
             void f()
             {
                    int y1,y2,y3;
                    y1 = x1;
                    y2 = x2;
                    y3 = x3;
             }
};

class Z: X
```

```
{
        public:
           void f()
           {
                 int z1,z2,z3;
                 z1 = x1;
                 z2 = x2;
                 z3 = x3;
           }
};
main()
{
              int m,n,p;
              Y y;
              m = y.x1;
              n = y.x2;
              p = y.x3;
              Z z;
              m = z.x1;
              n = z.x2;
              p = z.x3;
}
```

11.5 Debug the following program.

```
// Test program
#include <iostream.h>

class B1
{
       int b1;
  public:
       void display();
       {
                cout << b1 <<"\n";
       }
};

class B2
{
       int b2;
  public:
       void display();
```

```
        {
                cout << b2 <<"\n";
        }
};
class D: public B1, public B2
{
        // nothing here
};
main()
{
        D d;
        d.display()
        d.B1::display();
        d.B2::display();
}
```

Programming Exercises

11.1 *Assume that a bank maintains two kinds of accounts for customers, one called as savings account and the other as current account. The savings account provides compound interest and withdrawal facilities but no cheque book facility. The current account provides cheque book facility but no interest. Current account holders should also maintain a minimum balance and if the balance falls below this level, a service charge is imposed.*

Create a class ***account*** *that stores customer name, account number and type of account. From this derive the classes* ***cur_acct*** *and* ***sav_acct*** *to make them more specific to their requirements. Include necessary member functions in order to achieve the following tasks:*

(a) *Accept deposit from a customer and update the balance.*
(b) *Display the balance.*
(c) *Compute and deposit interest.*
(d) *Permit withdrawal and update the balance.*
(e) *Check for the minimum balance, impose penalty, necessary, and update the balance.*

Do not use any constructors. Use member functions to initialize the class members.

11.2 *Modify the program of Exercise 11.1 to include constructors for all the three classes.*

11.3 *An educational institution wishes to maintain a database of its employees. The database is divided into a number of classes whose hierarchical relationships are shown in Fig. 11.14. The figure also shows the minimum information required for each class. Specify all the classes and define functions to create the database and retrieve individual information as and when required.*

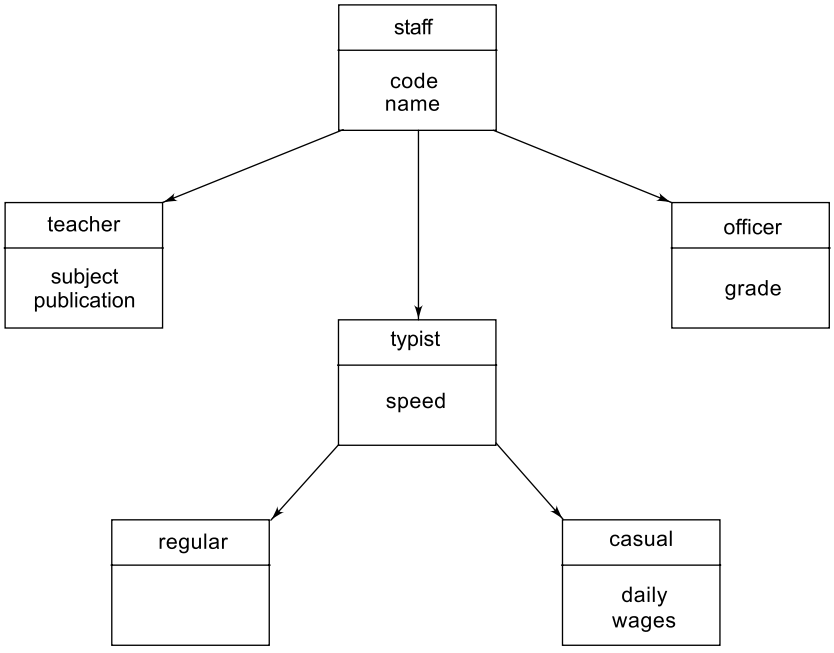


Fig. 11.14 ⇔ *Class relationships (for Exercise 11.19)*

11.4 *The database created in Exercise 11.3 does not include educational information of the staff. It has been decided to add this information to teachers and officers (and not for typists) which will help the management in decision making with regard to training, promotion, etc. Add another data class called* ***education*** *that holds two pieces of educational information, namely, highest qualification in general education and highest professional qualification. This class should be inherited by the classes* ***teacher*** *and* ***officer****. Modify the program of Exercise 11.19 to incorporate these additions.*

11.5 *Consider a class network of Fig. 11.15. The class* ***master*** *derives information from both account and admin classes which in turn derive information from the class* ***person****. Define all the four classes and write a program to create, update and display the information contained in* ***master*** *objects.*

11.6 *In Exercise 11.3, the classes* ***teacher****,* ***officer****, and* ***typist*** *are derived from the class* ***staff****. As we know, we can use container classes in place of inheritance in some situations. Redesign the program of Exercise 11.3 such that the classes* ***teacher****,* ***officer****, and* ***typist*** *contain the objects of* ***staff****.*

11.7 *We have learned that OOP is well suited for designing simulation programs. Using the techniques and tricks learned so far, design a program that would simulate a simple real-world system familiar to you.*

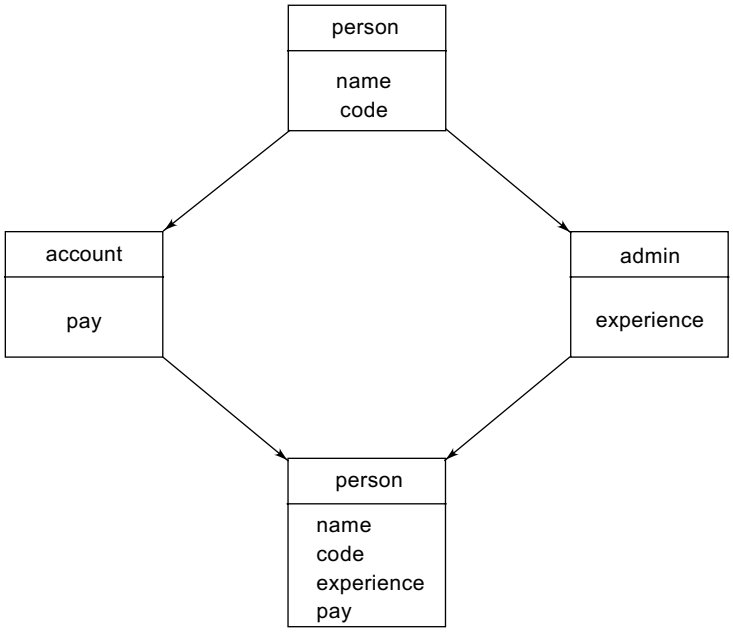


Fig. 11.15 ⇔ *Multipath inheritance (for Exercise 11.21)*

12

Managing Console I/O Operations

Key Concepts

- Streams
- Stream classes
- Unformatted output
- Character-oriented input/output
- Line-oriented input/output
- Formatted output
- Formatting functions
- Formatting flags
- Manipulators
- User-defined manipulators

12.1 Introduction

Every program takes some data as input and generates processed data as output following the familiar input-process-output cycle. It is, therefore, essential to know how to provide the input data and how to present the results in a desired form. We have, in the earlier chapters, used **cin** and **cout** with the operators >> and << for the input and output operations. But we have not so far discussed as to how to control the way the output is printed. C++ supports a rich set of I/O functions and operations to do this. Since these functions use the advanced features of C++ (such as classes, derived classes and virtual functions), we need to know a lot about them before really implementing the C++ I/O operations.

Remember, C++ supports all of C's rich set of I/O functions. We can use any of them in the C++ programs. But we restrained from using them due to two reasons. First, I/O methods in C++ support the concepts of OOP and secondly, I/O methods in C cannot handle the user-defined data types such as class objects.

C++ uses the concept of *stream* and *stream classes* to implement its I/O operations with the console and disk files. We will discuss in this chapter, how stream classes support the console-oriented input-output operations.

12.2 C++ Streams

The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives. Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as *stream*.

A stream is a sequence of bytes. It acts either as a *source* from which the input data can be obtained or as a *destination* to which the output data can be sent. The source stream that provides data to the program is called the *input stream* and the destination stream that receives output from the program is called the *output stream*. In other words, a program *extracts* the bytes from an input stream and *inserts* bytes into an output stream as illustrated in Fig. 12.1.

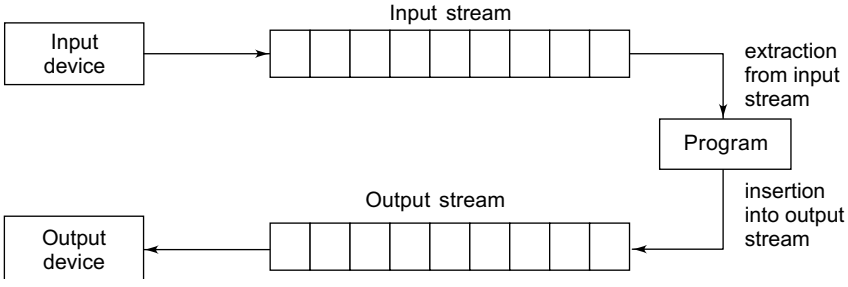


Fig. 12.1 ⇔ *Tata streams*

The data in the input stream can come from the keyboard or any other storage device. Similarly, the data in the output stream can go to the screen or any other storage device. As mentioned earlier, a stream acts as an interface between the program and the input/output device. Therefore, a C++ program handles data (input or output) independent of the devices used.

C++ contains several pre-defined streams that are automatically opened when a program begins its execution. These include cin and cout which have been used very often in our earlier programs. We know that cin represents the input stream connected to the standard input device (usually the keyboard) and cout represents the output stream connected to the standard output device (usually the screen). Note that the keyboard and the screen are default options. We can redirect streams to other devices or files, if necessary.

12.3 C++ Stream Classes

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called *stream classes*. Figure 12.2 shows the hierarchy of the stream classes used for input and output operations with the console unit. These classes are declared in the header file *iostream*. This file should be included in all the programs that communicate with the console unit.

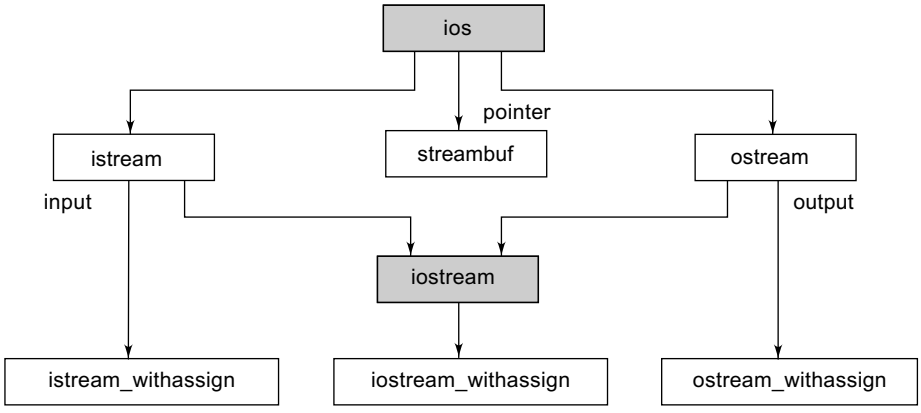


Fig. 12.2 ⇔ *Stream classes for console I/O operations*

As seen in the Fig. 12.2, **ios** is the base class for **istream** (input stream) and **ostream** (output stream) which are, in turn, base classes for **iostream** (input/output stream). The class **ios** is declared as the virtual base class so that only one copy of its members are inherited by the **iostream**.

The class **ios** provides the basic support for formatted and unformatted I/O operations. The class **istream** provides the facilities for formatted and unformatted input while the class **ostream** (through inheritance) provides the facilities for formatted output. The class **iostream** provides the facilities for handling both input and output streams. Three classes, namely, **istream_withassign, ostream_withassign**, and **iostream_withassign** add assignment operators to these classes. Table 12.1 gives the details of these classes.

12.4 Unformatted I/O Operations

12.4.1 Overloaded Operators >> and <<

We have used the objects **cin** and **cout** (pre-defined in the *iostream* file) for the input and output of data of various types. This has been made possible by overloading the operators >> and << to recognize all the basic C++ types. The >> operator is overloaded in the

Table 12.1 *Stream classes for console operations*

Class name	*Contents*
ios (General input/output stream class)	✹ Contains basic facilities that are used by all other input and output classes ✹ Also contains a pointer to a buffer object **(streambuf** object) ✹ Declares constants and functions that are necessary for handling formatted input and output operations

(*Contd.*)

Table 12.1 *(Contd.)*

Class name	*Contents*
istream (input stream)	✹ Inherits the properties of **ios** ✹ Declares input functions such as **get(), getline()** and **read()** ✹ Contains overloaded extraction operator >>
ostream (output stream)	✹ Inherits the properties of ios ✹ Declares output functions **put()** and **write()** ✹ Contains overloaded insertion operator <<
iostream (input/output stream)	✹ Inherits the properties of **ios istream** and **ostream** through multiple inheritance and thus contains all the input and output functions
streambuf	✹ Provides an interface to physical devices through buffers ✹ Acts as a base for **filebuf** class used ios files

istream class and << is overloaded in the **ostream** class. The following is the general format for reading data from the keyboard:

```
cin >> variable1 >> variable2 >> .... >> variableN
```

variable1, variable2, ... are valid C++ variable names that have been declared already. This statement will cause the computer to stop the execution and look for input data from the keyboard. The input data for this statement would be:

```
data1 data2 ..... dataN
```

The input data are separated by white spaces and should match the type of variable in the **cin** list. Spaces, newlines and tabs will be skipped.

The operator >> reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type. For example, consider the following code:

```
int code;
cin >> code;
```

Suppose the following data is given as input:

```
4258D
```

The operator will read the characters upto 8 and the value 4258 is assigned to **code.** The character D remains in the input stream and will be input to the next **cin** statement. The general form for displaying data on the screen is:

```
cout <<item1 <<item2 <<....<<itemN
```

The items *item1* through *itemN* may be variables or constants of any basic type. We have used such statements in a number of examples illustrated in previous chapters.

12.4.2 put() and get() Functions

The classes **istream** and **ostream** define two member functions **get**() and **put**() respectively to handle the single character input/output operations. There are two types of **get**() functions. We can use both **get(char *)** and **get(void)** prototypes to fetch a character including the blank space, tab and the newline character. The **get(char *)** version assigns the input character to its argument and the **get(void)** version returns the input character.

Since these functions are members of the input/output stream classes, we must invoke them using an appropriate object.

Example:

```
char c;
cin.get(c);              // get a character from keyboard
                         // and assign it to c
while(c != '\n')
{
       cout << c;        // display the character on screen
       cin.get(c);       // get another character
}
```

This code reads and displays a line of text (terminated by a newline character). Remember, the operator >> can also be used to read a character but it will skip the white spaces and newline character. The above **while** loop will not work properly if the statement

```
cin >> c;
```

is used in place of

```
cin.get(c);
```

note

Try using both of them and compare the results.

The **get(void)** version is used as follows:

```
.....
char c;
c = cin.get();   // cin.get(c); replaced
.....
.....
```

The value returned by the function **get**() is assigned to the variable **c**.

The function **put**(), a member of **ostream** class, can be used to output a line of text, character by character. For example,

```
cout.put('x');
```

displays the character **x** and

```
cout.put(ch);
```

displays the value of variable **ch.**

The variable **ch** must contain a character value. We can also use a number as an argument to the function **put**(). For example,

```
cout.put(68);
```

displays the character D. This statement will convert the **int** value 68 to a **char** value and display the character whose ASCII value is 68.

The following segment of a program reads a line of text from the keyboard and displays it on the screen.

```
char c;
cin.get(c);           // read a character

while(c != '\n')
{
     cout.put(c);    // display the character on screen
     cin.get(c);
}
```

Program 12.1 illustrates the use of these two character handling functions.

CHARACTER I/O WITH GET() AND PUT()

```
#include <iostream>

using namespace std;

int main()
{
     int count = 0;
     char c;

     cout << "INPUT TEXT\n";

     cin.get(c);

     while(c != '\n')
     {
            cout.put(c);
            count++;
            cin.get(c);
     }
     cout << "\nNumber of characters = " << count << "\n";

     return 0;
}
```

PROGRAM 12.1

```
Input
    Object Oriented Programming
Output
    Object Oriented Programming
           Number of characters = 27
```

note

When we type a line of input, the text is sent to the program as soon as we press the RETURN key. The program then reads one character at a time using the statement **cin. get(c);** and displays it using the statement **cout.put(c);.** The process is terminated when the newline character is encountered.

12.4.3 getline() and write() Functions

We can read and display a line of text more efficiently using the line-oriented input/output functions **getline**() and **write**(). The **getline**() function reads a whole line of text that ends with a newline character (transmitted by the RETURN key). This function can be invoked by using the object **cin** as follows:

```
cin.getline (line, size);
```

This function call invokes the function **getline**() which reads character input into the variable line. The reading is terminated as soon as either the newline character '\n' is encountered or size-1 characters are read (whichever occurs first). The newline character is read but not saved. Instead, it is replaced by the null character. For example, consider the following code:

```
char  name[20];
cin.getline(name, 20);
```

Assume that we have given the following input through the keyboard:

```
Bjarne Stroustrup <press RETURN>
```

This input will be read correctly and assigned to the character array **name**. Let us suppose the input is as follows:

```
Object Oriented Programming  <press RETURN >
```

In this case, the input will be terminated after reading the following 19 characters:

```
Object Oriented Pro
```

Remember, the two blank spaces contained in the string are also taken into account.

We can also read strings using the operator >> as follows:

```
cin >> name;
```

But remember **cin** can read strings that do not contain white spaces. This means that **cin** can read just one word and not a series of words such as "Bjarne Stroustrup". But it can read the following string correctly:

```
Bjarne_Stroustrup
```

After reading the string, **cin** automatically adds the terminating null character to the character array.

Program 12.2 demonstrates the use of >> and **getline**() for reading the strings.

READING STRINGS WITH GETLINE()

```
#include <iostream>

using namespace std;

int main()
{
     int size = 20;
     char city[20];

     cout << "Enter city name: \n";
     cin  >> city;
     cout << "City name:" << city << "\n\n";

     cout << "Enter city name again: \n";
     cin.getline(city, size);
     cout << "City name now: " << city << "\n\n";

     cout << "Enter another city name: \n";
     cin.getline(city, size);
     cout << "New city name: " << city << "\n\n";

     return 0;
}
```

PROGRAM 12.2

The output of Program 12.2 would be:

First run

```
Enter city name:
Delhi
City name: Delhi

Enter city name again:
City name now:
Enter another city name:
Chennai
New city name: Chennai
```

Second run

```
Enter city name:
```

```
New Delhi
City name: New

Enter city name again:
City name now: Delhi

Enter another city name:
Greater Bombay
New city name: Greater Bombay
```

note

During first run, the newline character '\n' at the end of "Delhi" which is waiting in the input queue is read by the **getline()** that follows immediately and therefore it does not wait for any response to the prompt 'Enter city name again:'. The character '\n' is read as an empty line. During the second run, the word "Delhi" (that was not read by cin) is read by the function **getline()** and, therefore, here again it does not wait for any input to the prompt 'Enter city name again:'. Note that the line of text "Greater Bombay" is correctly read by the second **cin.getline(city,size);** statement.

The **write()** function displays an entire line and has the following form:

```
court.write (line, size)
```

The first argument line represents the name of the string to be displayed and the second argument size indicates the number of characters to display. Note that it does not stop displaying the characters automatically when the null character is encountered. If the size is greater than the length of line, then it displays beyond the bounds of line. Program 12.3 illustrates how **write**() method displays a string.

DISPLAYING STRINGS WITH WRITE()

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
       char * string1 = "C++ ";
       char * string2 = "Programming";
       int m = strlen(string1);
       int n = strlen(string2);

       for(int i=1; i<n; i++)
       {
            cout.write(string2,i);
            cout << "\n";
```

```
        }

        for(i=n; i>0; i--)
        {
             cout.write(string2,i);
             cout << "\n";
        }

    // concatenating strings
    cout.write(string1,m).write(string2,n);

    cout << "\n";

    // crossing the boundary
    cout.write(string1,10);

    return 0;
}
```

PROGRAM 12.3

Look at the output of Program 12.3:

```
P
Pr
Pro
Prog
Progr
Progra
Program
Programm
Programmi
Programmin
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr
P
C++ Programming
C++ Progr
```

The last line of the output indicates that the statement

```
cout.write(string1, 10);
```

displays more characters than what is contained in **string1.**

It is possible to concatenate two strings using the **write**() function. The statement

```
cout.write(string1, m).write(string2, n);
```

is equivalent to the following two statements:

```
cout.write(string1, m);
cout.write(string2, n);
```

12.5 Formatted Console I/O Operations

C++ supports a number of features that could be used for formatting the output. These features include:

- **ios** class functions and flags.
- Manipulators.
- User-defined output functions.

The **ios** class contains a large number of member functions that would help us to format the output in a number of ways. The most important ones among them are listed in Table 12.2.

Table 12.2 **ios** *format functions*

Function	*Task*
Width ()	To specify the required field size for displaying an output value
precision ()	To specify the number of digits to be displayed after the decimal point of a float value
fill()	To specify a character that is used to fill the unused portion of a field
setf()	To specify format flags that can control the form of output display (such as left-justification and right-justification)
unsetf()	To clear the flags specified

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream. Table 12.3 shows some important manipulator functions that are frequently used. To access these manipulators, the file iomanip should be included in the program.

Table 12.3 *Manipulators*

Manipulators	*Equivalent ios function*
setw()	**width()**
setprecision()	**precision()**
setfill()	**fill()**
setiosflags()	**setf()**
resetiosflags()	**unsetf()**

In addition to these functions supported by the C++ library, we can create our own manipulator functions to provide any special output formats. The following sections will provide details of how to use the pre-defined formatting functions and how to create new ones.

12.5.1 Defining Field Width: width()

We can use the **width()** function to define the width of a field necessary for the output of an item. Since, it is a member function, we have to use an object to invoke it, as shown below:

```
cout.width(w);
```

where w is the field width (number of columns). The output will be printed in a field of w characters wide at the right end of the field. The **width**() function can specify the field width for only one item (the item that follows immediately). After printing one item (as per the specifications) it will revert back to the default. For example, the statements

```
cout.width(5);
cout << 543  <<  12 << "\n";
```

will produce the following output:

		5	4	3	1	2

The value 543 is printed right-justified in the first five columns. The specification width(5) does not retain the setting for printing the number 12. This can be improved as follows:

```
cout.width(5);
cout << 543;
cout.width(5);
cout << 12  << "\n";
```

This produces the following output:

		5	4	3				1	2

Remember that the field width should be specified for each item separately. C++ never truncates the values and therefore, if the specified field width is smaller than the size of the value to be printed, C++ expands the field to fit the value. Program 12.4 demonstrates how the function **width**() works.

SPECIFYING FIELD SIZE WITH WIDTH()

```
#include <iostream>
using namespace std;
int main()
{
     int items[4] = {10,8,12,15};
     int cost[4] = {75,100,60,99};

     cout.width(5);
```

```
        cout <<  "ITEMS";
        cout.width(8);
        cout << "COST";

        cout.width(15);
        cout << "TOTAL VALUE"  << "\n";

        int sum = 0;

        for(int i=0; i<4; i++)
        {
               cout.width(5);
               cout << items[i];

               cout.width(8);
               cout << cost[i];

               int value = items[i] * cost[i];
               cout.width(15);
               cout << value << "\n";
               sum = sum + value;
        }
         cout << "\n Grand Total = ";

        cout.width(2);
        cout << sum << "\n";

        return 0;
    }
```

PROGRAM 12.4

The output of Program 12.4 would be:

```
ITEMS          COST      TOTAL VALUE

   10            75              750
    8           100              800
   12            60              720
   15            99             1485

Grand Total = 3755
```

note

A field of width two has been used for printing the value of sum and the result is not truncated. A good gesture of C++ !

12.5.2 Setting Precision: precision()

By default, the floating numbers are printed with six digits after the decimal point. However, we can specify the number of digits to be displayed after the decimal point while printing the floating-point numbers. This can be done by using the **precision()** member function as follows:

```
cout.precision(d);
```

where d is the number of digits to the right of the decimal point. For example, the statements

```
cout.precision(3);
cout << sqrt(2) << "\n";
cout << 3.14159 << "\n";
cout << 2.50032 << "\n";
```

will produce the following output:

```
1.141  (truncated)
3.142  (rounded to the nearest cent)
2.5    (no trailing zeros)
```

Not that, unlike the function **width()**, **precision()** retains the setting in effect until it is reset. That is why we have declared only one statement for the precision setting which is used by all the three outputs.

We can set different values to different precision as follows:

```
cout.precision(3);
cout << sqrt(2)  << "\n";
cout.precision(5);              // Reset the precision
cout << 3.14159  << "\n";
```

We can also combine the field specification with the precision setting. Example:

```
cout.precision(2);
cout.width(5);
cout << 1.2345;
```

The first two statements instruct: "print two digits after the decimal point in a field of five character width". Thus, the output will be:

Program 12.5 shows how the functions **width()** and **precision()** are jointly used to control the output format.

PRECISION SETTING WITH PRECISION()

```
#include <iostream>
#include <cmath>

using namespace std;
```

```
int main()
{
  cout << "Precision set to 3 digits \n\n";
  cout.precision(3);

  cout.width(10);
  cout << "VALUE";
  cout.width(15);
  cout << "SQRT_OF_VALUE" << "\n";

  for(int n=1; n<=5; n++)
  {
         cout.width(8);
         cout << n;
         cout.width(13);
         cout << sqrt(n)  << "\n";
  }
  cout << "\n Precision set to 5 digits \n\n";
  cout.precision(5);              // precision parameter changed
  cout << " sqrt(10) = " << sqrt(10) << "\n\n";

  cout.precision(0);              // precision set to default
  cout << " sqrt(10) = " << sqrt(10) << " (default setting)\n";

  return 0;
}
```

PROGRAM 12.5

Here is the output of Program 12.5

```
Precision set to 3 digits
        VALUE     SQRT_OF_VALUE
          1               1
          2            1.41
          3            1.73
          4               2
          5            2.24
Precision set to 5 digits

sqrt(10) = 3.1623

sqrt(10) = 3.162278 (default setting)
```

note

Observe the following from the output:

1. The output is rounded to the nearest cent (i.e., 1.6666 will be 1.67 for two digit precision but 1.3333 will be 1.33).
2. Trailing zeros are truncated.
3. Precision setting stays in effect until it is reset.
4. Default precision is 6 digits.

12.5.3 Filling and Padding: fill()

We have been printing the values using much larger field widths than required by the values. The unused positions of the field are filled with white spaces, by default. However, we can use the fill() function to fill the unused positions by any desired character. It is used in the following form:

```
cout.fill (ch);
```

Where *ch* represents the character which is used for filling the unused positions. Example:

```
cout.fill('*');
cout.width(10);
cout << 5250  << "\n";
```

The output would be:

*	*	*	*	*	*	5	2	5	0

Financial institutions and banks use this kind of padding while printing cheques so that no one can change the amount easily. Like **precision()**, **fill()** stays in effect till we change it. See Program 12.6 and its output.

PADDING WITH FILL()

```
#include <iostream>

using namespace std;

int main( )
{  cout.fill('<');

   cout.precision(3);
          for(int n=1; n<=6; n++)
          {
           cout.width(5);
           cout << n;
           cout.width(10);
           cout << 1.0 / float(n) << "\n";
```

```
            if (n == 3)
               cout.fill ('>');
         }
         cout << "\nPadding changed \n\n";
         cout.fill ('#');    // fill( ) reset
         cout.width (15);
         cout << 12.345678   << "\n";

         return 0;
}
```

PROGRAM 12.6

The output of Program 12.6 would be:

```
<<<<1<<<<<<<<<1
<<<<2<<<<<<<0.5
<<<<3<<<<<0.333
>>>>4>>>>>>0.25
>>>>5>>>>>>>0.2
>>>>6>>>>>0.167

Padding changed

#########12.346
```

12.5.4 Formatting Flags, Bit-fields and setf()

We have seen that when the function **width()** is used, the value (whether text or number) is printed right-justified in the field width created. But, it is a usual practice to print the text left-justified. How do we get a value printed left-justified? Or, how do we get a floating-point number printed in the scientific notation?

The **setf()**, a member function of the **ios** class, can provide answers to these and many other formatting questions. The **setf()** (*setf* stands for set flags) function can be used as follows:

```
cout.setf(arg1,arg2)
```

The *arg1* is one of the formatting *flags* defined in the class **ios**. The formatting flag specifies the format action required for the output. Another **ios** constant, *arg2*, known as bit *field* specifies the group to which the formatting flag belongs.

Table 12.4 shows the bit fields, flags and their format actions. There are three bit fields and each has a group of format flags which are mutually exclusive. Examples:

```
cout.setf(ios::left, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
```

Note that the first argument should be one of the group members of the second argument.

Consider the following segment of code:

```
cout.fill('*');
cout.setf(ios::left, ios::adjustfield);
cout.width(15);
cout << "TABLE 1" <<  "\n";
```

Table 12.4 *Flags and bit fields for setf() function*

Format required	*Flag (arg1)*	*Bit-field (arg2)*
Left-justified output	ios :: left	ios :: adjustfield
Right-justified output	ios :: right	ios :: adjustfield
Padding after sign or base Indicator (like +##20)	ios :: internal	ios :: adjustfield
Scientific notation	ios :: scientific	ios :: floatfield
Fixed point notation	ios :: fixed	ios :: floatfield
Decimal base	ios :: dec	ios :: basefield
Octal base	ios :: oct	ios :: basefield
Hexadecimal base	ios :: hex	ios :: basefield

This will produce the following output:

T	A	B	L	E		1	*	*	*	*	*	*	*	*

The statements

```
cout.fill ('*');
cout.precision(3);
cout.setf(ios::internal, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
cout.width(15);

cout << -12.34567  << "\n";
```

will produce the following output:

-	*	*	*	*	*	1	.	2	3	5	e	+	0	1

note

The sign is left-justified and the value is right left- justified. The space between them is padded with stars. The value is printed accurate to three decimal places in the scientific notation.

12.5.5 Displaying Trailing Zeros and Plus Sign

If we print the numbers 12.75, 25.00 and 15.50 using a field width of, say, eight positions, with two digits precision, then the output will be as follows:

			1	0	·	7	5
						2	5
				1	5	·	5

Note that the trailing zeros in the second and third items have been truncated.

Certain situations, such as a list of prices of items or the salary statement of employees, require trailing zeros to be shown. The above output would look better if they are printed as follows:

```
10.75
25.00
15.50
```

The **setf()** can be used with the flag **ios::showpoint** as a single argument to achieve this form of output. For example,

```
cout.setf(ios::showpoint);          // display trailing zeros
```

would cause cout to display trailing zeros and trailing decimal point. Under default precision, the value 3.25 will be displayed as 3.250000. Remember, the default precision assumes a precision of six digits.

Similarly, a plus sign can be printed before a positive number using the following statement:

```
cout.setf(ios::showpos);          // show +sign
```

For example, the statements

```
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout.precision(3);
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::internal, ios::adjustfield);
cout.width(10);
cout << 275.5 << "\n";
```

will produce the following output:

+			2	7	5	·	5	0	0

The flags such as **showpoint** and **showpos** do not have any bit fields and therefore are used as single arguments in **setf()**. This is possible because the **setf()** has been declared as an overloaded function in the class **ios**. Table 12.5 lists the flags that do not possess a named bit field. These flags are not mutually exclusive and therefore can be set or cleared independently.

Table 12.5 *Flags that do not have bit fields*

Flag	*Meaning*
ios :: showbase	Use base indicator on output
ios :: showpos	Print + before positive numbers
ios :: showpoint	Show trailing decimal point and zeroes
ios :: uppercase	Use uppercase letters for hex output
ios :: skipus	Skip white space on input
ios :: unitbuf	Flush all streams after insertion
ios :: stdio	Flush **stdout** and **stderr** after insertion

Program 12.7 demonstrates the setting of various formatting flags using the overloaded **setf()** function.

FORMATTING WITH FLAGS IN setf()

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
  cout.fill('*');
  cout.setf(ios::left, ios::adjustfield);
  cout.width(10);
  cout << "VALUE";

  cout.setf(ios::right, ios::adjustfield);
  cout.width(15);
  cout << "SQRT OF VALUE" << "\n";

  cout.fill('.');
  cout.precision(4);
  cout.setf(ios::showpoint);
  cout.setf(ios::showpos);
  cout.setf(ios::fixed, ios::floatfield);

  for(int n=1; n<=10; n++)
  {
        cout.setf(ios::internal, ios::adjustfield);
        cout.width(5);
        cout << n;

        cout.setf(ios::right, ios::adjustfield);
        cout.width(20);
```

```
            cout << sqrt(n)  << "\n";
        }

        // floatfield changed
        cout.setf(ios::scientific, ios::floatfield);
        cout << "\nSQRT(100) = " << sqrt(100) << "\n";

        return 0;
    }
```

PROGRAM 12.7

The output of Program 12.7 would be:

```
VALUE*********SQRT OF VALUE
+...1.............+1.0000
+...2.............+1.4142
+...3.............+1.7321
+...4.............+2.0000
+...5.............+2.2361
+...6.............+2.4495
+...7.............+2.6458
+...8.............+2.8284
+...9.............+3.0000
+..10.............+3.1623

SQRT(100) = +1.0000e+001
```

note

1. The flags set by **setf()** remain effective until they are reset or unset.
2. A format flag can be reset any number of times in a program.
3. We can apply more than one format controls jointly on an output value.
4. The setf() sets the specified flags and leaves others unchanged.

12.6 Managing Output with Manipulators

The header file *iomanip* provides a set of functions called *manipulators* which can be used to manipulate the output formats. They provide the same features as that of the **ios** member functions and flags. Some manipulators are more convenient to use than their counterparts in the class **ios**. For example, two or more manipulators can be used as a chain in one statement as shown below:

```
cout << manip1 << manip2 << manip3 << item;
cout << manip1 << item1  << manip2 << item2;
```

This kind of concatenation is useful when we want to display several columns of output.

The most commonly used manipulators are shown in Table 12.6. The table also gives their meaning and equivalents. To access these manipulators, we must include the file *iomanip* in the program.

Table 12.6 *Manipulators and their meanings*

Manipulator	*Meaning*	*Equivalent*
setw (int *w*)	Set the field width to w.	width( )
setprecision(int *d*)	Set the floating point precision to *d*.	precision( )
setfill(int *c*)	Set the fill character to c.	fill( )
setiosflags(long *f*)	Set the format flag *f*.	setf( )
resetiosflags(long *f*)	Clear the flag specified by *f*.	unsetf( )
endl	Insert new line and flush stream.	"\n"

Some examples of manipulators are given below:

```
cout << setw(10) << 12345;
```

This statement prints the value 12345 right-justified in a field width of 10 characters. The output can be made left-justified by modifying the statement as follows:

```
cout << setw(10) << setiosflags(ios::left) << 12345;
```

One statement can be used to format output for two or more values. For example, the statement

```
cout << setw(5)  << setprecision(2) << 1.2345
     << setw(10) << setprecision(4) << sqrt(2)
     << setw(15) << setiosflags(ios::scientific) << sqrt(3);
     << endl;
```

will print all the three values in one line with the field sizes of 5, 10, and 15 respectively. Note that each output is controlled by different sets of format specifications.

We can jointly use the manipulators and the **ios** functions in a program. The following segment of code is valid:

```
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout << setprecision(4);
cout << setiosflags(ios::scientific);
cout << setw(10) << 123.45678;
```

note

There is a major difference in the way the manipulators are implemented as compared to the **ios** member functions. The **ios** member function return the previous format state which can be used later, if necessary. But the manipulator does not return the previous format state. In case, we need to save the old format states, we must use the **ios** member functions rather than the manipulators. Example:

```
cout.precision(2);            // previous state
int p = cout.precision(4);   // current state;
```

When these statements are executed, **p** will hold the value of 2 (previous state) and the new format state will be 4. We can restore the previous format state as follows:

```
cout.precision(p);           // p = 2
```

Program 12.8 illustrates the formatting of the output values using both manipulators and **ios** functions.

FORMATTING WITH MANIPULATORS

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
   cout.setf(ios::showpoint);

   cout << setw(5)  <<  "n"
        << setw(15) << "Inverse_of_n"
        << setw(15) << "Sum_of_terms\n\n";

   double term, sum = 0;

   for(int n=1; n<=10; n++)
   {
          term = 1.0 / float(n);
          sum  = sum + term;

          cout << setw(5) << n
               << setw(14) << setprecision(4)
               << setiosflags(ios::scientific) << term
               << setw(13) << resetiosflags(ios::scientific)
               << sum << endl;
   }
   return 0;
}
```

PROGRAM 12.8

The output of Program 12.8 would be:

```
n     Inverse_of_n      Sum_of_terms

1     1.0000e+000          1.0000
```

```
 2    5.0000e-001        1.5000
 3    3.3333e-001        1.8333
 4    2.5000e-001        2.0833
 5    2.0000e-001        2.2833
 6    1.6667e-001        2.4500
 7    1.4286e-001        2.5929
 8    1.2500e-001        2.7179
 9    1.1111e-001        2.8290
10    1.0000e-001        2.9290
```

12.6.1 Designing Our Own Manipulators

We can design our own manipulators for certain special purposes. The general form for creating a manipulator without any arguments is:

```
ostream & manipulator (ostream & output)
{
   .....
   .....  (code)
   .....
   return output;
}
```

Here, the *manipulator* is the name of the manipulator under creation. The following function defines a manipulator called **unit** that displays "inches":

```
ostream & unit(ostream & output)
{
        output << " inches";
        return output;
}
```

The statement

```
cout << 36 << unit;
```

will produce the following output

```
36 inches
```

We can also create manipulators that could represent a sequence of operations. Example:

```
ostream & show(ostream & output)
{
        output.setf(ios::showpoint);
        output.setf(ios::showpos);
        output << setw(10);
        return output;
}
```

This function defines a manipulator called **show** that turns on the flags **showpoint** and **showpos** declared in the class **ios** and sets the field width to 10.

Program 12.9 illustrates the creation and use of the user-defined manipulators. The program creates two manipulators called **currency** and **form** which are used in the **main** program.

USER-DEFINED MANIPULATORS

```
#include <iostream>
#include <iomanip>

using namespace std;

// user-defined manipulators

ostream & currency(ostream & output)
{
        output << "Rs";
        return output;
}

ostream & form(ostream & output)
{
        output.setf(ios::showpos);
        output.setf(ios::showpoint);
        output.fill('*');
        output.precision(2);
        output << setiosflags(ios::fixed)
               << setw(10);
        return output;
}

int main()
{
        cout << currency << form << 7864.5;

        return 0;
}
```

PROGRAM 12.9

The output of Program 12.9 would be:

```
Rs**+7864.50
```

Note that **form** represents a complex set of format functions and manipulators.

SUMMARY

- In C++, the I/O system is designed to work with different I/O devices. This I/O system supplies an interface called 'stream' to the programmer, which is independent of the actual device being used.
- A stream is a sequence of bytes and serves as a source or destination for an I/O data.
- The source stream that provides data to the program is called the *input stream* and the destination stream that receives output from the program is called the *output stream*.
- The C++ I/O system contains a hierarchy of stream classes used for input and output operations. These classes are declared in the header file **'iostream'**.
- **cin** represents the input stream connected to the standard input device and **cout** represents the output stream connected to the standard output device.
- The **istream** and **ostream** classes define two member functions **get()** and **put()** to handle the single character I/O operations.
- The >> operator is overloaded in the **istream** class as an extraction operator and the << operator is overloaded in the **ostream** class as an insertion operator.
- We can read and write a line of text more efficiently using the line oriented I/O functions **getline()** and **write()** respectively.
- The **ios** class contains the member functions such as **width()**, **precision()**, **fill()**, **setf()**, **unsetf()** to format the output.
- The header file **'iomanip'** provides a set of manipulator functions to manipulate output formats. They provide the same features as that of **ios** class functions.
- We can also design our own manipulators for certain special purposes.

Key Terms

- adjustfield
- basefield
- bit-fields
- console I/O operations
- decimal base
- destination stream
- field width
- **fill()**
- filling
- fixed point notation
- flags
- floatfield
- formatted console I/O
- formatting flags
- formatting functions
- **get()**

- **getline()**
- hexadecimal base
- input stream
- internal
- **ios**
- iomanip
- iostream
- istream
- left-justified
- manipulator
- octal base
- ostream
- output stream
- padding
- **precision()**
- **put()**
- **resetiosflags()**
- right-justified
- scientific notation
- **setf()**
- **setfill()**
- **setiosflags()**
- **setprecision()**
- setting precision
- **setw()**
- showbase
- showpoint
- showpos
- skipus
- source stream
- standard input device
- standard output device
- stream classes
- streambuf
- streams
- unitbuf
- **unsetf()**
- **width()**
- **write()**

Review Questions

12.1 *What is a stream?*

12.2 *Describe briefly the features of I/O system supported by C++.*

12.3 *How do the I/O facilities in C++ differ from that in C?*

12.4 *Why are the words such as **cin** and **cout** not considered as keywords?*

12.5 *How is **cout** able to display various types of data without any special instructions?*

12.6 *Why is it necessary to include the file iostream in all our programs?*

12.7 *Discuss the various forms of **get()** function supported by the input stream. How are they used?*

12.8 *How do the following two statements differ in operation?*

```
cin >> c;
cin.get(c);
```

12.9 *Both **cin** and **getline()** function can be used for reading a string. Comment.*

12.10 *Discuss the implications of size parameter in the following statement:*

```
cout.write(line, size);
```

12.11 *What does the following statement do?*

```
cout.write(s1,m).write(s2,n);
```

12.12 *What role does the* ***iomanip*** *file play?*

12.13 *What is the role of* ***file()*** *function? When do we use this function?*

12.14 *Discuss the syntax of* ***set()*** *function.*

12.15 *What is the basic difference between manipulators and* ***ios*** *member functions in implementation? Give examples.*

12.16 *State whether the following statements are TRUE or FALSE.*

(a) *A C++ stream is a file.*

(b) *C++ never truncates data.*

(c) *The main advantage of* ***width()*** *function is that we can use one width specification for more than one items.*

(d) *The* ***get(void)*** *function provides a single-character input that does not skip over the white spaces.*

(e) *The header file* ***iomanip*** *can be used in place of iostream.*

(f) *We cannot use both the C I/O functions and C++ I/O functions in the same program.*

(g) *A programmer can define a manipulator that could represent a set of format functions.*

12.17 *What will be the result of the following programs segment?*

```
for(i=0.25; i<=1.0; i=i+0.25)
{
     cout.precision(5);
     cout.width(7);
     cout << i;
     cout.width(10);
     cout <<i*i<< "\n";
}
cout << setw(10) << "TOTAL ="
     << setw(20) << setprecision(2) << 1234.567
     << endl;
```

12.18 *Discuss the syntax for creating user-defined manipulators. Design a single manipulator to provide the following output specifications for printing float values:*

(a) *10 columns width*

(b) *Right-justified*

(c) *Two digits precision*

(d) *Filling of unused places with* ***

(e) *Trailing zeros shown*

Debugging Exercises

12.1 To get the output Buffer1: Jack and Jerry Buffer2: Tom and Mono, what do you have to do in the following program?

```
#include <iostream.h>
void main()
{
        char buffer1[80];
        char buffer2[80];

        cout << "Enter value for buffer1 : ";
        cin >> buffer1;
        cout << "Buffer1 : " << buffer1 << endl;

        cout << "Enter value for buffer2 : ";
        cin.getline(buffer2, 80);
        cout << "Buffer2 : " << buffer2 << endl;
}
```

12.2 Will the statement cout.setf(ios::right) work or not?

```
#include <iostream.h>
void main()
{
        cout.width(5);
        cout << "99" << endl;

        cout.setf(ios::left);
        cout.width(5);
        cout << "99" << endl;

        cout.setf(ios::right);
        cout << "99" << endl;
}
```

12.3 State errors, if any, in the following statements.

(a) `cout << (void*) amount;`
(b) `cout << put("John");`
(c) `cout << width();`
(d) `int p = cout.width(10);`

(e) `cout.width(10).precision(3);`
(f) `cout.setf(ios::scientific,ios::left);`
(g) `ch = cin.get();`
(h) `cin.get().get();`
(i) `cin.get(c).get();`
(j) `cout << setw(5) << setprecision(2);`
(k) `cout << resetiosflags(ios::left |ios::showpos);`

Programming Exercises

12.1 *Write a program to read a list containing item name, item code, and cost interactively and produce a three column output as shown below.*

NAME	CODE	COST
Turbo C++	1001	250.95
C Primer	905	95.70
.....	...	.....
.....	...	.....
.....	...	.....

Note that the name and code are left-justified and the cost is right-justified with a precision of two digits. Trailing zeros are shown.

12.2 *Modify the above program to fill the unused spaces with hyphens.*

12.3 *Write a program which reads a text from the keyboard and displays the following information on the screen in two columns:*

(a) *Number of lines*
(b) *Number of words*
(c) *Number of characters*

Strings should be left-justified and numbers should be right-justified in a suitable field width.

13

Database Management System

Key Concepts

- Introduction
- File Oriented v/s Database Approach
- Data Models
- Architecture of a Database System
- Data Independence
- Data Dictionary
- DBA
- DBMS Keys
- Data Definition Language
- Data Manipulation Language
- Summary

13.1 Introduction

Data storage is an important function of a computer system. While the facility of storing the data is provided by hardware storage devices, we cannot simply dump the entire data in them. We must logically organize the data in such a way that future data access and manipulation becomes simpler and efficient. A Database Management System (DBMS) is one such dedicated program that manages the collection of a large number of data elements in a systematic manner.

Even though we do not realize but databases are used in our day-to-day lives in one form or the other. For instance, while performing a bank transaction, the banking software application accesses the underlying database to fetch our account details. Similarly, the point-of-sale application at the retail store and even our mobile phones make use of the underlying database to store and retrieve relevant information. Thus, from the point of view of application development, the design and implementation of a database system is quite significant and indispensable.

By definition, we may refer to DBMS as a collection of programs that facilitate the systematic creation and maintenance of a database. These programs help in designing the database structure, storing the data in the database and manipulating the data as per requirement. There are several different database modelling techniques and design methodologies that help us in designing and implementing an efficient database system. The choice of a particular design methodology should be done after carefully studying the current environment and its associated constraints. We shall gradually understand these concepts as we progress through this chapter.

13.2 File-oriented v/s Database Approach

Contrary to the modern database approach, the traditional programming applications relied on file-oriented data systems for the purpose of data storage. An application maintained its own set of files for data storage in isolation from the other applications which could be maintaining similar data. For example, the payroll application of the HR department would maintain a set of files for storing employee data which was partially similar to the data maintained in separate set of files by the MIS application of the operations team. The obvious disadvantage of such an approach was data redundancy, which resulted in wastage of memory space. Further, it also led to duplication of efforts in trying to update the same data across discrete sets of files.

The database approach removes these bottlenecks of the file-based systems by maintaining a single repository for data storage. The stored data is commonly accessed by all the applications that use the database system as their back-end, thus preventing data redundancy and ensuring data consistency.

Let us explore the various areas where the database approach holds an edge over the file oriented approach.

Data Definition

The process of storing data in a computer system first requires the design and specification of the data system. It involves specification of information like data type, data format, data constraints, etc. In file-based approach, this data-design specification is part of the application program itself. Thus, the particular file-based data system can only be used with the application that defines it. However, this is not the case with the database approach.

A database management system stores information about the structure and format of the data in separate files. This information is referred as *meta data*, that is data about data. The DBMS software makes use of both the meta data as well as the actual data stored in the database files to respond to the data requests raised by application programs. As a result, the application programs are not required to separately code the data-design specifications. These are specified only once in the DBMS software and used by different applications linked with the database. Further, the structure of the data can be easily modified by simply updating the meta files. However, this is not the case with file-based systems, which require explicit changes in the application code for updating the structure and format of the data.

Redundancy

Unlike file-based systems, where the same data is maintained by different application programs in their respective files, the database approach prevents such duplication of data by storing the data in a common central repository. This central repository is accessed and updated by different applications through the DBMS interface. As a result, the DBMS approach prevents data redundancy and ensures efficient utilization of memory space.

Consistency

Data inconsistency was one of the major bottlenecks of file-based data systems. If an application made any updates to the data then there was no means of automatically replicating it across other applications' data files. This had to be done manually and was thus error prone. The database approach eliminates this disadvantage by maintaining a single repository of data that is shared by different applications. Thus, any change in data values made by an application is instantly visible to all other applications accessing the same database. This ensures data consistency at all times.

Integrity

Data integrity refers to accuracy, validity and completeness of the stored data. The database approach holds an advantage on the file-based systems in the data-integrity front. While it is quite difficult to achieve high levels of data integrity in the file-oriented systems, the same is an inherent characteristic of the database systems.

In DBMS, data integrity is ensured with the help of integrity constraints. For instance, we may restrict the 'Age' column in a table to store only positive values. Similarly, we may specify that the 'Date' column stores the date values in the DD/MM/YYYY format. This ensures that the data is always valid and meaningful.

13.3 Data Models

A data model refers to the structure of a database system describing how data objects are arranged inside the database. It also describes several other concepts related to the database system, such as constraints, relationships, etc. The various types of data models are

- Entity-Relationship (ER) model
- Hierarchical model
- Network model

ER Model

The ER model realizes all real-world objects and concepts as entities and defines relationships amongst these entities. It is primarily used for designing a relational database system. It represents the overall logical structure of the entire database system.

The key terminologies related to the ER model are the following:

- ***Entity:*** It represents a real-world object, such as an employee, a bank account, etc.

- ***Entity Set:*** It is a collection of entities of similar type, such as a group of employees.
- ***Attributes:*** It refers to the characteristics that represent an entity. For example, Name, Employee Number, Department, etc., are the attributes of the Employee entity.
- ***Relationship:*** It specifies how two or more entities or entity sets are related to each other. For example, 'Manages', 'Supervises' and 'Works' can act as relationships between Employee and Department entity sets.
- ***Relationship set:*** It is a group of relationships of similar types.
- ***ER diagram:*** It is a diagrammatic representation of the logical structure of a database. It represents entity sets by rectangles, relationship sets by diamonds and attributes by ellipses.

Figure 13.1 shows a sample ER diagram:

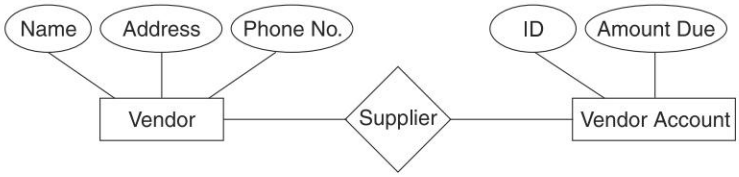
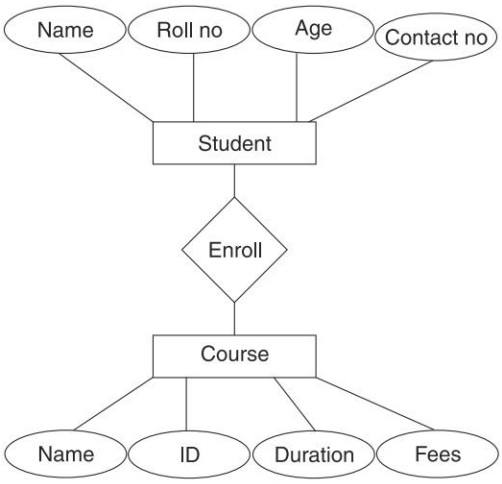


Fig. 13.1 ⇔ *ER Diagram*

In the above diagram, Vendor and Vendor Account are entity sets while Name, Address, ID etc., are attributes of the entity sets. Supplier is the relationship set that defines the relationship between the two entity sets. A complete ER diagram for a given scenario can be used as a blueprint or a design document for creating a relational database system.

Example 13.1: Consider the example of a training institute that runs several different courses for students. Each student is required to enrol into any one of the courses at the time of admission. Suggest the entity set, attributes and relationship for this scenario. Also, draw a tentative ER diagram.



ER Diagram

Solution:

Entity set 1: Course

Attributes: Name, ID, Duration, Fees

Entity set 2: Student

Attributes: Name, Roll Number, Age, Contact Number

Relationship: Enroll

Mapping Constraints

An ER scheme may define certain constraints while realizing a real-world scenario with the help of entities and relationships. Such constraints are expressed with the help of mapping cardinalities, which specify the number of entities to which an entity can be associated through a relationship. The various mapping cardinalities are the following:

- ***One-to-one:*** An entity in the entity set *A* is associated with only one entity in another entity set *B*.
- ***One-to-many:*** An entity in the entity set *A* is associated with multiple entities in another entity set *B*.
- ***Many-to-one:*** Multiple entities in the entity set *A* are associated with one entity in another entity set *B*.
- ***Many-to-many:*** Multiple entities in the entity set *A* are associated with multiple entities in another entity set *B*.

Example 13.2: Suggest a real-life scenario for each of the four mapping cardinalities.

Solution:

Mapping Cardinality	*Entity Set 1*	*Relationship*	*Entity Set 2*
One-to-one	Manager	Manages	Department
One-to-many	Bank Account	Contains	Transactions
Many-to-one	Employees	Working In	Department
Many-to-many	Employees	Working on	Projects

Hierarchical Model

As the name suggests, a hierarchical data model arranges the data in hierarchical or tree format. It follows the simple parent-child relationship for arranging data where each parent can have zero or many children while each child has only one parent. Hierarchical data model is suitable in situations where there is a direct relationship between records. To understand the hierarchical data model, let us take the analogy of the product range of an electronics products manufacturing company, as shown in Figure 13.2.

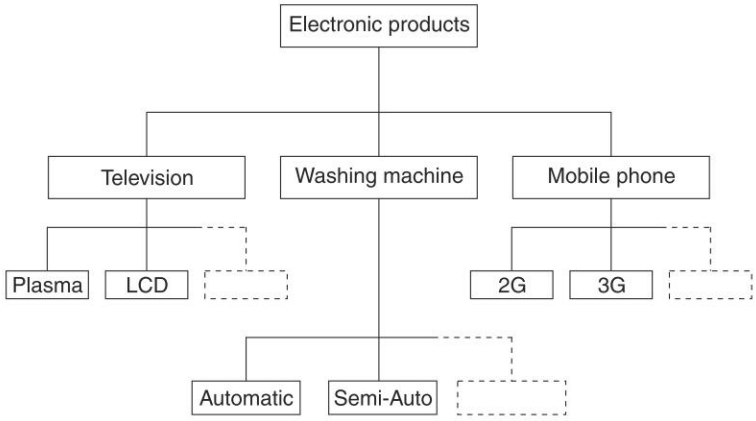


Fig. 13.2 ⇔ *Hierarchical Structure*

As shown in the above figure, the related products are grouped inside their parent categories in a hierarchical fashion. It is important to note that hierarchical database design is suitable in situations where there is a hierarchical relationship between the data items.

Some of the key advantages of hierarchical data model are the following:

- It is simple to understand and implement.
- It is particularly suitable in situations where the data items and their relationships are already known, and are not expected to change.
- It helps to build faster and efficient databases.

Apart from the above advantages, the hierarchical model also has certain limitations, which are given here

- It is less flexible and difficult to change.
- There is comparatively lesser scope of query optimization in hierarchically modeled databases.

Network Model

If the hierarchical data model is analogous to a tree structure, the network data model is analogous to a graph. That means, unlike the hierarchical model, a child node in a network model can have multiple parent nodes. This signifies a many-to-many relationship between various data items. Thus, flexibility becomes one of the most significant advantages of a network model.

The network data model was formally put forward by the Conference on Data Systems Languages (CODASYL) in 1971. With a many-to-many relationship concept, it removes any sort of restrictions on the database structure, as is prevalent in the hierarchical model. Thus, it facilitates convenient modeling of relationships between entities. Figure 13.3 represents a network data model:

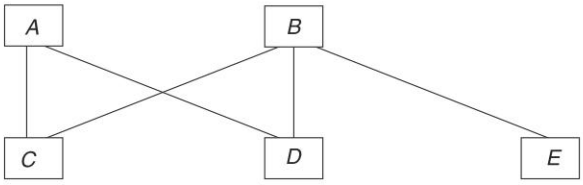


Fig. 13.3 ⇔ *Network model*

13.4 Architecture of a Database System

The architecture of a database system depicts the structure and layout of the data stored and the mechanisms used for accessing the stored data. It shows how various functional components are arranged inside the database system to facilitate typical database-related operations. Figure 13.4 shows the generic view of the database system architecture.

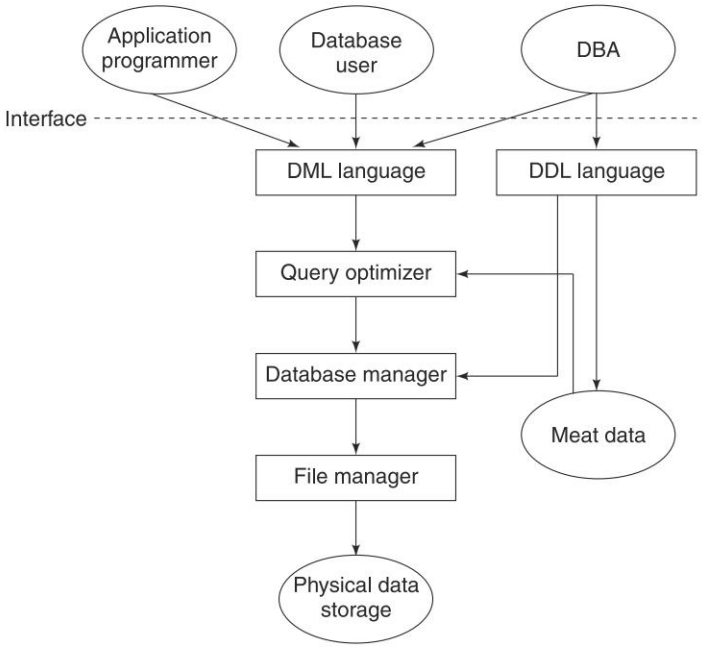


Fig. 13.4 ⇔ *Database System Architecture*

As shown in the above figure, a database system comprises of the following typical components:

- ***DML Language:*** Also known as query language, it is used for manipulating (inserting, modifying, and deleting) data stored in various tables of the database.

- ***DDL Language:*** It is used for defining the structure of the database, i.e., tables and their attributes, relationships, constraints, etc.
- ***Query Optimizer:*** It optimizes the execution of a query. There could be multiple ways of executing a particular query; the query optimizer chooses the fastest and most efficient methods amongst them.
- ***Database Manager:*** It acts as a controller for performing database-related operations.
- ***File Manager:*** It manages the database files.
- ***Physical Database:*** It is the physical storage device on which data is actually stored.
- ***Meta Data:*** It is the data dictionary used for storing meta information, i.e., data about data.
- ***User Interface:*** It is the entry point from where users and applications interact with the database.

There is another approach of looking at the database architecture, i.e. in terms of schemas, as explained next.

Three-Schema Architecture

The three-schema architecture was put forward by the ANSI/SPARC committee in the late 1970's. The objective of this architecture was to segregate the physical database, its logical structure and the users and applications from each other. The three-schema architecture divides the entire database schema across three levels, which are as follows:

- ***Internal Level:*** Contains the physical schema, i.e., the physical storage structure of the database.
- ***Conceptual Level:*** Contains the logical schema, i.e., the logical database constructs used for designing the database. Also known as community level, it comprises of data objects, relationships, constraints, database operations, etc.
- ***External Level:*** Contains the external schema specifying how data is presented to the users in different views.

The implementation of the three-schema architecture helps realize the all important concept of data independence in database systems.

13.5 Data Independence

Data dependence is a situation where design and development of an application program relies heavily on the associated storage structures. It was widely prevalent in file-oriented data storage systems. The design and coding of the application was done keeping in mind the structure of the file-based storage systems. If any change was made in the structure and layout of the storage system then it necessitated appropriate change in the application code as well. Obviously, such huge amount of data dependence was an overhead and posed difficulties in change management.

DBMS eliminated this disadvantage by supporting the concept of data independence. It segregates the programming and storage environments in such a way that changes in the

underlying storage structures do not affect the front-end application. The application continues to work with the data in the same manner as earlier. Thus, application programs are isolated from the storage structures. Data independence is of two types:

- ***Logical Data Independence:*** Allows modification of the conceptual schema without requiring any changes to be made to the external schema.
- ***Physical Data Independence:*** Allows modification of the internal schema without requiring any changes to be made to the conceptual schema.

13.6 Data Dictionary

Data dictionary is a key part of a database system that contains information about the database itself, as well as the data elements stored in it. Thus, a data dictionary is nothing but a collection of meta data or 'data about data'. The data dictionary of a database may be created in any of the following forms:

- Tables
- Text or XML files
- Spreadsheet

The main objective of a data dictionary is to store accurate and complete meta information so that the users who want to work with the database can access it as a ready reference. Typically, a data dictionary may include the following:

- Table name
- Column names
- Data-type information
- Default field values
- Data constraints
- Schema information
- User information and their privileges

Most of the database systems automatically update the data dictionary when a user issues any Data Definition Language (DDL) statement. A data dictionary might be of interest to any one of the following users:

- Database developers
- Application developers
- Database Administrator (DBA)

13.7 DBA

A database administrator, or DBA, is the person responsible for development and management of a database system. A DBA has the centralized control over the database and is primarily responsible for its maintenance and support. The typical roles and responsibilities of a DBA include the following:

- Implementing database models
- Designing and creating database schemas

- Managing user rights and privileges
- Implementing backup and recovery procedures
- Establishing and enforcing database standards
- Implementing integrity constraints
- Database performance tuning
- Executing all database control and administration tasks
- Ensuring database availability
- Planning and implementing database migration as and when required
- Managing database storage devices
- Implementing and reviewing database security policies
- Updating database documentation

The actual roles and responsibilities of a DBA may vary from one organization to the other.

13.8 DBMS Keys

A key is an attribute (column) or a combination of attributes that is used for accessing records or rows in a database table. The concept of keys particularly helps in joining multiple tables together so as to retrieve the required data values. The various types of keys are

- Super key
- Candidate key
- Primary key
- Foreign key
- Secondary key

Super Key

A table in a relational database system may comprise of an attribute or a column that contains unique values. If not for a single attribute, a combination of multiple attributes may generate unique identifier values in a table. These unique values help us in uniquely identifying each record or row in the table. Such attributes or combination of attributes that help uniquely identify all the records in a table are called super keys.

Consider a database table containing employee information, as shown in Table 13.1.

Table 13.1 *Employee Table*

Emp_id	*First_name*	*Last_name*	*Address*
079	Vikas	Verma	Delhi
081	Rahul	Aggarwal	Mumbai
002	Manish	Aggarwal	Agra
101	Vikas	Jain	Karnal
012	Neha	Malik	Delhi
084	Jim	Andrews	Pune

In the above table, the various super keys are

- Emp_id
- Emp_id, First_name
- Emp_id, Last_name
- Emp_id, Address, and so on

Candidate Key

It is possible that a super key comprises of certain attributes which themselves are capable of uniquely identifying the rows in a table. Candidate keys are all those super keys whose subset of attributes cannot uniquely identify rows in a table. Hence, we may say that candidate keys are nothing but minimalized super keys.

In the Employee table, the following can be considered as the candidate keys:

- Emp_id
- First_name, Last_name, Address

As we can observe clearly, the candidate key *First_name, Last_name, Address* can uniquely identify table rows but none of the three attributes can identify the table rows individually.

Primary Key

A candidate key that is actually used for unique identification of rows in a table is called primary key. A database designer needs to specifically designate one of the candidate keys a as primary key. As already explained, a primary key may comprise of single or multiple attributes.

In the Employee table, we may define Emp_id as the primary key.

Foreign Key

The concept of a foreign key helps to establish a cross reference or a relationship between two tables in a relational database system. An attribute or a combination of attributes in a table is considered as a foreign key if it is a primary or a candidate key in another table. All the foreign key values must be present in the primary or candidate key of the referenced table; however, the opposite may not hold true. The table being referenced is considered as the parent or a master table, while the referencing table is considered as the child table.

Consider a table containing employee skill set information, as shown in Table 13.2.

Table 13.2 *Employee Skill Set*

Skill_set	*Emp_id*	*Dept_name*
CRM	079	Sales
CRM	081	Sales
Marketing	081	Sales
Taxation	002	Accounts
Taxation	101	Accounts
Trainer	101	Accounts
Trainer	012	HR
IT	084	Support

In the above table, Emp_id is a foreign key that references back to the Emp_id attribute of the Employee table, shown in Table 13.1. Thus, by observing the relationship between the two tables we may say that Emp_id 081 possesses two skill sets, CRM and Marketing, and it belongs to Rahul Aggarwal.

Secondary Key

The attributes that are not super keys and help in identifying non-unique records in a table are termed secondary keys. In the Employee table, First_name and Last_name can be considered as secondary keys.

13.9 Data Definition Language

Data Definition Language (DDL) is a database language that is used for defining the structure of a database. It helps specify the meta data pertaining to the actual data that will be stored in the database. It comprises of a number of commands that allow us to create, modify or delete databases, tables, indexes, views, etc. DDL commands are executed by the DDL compiler.

Some of the basic SQL-based DDL commands are

- CREATE
- ALTER
- DROP

CREATE

The CREATE command is used for creating database objects. For example, consider the following command:

```
CREATE DATABASE Employee
```

The above command creates a new database named Employee.

Similarly, the following command creates a new table named emp_details:

```
CREATE TABLE emp_details (first_name char (30) not null, last_name char
(30) not null, emp_id int not null)
```

The above command creates a new table named emp_details containing three columns, first_name, last_name and emp_id. The first_name and last_name columns contain text strings of length up to 30 characters while the emp_id column contains integer-based ID values. Table 13.3 represents the emp_details table:

Table 13.3 *emp_details Table*

first_name	*last_name*	*emp_id*

Since we have not yet added any data in the emp_details table, it is shown as empty.

ALTER

The ALTER command is used to modify the structure of an already created table. For instance, we may use the ALTER command to add new columns or delete existing columns in a table. Consider the following command:

```
ALTER TABLE emp_details ADD department char (15) not null
```

The above command adds a new column named department into the emp_details table. Table 13.4 shows the modified structure of the emp_details table:

Table 13.4 *Modified emp_details Table*

first_name	*last_name*	*emp_id*	*department*

DROP

The DROP command is used for destroying or deleting the database objects. For example, consider the following command:

```
DROP DATABASE Employee
```

The above command deletes the database named Employee.

Similarly, the following command deletes the emp_details table:

```
DROP TABLE emp_details
```

With the execution of the DROP command, the contents of the database objects are also deleted.

13.10 Data Manipulation Language

Data Manipulation Language (DML) is a database language that is used for accessing and storing information in a database. It comprises of a number of commands that allow us to retrieve, insert, modify and delete information in a database. DML commands are executed by the DML compiler.

Some of the basic SQL-based DML commands are

- INSERT
- SELECT
- UPDATE
- DELETE

INSERT

The INSERT command is used for inserting records or rows in a database table. For example, consider the following command:

```
INSERT INTO emp_details values ('Rahul','Sharma', 199, 'Sales')
```

In the above command, the values to be inserted are specified in the same order as the order of the attributes or columns in the emp_details table. Table 13.5 shows the contents of emp_details table after a series of INSERT commands:

Table 13.5 *Records of emp_details Table*

first_name	*last_name*	*emp_id*	*department*
Rahul	Sharma	199	Sales
Anil	Kumar	55	Accounts
Geetika	Sharma	66	Operations
Rajiv	Bhatia	22	Operations

SELECT

The SELECT command is used to extract and display table information in different ways. For example, consider the following command:

```
SELECT * FROM emp_details
```

The above command will display the entire contents of the emp_details table.

However, if we want to display only some rows of the emp_details table then we need to specify the corresponding selection criteria, as shown below:

```
SELECT * FROM emp_details WHERE emp_id > 60
```

The above command will display all the rows in which the value of emp_id attribute is greater than 60. Table 13.6 shows the output:

Table 13.6 *Extracted Records of emp_details Table*

first_name	*last_name*	*emp_id*	*department*
Rahul	Sharma	199	Sales
Geetika	Sharma	66	Operations

Further, we may also extract only specific attribute values or columns from a table, as shown below:

```
SELECT first_name FROM emp_details
```

The above command will fetch and display all the values under the first_name column and ignore all other column values. Table 13.7 shows the output:

Table 13.7 *first_name column of emp_details Table*

first_name
Rahul
Anil
Geetika
Rajiv

UPDATE

The UPDATE command is used to modify the existing information contained in a table. For example, consider the following statement:

```
UPDATE emp_details SET emp_id = emp_id + 100
```

The above command will increase all the employee ID values stored in the table by 100. Table 13.8 shows the modified emp_details table:

Table 13.8 *Updated emp_id Column*

first_name	*last_name*	*emp_id*	*department*
Rahul	Sharma	299	Sales
Anil	Kumar	155	Accounts
Geetika	Sharma	166	Operations
Rajiv	Bhatia	122	Operations

Similarly, we can also update only some records in a table by specifying certain updation criteria. For example, consider the following statement:

```
UPDATE emp_details SET department = 'Sales' WHERE emp_id = 166
```

The above command will change the department of the employee having ID 166 to Sales.

DELETE

The DELETE command is used for removing a record or row from a table. For example, consider the following statement:

```
DELETE FROM emp_details WHERE cust_id = 166
```

The above command will remove details of the employee having ID 166 from the table. Table 13.9 shows the remaining records of the emp_details table:

Table 13.9 *emp_details Table with Record Deleted*

first_name	*last_name*	*emp_id*	*department*
Rahul	Sharma	199	Sales
Anil	Kumar	155	Accounts
Rajiv	Bhatia	122	Operations

Example 13.3: The following table named bluesky contains details of the employees of the Blue Sky organization.

first_name	*last_name*	*designation*	*email_id*	*emp_id*	*Qualification*
Subrata	Bakshi	SSE	subrata@bluesky.com	950	B Tech
Neha	Sharma	SE	neha@bluesky.com	986	MCA
Yogender	Kumar	SE	yogender@bluesky.com	903	MCA
Tanmay	Mukherjee	ME	tanmay@bluesky.com	994	B Tech
Mohammad	Afzal	ME	afzal@bluesky.com	936	B Tech
Tulsi	Krishnamoorty	DM	tulsi@bluesky.com	937	M Tech
Vinay	Ahuja	SSE	vinay@bluesky.com	938	MCA
Rajesh	Taneja	SSE	rajesh@bluesky.com	966	MCA
Shilpa	Sareen	SE	shilpa@bluesky.com	943	B Tech

Write the SQL queries along with the output for the following scenarios:

(a) Display first name and e-mail id of all employees having B Tech qualification.
(b) Change the e-mail id of Mohammad Afzal to m.afzal@bluesky.com.
(c) Delete the records of employees having designation of SSE.

Solution:

(a) SQL Query:

```
SELECT first_name, email_id FROM bluesky WHERE qualification = 'Btech'
```

Output:

first_name	*email_id*
Subrata	subrata@bluesky.com
Tanmay	tanmay@bluesky.com
Mohammad	afzal@bluesky.com
Shilpa	shilpa@bluesky.com

(b) SQL Query:

```
UPDATE bluesky SET email_id = 'm.afzal@bluesky.com' WHERE emp_id = 936
```

Output:

first_name	*last_name*	*designation*	*email_id*	*emp_id*	*qualification*
.	.	.	.	.	.
Mohammad	Afzal	ME	m.afzal@bluesky.com	936	B Tech
.	.	.	.	.	.

(c) SQL Query:

```
DELETE FROM bluesky WHERE designation = 'SSE'
```

Output:

first_name	*last_name*	*designation*	*email_id*	*emp_id*	*qualification*
Neha	Sharma	SE	neha@bluesky.com	986	MCA
Yogender	Kumar	SE	yogender@bluesky.com	903	MCA
Tanmay	Mukherjee	ME	tanmay@bluesky.com	994	B Tech
Mohammad	Afzal	ME	afzal@bluesky.com	936	B Tech
Tulsi	Krishnamoorty	DM	tulsi@bluesky.com	937	M Tech
Shilpa	Sareen	SE	shilpa@bluesky.com	943	B Tech

SUMMARY

- Database is a collection of related data organized for efficient access and manipulation.
- The logical arrangement of related data items in a database is an advantage over the conventional file-oriented storage systems that store the data in discrete files.
- The architecture of a database system depicts the structure and layout of the data stored and the mechanisms used for accessing the stored data.
- A data model refers to the structure of a database system describing how data objects are arranged inside the database.
- ER model realizes all real-world objects and concepts as entities and defines relationships amongst these entities.
- A hierarchical data model arranges the data in hierarchical or tree format.
- A network data model is analogous to a graph and signifies many-to-many relationship between the various data items.
- The three-schema database architecture segregates the physical database, its logical structure and the users and applications from each other.
- Data independence is a situation where changes in the underlying storage structures do not affect the front-end application.
- A data dictionary contains information about the database itself, as well as the data elements stored in it.
- A DBA is the person responsible for the overall development and management of a database system.
- Super key is an attribute or a combination of attributes that help uniquely identify the records in a table.
- Candidate key comprises of all the super keys whose subset of attributes cannot uniquely identify rows in a table.
- Primary key is the candidate key that is actually used for unique identification of records in a table.
- An attribute or a combination of attributes in a table is considered as a foreign key if it is a primary or a candidate key in another referenced table.
- Secondary key comprises of the attributes that are not super keys and help in identifying non-unique records in a table.
- DDL is a database language used for defining the structure of a database.
- DML is a database language used for accessing and storing information in a database.

Key Terms

- redundancy
- consistency
- integrity
- data model
- ER model
- entity
- relationship
- ER diagram
- hierarchical data model
- network data model
- three-schema architecture
- meta data
- data dictionary
- DBA
- super key
- candidate key
- primary key
- foreign key
- secondary key
- DDL
- DML
- CREATE
- ALTER
- DROP
- INSERT
- UPDATE
- SELECT
- DELETE

Review Questions

13.1 *List the key points of distinction between file-oriented and database approach to data storage.*

13.2 *What is a data model? Explain its various types.*

13.3 *Why is the hierarchical data model considered inflexible?*

13.4 *Explain the generic architecture of a database system with the help of a diagram.*

13.5 *Explain the three levels of the three-schema architecture.*

13.6 *Briefly describe data independence.*

13.7 *What are the typical roles and responsibilities of a DBA?*

13.8 *What is a primary key? Explain with the help of an example.*

13.9 *What is a foreign key? Explain with the help of an example.*

13.10 *When can a super key be referred as a candidate key?*

13.11 *Explain DDL with the help of examples.*

13.12 *Explain DML with the help of examples.*

Fill in the Blanks

13.1 The DBMS approach prevents data _____ and ensures data _____.

13.2 The ER model realizes all real-world objects and concepts as _____ and defines _____ amongst them.

13.3 _____ are the characteristics that represent an entity.

13.4 A group of relationships is termed _____.

13.5 In _____ mapping cardinality, an entity in the entity set *A* is associated with only one entity in another entity set *B*.

13.6 The _____ data model follows the parent–child relationship for arranging data elements.

13.7 The _____ language is used for defining the structure of the database.

13.8 Meta data is nothing but _____ about _____.

13.9 Minimalized super keys are termed as _____ keys.

13.10 Secondary keys are used for retrieving _____ records from a database table.

13.11 The _____ SQL command is used for deleting the database objects.

13.12 The record values can be modified with the help of _____ SQL command.

Multiple Choice Questions

13.1 Which of the following can be considered as an entity as per the ER model?

A. Account Number
B. Project
C. Age Groups
D. All of the above

13.2 In an ER diagram, a relationship set is represented by which of the following shapes?

A. Oval
B. Ellipse
C. Rectangle
D. Diamond

13.3 Consider the following:

Entity Set 1	Relationship	Entity Set 2
Patients	Admitted in	Ward

Which type of mapping cardinality is being depicted here?

A. One-to-one
B. One-to-many
C. Many-to-one
D. Many-to-many

13.4 Which of the following is not one of the levels of the three-schema architecture?

A. Internal
B. External
C. Logical
D. Conceptual

13.5 Which of the following does not stand a chance of becoming a primary key?

A. Super key
B. Candidate key
C. Secondary key
D. All of the above can become primary keys

13.6 Which of the following SQL commands is used for deleting a table?

A. DELETE
B. DROP
C. REMOVE
D. ERASE

13.7 Which of the following is not an SQL command?

A. ALTER
B. DROP
C. DELETE
D. CHOOSE

Answers to Fill in the Blanks

1. redundancy, consistency
2. entities, relationships
3. Attributes
4. relationship set
5. one-to-one
6. hierarchical
7. DDL
8. data, data
9. candidate
10. non-unique
11. DROP
12. UPDATE

Answers to MCQs

1. D
2. D
3. C
4. C
5. C
6. B
7. D

14

Computer Networking

Key Concepts

- Networking
- Networking Goals
- Protocols Used
- TCP/IP Model
- OSI Model
- Internetworking Concepts
- Internetworking Devices
- Internet
- Internet Applications
- Understanding World Wide Web
- Web Browsers
- Network Security and E-Commerce

14.1 Introduction

Computers were originally developed as stand-alone, single-user systems. These systems were able to receive user's data and then process the data to generate useful information. Only a single user that is, the owner of the computer system was able to access the processed data for self use. As the use of computers spread across government offices and business organizations, a number of issues were raised.

- What if a user wants to share his computer generated information with other colleagues?
- What if a user wants to explore the possibility of using certain information stored somewhere else?

Subsequent developments in software, hardware and communication technologies addressed the above issues by enabling the computers to communicate with each other and exchange information quickly and accurately and at any time. This field of computer science that focuses on exchange of data between different computer systems is known as networking.

The advent of networks led to the mushrooming of a large number of isolated networks that could not interact with each other due to hardware and software incompatibilities. Then, a

revolutionary concept called internetworking or internet came to the fore that allowed computer systems based on different networks to interact with each other without any problems.

There are a number of standard network hardware devices as well as software protocols that enable the realization of the concept of global virtual network called the Internet. We shall study these hardware devices and protocols later in this chapter. Further, we will also study the power of Internet, its application areas and the key fundamentals of network security.

14.2 Networking Goals

The primary task of computer networking is to facilitate exchange of data between computer systems and networking devices. With this key functionality it meets a number of goals, which are:

- **Resource sharing** It enables data and other computer-related resources present at remote locations to be shared amongst the various users of the network.
- **Communication** It allows network users present at physically distant locations to instantly communicate with each other through e-mail and chat applications.
- **Load sharing** Through distributed computing, it allows computing tasks to be divided across a network of computers, thus reducing the overall processing load.
- **Reduced costs** Resources such as printers, scanners, storage disks, etc. are networked and shared by a number of users, thus leading to cost reduction.
- **Reliability and availability** It allows multiple server computers or other resources to be networked in parallel, so that if one of them goes down, the other one can fill its place.

14.3 Protocols Used for the Internet

The Internet is a collection of computers, which are connected to each other for the purpose of sharing of information. To share information, data has to be transferred from one computer to another. When data is transferred between computers, certain rules called protocols have to be followed. The protocols perform different tasks, which are arranged in a vertical stack and each task is performed at a different layer. The computer, which sends the data to another computer, is called source computer. On the other hand, the computer, which receives the data from another computer, is called destination computer. At the time of transferring of data from the source to the destination computer, a protocol is responsible for determining the network to which the destination computer belongs. It is also responsible for defining the procedure, which must be followed for dividing the data into packets. The protocols are also responsible for detection of errors in the data packets and the correction of these errors or loss of data packets. In addition, it is the responsibility of the protocols to identify loss of connection between the computers in a network and help the users to establish the connection again.

Each computer on a network has a unique address, which is known as the Internet Protocol (IP) address. An IP address is a group of four numbers and the numbers are separated from each other by a dot. When any data is sent from one computer to another computer over the

network, it is divided into small modules known as packets or datagrams. These packets are transmitted on the network by the Internet Protocol. Each packet transmitted on the network contains the addresses of both source and destination computer. A gateway present on the network reads the address of the destination computer and sends the data to the specified address. Gateway is a computer, which contains the software required for the transmission of data over different networks. Each packet on the network is an independent entity, so they are transferred through different routes to reach the destination computer. The packets received at the destination are not in the same sequence in which they were transmitted. As a result, these packets are arranged in a right sequence by a protocol known as Transmission Control Protocol (TCP) and then are merged at the destination to form the complete data. TCP and IP work in coordination with other protocols such as Telnet and User Datagram Protocol (UDP) but are considered as the most fundamental of all protocols. All these protocols are collectively known as TCP/IP suite. A model known as TCP/IP model determines how the protocols of the TCP/IP suite will work together for the transfer of data between computers in a network.

14.3.1 TCP/IP Model

The TCP/IP model was initially developed by US Defence Advanced Research Projects Agency (DARPA). This model is also known as the Internet Reference model or DoD model. It consists of four layers, namely, application layer, transport layer, network layer, and physical layer. Figure 14.1 shows the four layers of TCP/IP model.

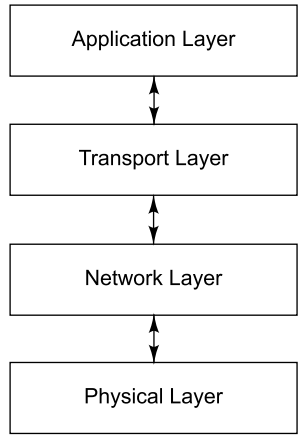


Fig. 14.1 ⇔ *Layers of TCP/IP model*

The physical layer in the TCP/IP model is responsible for interacting with the medium of transmission of data, whereas the application layer helps in interacting with the users. The four layers of TCP/IP and the functions performed by these layers are as follows:

- **Application layer** The application layer is responsible for managing all the user interface requirements. Many of the protocols, such as telnet, FTP, SMTP, DNS, NFS, LPD, SNMP and DHCP work on this layers.
- **Transport layer** The transport layer is responsible for the delivery of packets or datagrams. It also hides the packet routing details from the upper layer, i.e., the application layer. In addition, the transport layer allows detection and correction of errors and helps to achieve end-to-end communication between devices. The transport layer connects the application layer to the network layer through two protocols, namely, TCP and UDP.
- **Network layer** The network layer is also known as Internetworking layer or IP layer. It contains three protocols that perform different functions. The three protocols of Network layer are as follows:
  a*. *Internet Protocol (IP) IP is a connectionless protocol that is responsible for the delivery of packets. The IP protocol contains all the address and control information for each transmitted packet.

b. ***Internet Control Message Protocol (ICMP)*** The ICMP protocol is responsible for reporting errors, sending error messages and controlling the flow of packets. It is more reliable than the IP as it is capable of determining errors during data transmission.

c. ***Address Resolution Protocol (ARP)*** It is responsible for determining the Media Access Control (MAC) address corresponding to an IP address. It sends an ARP request on the network for a particular IP address and the device, which identifies the IP address as its own, returns an ARP reply along with its MAC address.

- **Physical layer** The physical layer is responsible for collecting packets so that the frames, which are transmitted on the network, can be formed. It performs all the functions required to transmit the data on the network and determines the ways for accessing the medium through which data will be transmitted. This layer does not contain any protocols but instead of protocols, it contains some standards such as RS-232C, V.35 and IEEE 802.3.

14.3.2 OSI Model

Open System Interconnection (OSI) is a layered design that defines the functions of the protocols used in a computer network. It consists of seven layers where each layer provides some services to the next layer. Figure 14.2 shows the seven layers of OSI model.

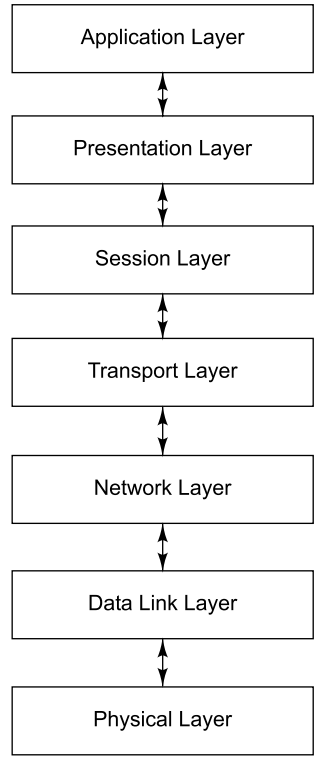


Fig. 14.2 ⇔ *OSI model*

The seven layers of the OSI model and the functions performed by them are as follows:

- **Application layer** The application layer provides an interface through which users can communicate and transfer data on the network.
- **Presentation layer** The presentation layer determines the way in which the data is presented to different computers. It converts the data into a particular format, which is supported by a specific computer. It is also responsible for encrypting, decrypting, compressing and decompressing of data.
- **Session layer** The session layer manages the communication between the computers on the network. This layer is responsible for notifying the errors, which may have occurred in the layers above it. It is also responsible for setting up and breaking the connection between the computers or devices.
- **Transport layer** The transport layer is responsible for the delivery of packets in a proper sequence. It also provides proper rectification of errors and manages the flow of packets over the network. This layer ensures that data is properly delivered at the destination. It also keeps track of all the packets, which fail to reach the destination, and transmits them again.
- **Network layer** The network layer is responsible for identifying the ways in which the data is transmitted over the network from one device to another. It prevents the overloading of packets on the network and maintains the proper flow so that all the

resources on the network can be used efficiently by all. The network layer directs the packets to the destination device on the basis of the IP address of the device. It detects the errors, which occur during the transmission of packets. This layer is also responsible for breaking the large size packets into smaller packets when the device is unable to accept the packets due to their large size.

- **Data link layer** The data link layer specifies the actions, which must be performed to maintain the network communication. It collects the packets to form frames, which are then transmitted over the network. It also finds out and then corrects the errors, which occur during the transmission of packets.
- **Physical layer** The physical layer describes all the physical requirements for the transmission of data between devices on a network. For example, the physical layer can specify the layout of pins, hubs and cables. This layer defines the relation between a single device and the transmission medium. It specifies the way in which a device must transmit data and also the way in which another device must receive data. This layer is responsible for the setting up and ending up of a network connection.

14.4 Internetworking Concepts

Internetworking is the process of connecting two or more networks with each other so that data can be exchanged between their computer systems. Thus, internetworking builds a network of networks called internetwork or simply internet. The need for internetworking arose when different isolated networks became somewhat disadvantageous due to their limited outreach. For instance, if a user working on a computer in Network A had to access a printer present on Network B, then he had to physically move to a computer system in Network B. This was obviously considered inconvenient because a task of similar nature could easily be performed in the user's own network without any physical movement.

14.4.1 Internetworking Issues

A number of issues are required to be addressed for implementing a seamless internetworking environment. Some of these issues are:

- The two networks may have different infrastructural set ups or topologies. For instance, one could be a LAN while the other could be a WAN.
- The two networks may be using different protocols for message transfer.
- The underlying hardware of the two networks may be different and incompatible.

Thus, internetworking requires a number of network incompatibilities to be resolved both at hardware as well as software levels.

14.4.2 Handling Internetworking Issues

The incompatibilities between communicating networks requires dedicated hardware devices to facilitate data transfer. Routers and gateways are two such devices that are used for internetworking purposes. A router is used for choosing the correct network path for data packets while a gateway is used for protocol translation of data packets.

The software incompatibilities between communicating devices requires some standardized protocol to be developed for intercommunication. TCP/IP is one such protocol that is widely used for Internetworking purposes.

14.5 Internetworking Devices

14.5.1 Router

As already explained, a router is used for transferring data packets across different computer networks. A router reads the address specified on a data packet to ascertain where it is heading and subsequently forwards it to its target network. There could be a large number of routers used in a dense interconnected networks system.

A router is similar to a computer having a CPU and memory of its own. It contains a number of interfaces to which different networks connect for internetworking. The interconnecting networks may be based on different technologies (Ethernet or FDDI) or have different topologies (LAN or WAN). The router simply reads the address where the incoming data packets are headed and routes them to their target network. Routers play a vital role in realizing the concept of a huge virtual network called Internet. Similarly, they are equally indispensable while catering to trivial requirements of interconnecting LANs of different departments of an organization.

14.5.2 Gateway

While a router interconnects different network types with each other, it does require all the interconnecting networks to be following the same protocol, such as TCP/IP. This limitation of a router is removed by a gateway which has the capability of interconnecting different types of networks that are based on different protocols. Thus, gateways are a step ahead of routers. While a router operates on the network, data link and physical layers of the OSI model, a gateway operates at all the seven layers.

To facilitate internetworking between cross protocol networks, a gateway is required to possess some kind of translation mechanism that can translate the incoming data packets into their respective target network's protocol. This is achieved with the help of protocol converters.

14.5.3 Networking Devices

While routers and gateways are used for the purpose of internetworking, there are several other hardware devices that are used by individual computer networks for facilitating data transmission. Some of these key networking devices are:

- **Modem** It is primarily used for connecting a computer system to the Internet. The primary objective of a modem is to convert the computer generated digital messages into analog form for their transfer through an analog communication channel. Similarly, it is also responsible for receiving the analog signals from the network and decoding them back to their digital format.

- **Hub** A hub is the common convergence point for the various devices present on a network. It exposes a large number of ports to receive data packets, which are then simply replicated to other networked computers or devices.
- **Switch** It is a more clever device than a hub and also works at higher layers in the OSI model as compared to a hub. Unlike a hub, it intelligently forwards the incoming data packet to its destination instead of replicating it across multiple ports. Thus a switch helps in preserving network bandwidth.
- **Network adapter** Also called Ethernet adapter, it is primarily used to interface a computer system with a computer network.

14.6 Internet

Internet is a popular buzzword among many people today. Almost everyone working in government offices and business organisations is using the Internet for exchange of information in one form or the other.

World Wide Web is another popular phrase among the computer users. It is commonly referred to as *the Web*. Most people consider the Internet and the World Wide Web to be synonynous, but they are not. Although these two terms are used interchangeably, they actually describe two different but related things.

The Internet is a massive network of networks that links together thousands of independent networks thus bringing millions of computers on a single network to provide a global communication system. It acts as a facilitator for exchange of information between computers that are connected to the Internet. It is like a network of roads in a country that facilitates the movement of vehicles around the country.

We can create special documents known as *hypertext documents* containing text, graphics, sounds and video on a computer. The storage location of these documents is known as *website*. The World Wide Web is the network of all such websites all around the world. It is popularly known as WWW or Web. The websites are spread across the Internet and therefore, the information contained in the websites can be transmitted through the Internet. It is like transporting the goods stored in the warehouses using the road network. So the Web is just a portion on the Internet and not same as the Internet.

14.7 Internet Applications

Nowadays the Internet is used in almost all the fields for different purposes. Each and every field uses one or the other services provided by the Internet. The Internet is extensively used in the following fields:

- Business
- Education
- Communication
- Entertainment
- Government

14.7.1 The Internet in Business

In business, the Internet can be used for many purposes. An organisation can provide details about its products on the Internet that can be either used by the other organisations interested in developing business links with it or by the prospective customers. Business transactions such as sale and purchase of products and online payment can also be performed using the Internet. This service of the Internet is called e-business, which can be further classified into the following categories:

- **Business-to-business (B2B)** B2B e-business refers to the business transactions that take place between two business organisations. In B2B, a large website acts as a market place and helps the buyers and suppliers interact at the organisational level. The website acting as a market place helps the buyers to find new suppliers and the suppliers to search for new buyers. It also saves the time and cost of interaction between the organisations. For example, a supplier business organisation can provide certain raw materials to a manufacturing business organisation through its website.
- **Business-to-consumer (B2C)** B2C e-business refers to the business transactions that take place between an organisation and a consumer directly. In B2C, a consumer can shop online for the products offered by a business organisation. It provides all the information regarding the available products through a website and allows the consumers to order and pay for the products online, thus facilitating fast and convenient shopping. For example, the Asian Sky Shop sells the various products offered by different business organisations online and any user who wants to purchase a product can buy it online.
- **Consumer-to-consumer (C2C)** C2C refers to the business transactions that take place between two consumers but with the help of a third party. In C2C, a consumer provides information about a product, which is to be sold, on the website of the third party. Another consumer can buy the same item through bidding on the website of the third party. The consumer, who provides an item for sale on the website, is known as seller; whereas the consumer, who bids for the item, is known as buyer. For example, e-bay is a website on which a consumer can provide information about the products, which s/he needs to sell. The best bidder gets to buy the listed product.
- **Consumer-to-business (C2B)** C2B is a business model that allows individual consumers to offer their products and services to companies in return of which they get payment from the companies. One of the popular examples of C2B model is the online advertising site Google Adsense. It allows individuals to display advertising content or promotional materials on their personal websites. The administration and payment of these ads are done by Google itself. Also, platforms like Google Video and Fotolia are good examples of C2B, where individuals can sell digital content including images, animations, videos, etc. to companies.

14.7.2 The Internet in Education

In the field of education, the Internet is widely used for learning and teaching. The Internet not only helps the students search information on various topics of their interest but also proves useful for the students pursuing distance education. The distance education institutes provide notes, lectures and syllabus to students through their respective websites. The students just have to access the website of the institute to get all the required information from it. If the

website of an educational institution supports e-learning, then the students can participate in online lectures through simulations, Web Based Training (WBT), etc.

The Internet also provides the Usenet service, which contains a large number of Newsgroups through which a user can submit as well as obtain the articles on different topics. The members of a newsgroup connect to each other and have discussions through the Usenet network. Usenet contains a number of message boards on which the articles are placed and the software known as newsreader is used to read the articles published on message board. Most of the newsgroups allow the users to submit their articles on the selected topics such as scientific research, social issues, religion and politics. Moreover, some newsgroups also allow the users to submit their articles on a topic of their own choice. Newsgroup not only helps the users in gaining knowledge but also allows them to make online friends.

The Internet also provides an application similar to Newsgroups known as Discussion forum. The discussion forum also allows a large number of people to hold discussions or place their articles on a particular topic similar to Newsgroups. But the only difference between the Discussion forums and Newsgroups is that the Discussion forums display articles according to the time or the thread of receiving the article. The thread refers to the grouping together of all the messages received on a particular topic. Some discussion forums allow the users to place their articles even without having a membership of the Discussion forum, while the other Discussion forums require the users to have membership along with valid username and password. The members of such Discussion forums have special facilities such as to make alterations in their previous articles, to initiate a new topic and to delete the previous articles submitted by them.

Both Newsgroup and Discussion forums are used by students and other users of the Internet to share their knowledge with each other by participating in a discussion on a specific topic. The extensive use of the Internet in education has led to the creation of what are known as Virtual Universities in many countries.

14.7.3 The Internet in Communication

The Internet is mostly used by the people as a fast and cheap means of communication. Many services provided by the Internet such as e-mail and instant messaging help the users to communicate quickly and cheaply over long distances. E-mail is an application of Internet that allows a user to send and receive text messages electronically. To use the email services, a user requires an account on a mail server. The account is created by the user by providing a username; a password and other personal information such as address and contact number. Each time the user wants to access the email service, she/he has to log on to the server using the username and the password provided during account creation. If the username or the password provided by a user is invalid, then that user is considered as an unauthorised user and is prohibited from using the service.

The Internet also provides another easy way of communication, i.e., communication through instant messaging. Instant messaging is a service of the Internet through which it is possible for a user to perform real-time communication with one or more users on the Internet. The real-time communication refers to the communication in which there is an immediate response to a message. In case of instant messaging, the communication between two users takes place

by instant sending and receiving of message. To use this service, the users have to log on to the instant messaging server. After a user has successfully logged on to the server, a chat room with a list of online users is made available to the user. An online user is a user who is available for chatting at a specific period of time. The user can then select an online user from the list and then send a message to that online user. If a response is received from the online user to whom a message was sent, then instant sending and receiving of message takes place. Chat rooms not only provide the sharing of text messages but also allow the users to share images and graphics online.

Apart from e-mail and instant messaging, Internet telephony and web conferencing are the other application areas of the Internet that facilitate, quick, cheap and efficient communication over long distances. Through these mediums, the users can talk to the other uses in real-time through an audio-visual interface.

14.7.4 The Internet in Entertainment

The Internet over the period of time has evolved as a great source of entertainment. It provides many entertainment resources to the users such as games, music and movies. The most popular entertainment resources on the Internet are the games, which are either free of cost or can be bought through the payment of a small price. Multi User Dungeon (MUD) is a virtual environment in which fantasy characters such as warriors, priests and thieves are adopted by end users for playing games. Each user represents a specific character and interacts with other characters with the help of text messages. The information regarding the game and the virtual environment is also provided to the users through commands displayed on the screen. MUD is also available with graphics that enhance the background of the game by providing it a 3-dimensional look. This feature is known as virtual reality because the background and the characters in the game resemble the real world entities.

Apart from games, the Internet also provides many other entertainment resources. Several websites provide easy access to any type of music and videos, which can be freely downloaded on the computers. The Internet also enables the users to share videos and photos with other users. Many websites on the Internet also provide information regarding the sports events taking place at specific period of time. These websites allow the users to access continuous score updates.

14.7.5 The Internet in Government

These days, the Internet is playing a crucial role in the functioning of the government organisations. Almost all the government organisations have set up their websites that provide information related to the organisation as well as help them in performing their operations. For example, nowadays people can submit the passport application form and file the income tax returns through the use of Internet. Moreover, Internet also enables the government agencies to share the data with each other.

The vast use of IT and Internet has paved the way for e-governance. More and more government agencies are adopting the concept of e-governance to improve their service delivery capabilities.

14.8 Understanding the World Wide Web

World Wide Web is a collection of web servers, which contain several web pages pertaining to different websites. The web pages contain hypertext, simple text, images, videos and graphics. The web pages are designed with the help of HyperText Markup Language (HTML). To view the web pages provided by a web server, the software known as web browser is required. To display the web pages, a web browser runs the HTML code segment written for a particular web page. Each web page on the Internet is provided its own address known as Uniform Resource Identifier (URI) or URL. This URL helps the web browser in locating a web page on the Internet. A URL string begins with the name of a protocol such as http or ftp that represents the protocol through which a web page is accessed. The rest of the URL string contains the domain server name of the web page being accessed and the location of the web page on the local web server.

14.9 Web Browsers

Web browser is the software, which is used to access the Internet and the WWW. It is basically used to access and view the web pages of the various websites available on the Internet. A web browser provides many advanced features that help achieve easy access to the Internet and WWW. When we open a web browser, the first page, which appears in the web browser window, is the home page set for that particular web browser.

The web browsers are categorised into two categories, text based and Graphical User Interface (GUI) based. The text based browsers are the browsers that display unformatted text contained in the HTML files. These types of browsers do not display images, which are inline with the text contained in the HTML files. However, the text based browsers have the ability of displaying the images that are not inline with the text contained in the HTML files. The text based browsers are simple to use and do not require computers with expensive hardware. They allow the downloading of graphic and sound files but only if the computer contains the software and the hardware required for such files. The GUI based browsers, on the other hand, display formatted text along with images, sounds and videos, which are contained in the HTML files. The user has to just click the mouse button to view or download image, sound and video files. The most commonly used web browsers are Internet Explorer (IE), Netscape Navigator and Mozilla Firefox.

The IE is the most widely used web browser that was developed in 1995 by Microsoft. The first version of IE, i.e., IE 2.0 could be installed and run on the computers with Macintosh and the 32-bit Windows operating systems. IE 2.0 was specially designed to access secure websites, and hence had the capabilities of tracing any kind of errors. To trace the errors and provide secure access to websites, IE 2.0 included a new protocol known as Secure Socket Layer (SSL) protocol. In 1996, the next version, i.e., IE 3.0 was developed, which had many advanced features, such as Internet Mail, Windows Address Book and Window Media Player. This version was basically developed for Windows 95 operating system. In 1997, the next version, IE 4.0 was developed that included Microsoft Outlook Express 4.0, which is e-mail software used for sending and receiving e-mail messages. Microsoft Outlook Express was included with IE 4.0 to provide enhanced Internet mail and news features. The latest version of IE is IE 8.0.

It is the most secure web browser as it contains many privacy and safety features as compared to any of the previous versions of IE. The IE 8.0 version supports the latest Windows operating system, i.e., Windows Vista.

To access Internet Explorer on a computer, we need to select Start→Programs→Internet Explorer. The Microsoft Internet Explorer window appears with the home page as shown in Fig. 14.3.

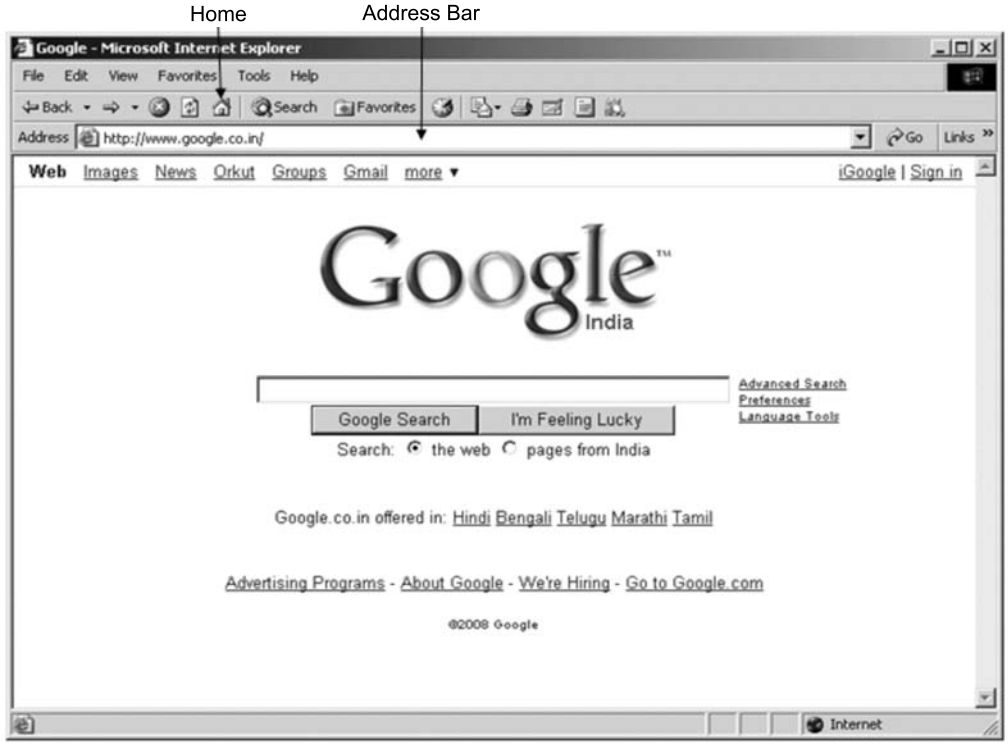


Fig. 14.3 ⇔ *Microsoft Internet Explorer window*

Figure 14.3 shows that the home page for the Internet Explorer has been set to the google.com website, which is a search engine that helps in searching information on the Internet. A user can change the home page according to the requirements using the Internet Options option of the Tools menu.

Another commonly used web browser is Netscape Navigator, which was also known as Mozilla during its development phase. This web browser was widely in use in the 1990's. The only advantage of Netscape Navigator over IE is that when a web page is being downloaded, unlike IE in which a blank screen appears some of the text and graphics contained in the web page appears in the case of Netscape Navigator. This prevents the wastage of time as the user can start reading the page even before it is completely downloaded. Initially, Netscape Navigator became very popular because of its advanced features and free availability to all the users. However, its usage declined later in 1995 when it was declared that the web browser was freely available only to the non-profit and educational organisations. Another reason for the decline

in the usage of Netscape Navigator is that it was not capable of fixing the errors automatically. On the other hand, IE 4.0 had the feature of automatically fixing errors. As a result, many people suddenly stopped using Netscape Navigator. In order to increase the usage of Netscape Navigator many new features such as mail and news reader were added to its older version. However, these new features affected the speed of the Web browser and increased its size. As a result, Netscape Navigato is rarely being used nowadays. To access Netscape Navigator, we need to select Start→ Programs→Netscape Communicator Professional Edition→Netscape Navigator. The Netscape window appears with the home page as shown in Fig. 14.4.

Fig. 14.4 ⇔ *The Netscape window with home page*

The third most commonly used web browser is the Mozilla Firefox, which was developed by Dave Hyatt and Blake Ross. Many versions of Mozilla Firefox web browser were developed before it was officially released in November 2004. The latest version of Mozilla Firefox that is currently being used is 2.0.0.14. This version includes many new features such as mail, news and HTML editing. The Firefox web browser uses XML User Interface (XUL), which supports features such as Cascade Spread Sheets (CSS) and JavaScript. XUL provides extensions and themes, which enable a user to increase the capabilities of the Mozilla Firefox web browser. Initially, the Firefox was named as m/b (Mozilla/Browser) but later its name was changed to Phoenix. The name Phoenix already existed for some BIOS software so the web browser was

renamed as Firebird but again the same problem persisted. This was already a name of a popular database server. So, in February 2004 another name, i.e., Mozilla Firefox was given to the web browser that persists till today. To access Mozilla Firefox, we need to select Start → Programs→Mozilla Firefox→Mozilla Firefox. The Mozilla Firefox window appears with the home page as shown in Fig. 14.5.

Fig. 14.5 ⇔ *Mozilla Firefox window with home page*

14.10 Network Security and Ecommerce

In today's highly networked world, network security is an important aspect that needs dedicated attention. Just as there are security threats in our real lives, there are threats while working in an internetworking environment. For example, there is a risk that somebody might impersonate you and try to withdraw funds your bank account. Similarly, there is a risk that some hacker may illegally gain access to your online bank account and siphon off funds. Just as a bank implements a number of security measures to ensure that no impostor gains access to the money saved in your bank account, there are security measures in the networked environment as well that prevent unauthorized and illegal access.

Before we learn about the various network security threats and their possible remedies, we must explore a vital application area of internetworking i.e., ecommerce. It is this aspect of Internet that requires considerable attention from the security standpoint.

14.10.1 Ecommerce

In simple words, we can define ecommerce as the buying and selling of goods or services online. While the goods are delivered through some physical distribution medium, the services may instantly be delivered online. However, in both cases the payment is made through electronic mode. The following scenarios are some of the typical instances of ecommerce:

- Buying books, groceries, etc., online
- Buying or selling shares electronically
- Transferring funds online (net banking)
- Making utility bills payment online

Ecommerce has given an altogether new dimension to how businesses are conducted these days. There is a considerably huge consumer segment that likes to buy products or services online. Thus, a business house must cater to both of these consumer segments equally.

14.10.2 Network Security Issues

Network security is important not just because internetworking involves electronic funds transfer but there are several other important security concerns that require equal attention, such as identity theft, breach of privacy, etc. The network security infrastructure and procedures must comply with the following basic requirements:

- **Confidentiality or privacy** A lot of the information being exchanged on the Internet may not be confidential, but the personal details of a user such as his name, e-mail address, credit card details, etc are confidential and must be guarded against any malicious or illegal access.
- **Integrity** The information available on the Internet must be reliable and must be prevented against any attempts of tampering. For instance, the user must be charged the same amount through his credit card as is mentioned in the product gallery or catalog.
- **Accountability or authenticity** The credentials of the user must be duly authenticated before granting him the desired access.

14.10.3 Network Security Techniques

There are several robust and logical network security techniques that help address the various network security issues. Some of these techniques are:

- **Firewall** A firewall is typically implemented to prevent unauthorized access to private networks (intranet) connected to the Internet. Implemented as a software, hardware, or a combination of both, it applies suitable security policies before allowing the requesting entity any access to a network resource. A firewall may be implemented through several different techniques, such as packet filter, application gateway, proxy server, etc.

- **Encryption** It is the process of encrypting or encoding the transmitted information in such a way that only the intended recipient having decoding information (a key) is able to decrypt and read the information. Encryption secures the transmission of user's personal data such as credit card details, e-mail address, phone number, etc, thus preventing any attempts to breach of privacy. PKI or Public Key Infrastructure is one of the fundamental techniques used to implementing encryption.
- **Digital Signature** It helps to meet the challenges of authenticity and integrity in an internetworking environment. It is like a digital stamp that makes the recipient of the digital information believe that the information was not tampered during its transmission.
- **VPN** VPN or Virtual Private Network, as the name suggests, is not a real but a virtual network that uses the public transmission medium such as the Internet along with its own security measures to realize an economical alternate to a dedicated private network. It is one of the best cost effective mediums being used by organizations world over to allow their globally dispersed regional offices to communicate and share information with each other in a secure manner.

SUMMARY

- A collection of networks in which a large number of computers are connected to each other is known as the Internet. The Internet marked its beginning with a network known as ARPAnet which was developed at Advanced Research Projects Agency (ARPA) of the U S in 1969. The first protocol used on the ARPAnet was TCP/IP. From 1975 to 1982, different scientists developed many new networks, such as Telnet, Usenet and Eunet. All these continuous developments led to the eventual development of the Internet.
- The Internet can be used to gather information on a wide variety of topics. Today the Internet is used in many fields such as business, education and entertainment. In business, the most popular use of the Internet is e-business through which an organisation and a consumer can communicate with each other and perform business transactions. In education field, the students use discussion forums and newsgroups on the Internet to gain specific information. The internet also extends its application in communication field by providing services such as e-mail and instant messaging through which a person at one location can communicate to another person located at a remote place. The Internet also provides entertainment through games, music and movies.
- To access the Internet and the WWW, the user requires a software known as web browser. Some commonly used web browsers are IE, Netscape Navigator and Mozilla Firefox. To gather information from the Internet, a user has to search for the information on the Internet. This is done with the help of search engines provided on the Internet. The most commonly used search engines are www.google.com, www.altavista.com and www.askjeeves.com.
- While the invention of computer itself was revolutionary, computer networking and particularly the Internet has added an all new dimension to it. A computer network may be defined as an arrangement of computer systems connected with each other for the purpose

of sharing data and resources. Internet is one step ahead of a network as it allows multiple computer networks to connect with each other. A number of network hardware devices such as router and gateway and standard networking protocols such as TCP/IP help realize the concept of Internet by allowing incompatible entities to communicate with each other.

⇔ Internet has entirely changed the way conventional business activities are conducted these days. Now, products or services can be bought by sitting at the comfort of our homes while the payments for the purchases can be made directly from out debit or credit card accounts. Growing use of Internet for business and personal usage has led to the need of an adequate security infrastructure that can thwart any malicious or illegal activities. This is achieved at both software and hardware levels through concepts like firewalls, encryption, digital signatures etc.

Key Terms

- Networking
- Internetworking
- OSI
- Router
- Gateway
- Ecommerce
- Firewall
- Encryption
- Digital signature
- VPN
- TCP/IP
- Internet
- World Wide Web

Review Exercises

14.1 *What is a computer network? List down its key advantages.*
14.2 *What are goals of networking?*
14.3 *Briefly explain the OSI model.*
14.4 *Elaborate upon the need of internetworking.*
14.5 *Briefly explain the various internetworking devices.*
14.6 *What is a TCP/IP model? Why is it used?*
14.7 *Given an introduction to the Internet and its various application areas.*
14.8 *What is difference between internet and world wide web?*
14.9 *What is network security? What is its significance?*
14.10 *Explain the various network security issues.*
14.11 *Briefly explain the various network security techniques.*
14.12 *What is ecommerce? What are its advantages?*

Fill in the Blanks

14.1 _____ provides the facility of information sharing and communication between users.

14.2 WWW refers to _____ _____ _____.

14.3 Each web page is accessed on a network using _____.

14.4 The methods and the rules followed to transfer data on a network are known as _____.

14.5 _____ is the unique address of a computer on the network.

14.6 _____ service of the Internet is used to perform real-time communication on the Internet.

14.7 _____ allows the users to search for some information on the Internet.

14.8 The TCP/IP model contains four layers, which are _____, _____, _____ and _____.

14.9 The two types of web browsers are _____ and _____.

14.10 An OSI model is also referred as _____.

Multiple Choice Questions

14.1 Which of the services of Internet allows the users to gather information from the Internet?

A. Email B. Discussion forums
C. WWW D. Instant messaging

14.2 What is the address of a computer on a network known as?

A. URL B. IP address C. Host D. Domain name

14.3 Which of the following tasks can be performed using the Internet?

A. Book air tickets B. Shop for clothing
C. Check the bank statement D. All of the above

14.4 At which layer of the TCP/IP model IP, ICMP and ARP protocols function?

A. Physical layer B. Application layer
C. Network layer D. Transport layer

14.5 How many layers are there in a TCP/IP model?

A. 2 layers B. 5 layers C. 7 layers D. 4 layers

14.6 Which of the following are the protocols that work at the application layer of the TCP/IP model?

A. FTP B. SMTP C. IP D. ARP

14.7 How many layers does an OSI model contain?

A. 2 layers B. 5 layers C. 7 layers D. 4 layers

14.8 Which layer in the OSI model prevents the overloading of packets on the network?

A. Data link layer
B. Application layer
C. Network layer
D. Session layer

14.9 Which of the following are the examples of a web browser?

A. Net navigator
B. Mozilla foxfire
C. IE
D. Netscape communicator

14.10 Which layer in a TCP/IP model does not contain any protocols?

A. Transport layer
B. Application layer
C. Network layer
D. Physical layer

14.11 The term ISP stands for?

A. Information System Protocol
B. Internet System Protocol
C. Internet Service Provider
D. Information Service Provider

A

C++ Operator Precedence

The Table A.1 below lists all the operators supported by ANSI C++ according to their precedence (i.e. order of evaluation). Operators listed first have higher precedence than those listed next. Operators at the same level of precedence (between horizontal lines) evaluate either left to right or right to left according to their associativity.

Table A.1 *C++ Operators*

Operator	*Meaning*	*Associativity*	*Use*
: :	global scope	right to left	::name
: :	class, namespace scope	left to right	name : : member
.	direct member	left to right	object.member
–>	indirect member		pointer->member
[ ]	subscript		pointer[expr]
( )	function call		expr(arg)
( )	type construction		type(expr)
++	postfix increment		m++
—	postfix decrement		m––
Sizeof	size of object	right to left	sizeof expr
sizeof	size of type		sizeof (type)
++	prefix increment		++m
––	prefix decrement		––m
typeid	type identification		typeid(expr)
const_cast	specialized cast		const_cast<expr>
dynamic_cast	specialized cast		dynamic_cast<expr>
reinterpret_cast	specialized cast		reinterpret_cast<expr>
static_cast	specialized cast		static_cast<expr>
( )	traditional cast		(type)expr
~	one's complement		~expr

(*Contd*)

!	logical NOT		! expr
–	unary minus		– expr
+	unary plus		+ expr
&	address of		& value
*	dereference		* expr
new	create object		new type
new [ ]	create array		new type [ ]
delete	destroy object	right to left	delete ptr
delete [ ]	destroy arrary		delete [ ] ptr
.*	member dereference	left to right	object.*ptr_to_member
–>*	indirect member dereference		ptr->*ptr_to_member
*	Multiply	left to right	expr1 * expr2
/	Divide		expr1 / expr2
%	Modulus		expr1 % expr2
+	add	left to right	expr1 + expr2
–	subtract		expr1 – expr2
<<	left shift	left to right	expr1 << expr2
>>	right shift		expr1 >> expr2
<	less than	left to right	expr1 < expr2
<=	less than or equal to		expr1 <= expr2
>	greater than		expr1 > expr2
>=	greater than or equal to		expr1 >= expr2
==	equal	left to right	expr1 == expr2
!=	not equal		expr1 != expr2
&	bitwise AND	left to right	expr1 & expr2
^	bitwise XOR	left to right	expr1 ^ expr2
|	bitwise OR	left to right	expr1 | expr2
&&	logical AND	left to right	expr1 && expr2
| |	logical OR	left to right	expr1 | | expr2
?:	conditional expression	left to right	expr1 ? expr2: expr3
=	assignment	right to left	x = expr
*=	multiply update		x *= expr
/=	divide update		x /= expr
%=	modulus update		x %= expr
+=	add update		x += expr
–=	substract update		x – = expr
<<=	left shift update		x <<= expr
>>=	right shift update		x >>= expr
&=	bitwise AND update		x &= expr
|=	bitwise OR update		x |= expr
^=	bitwise XOR update		x ^= expr
throw	throw exception	right to left	throw expr
,	comma	left to right	expr1, expr2

B

Projects

B.1 Minor Project 1: Menu Based Calculation System

Learning Objectives

The designing of the Menu Based Calculation System project will help the students to:

- Create C++ classes with static functions
- Generate and call static functions
- Use the functions of **Math.h** header file
- Develop and display the main menu and its submenus

Understanding the Menu Based Calculation System

The Menu Based Calculation System project is aimed at performing different types of calculations including normal and scientific calculations. In this project, two calculators, Standard and Scientific, are used for performing the calculations. The Standard calculator helps in performing simple calculations such as addition, multiplication, etc. while the Scientific calculator helps in performing mathematical operations such as finding the square or cube of a number.

The first screen contains a menu from which you can select the type of calculator: Standard, or Scientific. The first screen also provides the Quit option to terminate the execution of the application. Figure B.1 shows the first screen of the menu based calculation system.

To select a calculator, enter the integer corresponding to the calculator name. For instance, if you select **1**, the Standard calculator will open up, while selecting **2** will open the Scientific calculator.

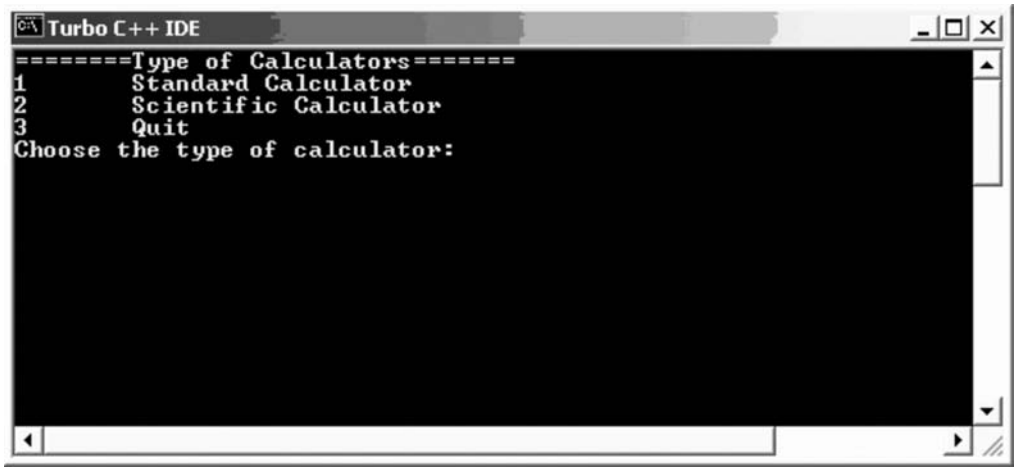


Fig. B.1

Developing the Menu based Calculation System

The code of the calculator application mainly comprises of two classes **stand_calc** and **scien_calc**. The stand_calc class helps to perform standard calculations. The scien_calc class, on the other hand, helps to perform scientific calculations. Both classes contain static functions so as to ensure that these functions can be called in the main function through class name.

Creating the stand_calc class

The stand_calc class aims at performing specific tasks related to standard calculations. These tasks are:

- Adding two numbers
- Subtracting the second number from the first number
- Multiplying two numbers
- Dividing the first number by the second number
- Modulus of the first number by the second number

To perform the above-mentioned tasks, the stand_calc class implements the following member functions:

Functions	*Description*
Addition	Returns the addition of two input numbers.
Subtraction	Returns the subtraction of two numbers accepted as input from the user.
Multiplication	Returns the multiplication of two numbers accepted as input from the user.
Division	Returns the output obtained after performing the division operation on the input numbers.
Modulus	Returns the output obtained after performing the modulus operation on the input numbers.

Creating the scien_calc class

You need to create the scien_calc class to perform tasks related to scientific calculations, which include finding the square or cube of a number, etc. The scien_calc class performs the following tasks:

- Determines the square of a number
- Determines the cube of a number
- Determines the first number to the power of the second number
- Determines the square root of a number
- Determines the factorial of a number
- Determines the value of sin, cos and tan by passing a number

To perform the above-mentioned tasks, the scien_calc class implements the following member functions:

Functions	*Description*
Square	Accepts a number and returns the square of that number
Cube	Accepts a number and returns the cube of that number
Power	Accepts two numbers and returns the first number to the power of the second number
sq_root	Accepts a number and returns its square root
Fact	Returns the factorial of an input number
sin_func	Returns the sin value of an input number
cos_func	Returns the cos value of an input number
tan_func	Returns the tan value of an input number

Calc

```
/* calc.cpp is a calculator. initially, it displays a main menu to choose the calculator
type. If a user chooses Standard calculator, then a menu appears for standard calculator
options. if a user chooses Scientific calculator, then a menu appears for scientific calculator
options and the last option is to Quit.
In standard calculator, options are to add, subtract, multiply etc. and in scientific
calculator, options are power, factorial, square root, etc.
In this program, preprocessor are defined for new calculation and old calculation. New
calculation will accept an operand whereas in old calculation, one operand is already
assumed from the result of previous calculation.
Exception handling is not implemented in this project, so do not enter a string when system
asks you for a number.
    */
    //File including and preprocessor declaration
    #include <iostream.h>
    #include <conio.h>
    #include <math.h>
    #include <stdlib.h>
    #define new_cal 1
```

```
#define old_cal 0
//stand_calc class to define standard calculator functions
class stand_calc
{
   /*Protyping of standard calculator functions. These functions are static, therefore
calling of these functions is possible with the name of the class. There is no need
to create an object of the class. */
   public:
   static double addition(double,double);
   static double subtract(double,double);
   static double multiplication(double,double);
   static double division(double ,double *);
   static double modulus(double *,double *);
};
//scien_calc class to define scientific calculator functions
class scien_calc
{
   public:
   static double square(double);
   static double cube(double);
   static double power(double,double);
   static double sq_root(double);
   static long int fact(double);
   static double sin_func(double);
   static double cos_func(double);
   static double tan_func(double);
};
//addition function will add two numbers
double stand_calc::addition(double a, double b)
{
   return(a+b);
}
//subtract function will subtract the second number from the first number
double stand_calc::subtract(double a, double b)
{
   return(a-b);
}
//multiplication function will multiply two numbers
double stand_calc::multiplication(double a, double b)
{
   return(a*b);
}
/*division function will divide the first number by the second number. This function
accepts two arguments, one is copy of a variable and another is pointer type because if
accepting divisor is zero, then this function will show a message to enter the divisor
again. Using pointer means that the entered value of the divisor for this function
should be updated at the main function also.*/
double stand_calc::division(double a, double *b)
```

```
{
   while(*b==0)
   {
          cout<<"\nCannot divide by zero.";
          cout<<"\nEnter second number again:";
          cin>>*b;
   }
   return(a/(*b));
}
/*Modulus function will divide the first number by the second number and return the
remainder part of the division. Similar to division function, it will not accept zero
in the divisor. Modulus cannot be performed on a double number, so we need to convert
it into an integer.*/
double stand_calc::modulus(double *a, double *b)
{
   while(*b==0)
   {
          cout<<"\nCannot divide by zero.";
          cout<<"\nEnter second number again:";
          cin>>*b;
   }
   //Converting double into an integer
   int x=(int)*a;
   int y=(int)*b;
   if(*a-x>0||*b-y>0)
          cout<<"\nConverting decimal number into an integer to perform modulus";
   *a=x;
   *b=y;
   return(x%y);
}
//Declaration of scien_calc class functions starts from here.
//square function of scien_calc class to return accepting number to the power 2
double scien_calc::square(double x)
{
   return(pow(x,2));
}
//cube function of scien_calc class to return accepting number to the power 3
double scien_calc::cube(double x)
{
   return(pow(x,3));
}
//power function of scien_calc class to return the first number to the power of the
second number
double scien_calc::power(double x,double y)
{
   return(pow(x,y));
}
```

```
//sq_rrot function of scien_calc class to return the square root of the entered number
double scien_calc::sq_root(double x)
{
   return(sqrt(x));
}
/*fact function of the scien_calc class to return a long integer as factorial of an
accepting number. This will convert accepting number into an integer before calculating
the factorial*/
long int scien_calc::fact(double x)
{
   int n=(int)x;
   long int f=1;
   while(n>1)
   {
          f*=n;
          n–;
   }
   return f;
}
//sin_func of the scien_calc class to return the sin value of x
double scien_calc::sin_func(double x)
{
   return(sin(x));
}
//cos_func of the scien_calc class to return the cos value of x
double scien_calc::cos_func(double x)
{
   return(cos(x));
}
//tan_func of the scien_calc class to return the tan value of x
double scien_calc::tan_func(double x)
{
   return(tan(x));
}

//Displaying the menus to enter the options and values
void main()
{
   double num1,num2,num3,temp;
   int choice1=0,choice2,flag;
   //Loop of main menu from where the program starts. It will show the menu to choose
the type of calculator.
   do
   {
          clrscr();
          cout<<"========Type of Calculators=======";
          cout<<"\n1\tStandard Calculator\n2\tScientific Calculator\n3\tQuit";
```

```
            cout<<"\nChoose the type of calculator:";
            cin>>choice1;
            flag=new_cal;
            //To perform an operation according to the entered option in the main menu
            switch(choice1)
            {
                    case 1:
                            //Loop to display the standard calculator menu
                            do
                            {
                                    clrscr();
                                    cout<<"==========Standard Calculator==========";
                                    cout<<"\n1\tAddition\n2\tSubtraction\n3\
tMultiplication\n4\tDivision\n5\tModulus\n6\tReturn to Previous Menu\n7\tQuit";
                                    //Option 8 will be displayed only when working on old
calculations. Here, already a number is saved in the calculator memory.
                                    if(flag==old_cal)
                                            cout<<"\n8\tClear Memory";
                                    cout<<"\nChoose the type of calculation:";
                                    cin>>choice2;
                                    //To perform operation and call functions of the
stand_calc class
                                    switch(choice2)
                                    {
                                            case 1:
                                                    //If a new calculation is there, then
accept the first number else previous calculation result will be the first number.
                                                    if (flag==new_cal)
                                                    {
                                                            cout<<"Enter first number:";
                                                            cin>>num1;
                                                    }
                                                    else
                                                    {
                                                            num1=temp;
                                                            cout<<"\nFirst number is
"<<num1<<endl;
                                                    }
                                                    cout<<"Enter second number:";
                                                    cin>>num2;
                                                    num3=stand_calc::addition(num1,num2);
                                                    cout<<"\nAddition of "<<num1<<" and
"<<num2<<" is "<<num3;
                                                    cout<<"\nPress any key to
continue.....";
                                                    getch();
```

```
                                                temp=num3;
                                                flag=old_cal;
                                                break;
                                        case 2:
                                                if (flag==new_cal)
                                                {
                                                        cout<<"Enter first number:";
                                                        cin>>num1;
                                                }
                                                else
                                                {
                                                        num1=temp;
                                                        cout<<"\nFirst number is
"<<num1<<endl;
                                                }
                                                cout<<"Enter second number:";
                                                cin>>num2;
num3=stand_calc::subtract(num1,num2);
                                                cout<<"\nSubtraction of "<<num2<<"
from "<<num1<<" is "<<num3;
                                                cout<<"\nPress any key to
continue.....";
                                                getch();
                                                temp=num3;
                                                flag=old_cal;
                                                break;
                                        case 3:
                                                if (flag==new_cal)
                                                {
                                                        cout<<"Enter first number:";
                                                        cin>>num1;
                                                }
                                                else
                                                {
                                                        num1=temp;
                                                        cout<<"\nFirst number is
"<<num1<<endl;
                                                }
                                                cout<<"Enter second number:";
                                                cin>>num2;
num3=stand_calc::multiplication(num1,num2);
                                                cout<<"\nMultiplication of "<<num1<<"
and "<<num2<<" is "<<num3;
                                                cout<<"\nPress any key to
continue.....";
                                                getch();
                                                temp=num3;
```

```
                                        flag=old_cal;
                                        break;
                                case 4:
                                        if (flag==new_cal)
                                        {
                                                cout<<"Enter first number:";
                                                cin>>num1;
                                        }
                                        else
                                        {
                                                num1=temp;
                                                cout<<"\nFirst number is
"<<num1<<endl;
                                        }
                                        cout<<"Enter second number:";
                                        cin>>num2;
num3=stand_calc::division(num1,&num2);
                                        cout<<"\nDivision of "<<num1<<" by
"<<num2<<" is "<<num3;
                                        cout<<"\nPress any key to
continue.....";
                                        getch();
                                        temp=num3;
                                        flag=old_cal;
                                        break;
                                case 5:
                                        if (flag==new_cal)
                                        {
                                                cout<<"Enter first number:";
                                                cin>>num1;
                                        }
                                        else
                                        {
                                                num1=temp;
                                                cout<<"\nFirst number is
"<<num1<<endl;
                                        }
                                        cout<<"Enter second number:";
                                        cin>>num2;
num3=stand_calc::modulus(&num1,&num2);
                                        cout<<"\nModulus of "<<num1<<" by
"<<num2<<" is "<<num3;
                                        cout<<"\nPress any key to
continue.....";
                                        getch();
                                        temp=num3;
                                        flag=old_cal;
```

```
                                                break;
                                        case 6:
                                                cout<<"\nReturning to previous menu.";
                                                cout<<"\nPress any key to
continue.....";
                                                getch();
                                                break;
                                        case 7:
                                                cout<<"\nQuitting.....";
                                                cout<<"\nPress any key to
continue.....";
                                                getch();
                                                exit(0);
                                        case 8:
                                                //If a new calculation is going on
then 8 is an invalid option, else 8 is an option to start a new calculation
                                                if(flag==new_cal)
                                                {
                                                        cout<<"\nInvalid choice.";
                                                        cout<<"\nPress any key to
continue.....";
                                                        getch();
                                                }
                                                else
                                                {
                                                        temp=0;
                                                        flag=new_cal;
                                                }
                                                break;
                                        default:
                                                cout<<"\nInvalid choice.";
                                                cout<<"\nPress any key to
continue.....";
                                                getch();
                                                break;
                                }
                        }while (choice2!=6);
                        break;
                case 2:
                        //Loop to display scientific calculator menu
                        do
                        {
                                clrscr();
                                cout<<"==========Scientific Calculator==========";
cout<<"\n1\tSquare\n2\tCube\n3\tPower\n4\tFactorial\n5\tSin\n6\tCos\n7\tTan\n8\tReturn
to previous menu\n9\tQuit";
                                if(flag==old_cal)
```

```
                cout<<"\n10\tClear Memory";
            cout<<"\nChoose the type of calculation:";
            cin>>choice2;
            switch(choice2)
            {
                case 1:
                    if (flag==new_cal)
                    {
                        cout<<"Enter number to find
square:";
                        cin>>num1;
                    }
                    else
                    {
                        num1=temp;
                        cout<<"\nNumber is
"<<num1<<endl;
                    }
                    num3=scien_calc::square(num1);
                    cout<<"\nSquare of "<<num1<<" is
"<<num3;
                    cout<<"\nPress any key to
continue.....";
                    getch();
                    temp=num3;
                    flag=old_cal;
                    break;
                case 2:
                    if (flag==new_cal)
                    {
                        cout<<"Enter number to find
cube:";
                        cin>>num1;
                    }
                    else
                    {
                        num1=temp;
                        cout<<"\nNumber is
"<<num1<<endl;
                    }
                    num3=scien_calc::cube(num1);
                    cout<<"\nCube of "<<num1<<" is
"<<num3;
                    cout<<"\nPress any key to
continue.....";
                    getch();
                    temp=num3;
```

```
                                                flag=old_cal;
                                                break;
                                        case 3:
                                                if (flag==new_cal)
                                                {
                                                        cout<<"Enter first number for
base to find power:";
                                                        cin>>num1;
                                                }
                                                else
                                                {
                                                        num1=temp;
                                                        cout<<"\nFirst number is
"<<num1<<endl;
                                                }
                                                cout<<"Enter second number for power
to find power:";
                                                cin>>num2;
                                                   num3=scien_calc::power(num1,num2);
                                                cout<<"\n"<<num1<<" to the power
"<<num2<<" is "<<num3;
                                                cout<<"\nPress any key to
continue.....";
                                                getch();
                                                temp=num3;
                                                flag=old_cal;
                                                break;
                                        case 4:
                                                if (flag==new_cal)
                                                {
                                                        cout<<"Enter number to find
factorial:";
                                                        cin>>num1;
                                                }
                                                else
                                                {
                                                        num1=temp;
                                                        cout<<"\nNumber to
find factorial is "<<num1<<endl;
                                                }
                                                long int num4=scien_calc::fact(num1);
                                                cout<<"\nFactorial of "<<num1<<" is
"<<num4;
                                                cout<<"\nPress any key to
continue.....";
                                                getch();
                                                temp=num4;
```

```
                                        flag=old_cal;
                                        break;
                                case 5:
                                        if (flag==new_cal)
                                        {
                                                cout<<"Enter number to find
sin value:";
                                                cin>>num1;
                                        }
                                        else
                                        {
                                                num1=temp;
                                                cout<<"\nNumber for sin value
is "<<num1<<endl;
                                        }
                                        num3=scien_calc::sin_func(num1);
                                        cout<<"\nSin value of "<<num1<<" is
"<<num3;
                                        cout<<"\nPress any key to
continue.....";
                                        getch();
                                        temp=num3;
                                        flag=old_cal;
                                        break;
                                case 6:
                                        if (flag==new_cal)
                                        {
                                                cout<<"Enter number to find
cos value:";
                                                cin>>num1;
                                        }
                                        else
                                        {
                                                num1=temp;
                                                cout<<"\nNumber for cos value
is "<<num1<<endl;
                                        }
                                        num3=scien_calc::cos_func(num1);
                                        cout<<"\nCos value of "<<num1<<" is
"<<num3;
                                        cout<<"\nPress any key to
continue.....";
                                        getch();
                                        temp=num3;
                                        flag=old_cal;
                                        break;
```

```
                                        case 7:
                                                if (flag==new_cal)
                                                {
                                                        cout<<"Enter number to find
tan value:";
                                                        cin>>num1;
                                                }
                                                else
                                                {
                                                        num1=temp;
                                                        cout<<"\nNumber for tan value
is "<<num1<<endl;
                                                }
                                                num3=scien_calc::tan_func(num1);
                                                cout<<"\nTan value of "<<num1<<" is
"<<num3;
                                                cout<<"\nPress any key to
continue.....";
                                                getch();
                                                temp=num3;
                                                flag=old_cal;
                                                break;
                                        case 8:
                                                cout<<"\nReturning to previous menu.";
                                                cout<<"\nPress any key to
continue.....";
                                                getch();
                                                break;
                                        case 9:
                                                cout<<"\nQuitting.....";
                                                cout<<"\nPress any key to
continue.....";
                                                getch();
                                                exit(0);
                                        case 10:
                                                if(flag==new_cal)
                                                {
                                                        cout<<"\nInvalid choice.";
                                                        cout<<"\nPress any key to
continue.....";
                                                        getch();
                                                }
                                                else
                                                {
                                                        temp=0;
                                                        flag=new_cal;
```

```
                                                }
                                                break;
                                        default:
                                                cout<<"\nInvalid choice.";
                                                cout<<"\nPress any key to
continue.............";
                                                getch();
                                                break;
                                        }
                                }while (choice2!=8);
                                break;
                        case 3:
                                cout<<"\nQuitting......";
                                cout<<"\nPress any key to continue........";
                                getch();
                                break;
                        default:
                                cout<<"\nInvalid Choice.";
                                cout<<"\nPress any key to continue........";
                                getch();
                                break;
                }
        }while (choice1!=3);
}
```

B.2 Major Project 1: Banking System

Learning Objectives

The designing of the Banking System project helps the students to:

- Create C++ classes and call the functions declared in the classes
- Develop and display main menu and its submenus
- Change the menu options during runtime
- Programmatically create files using File System objects
- Perform file transactions such as Updation, Deletion and Display from files
- Use **iomanip** header file in C++ to display formatted output of data using **setw()** function for setting width of the text to be displayed.

Understanding the Banking System Project

The Banking System application helps maintain the data related to the customers and performs the typical banking transactions. In the Banking System application, **FileSystem Object** is used to create two data files, which have the extension .dat. The data related to a customer is stored in the newrecords.dat data file. The data related to transactions, such as withdrawal

and deposit are stored in the transaction.dat data file. The following figure shows the main menu of the Banking System application:

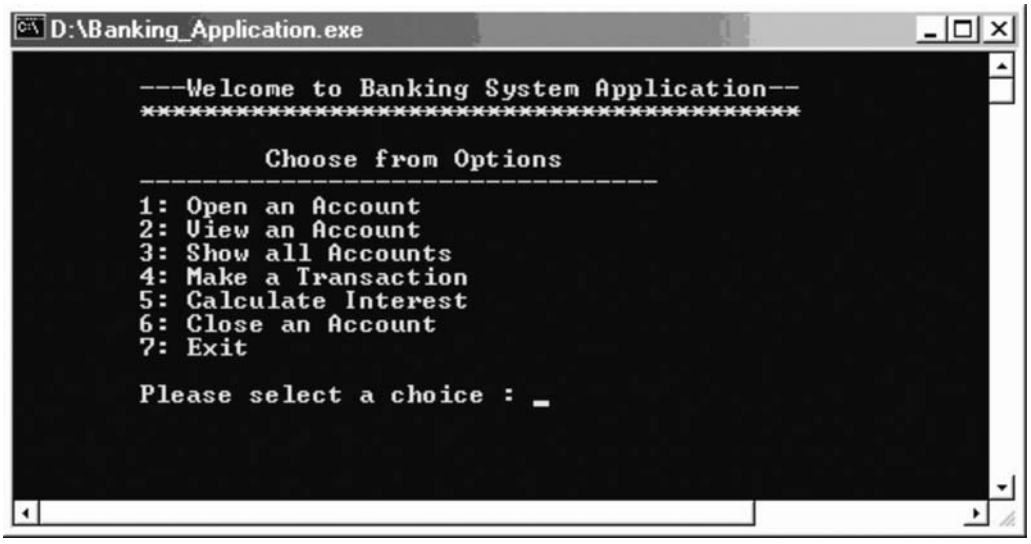


Fig. B.2

Developing the Banking System

The development of Banking System application involves the creation of the following classes:

- menus
- dispRecords
- accountTransactions

Creating the menus Class

You need to create the class **menus** to implement the functionality of displaying a main menu and closing the account before quitting the Banking System application. To create the class menus, you need to define the member functions, **showmenu** and **closemenu** and the variables required for displaying the main menu of the Banking System application. The showmenu member function helps to display the main menu to the users of the Banking System application. The closemenu member function helps to display the Closing Account menu when the user selects the Close an Account option from the main menu.

Creating the dispRecords Class

You need to create the **dispRecords** class to implement the functionality of displaying the information related to the customers of a bank and their accounts. In the dispRecords class, data related to customers is retrieved from the newrecords.dat data file for displaying customer

information or adding and closing of customer accounts. You can create the dispRecords class by defining the variables required for displaying customer and account information and the member functions such as **displayCustomer** and **deleteAccount**. The following table lists the member functions that need to be defined in the class dispRecords:

Functions	*Descriptions*
addDetails(int, char name[30], char address[60], float)	Adds the information related to a new customer of the bank who becomes an account holder.
displayCustomers(void)	Displays a list of all the account holders of the bank along with their account numbers and balance.
deleteAccount(int)	Deletes the information related to the account holder from the newrecords.dat data file.
updateBalance(int, float)	Updates the balance after a customer has performed a deposit or withdrawal transaction.
lastAccount()	Displays the account number of the last entry.
accountExists(int)	Checks whether an account exists or not.
getName(int)	Retrieves the name of the account holder.
getAddress(int)	Retrieves the address of the account holder.
getBalance(int)	Retrieves the balance of the account holder.
getRecord(int)	Returns the record number from the newrecords.dat data file when an employee of the bank enters the account number related to an account holder.
display(int)	Displays all the information related to an account holder from the newrecords.dat file on the basis of specified account number.

Creating the accountTransactions Class

You need to create the **accountTransactions** class so that transactions related to an account can be performed. The data related to the transactions are stored in the transaction.dat data file. The accountTransactions class also uses some member functions defined in the dispRecords class. In the class accountTransactions, the Object Oriented Programming (OOP) concepts of Polymorphism are used to manipulate data, which need to be stored in the transaction.dat data file. You can create the accountTransactions class by defining variables and member functions, which include **new_account** and **showAccount**. The following table lists the member functions of the accountTransactions class:

Functions	*Descriptions*
new_account(void)	Validates the information related to a new customer and adds the information to the transaction.dat data file using the addDetails member function.
closeAccount()	Closes the account of an account holder after verifying the account number.
showAccount(int)	Displays the headings Customer Name, Deposit and Withdrawal, Interest and Balance.

display_account(void)	Displays the data related to a specific account holder.
deleteAccount(int)	Deletes the data related to a transaction from the transaction.dat data file on the basis of the account number of that account holder.
transaction(void)	Helps to perform deposit and withdrawal transactions.
dateDiffer(int, int, int, int, int, int)	Checks the current and account creation dates. If the account in the bank has completed one year, then interest for that account is calculated.
getInterest(int, float)	Generates interest when one year has completed for a particular account.
showInterest(void)	Displays the interest generated using the getInterest member function. The showInterest member function also helps to update the balance of the account holder.

Banking_Application

```
/** A Banking System with normal transactions **/
   #include <iostream.h>
   #include <fstream.h>
   #include <process.h>
   #include <string.h>
   #include <stdlib.h>
   #include <stdio.h>
   #include <ctype.h>
   #include <conio.h>
   #include <dos.h>
   #include <iomanip.h>

   // The Menus Class displays the Menu
   class Menus
   {
      public :
                    void showmenu(void) ;
      private :
                    void closemenu(void) ;
   };

   // The Class displays all the Customer Account related functions
   class dispRecords
   {
      public :
                    void addDetails(int, char name[30], char address[60], float) ;
                    void displayCustomers(void) ;
                    void deleteAccount(int) ;
                    void updateBalance(int, float) ;
                    void updateCustomer(void) ;
```

```
                    int  lastAccount(void) ;
                    int  accountExists(int) ;
                    char *getName(int);
                    char *getAddress(int);
                    float getBalance(int) ;
                    int  getRecord(int) ;
                    void display(int) ;
                    void  displayList(void) ;
                    int   AccountNumber ;
                    char  name[50], address[50] ;
                    float intBalance ;
};

// The Class has all the transaction related methods
class accountTransactions
{
   public :
                    void new_account(void);
                    void closeAccount(void);
                    void display_account(void);
                    void transaction(void);
                    void  addDetails(int, int, int, int, char, char typeTransaction[15],
float, float, float);
                    void  deleteAccount(int);
                    int   dateDiffer(int, int, int, int, int, int);
                    float getInterest(int, float);
                    void  display(int);
                    void  showAccount(int);
                    int   AccountNumber;  //variable for Account Number
                    char  trantype[10];  // variable of cheque or cash input or output
                    int   dday, mmonth, yyear;  // transaction date
                    char  transactions;         // type of transactions - Deposit or
Withdrawal of Amount
                    float intInterest, intAmount, intBalance;
                    static float calcInterest;
                    void showInterest(void);//added
};

// showmenu() method to display the Main Menu in the application
void Menus :: showmenu(void)
{
   char choice;
   while (1)
   {
          clrscr();
          cout<<"\n      --Welcome to Banking System Application--     \n";
          cout<<" ****************************************\n\n";
          cout<<"        Choose from Options \n";
```

```
            cout<<" ———————————— \n";
            cout <<"       1: Open an Account\n";
            cout <<"       2: View an Account     \n";
            cout <<"       3: Show all Accounts   \n";
            cout <<"       4: Make a Transaction  \n";
            cout <<"       5: Calculate Interest\n";
            cout <<"       6: Close an Account\n";
            cout <<"       7: Exit\n\n";
            cout <<"       Please select a choice : ";
            choice = getche();

                    if (choice == '1')
                    {
                            accountTransactions objAT;
                            objAT.new_account();
                    }
                    else
                    if (choice == '2')
                    {
                            accountTransactions objAT;
                            objAT.display_account();
                    }
                    else
                    if (choice == '3')
                    {
                            dispRecords newRec;
                            newRec.displayCustomers();
                    }
                    else
                    if (choice == '4')
                    {
                            accountTransactions objAT;
                            objAT.transaction();
                    }
                    else
                    if (choice == '5')
                    {
                            accountTransactions objAT;
                            objAT.showInterest();
                    }
                    else
                    if (choice == '6') {
                            closemenu();
                    }
                    else
                    if (choice == '7') {
                            cout<<"\n      Thanks for using this application. Please
press any key to exit.\n";
```

```
                        getch();
                        break ;
                }
        }
}

// closemenu() method displays the Closing of the Account of the Customer in the
Application
void Menus :: closemenu(void)
{
   char choice;
   while (1)
   {
        clrscr() ;
        cout<<"         –Close Menu– \n" ;
        cout<<"         ********************* \n" ;
        cout <<"        1: Close/Delete an Account\n" ;
        cout <<"        0: Exit from this menu\n\n" ;
        cout <<"        Select a Choice: " ;
        choice = getche() ;

                if (choice == '1')
                {
                        accountTransactions at ;
                        at.closeAccount() ;
                        break ;
                }
                else
                if (choice == '0')
                {
                        cout<<"\n       You have entered 0 to go back to the previous
Menu. \n";
                        getch();
                        break ;
                }
        }
}

// lastAccount() method returns the Last Account Number from the newrecords.dat file
int dispRecords :: lastAccount(void)
{
   fstream filename ;
   filename.open("newrecords.dat", ios::in) ;
   filename.seekg(0,ios::beg) ;
   int count=0 ;
   while (filename.read((char *) this, sizeof(dispRecords)))
        count = AccountNumber ;
   filename.close() ;
```

```
   return count ;
}

// getRecord() method returns the record number from the newrecords.dat file when a
banking staff enters the Account Number
int dispRecords :: getRecord(int retrieve_AccNo)
{
   fstream filename;
   filename.open("newrecords.dat", ios::in) ;
   filename.seekg(0,ios::beg) ;
   int count=0 ;
   while (filename.read((char *) this, sizeof(dispRecords)))
   {
          count++ ;
          if (retrieve_AccNo == AccountNumber)
                 break ;
          /*keep on counting the record till the Account Number is found and exit from
the newrecords.dat file. */
   }
   filename.close() ;
   return count ;
}

// display() method displays all the details of the Account Number from the newrecords.
dat file
void dispRecords :: display(int retrieve_AccNo)
{
   int record ;
   record = getRecord(retrieve_AccNo) ;
   fstream filename ;
   filename.open("newrecords.dat", ios::in);
   filename.seekg(0,ios::end);
   int location;
   location = (record) * sizeof(dispRecords);
   filename.seekp(location);

   while (filename.read((char *) this, sizeof(dispRecords)))
   {
          if (retrieve_AccNo == AccountNumber)
          {
                 cout <<"\n     ACCOUNT NO. : " <<AccountNumber ;
                 cout <<"\n     Name    : "<<name ;
                 cout <<"\n     Address : " <<address ;
                 cout <<"\n     Balance : " <<intBalance ;
                 break ;
          }
   }
   filename.close() ;
```

```
}

// getName() method returns the Account Holder's Name from the newrecords.dat file
char *dispRecords :: getName(int retrieve_AccNo)
{
   fstream filename;
   filename.open("newrecords.dat", ios::in);
   filename.seekg(0,ios::beg);
   char retrieve_CustName[30];

          while (filename.read((char *) this, sizeof(dispRecords)))
          {
                 if (AccountNumber == retrieve_AccNo)
                 {
                        strcpy(retrieve_CustName,name);
                 }
          }
          filename.close();
          return retrieve_CustName;
}

// getAddress() method returns the Address of the Account Holder from the newrecords.
dat file
char *dispRecords :: getAddress(int retrieve_AccNo)
{
   fstream filename;
   filename.open("newrecords.dat", ios::in);
   filename.seekg(0,ios::beg);
   char retrieve_Address[60];

   while (filename.read((char *) this, sizeof(dispRecords)))
   {
          if (AccountNumber == retrieve_AccNo)
          {
                 strcpy(retrieve_Address,address);
          }
   }
   filename.close();
   return retrieve_Address;

}

/* getBalance() method returns the Balance of the Account Holder from the newrecords.
dat file*/
float dispRecords :: getBalance(int retrieve_AccNo)
{
   fstream filename ;
   filename.open("newrecords.dat", ios::in);
```

```
    filename.seekg(0,ios::beg);
    float iBalance ;
    while (filename.read((char *) this, sizeof(dispRecords)))
    {
            if (AccountNumber == retrieve_AccNo)
            {
                    iBalance = intBalance;
            }
    }
    filename.close();
    return iBalance;
}

// accountExists() method checks if the Account exists in the newrecords.dat file or
not.
int dispRecords :: accountExists(int retrieve_AccNo)
{
    fstream filename ;
    filename.open("newrecords.dat", ios::in);
    filename.seekg(0,ios::beg) ;
    int count=0 ;
    while (filename.read((char *) this, sizeof(dispRecords)))
    {
            if (AccountNumber == retrieve_AccNo)
            {
                    count = 1;
                    break;
            }
    }
    filename.close();
    return count;
}

/* displayList() method displays the output of all the Accounts in a proper format
for the Choice 3*/
void dispRecords :: displayList()
{
    cout<<"
                                    \n" ;
    int day1, month1, year1 ;
    struct date dateval;
    getdate(&dateval);
    day1 = dateval.da_day ;
    month1 = dateval.da_mon ;
    year1 = dateval.da_year ;
    cout <<"\n Date: " <<day1 <<"/" <<month1 <<"/" <<year1<<"\n";
    cout<<setw(80)<<"————————————————————————————————————\n";
    cout<<setw(23)<<" ACCOUNT NO.";
```

```
   cout<<setw(23)<<" NAME OF PERSON";
   cout<<setw(23)<<" BALANCE\n";
   cout<<setw(80)<<"--------------------------------------------------\n";
}

// displayCustomers() method displays all the Account Holders/Customers from the
newrecords.dat file
void dispRecords :: displayCustomers(void)
{
   clrscr() ;
   int len1;
   int row=8, check ;
   fstream filename ;

   FILE * pFile;
   pFile = fopen("newrecords.dat","r");
   if (pFile == NULL)
   {
       cout<<"\n  No Account exists. Please go back to the previous menu. \n";
          getch();
          return ;
          //fclose (pFile);

   } else {
          displayList();
          filename.open("newrecords.dat", ios::in);
          filename.seekg(0,ios::beg);
          while (filename.read((char *) this, sizeof(dispRecords)))
          {
                  check = 0 ;

                  cout.fill(' ');
                  cout <<setw(20);
                  cout.setf(ios::right,ios::adjustfield);
                  cout<<AccountNumber;
                  cout.fill(' ');
                  cout <<setw(25);
                  cout.setf(ios::internal,ios::adjustfield);
                  cout<<name;

                  cout <<setw(23);
                  cout.setf(ios::right,ios::adjustfield);
                  cout<<intBalance<<"\n" ;
                  row++ ;
                  if (row == 23)
                  {
                          check = 1 ;
                          row = 8 ;
```

```
                        cout <<" \n\n Continue the application... \n";
                        getch() ;
                        clrscr() ;
                        displayList() ;
                }
        }
    }
    filename.close() ;
    if (!check)
    {
            cout <<"\n\n Continue the application... \n";
            getch() ;
    }
}

// addDetails() method adds new records of Account Holders/Customers in the newrecords.
dat file
void dispRecords :: addDetails(int retrieve_AccNo, char retrieve_CustName[30], char
retrieve_Address[60], float iBalance)
{
    AccountNumber = retrieve_AccNo ;
    strcpy(name,retrieve_CustName) ;
    strcpy(address,retrieve_Address) ;
    intBalance = iBalance ;
    fstream filename ;
    filename.open("newrecords.dat", ios::out | ios::app) ;
    filename.write((char *) this, sizeof(dispRecords)) ;
    filename.close() ;
}

// deleteAccount() method deletes the particular record from the newrecords.dat file
on the basis of the Account Number.
void dispRecords :: deleteAccount(int retrieve_AccNo)
{
    fstream filename ;
    filename.open("newrecords.dat", ios::in) ;
    fstream temp ;
    temp.open("calculations.txt", ios::out) ;
    filename.seekg(0,ios::beg) ;
    while ( !filename.eof() )
    {
            filename.read((char *) this, sizeof(dispRecords)) ;
            if ( filename.eof() )
                    break ;
            if ( AccountNumber != retrieve_AccNo )
                    temp.write((char *) this, sizeof(dispRecords)) ;
    }
    filename.close() ;
```

```
    temp.close() ;
    filename.open("newrecords.dat", ios::out) ;
    temp.open("calculations.txt", ios::in) ;
    temp.seekg(0,ios::beg) ;
    while ( !temp.eof() )
    {
            temp.read((char *) this, sizeof(dispRecords)) ;
            if ( temp.eof() )
                    break ;
            filename.write((char *) this, sizeof(dispRecords)) ;
    }
    filename.close() ;
    temp.close() ;
}

// updateBalance() method updates the balance of the Account Number after a transaction
is done in the newrecords.dat file
void dispRecords :: updateBalance(int retrieve_AccNo, float iBalance)
{
    int record ;
    record = getRecord(retrieve_AccNo) ;
    fstream filename ;
    filename.open("newrecords.dat", ios::out | ios::ate) ;
    intBalance = iBalance ;
    int location ;
    location = (record-1) * sizeof(dispRecords) ;
    filename.seekp(location) ;
    filename.write((char *) this, sizeof(dispRecords)) ;
    filename.close() ;
}

// addDetails() method adds the details of a transaction in the transactions.dat file
void accountTransactions :: addDetails(int retrieve_AccNo, int day1, int month1, int
year1, char t_tran, char typeTransaction[10], float interest_accrued, float t_amount,
float iBalance)
{
    fstream filename ;
    filename.open("transactions.dat", ios::app) ;
    AccountNumber = retrieve_AccNo ;
    dday = day1 ;
    mmonth = month1 ;
    yyear = year1 ;
    transactions = t_tran ;
    strcpy(trantype,typeTransaction) ;
    intInterest = interest_accrued ;
    intAmount = t_amount ;
    intBalance = iBalance ;
    filename.write((char *) this, sizeof(accountTransactions)) ;
```

```
    filename.close();
}

// deleteAccount() method deletes the record of a transaction from the transactions.
dat file
void accountTransactions :: deleteAccount(int retrieve_AccNo)
{
    fstream filename ;
    filename.open("transactions.dat", ios::in) ;
    fstream temp ;
    temp.open("calculations.txt", ios::out) ;
    filename.seekg(0,ios::beg) ;
    while ( !filename.eof() )
    {
            filename.read((char *) this, sizeof(accountTransactions)) ;
            if ( filename.eof() )
                    break ;
            if ( AccountNumber != retrieve_AccNo )
                    temp.write((char *) this, sizeof(accountTransactions)) ;
    }
    filename.close() ;
    temp.close() ;
    filename.open("transactions.dat", ios::out) ;
    temp.open("calculations.txt", ios::in) ;
    temp.seekg(0,ios::beg) ;
    while ( !temp.eof() )
    {
            temp.read((char *) this, sizeof(accountTransactions)) ;
            if ( temp.eof() )
                    break ;
            filename.write((char *) this, sizeof(accountTransactions)) ;
    }
    filename.close() ;
    temp.close() ;
}

// new_account() method adds a new record in the newrecords file and transaction.dat
files(choice 1)
void accountTransactions :: new_account(void)
{
    char choice ;
    int i, check ;
    clrscr() ;
    dispRecords newRec ;
    cout <<"        Please press 0 to go back to previous menu. \n" ;
    cout<<"                                                    \n";
    cout<<"        -Open a New Bank Account-      \n";
    cout<<"        ********************   \n";
```

```
    int day1, month1, year1 ;
    struct date dateval;
    getdate(&dateval);
    day1 = dateval.da_day ;
    month1 = dateval.da_mon ;
    year1 = dateval.da_year ;
    int retrieve_AccNo ;
    retrieve_AccNo = newRec.lastAccount() ;
    retrieve_AccNo++ ;

    if (retrieve_AccNo == 1)
    {
            newRec.addDetails(retrieve_AccNo,"Ravi","Delhi",1.1) ;
            newRec.deleteAccount(retrieve_AccNo) ;
            addDetails(retrieve_AccNo,1,1,1997,'D',"default value",1.1,1.1,1.1) ;
            deleteAccount(retrieve_AccNo) ;
    }
    char retrieve_CustName[30], tran_acc[10], retrieve_Address[60] ;
    float t_bal, iBalance ;
    cout <<"        Date : "<<day1 <<"/" <<month1 <<"/" <<year1<<"\n" ;
    cout <<"        Account no. # " <<retrieve_AccNo;
    do
    {
            cout <<"\n\n   Please enter the Name of the Account Holder    : ";
            check = 1;
            gets(retrieve_CustName);
            if (retrieve_CustName[0] == '0')
            {
                    cout<<"\n\t    Invalid Customer Name.";
                    getch();
                    return;
            }
            strupr(retrieve_CustName);
            if (strlen(retrieve_CustName) == 0 || strlen(retrieve_CustName) > 30)
            {
                    check = 0;
                    cout<<"\t\n    Customer Name is either blank or its length is greater
than 30 characters.\n";
                    getch();
            }
    } while (!check);

    do
    {
            cout <<"\n     Please enter the Account Holder's Address : ";
            check = 1;
            gets(retrieve_Address);
            if (retrieve_Address[0] == '0')
            {
```

```
                cout<<"\n\t    Invalid Customer Address.";
                getch();
                return;
        }
        strupr(retrieve_Address);
        if (strlen(retrieve_Address) < 1 || strlen(retrieve_Address) > 60)
        {
                check = 0 ;
                cout<<"\t\n    Customer Address is either blank or its length is
greater than 60 characters.\n" ;
                getch() ;
        }
  } while (!check) ;

  do
  {
        char chr_VerifyingPerson[30] ;
        cout <<"\n     Please enter the Name of the Verifying Person of the Account
Holder : ";
        check = 1 ;
        gets(chr_VerifyingPerson);
        if (chr_VerifyingPerson[0] == '0')
        {
                cout<<"\n\t    Invalid Verifying Person Name.";
                getch();
                return;
        }
        strupr(chr_VerifyingPerson) ;
        if (strlen(chr_VerifyingPerson) < 1 || strlen(chr_VerifyingPerson) > 30)
        {
                check = 0 ;
                cout<<"\t\n    The Verifying Person's Name is either blank or greater
than 30 characters. Please try again.\n";
                getch() ;
        }
  } while (!check) ;

  do
  {
        cout <<"\n  Please enter the Deposit Amount while opening a New Account : ";
        check = 1 ;
        gets(tran_acc) ;
        t_bal = atof(tran_acc) ;
        iBalance = t_bal ;
        if (strlen(tran_acc) < 1) {
                cout<<"\n    Invalid Transaction value. Exiting from the current
Menu.\n ";
                getch();
                return ;
```

```
            }
            if (iBalance < 1000)
            {
                    check = 0 ;
                    cout<<"\t\n    The Minimum Deposit Amount should be Rs.1000. Please
try again. \n";
                    getch() ;
            }

    } while (!check) ;

    do
    {
            cout <<"\n     Do you want to save the record? (y/n) : " ;
            choice = getche() ;
            choice = toupper(choice) ;
    } while (choice != 'N' && choice != 'Y') ;
    if (choice == 'N' || choice == 'n')
    {
            cout<<"\n      The Customer Account is not created\n.
Please continue with the application.\n";
            getch();
            return ;
    }
    float t_amount, interest_accrued ;
    t_amount = iBalance ;
    interest_accrued = 0.0 ;
    char t_tran, typeTransaction[10] ;
    t_tran = 'D' ;
    strcpy(typeTransaction," ") ;

    newRec.addDetails(retrieve_AccNo,retrieve_CustName,retrieve_Address,iBalance) ;
    addDetails(retrieve_AccNo,day1,month1,year1,t_tran,typeTransaction,
interest_accrued,t_amount,iBalance);
    cout<<" \n\n           The New Account is successfully created.\n
Please continue with the application.\n";
    getch();
}

// showAccount() method formats the display of the records from the transactions.dat
file for a particular account(choice 2).
void accountTransactions :: showAccount(int retrieve_AccNo)
{
    cout<<"
                                                    \n";
    int day1, month1, year1 ;
    struct date dateval;
    getdate(&dateval);
```

```
    day1 = dateval.da_day ;
    month1 = dateval.da_mon ;
    year1 = dateval.da_year ;
    cout<<"Date: " <<day1 <<"/" <<month1 <<"/" <<year1<<"\n" ;
    cout <<"Account no. " <<retrieve_AccNo ;
    dispRecords newRec ;

    char retrieve_CustName[30] ;
    strcpy(retrieve_CustName,newRec.getName(retrieve_AccNo)) ;
    char retrieve_Address[60] ;
    strcpy(retrieve_Address,newRec.getAddress(retrieve_AccNo)) ;

    cout<<setw(25)<<"\n Account Holder's Name :   "<<retrieve_CustName;
    cout<<"\nAddress               :       "<<retrieve_Address<<"\n";
    cout<<setw(80)<<"\n--------------------------------------\n";
    cout<<setw(10)<<"Dated";
    cout<<setw(12)<<"Details";
    cout<<setw(12)<<"Deposited";
    cout<<setw(15)<<"Withdrawn";
    cout<<setw(12)<<"        ";
    cout<<setw(10)<<"Balance";
    cout<<setw(80)<<"\n--------------------------------------\n";
}

// display_account() method displays records from the transactions.dat file
void accountTransactions :: display_account(void)
{
    clrscr() ;
    char t_acc[10] ;
    int tran_acc, retrieve_AccNo;
    dispRecords obj2;
    cout <<"       Press 0 to go back to previous menu.\n" ;
    cout <<"       Please enter Account No. you want to view :   " ;
    gets(t_acc);
    tran_acc = atoi(t_acc);       /* converting Account Number to integer value */
    retrieve_AccNo = tran_acc;
    if (retrieve_AccNo == 0){
           cout<<"\n      You have pressed 0 to exit. \n";
           getch();
           return ;
    }
    clrscr();
    dispRecords newRec;
    accountTransactions aa;
    int row=8, check ;
    fstream filename ;

    FILE * pFile;
```

```
    pFile = fopen("newrecords.dat","r");
    if (pFile == NULL)
    {
        cout<<"\n  No such Account Exists. Please create a New Account. \n";
            getch();
            return ;

    } else if (!newRec.accountExists(retrieve_AccNo)) {
            cout <<"\t\n   Account does not exist.\n";
            getch();
            return;
    } else {

            showAccount(retrieve_AccNo) ;
            filename.open("transactions.dat", ios::in);
            /* Reading the transaction.dat file and displaying the details of a particular
Account */
            while (filename.read((char *) this, sizeof(accountTransactions)))
            {
                    if (AccountNumber == retrieve_AccNo)
                    {
                            check = 0 ;
                            cout<<setw(3)<<dday<<"/"<<mmonth<<"/"<<yyear ;
                            cout <<setw(10)<<trantype ;
                            if (transactions == 'D') {
                                    cout.setf(ios::right,ios::adjustfield);
                                    cout <<setw(15);
                                    cout<<intAmount;
                                    cout <<setw(20);
                                    cout<<" ";

                            } else {
                                    cout.setf(ios::right,ios::adjustfield);
                                    cout<<setw(25);

                                    cout<<intAmount;
                                    cout <<setw(10);
                                    cout<<"  ";
                            }
                            cout<<setw(15);
                            cout.setf(ios::right,ios::adjustfield);
                            cout <<intBalance <<"\n";
                            row++;

                            if (row == 23)
                            {
                                    check = 1 ;
                                    row = 8 ;
```

```
                                        cout <<"\n\n  Please continue with the application.
\n";
                                        getch();
                                        clrscr();
                                        showAccount(retrieve_AccNo);
                                }
                        }
                }
        }
   }
   filename.close() ;
   if (!check)
   {
          cout <<"\n\n   Press any key to continue with the application. \n" ;
          getch() ;
   }
}
// dateDiffer() method displays the difference between 2 dates.
int accountTransactions :: dateDiffer(int day1, int month1, int year1, int day2, int
month2, int year2)
{
   static int monthArr[] = {31,28,31,30,31,30,31,31,30,31,30,31};      //Array  of
months for storing the no. of days in each array
   int days = 0 ;
   while (day1 != day2 || month1 != month2 || year1 != year2)
   {
          /* checking if the two dates in days,months and years differ and incrementing
the number of days.*/
          days++ ;
          day1++ ;
          if (day1 > monthArr[month1-1])
          {
                 day1 = 1 ;
                 month1++ ;
          }
          if (month1 > 12)
          {
                 month1 = 1 ;
                 year1++ ;
          }
   } return days ;
}

// getInterest() function calculates interest on the balance from the transaction.
dat file
float accountTransactions :: getInterest(int retrieve_AccNo, float iBalance)
{
   fstream filename ;
   filename.open("transactions.dat", ios::in);
```

```
    dispRecords newRec;
    filename.seekg(0,ios::beg) ;
    int day1, month1, year1, month_day;
    while (filename.read((char *) this, sizeof(accountTransactions)))
    {
            if (AccountNumber == retrieve_AccNo)
            {
                    day1 = dday ;
                    month1 = mmonth ;
                    year1 = yyear ;
                    iBalance = newRec.getBalance(retrieve_AccNo);
                    break ;
            }
    }
    int day2, month2, year2;
    struct date dateval;
    getdate(&dateval);
    day2 = dateval.da_day;
    month2 = dateval.da_mon;
    year2 = dateval.da_year;
    float interest_accrued=0.0;
    int yeardiff = year2 - year1;

    if ((year2<year1) || (year2==year1 && month2<month1) || (year2==year1 && month2==month1
&& day2<day1)) {

            return interest_accrued;
    }
    month_day = dateDiffer(day1,month1,year1,day2,month2,year2);
    int months;
    if (month_day >= 30)
    {
            months = month_day/30;
    } else {
            months = month_day/30;
    }
            if(interest_accrued == 0 && yeardiff == 1) {
                    interest_accrued = ((iBalance*0.5)/100) * (months);
            } else if (yeardiff > 1 && yeardiff < 25 && interest_accrued == 0) {
                            interest_accrued = ((iBalance*0.5)/100) * (months);
            } else {
                    interest_accrued = 0;
            }

    filename.close();
    return interest_accrued;
}
```

```
/*Method for generating Interest and updation of the Balance and addDetails methods.
(Choice 5)*/
void accountTransactions :: showInterest(void)
{
  clrscr();
  char t_acc[10];
  int tran_acc, retrieve_AccNo, check;

  cout <<strupr("\n      Important Information: Interest should be generated only\n
once in a Year.\n\n\t If you have already generated interest for an Account,\n\t please
ignore that Account.\n\t Thank you.\n");
  cout <<"\n     Press 0 to go back to previous menu.\n" ;
  cout <<"\n     To view the transaction of the Account, please enter it: " ;
  gets(t_acc) ;
  tran_acc = atoi(t_acc) ;
  retrieve_AccNo = tran_acc ;
  if (retrieve_AccNo == 0)
         return ;
  clrscr() ;
  dispRecords newRec ;
  if (!newRec.accountExists(retrieve_AccNo))
  {
         cout <<"\t\n   Account does not exist.\n";
         getch();
         return;
  }
  cout <<"       Press 0 to go back to previous menu.\n";
  cout<<"                                                          \n";
  cout<<"\n      -Please enter the Account no. to generate interest-  \n";
  cout<<" *****************************************\n";
  int day1, month1, year1;
  struct date dateval;
  getdate(&dateval);
  day1 = dateval.da_day;
  month1 = dateval.da_mon;
  year1 = dateval.da_year;
  cout <<"       Date : "<<day1 <<"/" <<month1 <<"/" <<year1<<"\n";
  cout <<"       Account no. " <<retrieve_AccNo<<"\n";
  char retrieve_CustName[30] ;
  char retrieve_Address[60] ;
  float iBalance ;
  strcpy(retrieve_CustName,newRec.getName(retrieve_AccNo)) ;
  strcpy(retrieve_Address,newRec.getAddress(retrieve_AccNo)) ;
  iBalance = newRec.getBalance(retrieve_AccNo);

  cout <<"       Customer Name   :      " <<retrieve_CustName;
  cout <<"\n     Customer Address:      " <<retrieve_Address ;
  cout <<"\n     Bank Balance    :      " <<iBalance ;
```

```
   float interest_accrued;
   interest_accrued = getInterest(retrieve_AccNo,iBalance);
                                   /* Calculation of interest of the deposit amount*/
   cout<<"\n\tInterest generated: "<<interest_accrued;
   getch();
   iBalance = iBalance + interest_accrued;
   dispRecords obj2;
   /*Updating the Balance once Interest is generated in a year*/
   obj2.updateBalance(retrieve_AccNo, iBalance);
   /*Adding Interest as a Deposit when it is generated in a year.*/
   addDetails(retrieve_AccNo,day1,month1,year1,'D',"Interest",interest_accrued,
interest_accrued,iBalance);
}
/* This method does all the Deposit/Withdrawal transactions in the transaction.dat
file(Choice 4)*/
void accountTransactions :: transaction(void)
{
   clrscr();
   char t_acc[10];
   int tran_acc, retrieve_AccNo, check;
   cout <<"       Press 0 to go back to previous menu.\n" ;
   cout <<"       To view the transaction of the Account, please enter it: " ;
   gets(t_acc) ;
   tran_acc = atoi(t_acc) ;
   retrieve_AccNo = tran_acc ;
   if (retrieve_AccNo == 0)
          return ;
   clrscr() ;
   dispRecords newRec ;
   if (!newRec.accountExists(retrieve_AccNo))
   {
          cout <<"\t\n   Account does not exist.\n";
          getch();
          return;
   }
   cout <<"       Press 0 to go back to previous menu.\n";
   cout<<"                                                              \n";
   cout<<"\n      -Make correct entry for the Transaction below-  \n";
   cout<<" ****************************************\n";
   int day1, month1, year1;
   struct date dateval;
   getdate(&dateval);
   day1 = dateval.da_day;
   month1 = dateval.da_mon;
   year1 = dateval.da_year;
   cout <<"       Date : "<<day1 <<"/" <<month1 <<"/" <<year1<<"\n";
   cout <<"       Account no. " <<retrieve_AccNo<<"\n";
   char retrieve_CustName[30] ;
```

```
    char retrieve_Address[60] ;
    float iBalance;
    float interest_accrued = 0.0;
    strcpy(retrieve_CustName,newRec.getName(retrieve_AccNo)) ;
    strcpy(retrieve_Address,newRec.getAddress(retrieve_AccNo)) ;
    iBalance = newRec.getBalance(retrieve_AccNo);

    cout <<"       Customer Name   : " <<retrieve_CustName;
    cout <<"\n     Customer Address: " <<retrieve_Address ;
    cout <<"\n     Bank Balance: " <<iBalance ;
    char tranDetails, typeTransaction[10], tm[10] ;
    float t_amount, t_amt ;

    do
    {
           cout <<"\n  Please enter D for Deposit or W for Withdrawal of Amount : " ;
           tranDetails = getche() ;
           if(tranDetails == '0') {
                  cout<<"\n\n    You have pressed 0 to Exit.";
                  getch();
                  return;
           }
           tranDetails = toupper(tranDetails) ;
    } while (tranDetails != 'W' && tranDetails != 'D') ;

    do
    {
           cout <<"\n     The Transaction type is either Cash or Cheque..\n" ;
           check = 1 ;
           cout <<"       (Cash/Cheque) : " ;
           gets(typeTransaction) ;
           strupr(typeTransaction);
           if(typeTransaction[0] == '0') {
                  cout<<"\n\n    You have pressed 0 to Exit.";
                  getch();
                  return;
           }
           if (strlen(typeTransaction) < 1 || (strcmp(typeTransaction,"CASH") && strc
mp(typeTransaction,"CHEQUE")) )
           {
                  check = 0 ;
                  cout<<"\n      The Transaction is invalid. Please enter either Cash
or Cheque. \n" ;
                  getch() ;
           }

    } while (!check);
```

```
   do
   {
          cout <<"\n     Please enter the Transaction Amount :  \n";
          check = 1 ;
          cout <<"       Amount : Rs. ";
          gets(tm) ;
          t_amt = atof(tm) ;
          t_amount = t_amt ;

          if (t_amount < 1 || (tranDetails == 'W' && t_amount > iBalance) )
          {
                 check = 0;
                 cout<<"\n      Either Amount is not a numeric value or\n it is blank
or\n you are trying to withdraw amount more than in the Account.....  \n" ;
                 getch() ;
          }
   } while (!check) ;
   char choice ;

   do
   {
          cout <<"\n     Save the changes made in the transaction details? (y/n): ";
          choice = getche() ;
          choice = toupper(choice);
   } while (choice != 'N' && choice != 'Y') ;

   if (choice == 'N' || choice == 'n') {
          cout<<"\n       The Transaction is not saved. \n";
          getch();
          return ;
   }

   if (tranDetails == 'D') {
          cout<<"\n      The Amount is Deposited in the Bank.\n";
          getch();
          iBalance = iBalance + t_amount;
   } else {
          cout<<"\n      The Amount is Withdrawn from the Bank.\n";
          getch();
          iBalance = iBalance - t_amount;
          }
   newRec.updateBalance(retrieve_AccNo,iBalance);
   /* Adding record details for the Transaction done (deposit or withdrawal) and saving
it in the file*/
   addDetails(retrieve_AccNo,day1,month1,year1,tranDetails,typeTransaction,interes
t_accrued,t_amount,iBalance);
}
```

```
/* This method deletes the Account from both the dat files(Choice 6)*/
void accountTransactions :: closeAccount(void)
{
  clrscr() ;
  char t_acc[10] ;
  int tran_acc, retrieve_AccNo ;
  cout <<"       Press 0 to go back to previous menu.\n" ;
  cout <<"       Please enter the Account you want to close : " ;
  gets(t_acc);

  tran_acc = atoi(t_acc) ; /* changing account no. to integer type. */
  retrieve_AccNo = tran_acc ;
  clrscr() ;
  dispRecords newRec ;
  if (!newRec.accountExists(retrieve_AccNo))
  {
        cout <<"\t\n You have entered an invalid Account or it does not exist.\n";
        cout <<" Please try again.\n";
        getch();
        return ;
  }
  cout <<"\n     Press 0 to go back to previous menu\n" ;
  cout<<"\n      Closing this Account.\n";
  cout<<"**********************************\n\n";
  int day1, month1, year1 ;
  struct date dateval;
  getdate(&dateval);
  day1 = dateval.da_day ;
  month1 = dateval.da_mon ;
  year1 = dateval.da_year ;

  cout <<"Date: "<<day1 <<"/" <<month1 <<"/" <<year1<<"\n";
  char choice;
  newRec.display(retrieve_AccNo); /*Displaying the Account Details on the basis of
the retrieved Account Number*/

  do
  {
        cout <<"\n     Are you sure you want to close this Account? (y/n): ";
        choice = getche();
        choice = toupper(choice) ;
  } while (choice != 'N' && choice != 'Y');

  if (choice == 'N' || choice == 'n') {
        cout<<"\n      The Account is not closed.\n";
        getch();
        return;
  }
```

```
   newRec.deleteAccount(retrieve_AccNo);
   deleteAccount(retrieve_AccNo);
   cout <<"\t\n\n Record Deleted Successfully.\n";
   cout <<"       Please continue with the application....\n";
   getch();
}

/* The Login method checks for the username and the password for accessing the Banking
Application*/
   int login (void)
   {
   char username[9],ch;
   char username1[]="banking";
   int i=0;
   char a,b[9],pass[]="tatahill";
   cout<<"\n\n";
   cout<<"\n\t    Login to the Banking Application.\n";
   cout<<"\t      **********************************\n";
   cout<<"\n\n\tPlease enter Username    :       ";
   cin >> username;
   cout<<"\n\n\tPlease enter Password to authenticate yourself  :       ";
   fflush(stdin);
          do
                        {
                        ch=getch();
                        if(isalnum(ch))
                                {
                                b[i]=ch;
                                cout<<"*";
                                i++;
                                }
                        else
                                if(ch=='\r')
                                        b[i]='\0';
                                else if(ch=='\b')
                                        {
                                        i--;
                                        cout<<"\b\b";
                                        }
                        }
                while(ch!='\r');

   b[i]='\0';
   fflush(stdin);

          if((strcmp(b,pass)==0)&&(strcmp(username1,username)==0))
          {
                        cout<<"\n\n\t  You have entered successfully\n\n";
```

```
                        return(1);
                }
                else
                {
                        cout<<"\t\n\n          Incorrect Username or Password.";
                        cout<<"\n";
                        return(0);
                }

        }

/* This is the Main function which displays the Menu */
void main(void)
{
        clrscr();
        int val,ch;
        a: val=login();
        if (val==0)
        {
                cout<<"\n\t    Want to try again?\n";
                cout<<"\t      1.TRY AGAIN ";
                cout<<"\n\t    2.EXIT";
                cout<<"\n\n\t  Enter your choice and press enter:";
                cin>>ch;
                if (ch==1) {
                        clrscr();
                        goto a;

                } else {
                        exit(0);
                }
        }
        Menus obj1 ;
        obj1.showmenu();
}
```

BE-205

B.E. (First Semester) Examination, Dec., 2010

(Grading System)
(Common for all Branches)
Basic Computer Engineering

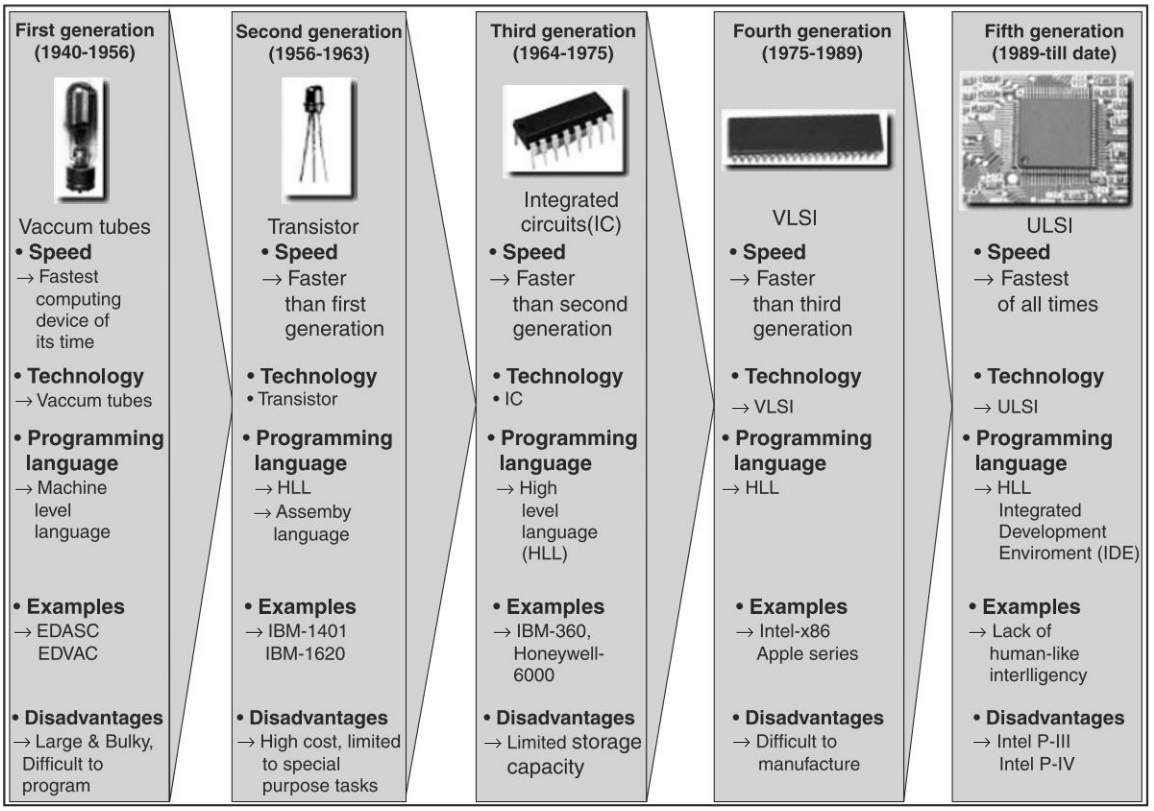
Time Three Hours *Maximum Marks 70*

Minimum Pass Marks 22 (D Grade)

1. (a) Explain the difference between four generations of computers and in what way each generation was better than earlier.

Ans. Difference between Generations of Computers

Here's a snapshot of the key differences between different generations of computers:



1. (b) What are the different types of memories? How the size of memory is specified? Define the access time of memory.

Ans. Different Types of Memories

Refer Section 1.8.

Specification of Memory Size

The 'smallest addressable unit' of memory is referred as byte. Most computers use groups of bytes, usually 2 or 4, known as 'words' to represent information.

Some of the typical memory size specifications are:

1 KB (KiloByte) = 1,024 bytes

1 MB (MegaByte) = 1024 KBs or 1,048,576 bytes

1 GB (GigaByte) = 1024 MBs or 1,073,741,824 bytes

1 TB (TeraByte) = 1024 GBs or 1,099,511,627,776 byte

Access Time of Memory

Access time refers to the time consumed in completing the memory read/write operation. It mainly depends on the type of storage device used and the mode of data access. The access time of primary memory is faster than that of the secondary memory.

2. (a) Why do computers have internal memory as part of the CPU and the internal bulk memory separately?

Ans. Reason for Having Separate Internal CPU Memory and Internal Bulk Memory

The access speed of internal CPU memory is substantially fast as compared to that of the bulk memory. Alternatively, the bulk memory offers very large storage capacity as compared to the internal CPU memory. This access speed and storage capacity trade-off is vital to the suitability of these memory devices in different scenarios. For instance, the internal CPU memory works in close coordination with the processor by supplying the desired program data at fast speed. Since the size of the program that is being currently executed is not large; the smaller storage capacity of the internal CPU memory does not pose any hindrance. Alternatively, the internal bulk memory serves an altogether different purpose. It allows the users to store the processed data for future use. The large size of the bulk memory ensures that the large quantities of data can be stored in it. The comparatively slower access speed of the bulk memory device is not much of a concern in such a case.

2. (b) What is the difference between Random Access and Sequential Access?

Ans. Difference between Random Access and Sequential Access

Random Access	*Sequential Access*
In case of random access of memory, data can be read or written in any order.	In case of sequential access of memory, data can be read or written only in a sequential fashion.
It allows quick access of data.	It allows comparatively slower access of data as the address location needs to be sequentially traversed.
Example: magnetic tape	*Example:* magnetic disk

2. (c) Difference between Address bus, Data bus and Control bus

Ans. **Data bus:** It is used for transferring data amongst the different internal components of a computer system.

Address bus: It is used for transferring memory addresses for read/write operations. It contains a number of address lines that determine the range of memory addresses that can be referenced using the address bus.

Control bus: It is used for carrying the commands issued by the processor and the status signals generated by the various devices in response to these commands.

2. (d) Explain the application of computer in multimedia and animation.

Ans. **Application of Computer in Multimedia and Animation**

Refer Section 1.13.7.

3. (a) What is an Operating System? Explain the services provided by an Operating System.

Ans. **Operating System**

Refer Section 2.1.

Services provided by an Operating System

The various services provided by an operating system are the following:

- **User interface:** It provides an interface to the users to interact with the computer system. The interface provided can be command-based (CUI) or graphical (GUI).
- **I/O operation:** It enables the users to perform input/output operations on the computer system.
- **File manipulation:** It allows the users to create and store files.
- **Memory management:** It manages the different memory storage devices and allows the users to perform read/write operations.
- **Program execution:** It allows the users to load and run their programs.
- **Security:** In a multi-user environment, an operating system implements authentication mechanisms for securing data and hardware resources.

3. (b) What is the main drawback of structured programming?

Ans. **Structured Programming and the OOP Approach**

Refer Sections 4.3 and 4.4.

4. (a) What is Inheritance? Explain its various types.

Ans. **Inheritance**

Refer Section 4.5.4.

Types of Inheritance

Refer Sections 11.1 to 11.8.

4. (b) Explain how operating system performs file management functions.

Ans. **How an Operating System performs File-Management Functions?**

Refer Section 2.6.

5. Explain the following:

(a) Objects as Function Arguments

Ans. Objects as Function Arguments

Refer Section 8.14.

5. (b) Classes and Objects

Ans. Classes

Class is a way of binding data and its associated functions together. A class definition creates an abstract data type that is similar to the other built-in data types. A typical class specification has two parts:

- **Class declaration:** Describes the type and scope of class members.
- **Class function definition:** Describes how class functions are implemented.

Following is a sample class declaration:

```
class item
{
       int number;
       float cost;
   public:
       void getdata(int a, float b);  //function prototype
       void putdata(void);
}
```

Objects

An object is an instance of a class. Just like we create instances (variables) of different built-in data types, objects are also instances of the abstract data types created using classes.

Following is an example of object creation:

```
item x;
item x, y, z;
```

Here, *x*, *y* and *z* are objects of the item class.

5. (c) Dynamic Initialization of Objects

Ans. Dynamic Initialization of Objects

Refer Section 9.6.

5. (d) Copy Constructor

Ans. Copy Constructor

Refer Section 9.7.

6. (a) Write a program which generates series of Prime Numbers.

Ans. Program for Generating a Series of Prime Numbers

```
#include <iostream>
#include <conio.h>
using namespace std;

void main()
{
  int num,i,j,count;
  cout<<"Enter the number of terms: ";
  cin>>num;
  cout<<"The various prime numbers between 1 and "<<num<<"
are:";

for(i=1;i<=num;i++)
{
  count=0;
  for(j=1;j<=i;j++)
    if(i%j==0)
      count++;
    if(count==2)
      cout<<"\t"<<i;
}
getch();
}
```

Output

```
Enter the number of terms: 30
The various prime numbers between 1 and 30 are:     2     3
5     7     11     13     17     19     23     29
```

6. (b) Explain the following:

(i) Pass by Value

(ii) Pass by Address

(iii) Pass by Reference

Ans. (i) Pass by Value

In this method, the value of a variable is passed to a function. The passed value is copied into another variable in the parameter list of the function. Thus, any change made to the new variable's value inside the function is not reflected back in the main function. The following code snippet depicts this scenario:

Code

```
void change(int n)
{
  n=100;
}
void main()
{
  int num=50;
  cout<<"Before function call, num = "<<num;
  change(num);
  cout<<"\nAfter function call, num = "<<num;
getch();
}
```

Output

```
Before function call, num = 50
After function call, num = 50
```

(ii) Pass by Address

In this method, the memory addresses of the variables rather than the copies of values are passed to the function. Thus, any changes made to the values stored at the memory addresses stay permanent and get reflected back to the main function. The following code snippet depicts this scenario:

Code

```
void change(int *n)
{
  *n=100;
}
void main()
{
  int num=50;
  cout<<"Before function call, num = "<<num;
  change(&num);
  cout<<"\nAfter function call, num = "<<num;
getch();
}
```

Output

```
Before function call, num = 50
After function call, num = 100
```

(iii) Pass by Reference

When arguments are passed by reference, the formal arguments in the called function become aliases to the actual arguments in the calling function. Thus, when a function is working with its own arguments, it is actually working on the original data. The following code snippet depicts this scenario:

Code

```
void change(int &n)
{
    n=100;
}
void main()
{
  int num=50;
  cout<<"Before function call, num = "<<num;
  change(num);
  cout<<"\nAfter function call, num = "<<num;
getch();
}
```

Output

```
Before function call, num = 50
After function call, num = 100
```

7. (a) What is the need for evaluation of a DBMS? List the technical criteria that are to be considered during the evaluation process.

Ans. Need for Evaluation of DBMS

DBMS evaluation is necessary due to the following reasons:

- Ensuring compatibility with the software system
- Ensuring that the performance of the DBMS system matches the requirement
- Assessing usability and maintenance trade-offs

Technical criteria for DBMS Evaluation

The technical criteria for DBMS evaluation mainly depends on the situation at hand which includes the associated software system and its environment.

Typically, DBMS evaluation criteria may include the following:

- **Design:** Realizes the real world more accurately
- **Structure:** Supports abstract and multimedia data types
- **User interface:** Allows creating standard and user-friendly interfaces

- **Maintenance:** Enhances database programmer's productivity by supporting code reusability
- **Portability:** Enhances system portability and supports extensibility

7. (b) Explain DDL and DML operation in database system.

Ans. DDL Operation

Refer Section 13.9.

DML Operation

Refer Section 13.10.

8. (a) Explain the Architecture of Database System.

Ans. Architecture of Database System

Refer Section 13.4.

8. (b) What is a Database Model? Explain any two types of data models with an example with an example for each.

Ans. Database Model and its Types

Refer Section 13.3.

9. (a) What are the services provided by Network Layer to Transport Layer? Explain.

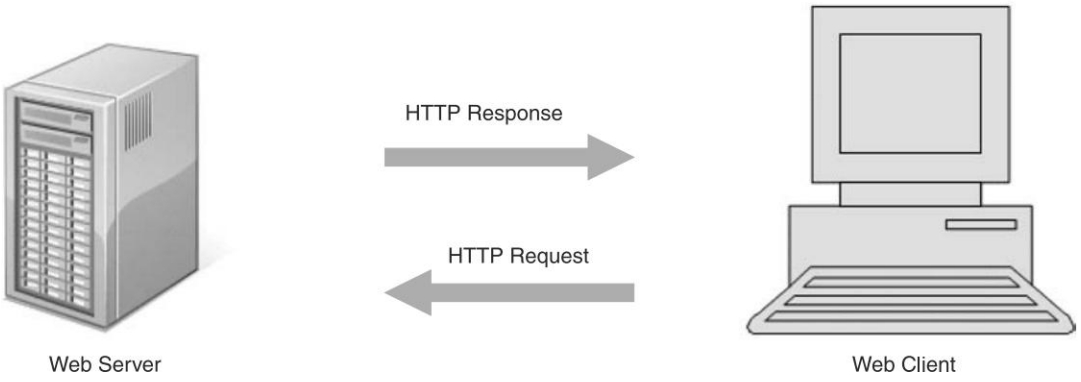
Ans. Services provided by Network Layer to Transport Layer

Refer Section 14.3.

9. (b) Explain the architecture of WWW as on client/server application.

Ans. World Wide Web

World Wide Web (WWW) is a collection of interlinked hypertext web pages accessed by the users through the Internet. These web pages comprise of hypertext, simple text, images, videos, animations, graphics, and many more elements. To view the Web pages hosted on a Web server, a software known as Web browser is required on the client computer. The following figure shows the architecture of WWW as a client-server application:



The various steps involved in accessing a Web page through WWW are the following:

1. User types the URL of the Web page on the Web browser.
2. An HTTP request is sent to the Web address pointed by the URL.
3. The Web server accepts the HTTP request and after validation generates the response object.
4. Based on the Web server's response the Web page is rendered by the Web browser on the client's computer.
5. If the user clicks a hyperlink on the Web page then an HTTP request is sent to the Web address pointed by that hyperlink.

10. (a) Explain TCP/IP Reference Model in detail.

Ans. TCP/IP Reference Model

Refer Section 14.3.1.

10. (b) What is e-Commerce? Explain the role of networking in e-Commerce.

Ans. e-Commerce

Refer Section 14.10.

10. (c) What are the various Networking Devices? Explain briefly.

Ans. Networking Devices

Refer Section 14.5.3.