Leading Books Publisher

COMPUTER SCIENCE

Grade XII

Free From Government
NOT FOR SALE

Khyber Pakhtunkhwa Textbook Board, Peshawar

رشوت دینے والا

اور رشوت

لینے والا دونوں جہنمی ہیں۔

COMPUTER SCIENCE

Grade XII

Abdul Haseeb

Leading Books Publisher
University Road, Peshawar

عبدالحسیب

Khyber Pakhtunkhwa Textbook Board, Peshawar

Approved by:
Directorate of Curriculum and Teacher Education, Abbottabad,
according to the National Curriculum 2006.
NOC No.5385-87/SS-V dated: 11.09.2015

Author: Mohammad Khalid
Head of Computer Science Department
at OPF Boys, College, Islamabad.

Rahman Ali
M.Phil, M.Sc
University of Peshawar

Review Supervision:
Syed Bashir Hussain Shah
Director Curriculum & Teacher Education Abbottabad

Reviewers: Mr. Muhammad Zahid, Associate Professor, Chairman
Department of Computer Science, GPGC No.1 Abbotabad
Mr. Muhammad Daud, Assistant Professor GPGC Mandian Abbotabad.
Dr. Muhammad Naeem, IT Department Hazara University Mansehra.
Mr. Waqar Ahmed, Assistant Subject Specialist Text Book Board Peshawar.
Mr. Ibrar Ahmed, Subject Specialist, DCTE Abbotabad.

Editor: Shakeel Ahmed, Subject Specialist (Maths)
Khyber Pakhtunkhwa Textbook Board Peshawar.

Printing Supervision:
Masood Ahmed, Chairman
Khyber Pakhtunkhwa Textbook Board Peshawar.
Saeedur Rehman, Member (E & P)
Khyber Pakhtunkhwa Textbook Board Peshawar.

Academic Year 2018-19
Website: www.kptbb.gov.pk
Email : membertbb@yahoo.com
Phone: 091-9217159-60

Contents

UNIT 1

OPERATING SYSTEM

After the completion of Unit-1, Students will be able to:

- Define an operating system.
- Describe commonly used operating systems (DOS, Windows, Unix, Macintosh).
- Explain the following types of operating systems:
  - Batch processing operating system
  - Multiprogramming operating system
  - Multitasking operating system
  - Time-Sharing operating system
  - Real Time operating system
  - Multiprocessor operating system
  - Parallel processing operating system
  - Distributed operating system
  - Embedded operating system
- Describe features/characteristics of Single-user and Multi-user operating systems.
- Describe the following functions of operating systems:
  - Process management
  - Memory management
  - File management
  - I/O system management
  - Secondary storage management
  - Network management
  - Protection system
  - Command-interpreter
- Define a process.
- Describe the new, running, waiting/block, ready and terminated states of a process.
- Differentiate between:
  - Thread and Process
  - Multi-threading and Multi-tasking
  - Multi-tasking and Multi-programming

1.1 INTRODUCTION

This unit is focused on introduction to Operating System and its functions and services. It also describes that how an operating system is used to execute processes. At the end, a differentiation is made between thread and process, multithreading, multitasking and multiprogramming.

Operating System is an important component of a computer system and has the same relation with computer hardware as human soul has with human body. As human body is useless without human soul, similarly, a computer is useless without an operating system.

1.1 An Operating System

An operating System (OS) is an intermediary between users and computer hardware. It provides users an environment in which a user can execute programs conveniently and efficiently. In technical terms, it is a software which manages hardware. An operating System controls the allocation of resources and services such as memory, processors, devices and information.

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.

Some commonly used operating systems that are used for personal computers are: Linux, Mac OS, Microsoft Windows XP, Microsoft Windows Vista, Windows 7 and UNIX.

A computer without an operating system is just an empty dump of metal. To run application programs on a computer, it must have an operating system installed on it. Operating systems perform basic operations, such as getting input from the input devices, transferring data from main memory to the processor for processing and then sending

output to the output devices. It also keeps track of files and directories on the hard disk, and controls peripheral devices.

Consider Figure 1.1 which shows that operating system is an interface between users and computer hardware.

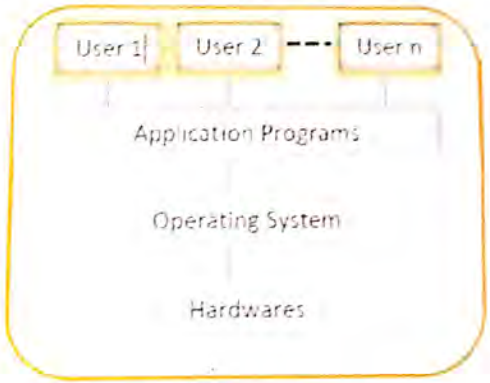


Figure 1.1: Operating System as an Interface

Operating system operates on top of hardware and acts as a virtual machine that hides the hardware level complexities from the users and provides them a convenient way of using the system.

1.1.2 Commonly used operating systems

The development of operating system has started since 1950's and different types of operating systems have been developed so far by different companies. The most commonly used operating systems are: DOS, Windows, UNIX and Macintosh.

a. DOS

DOS is the abbreviation for Disk Operating System which was the first operating system used for personal computers.

DOS operating system was developed in 1980 by Microsoft when, in July 1980, the IBM assigned a project to Microsoft for the development of a 16-bit operating system for their personal computer. Microsoft developed the first version of DOS for the personal computer and named it PC-DOS. After this, Microsoft developed a Microsoft version of the same PC-DOS operating system and named it MS-DOS. Both PC-DOS and MS-DOS are almost similar and now users refer them with only DOS.

In Figure 1.2 a snapshot of Microsoft DOS Operating System is shown with some basic commands: DIR and VER.

Figure 1.2 MS DOS Operating System

The most commonly used DOS commands are:

CD - changes the current directory
COPY - copies a file
DEL - deletes a file
DIR - lists directory contents
EDIT - starts an editor to create or edit plain text files
FORMAT - formats a disk to accept DOS files

HELP - displays information about a command
MKDIR - creates a new directory
RD - removes a directory
REN - renames a file
TYPE - displays contents of a file on the screen

b. Windows

Windows operating systems are the most commonly used operating systems that are based on Graphical User Interfaces (GUI).

i. Brief history of windows operating system

The history of windows operating system goes back to 1983. The first windows operating system "Windows 1.0" was launched on November 20, 1985. The graphical support with desktop icons was introduced in "windows 2.0" that was launched on December 9, 1987. On August 24, 1995, "windows 95" was launched. Windows 98, XP and Vista were released on June 25, 1998, October 25, 2001 and 2006 respectively. In October 2009 windows 7 was released and Windows 8 was released on October 2012.

ii. Features of windows operating system

- Windows operating system has improved graphical user interface that can be easily used by every type of user.
- Windows operating systems are user friendly.
- These are the best operating systems running on PCs.
- Windows series of operating systems are not very expensive for home users.
- Windows operating systems are supported by most of the software.

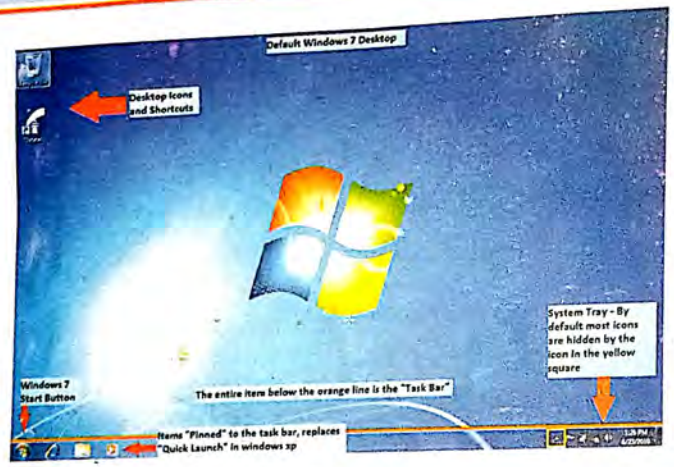


Figure 1.3: Windows 7 Desktop

c. UNIX Operating System

UNIX is a multitasking and multi-user operating system that was developed by Ken Thompson, Dennis Ritchie and their group members at Bell Labs in 1969. It was first developed in assembly language. Later on, in 1973, it was re-developed in C language. This version of UNIX has facilitated its further development and support to other hardware.

UNIX operating systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment for naive users.

Different versions of UNIX operating system have been developed so far which have some features in common. The most popular types of UNIX operating system are Sun Solaris, GNU/Linux and Mac OS X.

When you want to open the UNIX terminal window, click on **Application** menu and then select **Terminal** icon.

After selecting **Terminal** from the menu, the UNIX Terminal window will open having % prompt as shown in figure 1.4. The user will enter the command in this prompt to execute.

Figure 1.4 UNIX Prompt

d. Mac OS

Mac OS stands for Macintosh operating system. It is a series of operating systems having graphical user interfaces that have been developed by Apple Inc. On January 24, 1984, Apple Computer Inc. manufactured its first Macintosh personal computer and developed a system software for it which was later on renamed to Mac OS.

Mac OS can be divided into two groups which are: Mac OS Classic family and the Mac OS X family. So far, Mac OS has ten (10) versions with the names Mac OS 1, 2, 3, 4, 5, 6,7,8,9 and X. Figure 1.5 shows the desktop interface of Mac OS.

Figure 1.7: Mac OS Desktop

1.13 Types of Operating Systems

Operating system can be divided into different types which are given below.

a. Batch processing operating system

Batch processing system is that type of operating system which collects **jobs** in batches before being processed by the CPU. A job is a piece of work usually consisting of a program and the data to be run. All the Jobs are stored in **job queues** until the computer is ready to process them.

Once the CPU gets ready and fetches a job from the job queue, then there is no interaction of the user with the computer to interrupt the processing of the job until its completion.

Batches of jobs are formed by collecting jobs during working hours and then executed during the evening time or at the end of each day or whenever the computer is available. Once a batch job begins, it continues until it is completed or until an error occurs.

Payroll system and preparation of electricity, telephone and credit card bills are processed under the batch processing systems. In each of these systems, the individual wages, calls, units consumed and transactions performed are batched together and at the end of each month the total bill is calculated.

The example of batch processing system is IBM's OS 360.

b. Multiprogramming operating system

Simple batch operating system has the problem that the processor often remains idle when the process under execution is halt due to some external event. Also, Input/output devices are slow as compared to processor, therefore, most of the time the processor

remains idle. To eliminate this problem the concept of multiprogramming operating systems was introduced.

Multiprogramming operating system is that type of operating system which allows running of multiple computer programs simultaneously on a single processor. For example, you may be typing in MS Word, listening to music while in background Internet Explorer is downloading some pages from the Internet. Working in Windows 7, Vista, XP and Linux environment is multiprogramming.

In multiprogramming operating systems, several programs, called jobs, are active at the same time in the main memory (RAM) of the computer. This means that multiple jobs are loaded into RAM and executed by a single processor. Figure 1.6 shows main memory partitioned into five (05) frames holding four (04) jobs.

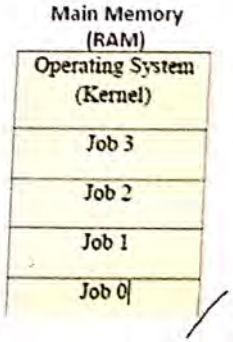


Figure 1.6 Main Memory Partitioning in Multiprogramming

Advantages of multi-programming

Multiprogramming operating systems make efficient use of the processor by interleaving multiple actives jobs that reside in main memory. It also attempts to increase CPU utilization by always having something for the CPU to execute. Main memory (RAM) is efficiently utilized to hold more jobs at a time.

Limitation of multi-programming

Multiprogramming requires context switching since several programs (processes) sequentially share the CPU. This typically introduces processing delay compared to a dedicated unshared process.

c. Multi-tasking operating system

Multitasking is the process in which the operating system performs multiple tasks simultaneously on a single processor in such a way that creates the impression of parallel executions. Practically, all modern OS has the ability of multitasking.

Multitasking does not mean that the computer has multiple processors to execute multiple processes simultaneously. It means that the operating system switches the jobs so quickly on a single processor that the user thinks all these processes are executing in parallel.

Multitasking can be either pre-emptive or cooperative. In pre-emptive version, the operating system divides the CPU time and allocates one slot to each of the programs. Operating systems that support preemptive multitasking are: Solaris, UNIX, Windows 2000, Windows XP, Mac OS X and Linux. In cooperative multitasking, processes cooperate with each other to achieve simultaneous execution. Operating system that supports cooperative multitasking is Windows 3.x.

d. Time-Sharing operating system

Time sharing operating systems are those operating systems which slice processor (CPU) time into small fixed time slices and assign the processes to the CPU for that time slice to be executed. If the process is completed in its allotted time slice, it quits the system and CPU is assigned to the new process waiting on the top of the ready queue. If the time slice expires and the process still remains incomplete, it is added at the back of the ready queue for its next turn.

This technique of sharing the CPU time is termed as time sharing, because, the time of the processor is shared among multiple jobs residing in main memory (RAM). Multiple users access the system simultaneously through their terminals and the operating system interleaves the execution of each program in a short burst of time called **time slice** or **quantum**.

Time sharing operating systems are needed for applications which need quick response time and users interactions such as transactions processing.

Advantages of Time sharing operating system

- Minimizes response time.
- Supports user interactivity.
- A process does not wait for a long time if it is preceded by a process of larger execution time.
- All processes get equal chances for execution.

Time Sharing Vs Multitasking Operating System

There is no universally accepted difference between time sharing and multi-tasking operating systems. Multi-tasking is used in a broad sense and thus Time-sharing operating system can be considered is a kind of multitasking operating system. Time sharing operating systems have a short time slices given to each process whereas in multitasking operating systems there is no time slice assigned to each process. The operating system (multitasking) switches among the processes residing in the main memory, RAM, in such a way that when a process needs input or output operation the operating system is switched to some other process residing in RAM.

e. Real-Time operating system

Real time operating systems are those operating systems which give quick response to users requests without delaying them. Unlike batch processing operating systems, real time operating systems produce immediate response to users requests as they are input to the system. The success rate of these operating systems is not only measured in terms of the correctness of the logically computed result, but also on the basis of time at which the results are produced. As these systems process the users input immediately, therefore, the systems supports high degree of interactivity and as a result they can be widely used in critical environments, such as, air traffic control systems, radar system, missiles systems etc.

As the results of real time operating systems are time-based, therefore, they are grouped into hard-real time operating systems and soft-real time operating systems. Hard-real time operating system guarantees the solution in the specified time constraint; otherwise, the results are catastrophic such as flight navigation, automobile, and spacecraft systems. In soft-real time operating systems, if the deadlines are missed by some amount of time still they are acceptable such as media streaming in distributed systems.

Features of real time operating system

- A real time operating system guarantees a solution or results within a specific time.
- These are multitasking operating systems.
- These operating systems are used for the execution of real-time applications, such as weather forecasting, geographical information retrieval, controlling machinery, scientific instruments and industrial systems etc.
- These systems are aimed on quick and predictable response to events.
- When these systems start execution of a task, they do not provide full control to the users to interact or interfere in that processing.

f. Multiprocessor operating system

The use of two or more central processing units (CPUs) within a single computer system is called **Multiprocessing**. A multiprocessing system can run multiple tasks in parallel on multiple CPU's and thus there should be an 'operating system to control such parallel executions. Multiprocessor operating systems are the operating systems that perform this task.

As multiprocessing operating systems execute multiple tasks in parallel, therefore, a single process can be divided into small independent units, called threads, and executed in parallel on multiple processors giving birth to the concept of **multithreading**.

The most commonly used multiprocessing systems are symmetric multiprocessing and asymmetric multiprocessing. In symmetric multiprocessing systems, all the CPUs are equal but in asymmetric multiprocessing systems there is one master CPU controlling other CPUs called slave CPUs.

Advantages of multiprocessing operating systems

If you have to run multiple programs at the same time, multi-processors can be very useful. Another advantage of multi-processors is that more and more programs these days are multithreaded. A multi-threaded application is a single program or application that can make use of more than one processors/cores. If one processor fails to work then the load is shared by other processors.

Limitations of multiprocessing operating system

Cost and power consumptions are the disadvantages of multiprocessing operating systems. A large main memory is required. A very sophisticated operating system is required to schedule, balance and co-ordinate the input, output and processing activities of multiple CPUs.

Multiprogramming Vs Multiprocessing

Multiprogramming is different from multiprocessing in the sense that in multiprogramming multiple programs are processed by a single processor in an interleaved passion giving the impression of being executed simultaneously while in multiprocessing operating systems multiple processors execute more than one programs simultaneously.

g. Parallel processing operating system

The operating system that is used to operate, control and manage a parallel computer system is called parallel processing operating system. A parallel computer system is a computer which has multiple processors that work co-operatively to solve a specific computational problem. The parallel execution can be achieved by executing multiple processes on different processors in parallel. Parallel operating systems are much similar to multiprocessing operating systems.

Parallel computing is used for problems which need many calculations simultaneously. This task is accomplished by dividing the computationally complex and large problems into smaller independent tasks which are executed concurrently on parallel processors.

Parallel processing can be achieved at different levels such as, bit-level, instruction level, data level and task level. The most commonly used parallel processing operating systems are UNIX and Microsoft Windows NT.

Time-sharing Vs parallel processing operating system

Time-sharing operating systems allows users to share a single processor simultaneously. Each user is allocated resources, CPU, for a particular time slot called **time slice**. In parallel processing operating systems, various processes are executed simultaneously on multiple processors. In parallel processing operating systems, there is no such scheme used for the allocation of resources (e.g. processor) in which it is shared among multiple processes.

h. Distributed operating system

A distributed operating system is an operating system that manages a group of independent computers and makes them appear to the users as a single computer.

When independent computers are networked together in such a way to work in a co-operative manner then they give rise to a distributed system. In a distributed environment, for performing a task, the computers communicate with each other and perform operations in such a way that the users do not feel that the computations are taking place on more than one machine. In a distributed environment, the resources should be shared; the processes should be scheduled on different machines, and the communication and synchronization mechanisms should be defined clearly. All these are the duties of a distributed operating system. The distributed operating system hides from the users that where a particular resource is, how the data is accessed, how a resource was moved to a new location and how the error was recovered.

Advantages of distributed system

- Communication and resource sharing is possible which eliminates the need of dedicated resources with each computer system.
- The distributed systems are economical in the sense that expensive resources are shared.
- The system is reliable because of the availability of multiple machines for performing the task of a failed system.
- The system has the potential for incremental growth.

Disadvantages of distributed system

- Network connectivity is an essential part of a distributed system which is a difficult and expensive task.
- Security and privacy is an issue.

i. Embedded operating system

Embedded operating systems are those operating systems which are designed for use in embedded computer systems. An embedded computer system is a computer which is a part of another system such as robot, missile etc.

These operating systems are designed for the operations of small machines like Personal Digital Assistant (PDA) with less autonomy. They can operate with limited number of resources. By design, these operating systems are very compact in size and efficient in performance. Windows CE, Embedded Linux and Android are the examples of embedded operating systems.

1.1.4 Single-user and Multi-user Operating Systems

Operating systems can be divided into two categories as for as number of users are concerned.

a. Single-user operating systems

Single-user operating systems are those operating systems which are usable by a single user at a time. These operating systems execute a single process at a time on a single processor. Batch processing operating systems are the examples of single user operating systems.

b. Multi-user operating systems

Multi-user operating systems are those operating systems which allow multiple users to access a computer system at the same time. Time-sharing operating system is an example of multi-user operating systems as they enable multiple users to access a single computer by sharing of time. UNIX is an example of multi user operating system.

1.2 OPERATING SYSTEM FUNCTIONS

As operating system is the main controller of a computer system, therefore, it provides a variety of services and functions to the computer and its users which are listed below:

- Process Management
- Memory Management
- File Management
- Input/output System Management
- Secondary Storage Management
- Network Management
- Protection System
- Command-interpreter

Process management

A program under execution is called process. Early computer systems were able to run only one process on a single processor, but, now the concepts of multitasking and multiprogramming have enabled computers to run multiple programs on a single processor. **Process management** is that function of operating system in which it deals with running multiple processes on a single computer. Since, most of the computer systems contain one processor with one core, multitasking is done by simply switching processes quickly. If there are more processes to run on a processor, then the operating system decides whether to shorten the time slice for each process or to make the processes wait long for their next turns of execution.

Memory management

Computer memory is a vital hardware resource of the computer system and it should be properly managed. It is grouped into Registers, CPU cache, Random Access Memory (RAM) and Disk Storage. A module of operating system called memory manager is responsible for the coordination of these types of memory by tracking which memory is

available, which is to be allocated or de-allocated and how to move data among them. This overall function of operating system is called **memory management**.

File management

The data is stored in computer memory in **files** which are further arranged and organized in sub directories and root directory resulting in a **hierarchical file system**. A module of operating system that organizes and keeps track of all the files in computers is called **file manager** and the process of managing these files is called **file management**. For example, a *hierarchical file system*, shown in figure 1.7 and 1.8 depict that how files are arranged into root directory and sub-directories. This hierarchical file system is also called **files system**.

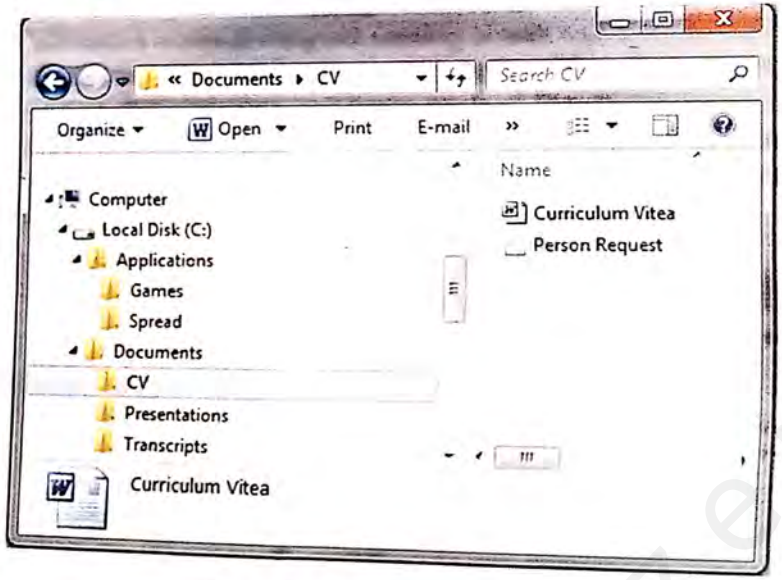


Figure 1.7: File Structure in Computer

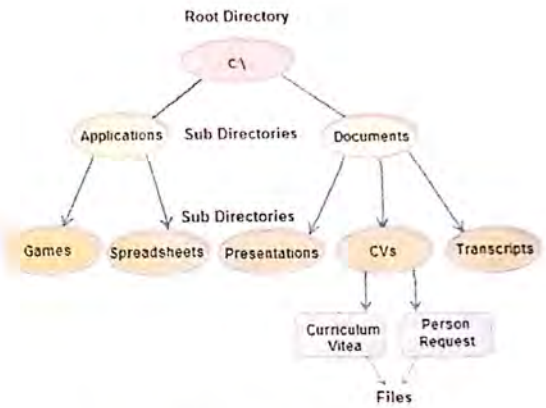


Figure 1.8: Hierarchical File System

Input/Output System Management

Every computer system has input/output devices such as keyboard, mouse, disk drives, printers, displays etc. All these devices need significant amount of management. The module of operating system used to manage input/output devices is called **Input/Output manager** or **device manager** and the process is called Input/Output management. I/O manager allocates resources on requests to applications to perform I/O operation. Device manager provides a convenient way for the use of input/output hardware and hides the hardware level complexities from the users.

The input/output hardware of computer cannot be accessed or used directly by the users. To make them useable, device drivers software must be installed on the computer. It is the device manager module of the operating system that make use of device driver software to access and manage Input/output devices. The relationship of operating system with drivers and hardware is shown in Figure 1.9.

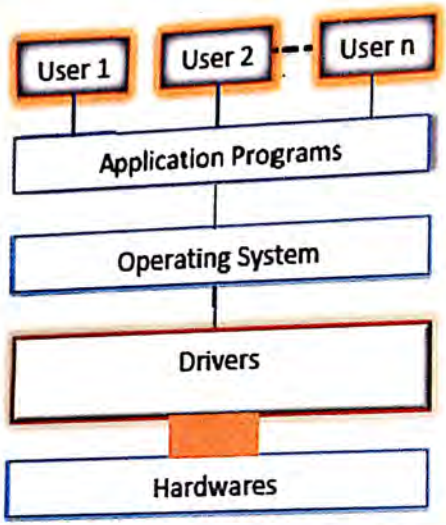


Figure 1.9 Device Drivers and Operating System

Secondary Storage Management

Secondary storage consists of those devices which stores data permanently for a long time. These devices includes tape drives, hard disks, CD and DVD disks, flash memory and other media designed to hold information for long life. The data stored in the secondary storage is accessed in primary storage and then into cache and CPU registers. The data loaded from secondary storage is ordinarily divided into fixed number of bytes or words. The function of operating system is to manage secondary storage devices and handle proper flow of data among primary and secondary storage devices is called **secondary storage management**.

In storage, each location has an address. The set of all addresses available to a program is called an address space which is managed by the secondary storage management module of the operating system.

Network management

Network management is not a core part of operating system but in modern operating systems **network manager** has become essential component to efficiently manage network related issues. Almost, all modern operating systems are capable of using transmission control protocol (TCP) and internet protocol (IP) for the network supporting services. The module of operating system that is responsible for sharing resources such as files, printers, and scanners using either wired or wireless connections is called **network manager** and the process is called **network management**.

Many operating systems support networking protocols. The most popular network operating systems are Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, and Novell NetWare.

Protection System

Nowadays, computer systems have multiple users connected through clients to the server computers. These users concurrently access and execute multiple processes which must be protected against each other so that the processes may not interfere each other data. The module of operating system responsible for this task is called **protection system**. Protection refers to mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system.

Command interpreter

All operating systems need to provide an interface to the users for the communication with the computer. The two commonly used methods, provided by the operating systems, for interfacing are Command Line Interface (CLI) and Graphical User Interface (GUI).

A **command line interface** or **CLI** is a method of interacting with an operating system or software using a command line interpreter. This command line interpreter may be a text terminal, terminal emulator, or remote shell client.

1.3 PROCESS MANAGEMENT

As discussed earlier, process management is one of the important functions of operating system. The following sections describe process in detail with a comparison to threads.

1.3.1 Process

Following are some definitions of process:

- A program under execution by the processor (CPU) is called process.
- An instance of a program running on a computer is called process.
- The entity that can be assigned to and executed on a processor is termed as process.

1.3.2 Process States

A process can be in one of the different states: new, ready, running, block, and exit. These states are usually represented in the form of models called process models.

a. New State

When a new process is created then it is said to be in new state.

b. Running State

A process is said to be in running state if it is under execution by the CPU. This means that the process actually using the CPU at that particular time.

c. Blocked (or waiting) State

A process is said to be in blocked state if it is waiting for some event to happen such as an Input/Output completion before it can proceed.

d. Ready State

A process is said to be in ready state if it is ready to be assigned to the CPU for processing.

e. Terminated state (or exit)

A process is said to be in terminated state if the CPU has finished its execution.

An example of a five state process model is given below:

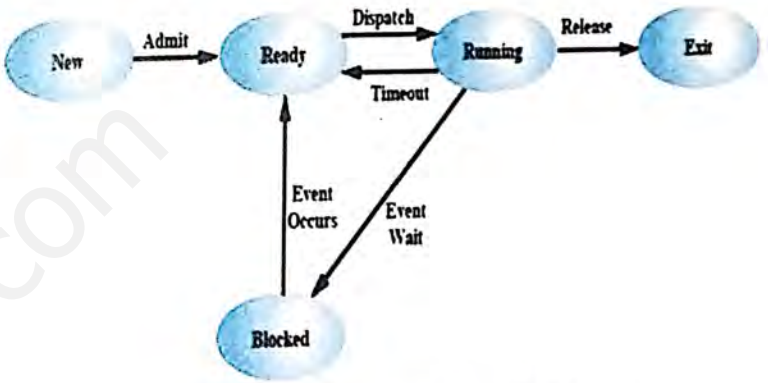


Figure 1.10: Five-State Process Model

1.3.3 Threads and processes

A **thread** can be defined as a dispatchable unit of a process that can be executed sequentially and is interruptible. It can also be defined as a single sequence stream within a process.

As threads are small independently executable units of processes, therefore, they have some of the properties of processes because of which they are sometimes called **lightweight processes**. In a process, threads allow multiple executions of streams.

Processes vs. Threads

- Processes are independent of one another while threads are not independent of one another.
- Unlike processes, all threads can access every address in the task.
- Processes might or might not assist one another because processes may originate from different users while threads are designed to assist one another.

1.3. 4 Multitasking and Multithreading

- The features of a multitasking operating system is to execute more than one **programs** simultaneously while the multithreading feature executes **different parts of the program**, called **threads**, simultaneously.
- As multitasking executes individual program, therefore, the multitasking operating systems use separate memory and data, while the multithreaded operating systems share the same memory and data variables for all threads of the same process.
- Implementation of multitasking is relatively easier in operating systems than the implementation of multithreading.

1.3. 5 Multitasking and Multiprogramming

Multiprogramming is the concept in which more than one user's programs are active at the same time in the main memory (RAM) of the computer and the operating system selects one job at a time from the ready queue and assigns it to the CPU for execution. The ready queue is always placed in the main memory of the computer. To understand this concept, consider the scenario of having the application programs such as MS Word, MS Excel, and MS Access opened with a user typing in MS Word document and leaving the others just in main memory without performing any task.

The concept of multitasking means performing multiple tasks in parallel. Always, a single CPU processes one task at a time but the switching of CPU between the processes is so fast that it gives the impression of simultaneous execution of multiple processes. While listening to music, we can browse Internet and print some document at the same time. In a broad sense, both the multiprogramming and multitasking are the same things with two different names.

Summary

- An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.
- The history of operating system goes back to 1950s and different types of operating system have been developed so far. DOS, Windows, Macintosh and UNIX are the examples of operating systems.
- Modern operating systems have been passed through a long evolutionary period starting from batch processing system, multiprogramming and then to distributed, parallel and embedded operating systems.
- Operating system acts like a controller and thus performs a wide range of functions and services such as process management, memory management, file management, I/O management, network management and ensuring security.
- A job under execution is called process. A process can be in one of ready, running or waiting state. It is the responsibility of the operating system to decide which process to be assigned to the CPU for execution and which to be suspended and placed back either in waiting or ready state.

- Multitasking, multiprogramming and multithreading are the features of nearly all modern operating systems which enable them to process multiple users jobs simultaneously in a parallel fashion.

Activities

ACTIVITY 1

The instructor should apply a few internal and external commands of MS-DOS to the students in computer lab and then provide a list of most commonly used MS-DOS commands and asks them to practice on these commands.

ACTIVITY 2

The instructor should practice the installation steps of Windows XP and Windows 7 Operating System in the class and then should, briefly, describe the main features of these operating systems to the students.

ACTIVITY 3

Utility programs are the programs that are considered as a component of operating system. They are installed automatically with the installation of operating systems. A few examples of utility programs are Notepad, WordPad, Calculator, Image Viewer and Antivirus. The instructor should briefly practice the students on these programs.

ACTIVITY 4

Drivers are the software that are used to interface computer hardware with the operating system and make them available to the users. The instructor should practice the installations of LAN card, Sound card and VGA card to the students.

Exercise

Q.1 Fill in the Blanks.

i. The concept of executing multiple programs simultaneously on multiple possessors is known as_____.

ii. Multitasking operating systems execute _____ simultaneously.

iii. The central theme of modern operating systems, based on the concept of switching among multiple programs in memory, is called _____.

iv. A process consists of five states i.e. new, ready, running, block and _____.

v. _____ is a multitasking, multi-user operating system for servers, desktops and laptops, originally developed in 1969 by a group of AT&T employees at Bell Labs.

Q.2 Select the correct choice for the following Multiple Choice Questions.

i. Operating system is an:
   a. Application software
   b. System software
   c. Driver
   d. None of the above

ii. DOS stands for:
   a. Doctor of system
   b. Document of system
   c. Disk operating system
   d. None of the above

iii. Which one of the following is correct for multiprocessing operating systems?
   a. Used on systems having only one processor
   b. Cannot execute more than one processes at a time
   c. Run on systems which having more than one processors
   d. None of the above

iv. Multiprogramming refers to:
   a. Having several programs in RAM at the same time
   b. Multiprocessing
   c. Writing programs in multiple languages
   d. None of the above

v. The most recently launched operating system of Microsoft is:
   a. Windows 98
   b. Windows 8
   c. DOS
   d. Windows NT

vi. Which of the following is not an operating system?
   a. Linux
   b. MS Word
   c. UNIX
   d. Windows XP Professional

vii. Which type of software enables you to solve specific problems or perform specific tasks?
   a. System software
   b. Operating system
   c. Utility software
   d. Application software

viii. Which type of software controls your application software and manages how your hardware devices work together?
   a. System software
   b. Operating system
   c. Utility software
   d. Application software

ix. A technique in which a process, executing an application, is divided into threads that can run concurrently is called:
   a. Multithreading
   b. Multiprocessing
   c. Batch processing
   d. None of the above

Q.3 Write TRUE/FALSE against the following statements.

i. The operating system acts as an interface between the computer hardware and the human user.
ii. Multiprogramming allows the processor to make use of idle time caused by I/O wait.
iii. An operating system controls the execution of applications and acts as an interface between applications and the computer hardware.
iv. In the early computers, the operating systems did not exist.
v. Multi-programming do not provide better utilization of system resources than multiprogramming.
vi. In a time sharing system, when a process is assigned to the CPU for execution, it remains there till its completion.
vii. A process is a dispatchable unit of thread.
viii. A process is directly assigned to the CPU for processing from a block state when the event for which it is waiting happens.

ix. A process is said to be in ready state if it is waiting for some external event which has not yet been occurred.

x. Batch processing systems does not support interactivity.

Q.4 What is an Operating system? Explain different types of operating systems.

Q.5 Differentiate between the following:

a) Single-user and Multi-users operating systems

b) Thread and Process

c) Multiprogramming and Multiprocessing

Q.6 Define distributed operating system and describe its advantages and disadvantages.

Q.7 Write short notes on the following.

a) Different states of a process

b) DOS

c) Real time processing systems

d) Embedded operating systems

Q.8 Describe functions of an operating system.

Q.9 Compare DOS, Windows and UNIX operating systems.

UNIT 2

SYSTEM DEVELOPMENT LIFE CYCLE

After the completion of Unit-2, Students will be able to:

- Define a system.
- Explain System Development Life Cycle and its importance.
- Describe objectives of SDLC.
- Describe stakeholder of SDLC and their roles.
- Explain the following phases of SDLC:
  - Planning
  - Feasibility
  - Analysis
  - Requirement engineering
  - Requirement gathering
    - Functional requirements
    - Non-functional requirements
  - Requirement validation
  - Requirement management
  - Design (algorithm, flowchart and pseudo code)
  - Coding
  - Testing/verification
  - Deployment/implementation
  - Maintenance/support
- Explain the role of the following in :
  - Management
  - Project manager
  - System analyst
  - Programmer
  - Software tester
  - Customer

2.1 INTRODUCTION

Systems are created to solve problems. Nowadays, systems are so big and complex that teams of architects, analysts, programmers, testers and users must work together to create the millions of lines of custom-written code that drive our enterprises. To manage this, a number of system development life cycle (SDLC) models have been created. This Unit is focused on SDLC, its importance, objectives, stakeholders and logical phases that are followed for the development of a software product.

2.1.1 A System

The term system is originated from the Greek term systēma, which means to "place together." It can be defined as a set of interrelated components having a clearly defined boundary that work together to achieve a common set of objectives.

A system can be developed by applying a set of methods, procedures and routines in a proper sequence to carry out some specific task. When all these functions are applied to build software then the system will be called as a software system.

System Development Life Cycle (SDLC)

The systems development life cycle (SDLC) is a conceptual model used in project management that describes the stages involved in an information system development project from an initial feasibility study through maintenance of the completed application.

2.1.2 System development life cycle and its importance

System Development Life Cycle (SDLC) is a step wise process of creating computer systems. It is also known as information system development or application development. It is a conceptual model which represents the necessary steps used for the development process of a software system.

The SDLC is a problem-solving process through which a series of steps helps to produce a new computer information system. The entire steps conducted in a sequence should provide the answers to a problem or opportunity. This step-wise procedure to build a system has a lot of importance.

Importance of SDLC

The following points summarize the importance of the use of SDLC.

- SDLC is important because it breaks down the entire life cycle of software development into phases thus making it easier for the development team members to easily evaluate each part of software development.
- SDLC makes it easier for programmers to work concurrently on each phase.
- It provides a rough time estimate that when the software will be available for use.
- It delivers quality software which meet or exceed customer expectations.
- It provides the basic framework for the developing of quality software.
- SDLC helps the project managers to establish a project management structure to be followed strictly during the system development.
- SDLC clearly defines and assigns the roles and responsibilities of all the involved parties.
- It ensures that the requirements for the development of the software system are well defined and subsequently satisfied.

2.1.3 Objectives of SDLC

The objectives of the System Development Life Cycle (SDLC) are as follows:

- Delivery of quality software that meet the customer expectations.
- Delivery of inexpensive and cost-effective software which are easily maintainable.
- Maximize productivity in terms of the software systems delivered.

- One of the major objectives of SDLC is to establish an appropriate level of management authority to direct, coordinate, control, review, and approve the software development project.
- SDLC should ensure the project management accountability.
- Proper documentation of all the requirements needed for the development of the new software system.
- Ensuring that projects are developed within the current and planned information technology infrastructure.
- SDLC should identify the potential project risks in advance so that proper planning should be done.

2.1.4 Stakeholders of SDLC

Those entities which are either within the organization or outside of the organization that sponsor a project, or have an interest or have the intention to get it after its successful completion, or may have a positive or negative influence in the project completion are called **stakeholders**. Project stakeholders include the customers, the user group, the project manager, the development team and the testers. All those who have some interest in the project can be considered as stakeholders of that project. The individuals as well as the organizations that are actively involved in the project, or whose interests may be affected as a result of project execution or project completion are the part of stakeholders. It is the duty of the project management team to identify the stakeholders, determine their requirements, expectations and manage their influence in relation to the requirements to ensure a successful project.

Role of stakeholders

The basic roles of the stakeholders are:

- For the development of a software system, resources such as time, money, equipment etc. are needed which should be provided to the project team by the stakeholders.

- Stakeholders educate the developers about their business.
- They spend more time to provide information and clarify requirements to the analysts and developers.
- The stakeholders should be specific and precise about the requirements.
- Make timely decisions.
- Respect a developer's assessment of cost and feasibility.
- Set requirement priorities.
- Review and provide timely feedback.
- Promptly communicate changes to requirements.

2.1.5 System development life cycle phases

The most commonly used phases of SDLC are given below.

- Planning
- Feasibility Study
- Analysis
- Requirement Engineering
- Design
- Coding
- Testing / Verification
- Deployment / Implementation
- Maintenance / Support

These phases are diagrammatically shown in Figure 2.1.

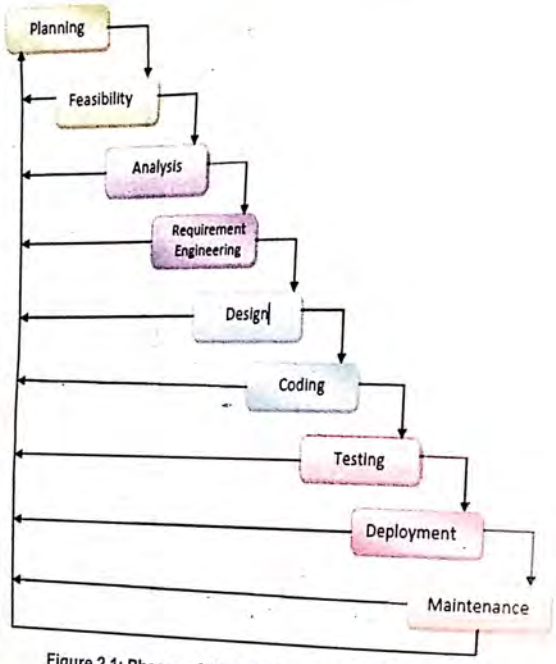


Figure 2.1: Phases of System Development Life Cycle

a. Planning phase

Planning phase is the first phase of System Development Life Cycle. During this phase, the group that is responsible for creating the software system must first determine what the system needs to do for the organization. Often, the planning phase tries to find answers for the questions such as:

- What do we need this system for?
- What the new software system will do for the organization?
- How this new software system will be developed?

In this initial phase of SDLC, all the resources, both human and technology resources are put together and a project plan is devised by the project manager.

Activities of the planning phase

- Define business problem or opportunity
- Define Project Scope and Constraints
- Produce detailed project schedule (time) and cost
- Define project benefits
- Write a report to management

b. Feasibility

Analysis and evaluation of a proposed project or system, to determine, whether it is technically, economically and operationally feasible within the estimated cost and time is called feasibility study. Feasibility study is one of the important steps in SDLC. It is divided into:

i. Technical feasibility

It refers to the technical resources needed for the development of the proposed software system.

ii. Economic feasibility

This refers to the cost of the project. Here, it tries to determine; total cost of ownership (TCO), tangible benefits and intangible benefits.

iii. Operational feasibility

In this type of feasibility it is determined that whether the proposed system will be used effectively after it has been developed. It means whether the organization has much experienced personals to operate the system or not?

iv. Schedule feasibility

It means that a project can be implemented in an acceptable time frame.

c. Analysis phase

In this phase, the incharge of the project team must decide if the project should go ahead with the available resources or not. Analysis is also looking at the existing system to see what and how it is doing its job. The project team asks the following questions during the analysis.

- Can the proposed software system be developed with the available resources and budget?
- Will this system significantly improve the organization?
- Does the existing system even need to be replaced?

Activities of the analysis phase

- Gather information to learn problem domain
- Define system requirements.
- Prioritize requirements.
- Build prototypes for feasibility and discovery of requirements.
- Generate and evaluate alternative solutions for the problem in hand.
- Review recommendations with top level management to decide about the project.

d. Requirement engineering

Requirements Engineering (RE) is a set of activities used to identify and communicate the purpose of a software system, and the contexts in which it will be used. Requirement engineering consists of the following steps.

- Requirement gathering
- Requirement validation
- Requirements management

Requirement gathering

Requirement gathering is usually the first part of any software product. In this phase, meetings with the customers or prospective customers are arranged; the market requirements and features that are in demand are analyzed. It is also to find out if there is a real need in the market for the software product that is proposed to be developed.

These requirements are of two types:

- Functional requirements
- Non-Functional Requirements

Functional requirements: Functional requirement are those requirements of a software system which describe a function of a software system or its component. It includes calculations, technical details, data manipulation and processing and other specific functionality that define what a system is supposed to accomplish. Examples of functional requirements are:

a) Interface requirements

Field accepts numeric data entry.

Field only accepts dates before the current date.

Screen can print on-screen data to the printer.

b) Business requirements

Data must be entered before a request can be approved

c) Regulatory/Compliance requirements

The database will have a functional audit trail.

The system will limit access to authorized users.

The spreadsheet can secure data with electronic signatures.

d) Security requirements

Member of the Data Entry group can enter requests but not approve or delete requests.

Members of the Managers group can enter or approve a request, but not delete requests.

Members of the Administrators group cannot enter or approve requests, but can delete requests.

Non-functional requirements: Non-functional requirements are those requirements which specify criteria for the judgment of the operations of a system. It describes that how well the system performs its duties. Nonfunctional requirements are often called qualities of a system. These requirements depend upon the nature of the software.

Different types of non-functional requirement are:

- Accessibility Requirements
- Accuracy Requirements
- Backup and Recovery Requirements
- Memory Capacity Requirements
- Compatibility Requirements
- Error-Handling Requirements
- Maintainability Requirements
- Performance Requirements
- Security Requirements

Requirements validation

Requirement validation is concerned with examining the requirements to certify that they meet the intentions of the stakeholders, and to ensure that they define the right system, the essence of the agreement and understandings between developer and acquirer about what to build, in a manner that ensures a common understanding across the project team

and among the stakeholders. The validation differs from verification in the sense that verification occurs after requirements have been accepted.

In requirements validation, the requirements elicited are reviewed to check that requirements are complete, correct, unambiguous, consistent, prioritized, modifiable, and traceable.

Re quirements management

Requirements management is performed to ensure that the software continues to meet the expectations of the acquirer and users. The requirements document should always accurately reflect those expectations, as changes occur, in order to communicate to the development team.

Requirements management, therefore, needs to gather new requirements that arise from changing expectations, new regulations, or other sources of change. It needs to analyze the impact of those changes on the project. The conflicts that arise among the acquirer and the user needs to be addressed and changes should be negotiated and validated before such requirements are added to the baseline.

e. Design phase

During this phase, the system is designed to satisfy the functional requirements identified in the previous phase. Since problems in the design phase can be very expensive to solve in later stages of the software development, therefore, a variety of elements are considered in the design to minimize the risk. These include:

- Identifying potential risks and defining mitigating design features.
- Performing a security risk assessment.
- Developing a conversion plan to migrate current data to the new system.
- Determining the operating environment.

- Defining major subsystems and their inputs and outputs.
- Allocating processes to resources.

The design phase normally consists of three different architectures. These are:

- Algorithms
- Flow chart
- Pseudo code

i. Algorithms

An algorithm is a specific set of instructions for carrying out a procedure or solving a problem.

For Example the following algorithm will find average of the 5 numbers.

```
Sum = 0, Average = 0
Read a, b, c, d, e
Sum = a+b+c+d+e
Average = Sum/5
Print Average
```

ii. Flowcharts

A flowchart is a type of diagram that represents an algorithm or a process. It shows the steps of the algorithms with the help of boxes and their order by arrows connecting the boxes. This diagrammatic representation of the algorithm gives a step-by-step solution to a given problem. The operations are represented in these boxes, and the flow of control on the arrows. These flowcharts are used for analyzing, designing, documenting or managing a process or program in various fields.

Some most commonly used flowchart symbols are shown in Figure 2.3.

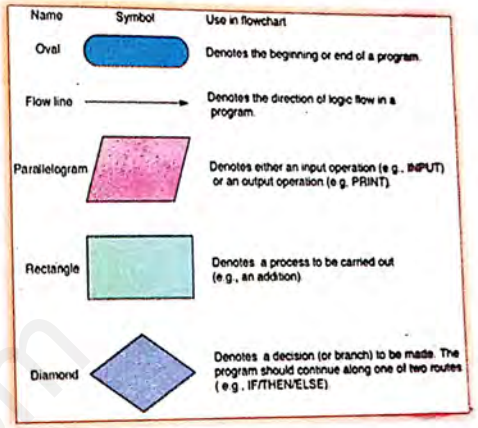


Figure 2.2 Flowchart Symbols

Example: Flowchart for the above algorithm will be as follows.

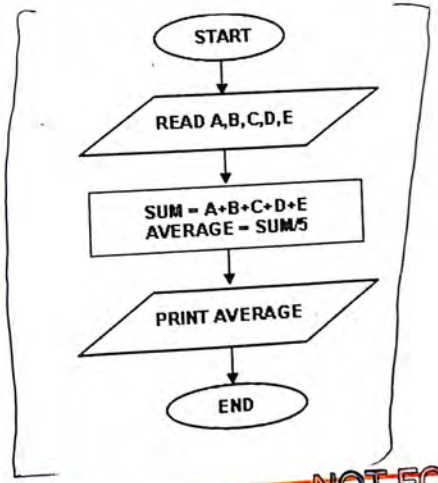


Figure 2.3 Flowchart

iii. Pseudo code

Pseudo code is an outline of a program, written in a form that can easily be converted into real programming statements. Pseudo code neither is compiled nor executed, and there is no specific formatting and syntax rules for it.

Pseudo code is beneficial in the sense that it enables the programmer to focus on the algorithms without worrying about all the syntactic details of a particular programming language. In fact, one can write pseudo code without even knowing what programming language will be used for the final implementation.

The following example shows a pseudo code to display the result of a student as "Pass" if the percentage is greater than or equal to 60% and "Fail" otherwise.

```
IF Percentage >= 60%
Print "Pass"
Else
Print "Fail"
```

f. Coding

Coding is the process of designing, writing, testing, debugging, and maintaining the source code of computer programs. This source code is written in programming languages. The purpose of coding is to create a program that shows a certain desired behavior. For coding some other specialties are also needed. These include knowledge of the application domain, specialized algorithms and formal logic. Coding is also called computer programming.

For coding computer programming languages are used. There are different types of programming languages such as procedural languages and object oriented programming (OOP) languages. Nowadays, object oriented programming languages are the most

commonly used languages. C++, Java, C#.Net, and VB.Net etc. are the examples of this group of languages.

The coding style of the C++ programming language is shown as follows.

Example 1

```
/* program to add N numbers*/
#include <iostream.h>
#include<conio.h>
int main ()
{
int N, sum=0, y ;
cout<< "Enter value for N"<<endl;
cin>>N ;
for (int x=1 ; x<=N ; ++x)
{
cin>>y ;
sum= sum+y ;
}
cout<< "The sum of N values is: "<<sum;
getch();
return 0;
}
```

Example 2

```
/* program to display result s of the students*/
#include <iostream.h>
#include<conio.h>
int main ()
```

```
{
float Percentage;
cout<< "Enter Percentage of the student"<<endl;
cin>> Percentage;
if (Percentage >=60.0)
cout<< "Passed";
else
cout<< "Failed";
getch();
return 0;
}
```

g. Testing/Verification

The execution of a program to find its errors is called testing. Here, the bugs are identified in the programmed modules. The purpose of testing is to evaluate an attribute or capability of a program or system and determine that whether it meets its required results. Testing/verification the software is actually operating the software under controlled conditions. It is the process of checking the items for consistency by evaluating the results against pre-specified requirements.

h. Deployment / Implementation

Software deployment is a set of activities that are used to make the software system available for use. The deployment is also called implementation. Its activities occur at the producer site or at the consumer site or both. Because every software system is unique, the precise processes or procedures within each activity are different from others and thus cannot be clearly defined.

The main activities that are involved during deployment are:

- Installation and activation of the equipments and software.

- In some cases the users and the computer operation personals are trained on the developed software system.
- Conversion: The process of changing from the old system to the new one is called conversion.

i. Maintenance / Support

Keeping a system in its proper working condition is called maintenance. In other words, it can also be defined as the modification of a software product after delivery to correct faults, to improve performance. Maintenance is an important activity that takes place throughout the life of software in both the computer equipments and computer software. Programmers spend more time in maintaining programs than writing them.

In SDLC, the system maintenance is an ongoing process. The system is monitored continually for performance in accordance with user requirements and needed system modifications are incorporated. When modifications are identified, the system may reenter the planning phase. This process continues until a complete solution is provided to the customer. Maintenance can be either be repairing or modification or some enhancement in the existing system.

2.1.6 Role of the Personnel involved in SDLC

The activities of all the phases of software development life cycle are being performed by different groups of people and individual personnel. These personnel are professionals in performing their particular jobs and include:

- Management
- Project Manager
- System Analyst
- Programmer
- Software Tester
- Customer

a. Management

Organizing, coordinating and controlling the activities of a software development by the managers and executives in accordance with certain standard procedures is called management. A strong management has the ability to satisfy the customers and acquirers of the software system. Also, a proposed project or a product will only meet its objectives if managed properly, otherwise, will result in failure.

The role of good Management is to:

- Provide consistency of success of the software with regard to Time, Cost, and Quality objectives.
- Ensure that customer expectations are met.
- Collect historical information and data for future use.
- Provide a method of thought for ensuring all requirements are addressed through a comprehensive work definition process.
- Reduce risks associated with the project.
- Minimize scope creep by providing a process for managing changes.

b. Project Manager

A **project manager** is a professional in the field of project management responsible for planning, execution, and closing of any project. Apart from management skills, a software project manager will typically have an extensive background in software development. He is also expected to be familiar with the whole software development life cycle process. The specific responsibilities of the project manager vary depending on the company size, the company maturity, and the company culture.

The role of project manager is to manage:

- Project plan

- Project stakeholders
- Project team
- Project risk
- Project schedule
- Project budget
- Project conflicts

c. System Analyst

A systems analyst is a professional in the field of software development that studies the problems, plans solutions for them, recommends software systems, and coordinates development to meet business or other requirements. System analyst has expertise in a variety of programming languages, operating systems, and computer hardware platforms. Because they often write user requests into technical specifications, the systems analyst is the contact person between vendor and information technology professionals. Analyst may be responsible for developing cost analysis, design considerations, and implementation time.

The role of system analyst is to:

- Plan a system flow from the ground up.
- Interact with customers to learn and document requirements that are then used to produce business requirements documents.
- Write technical requirements from a critical phase.
- Interact with designers to understand software limitations.
- Help programmers during system development, e.g. provide use cases, flowcharts or even database design.
- Manage system testing.
- Document requirements and contribute to user manuals.

- Whenever a development process is conducted, the system analyst is responsible for designing components and providing that information to the developer.

d. Programmer

A programmer is a technical person that writes computer programs in computer programming languages to develop software. A programmer writes, tests, debugs, and maintains the detailed instructions that are executed by the computer to perform their functions. Other names for programmer are coder and developer.

The roles of a programmer are:

- Writing, testing, and maintaining the instructions of computer programs.
- Updating, repairing, modifying and expanding existing programs
- Testing the code by running to ensure its correctness
- Developing new methods and approaches to computer programming
- Consultancy with outside parties in relation to the construction of computer programming methods and the programs themselves.
- Supervision of the information systems (in some cases).
- Maintenance of computer databases is another type of specific duty which à computer programmer may find himself/herself responsible.
- Preparing graphs, tables and analytical data displays which show the progress of a computer program.

e. Software Tester

A software tester is a computer programmer having specialty in testing the computer programs using different testing techniques. Software tester is responsible for understanding requirements, creating test scenarios, test scripts, preparing test data, executing test scripts and reporting defects and reporting results to test lead.
The roles of a software tester are:

- Create test scenarios, test conditions and expected results and test cases.
- Run and maintain automated test scripts.
- Create required test data and maintain common test data sheet of the project.
- Create test deliverables as per testing standards followed by the company or project.
- Execute test scripts and document test results.
- Inform testing team leader on any issues that could potentially impact quality, schedule or budget of the project.

f. Customer

A Customer is an individual or an organization that is a current or potential buyer or user of the software product. Customers usually purchase software from software manufacturer companies (software houses), users groups and individuals. Customers are also called clients but the only difference between the two is that the customers purchase the software products and the clients purchase services. Customers are the real evaluators of a software product by using it and identifying its merits and demerits.

Summary

- System can be defined as a set of interrelated components having clearly defined boundary that work together to achieve a common set of objectives.
- The systems development life cycle (SDLC) is a conceptual model used in project management that describes the stages involved in an information system development project, from an initial feasibility study through maintenance.
- The advantage of System Development Life Cycle (SDLC) is that it provides a step wise process to the development of computer Software.
- The main objective of SDLC is to obtain quality software that cost low price and satisfy the customers.
- Stakeholders are those entities which are either within the organization or outside of the organization that sponsor a project, or have an interest or have the intention to get it after its successful completion, or may have a positive or negative influence in the project completion.
- Planning phase is the first phase of Software Development Life Cycle in which it is determined what the system needs to do for the organization.
- Analysis and evaluation of a proposed project or system is to determine, whether it is technically, economically and operationally feasible within the estimated cost and time is called feasibility study.
- Requirements Engineering (RE) is a set of activities used to identify and communicate the purpose of a software system, and the contexts in which it will be used.
- Analysis is the process of looking at the existing system to determine what and how it is doing its job.
- Design phase is that phase of SDLC in which the system is designed to satisfy the functional requirements identified in the requirement engineering phase.
- Coding is the process of designing, writing, testing, debugging, and maintaining the source code or computer programs.

- Testing is the process of executing users programs to find its errors. In this process, bugs are identified in the programmed modules.
- Software deployment is a set of activities that are used to make the software system available to the users for use. The deployment is also called implementation.
- Software maintenance is the process of keeping a system in its proper working condition.

Activities

For the following activities the instructor should divide the students into groups. Each group should be assigned one of the projects discussed in the activities. The groups are to prepare a description of what needs to be carried out for their projects for each phase of the software development life cycle (SDLC).

ACTIVITY 1

An automotive manufacturing company contacts your IT Company to develop a new system for stock control of automotive parts. The current system is manual.

ACTIVITY 2

A government department contacts your IT Company to develop a new system for maintaining patients' appointments at the hospital. The current system is manual.

ACTIVITY 3

A college contacts your IT department to develop a new system for lecturers to enter student's grades and attendance. The current system is manual. List the personnel involved in this project.

ACTIVITY 4

A cinema manager wants to upgrade his current system of booking seats so that people will also be able to book through the internet. He has contacted your company for the job.

ACTIVITY 5

A large travel agency has contacted your IT department within the travel agency to upgrade the current outdated computerized system which the company is using to book seats.

ACTIVITY 6

Teacher should help the students in drawing flowcharts for the above five (5) projects mentioned in activities 1, 2, 3, 4, and 5, and assign some new problems as well as assignments to draw flowcharts for them.

Exercise

Q.1 Fill in the Blanks.

i. The requirements which define the function of a software system are called_____ requirements.

ii. All the activities that make a software system available for use is known as software_____.

iii. The person who is familiar with a variety of programming languages, operating systems and computer hardware is known as _____.

iv. A software tester is basically a _____ having a specialty in testing the computer programs.

v. A project manager is a professional in the field of _____.

Q.2 Select the correct choice for the following Multiple Choice Questions.

i. The first step in the system development life cycle is:

a) Analysis
b) Design
c) Problem Identification
d) Development and Documentation

ii. The organized process or set of steps that needs to be followed to develop an information system is known as:

a) Analytical cycle
b) Design cycle
c) Program specifications
d) System development life cycle

iii. Enhancements, upgradation and bugs fixation are done during the _____ step in the SDLC.

a) Maintenance and Evaluation
b) Problem Identification
c) Design
d) Development and Documentation

iv. The _____ determines whether the project should go forward.

a) Feasibility
b) Problem identification
c) System evaluation
d) Program specification

v. _____ spend most of their time in the beginning stages of the SDLC, talking with end-users, gathering requirements, documenting systems and proposing solutions.

a) Project managers
b) System analysts
c) Network engineers
d) Database administrators

vi. The entities having a positive or negative influence in the project completion are known as _____.

a) Stakeholders
b) Stake supervisors
c) Stake owners
d) None of the above

vii. System maintenance is performed in response to _____.

a) Business changes
b) Hardware and software changes
c) User's requests for additional features
d) All of the above

Q.3 Write TRUE/FALSE against the following statements.

i. A stakeholder is any person or organization who has a direct or indirect interest in the project.
ii. System maintenance is performed to gather the requirements of the proposed system.
iii. The how, when, what type of questions are answered in the planning phase of SDLC.
iv. The three architectures i.e. algorithms, pseudo code and flowcharts are used in the implementation phase of SDLC.
v. Each phase of SDLC is accomplished as a discrete, separate step.

Q.4 Define software development life cycle (SDLC). What are its objectives?

Q.5 What is a system? Where exactly the testing activities begin in SDLC?

Q.6 Why software development life cycle is important for the development of software?

Q.7 Who are stakeholders of SDLC? Describe their responsibilities.

Q.8 What is meant by the term software requirement? Differentiate between functional and non-functional requirements.

Q.9 Design a flowchart for the following algorithm.

```
sum = 0, N=5
x = 1
while x ≤ N do
sum = sum + x
x = x + 1
end while
print sum
```


UNIT 3

PROGRAMMING USING C++

After the completion of Unit-3, Students will be able to

- Define Program.
- Define header file and reserved words.
- Describe the structure of a C++ program.
- Know the use of statement terminator (;).
- Explain the purpose of comments and their syntax.
- Differentiate between Constant and Variable.
- Explain the rules for specifying variable names.
- Know different Data types in C++ and their size in bytes.
- Define constant qualifier- const.
- Explain the process of declaring and initializing variables
- Use type casting.
- Explain the use of **cout** and **cin** statements.
- Define **getch**(), **gets**() and **puts**() functions.
- Define and use escape sequences (\a, \b, \r, \t, \\, \', \").
- Make use of commonly used I/O handling functions.
- Use manipulators **endl** and **setw**.
- Define and use different types of operators in C++.
- Identify Unary, binary and ternary operators.
- Define and explain expression and compound expression.
- Explain the order of precedence of operators.

3.1 INTRODUCTION

This Unit is about the introduction to Object Oriented Programming Language, C++, and covers its basics in detail. The Unit is focused on basic C++ program structure, variables used in C++, input/output statements and the use of operators in the language.

Bjarne Stroustrup at Bell Labs initially developed C++ during the early 1980's. It was designed to support the features of C such as efficiency and low-level support for system level coding. Added to this were features such as classes with inheritance and virtual functions, derived from the Simula language, and operator overloading, derived from Algol. C++ is best described as a superset of C, with full support for object-oriented programming. This language is in wide spread use. The first commercial release of the C++ language was in October of 1985.

3.1.1 Computer program

A computer program is a set of instructions written in computer languages to perform a specified task for a computer. A computer program tells the computer that what to do and in which order to do. Different types of programming languages are used to develop programs. Some commonly used programming languages are C++, JAVA, SQL, HTML, etc.

3.1.2 Header files and Reserved words

Header files and reserved words are the two important components of almost every C++programs.

a. Header files

Header files also known as **include files,** are standard library files that have an extension of (.h) and which are used to hold declarations for other files.

Consider the following program:

```
// header file example
#include <iostream.h>
int main()
{
cout << "Hello, world!" << endl;
return 0;
}
```

Output of the program

Hello, world!

This program prints the string "Hello, world!" to the screen using *cout*. However, this program never defines *cout*, so how does the compiler know about the object *cout*? The answer for this is that *cout* has been declared in a header file called "*iostream*". When the line *#include <iostream.h>* is used in the program, the compiler locates and read all the declarations from a header file named "*iostream*".

b. Reserved words

Reserved words or keywords are those words which have their special meaning within the C++ language and are reserved for some specific purpose. C++ reserved words cannot be used for any other purpose in a C++ program and even cannot be used as variables. Here is a list of C++ keywords shown in Table 3.1.

asm	auto	bool	break	case	catch	char	class
const	const_cast	continue	default	delete	do	double	dynamic_cast
else	enum	explicit	export	extern	false	float	for
friend	goto	if	inline	int	long	mutable	namespace
new	operator	private	protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast	struct	switch	template
this	throw	true	try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile	wchar_t	while	

Table 3.1: List of C++ Keywords

3.1.3 Structure of a C++ program

General syntax of a C++ program is given below.

```
Preprocessor directives <header file>
int main()
{
Body of the program
}
```

Each C++ program has three main components. These are: **Preprocessor directives**, **main** function and body of the **main** function. Now, consider the following "Hello World" example to explain these components.

```
#include <iostream.h>
#include <conio.h>
int main()
{
cout << "Hello World!";
getch();
return 0;
}
```

Output of the Program

Hello World!

The result of the above program is that it prints "Hello World!" on the screen. It is one of the simplest programs that can be written in C++, but it contains the fundamental components of almost every C++ program.

a. Preprocessor directives (#include, #define)

A preprocessor is a collection of special statements which are executed before the compilation process.

Almost every C++ program contains a preprocessor directive. The *#include* preprocessor directives is commonly used to insert *header files* with extension '.h'. These header files are already stored in computer in *include* directory. In the above program, two *#include* directives have been used, *#include<iostream.h>* and *#include<conio.h>*. *#include<iostream.h>* is used for the C++ object *cout* and *#include<conio.h>* for the *built-in* function *getch()*.

In C++, the other commonly used preprocessor directive is *#define* which is used to define symbolic constants. Its general format is:

***#define* identifier value**

For example:

```
#define PI 3.14159
#define NEWLINE '\n'
```

This defines two new constants: PI and NEWLINE. Once they are defined, they can be used in the rest of the program as if they were any other regular constant, for example:

```
#include <iostream.h>
#include <conio.h>
#define PI 3.14159
#define NEWLINE '\n'
int main()
{
```

```
double r=5.0; // radius
double circle;
circle = 2 * PI * r;
cout << "Area of the Circle: "<<circle;
cout << NEWLINE;
getch();
return 0;
}
```

Output of the Program

Area of the Circle: 31.4159

The *#define* directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (;) at its end.

b. main( ) Function

The *main* function is the point by where all C++ programs start their execution. It does not matter whether there are other functions with other names defined before or after it, the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a *main* function.

The word *main* is followed in the code by a pair of parentheses ( ).

c. Body of the main( ) function

After the parentheses of the *main* function, the body of *main* function starts which is enclosed in braces { }. When this function is executed, the instructions written within it perform their task. Inside the body of *main* function, C++ statements are written.

C++ Statements

A C++ *statement* is a simple or compound expression that can actually produce some effect. It is actually an instruction to the computer for taking an action.

For example *cout << "Hello World!";* is a C++ statement. The following are some examples of statements.

cout << "Hello World!";

In fact, this statement performs the only action of displaying the result "Hello World!" on the screen.

cout is the name of the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream. *cout* is declared in the *<iostream.h>* standard header file.

getch ();

This is a standard library function defined in the header file *<conio.h>* and is used to wait for the user to input a character from the keyboard.

return 0;

The return statement causes the *main* function to return control to the operating system. Return may be followed by a return code. The code 0 returned by the *return* statement tells the operating system that the program terminates normally.

The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, the following program can also be written in a single line.

```
int main()
{
```

```
cout << " Hello World!";
return 0;
}
```

Its single line version will be:

```
int main() { cout << "Hello World!"; return 0; }
```

3.1.4 Statement terminator (;)

Each C++ statement is terminated by a symbol named semicolon (;) which is called *statement terminator*. In C++, the separation between statements is specified with this ending semicolon (;) at the end of each statement. It does not matter to write more than one statement on a single line but matter if you do not separate them with semicolons. The use of each statement on a separate line is only to add clarity in the program.

3.1.5 Comments and their syntax in C++

Comments are parts of the source code ignored by the compiler. They do nothing but simply increase the readability and understandability of a program. Their purpose is only to allow the programmers to insert notes or descriptions within the source code.
C++ supports two types of comments:

a. Single line comment

It is represented by the double slash symbol //. It discards everything from where the pair of slashes signs // is found up to the end of the same line.

b. Multiline comment

These types of comments are represented by the symbols /* ..... */. It ignores everything between the /* characters and the first appearance of the */ characters, with the possibility of including more than one line.

Consider the following program to demonstrate comments.

```
/* program to demonstrate comments in C++*/
#include <iostream.h>
#include<conio.h>
int main()
{
cout << "Welcome! "; // prints Welcome!
cout << "I'm a C++ program demonstrating comments";
getch();
return 0;
}
```

Output of the Program

```
Welcome!
I'm a C++ program
```

In the above example, the first two lines are multi-line comments while the comment at line 7 is a single line comment.
Comments are always ignored by the compiler when it compiles the program. If comments are included within the source code of your programs without using // or /*, the compiler will take them as if they were C++ expressions.

3.2. C++ Constants and Variables

Variables and constants are the important components of every programming language.

3.2.1 Constants and Variables

a. Constants

Constants are the expressions which have fixed value. Constants are generally categorized into: Literal, Boolean and Symbolic constants. They are used to express particular values within the source code of a program.

Consider the following line of code:

VarA = 555;

In this line of code, 555 is a **literal constant.** The literal constants are further categorized into the following types.

- String constants
- Numeric constants
- Character constants

i. String constants

A string constant is a sequence of characters enclosed in double quotation marks is called string constant. Its maximum length is 256 characters. Following are some examples of valid string constants.

"Welcome to the first C++ Program"
"The result="

ii. Numeric constants

Numeric constants consist of positive or negative signed numerals. There are four types of numeric constants. These are:

- Integer constant
- Floating point constant
- Hexadecimal constant
- Octal constant

- ***Integer constants***: Integer constants are those positives or negative signed numbers that do not contain a decimal point e.g. 2010, -321 etc.

- ***Floating point constants:*** The numeric constants having a decimal point are called floating point constants e.g. 33.55 and -0.22 etc. These constants can also be either positive signed or negative signed. These constants can also be represented in its exponential form by the use of alphabet 'e' or 'E' to denote the exponents of the numbers. For example.

  9010E10
  7810.11E-11
  -10.990e8
  -1.001e-1

- ***Hexadecimal constants:*** The constants represented in hexadecimal number system are called hexadecimal constants. To represent a hexadecimal constant, begin the constant with 0x or 0X, followed by a sequence of digits in the range 0 through 9 and a through f (or A through F). The digits a to f (or A to F) represent values in the range 10 through 15. For example.

```
int i = 0x3fff; // Hexadecimal constant
int j = 0X3FFF; // Hexadecimal constant
```

- ***Octal constants:*** The representation of constants in octal number system is called octal constants. To specify an octal constant, begin the specification with 0, followed by a sequence of digits in the range 0 through 7. Consider the following example:

```
int i = 0377; // Octal constant
int j = 0397; // Error: 9 is not an octal digit
```

Octal numbers are integer numbers of base 8 and their digits are 0 to 7.

iii. Character constants

A character enclosed within single quotes is called character constant.
Consider the following examples:

'A', 'a', '.', '?'

b. Variables

A variable in C++ is a name for a piece of memory that can be used to store information. A variable can be thought as a mailbox where information can be put and retrieved from. All computers have memory, called Random Access Memory (RAM) that is available for programs to use. When a variable is declared, a piece of that memory is set aside for that variable.

Consider the following segment of code.

```
int a = 5;
int b = 2;
int result = a - b;
```

In this segment, a, b and result are the integer variables that reside in RAM and store integer values.

3.2.2 Rules for naming variables

The symbol used for a variable is called an identifier or variable name. A set of rules are used for naming variables. These rules are:

- Valid identifier is a sequence of one or more letters, digits or underscores characters (_).

- Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid.
- Variable identifiers always have to begin with a letter. They can also begin with an underscore character(_).
- In no case, a variable can begin with a digit.
- A Reserved Word of the C++ language cannot be used as an identifier.
- Names should be meaningful.
- Names should not be too long.

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

C++ language is a *"case sensitive"* language which means that an identifier written in capital letters is not equivalent to the one written in small letters.

3.2. 3 Declaration and Initialization of variables

a. Declaration of variables

In order to use a variable in C++, we must first declare it. The declaration of a variable specifies that which type of data this variable will store/use. The syntax to declare a new variable is to write the specifier of the desired data type such as *int, bool, float* etc. followed by a valid identifier. For example:

Data type variable_name;

int a;

When this statement is executed by the CPU, a piece of memory from RAM will be set aside and it will be named as a. More than one variables of the same type can be declared in a single statement by separating them with commas. For example:
int a, b, c;

This line of code declares three variables *a*, *b* and *c*, all of type *int*. This declaration is exactly the same as shown in the following segment of code:

```
int a;
int b;
int c;
```

Consider the following example to see the use of variable declaration.

```
#include <iostream.h>
#include <conio.h>
int main()
{
// declaring variables:
int a, b;
int result;
// process:
a = 55;
b = 22;
result = a - b;
// print out the result:
cout << "The Result is= "<<result;
// terminate the program:
getch();
return 0;
}
```

Output of the Program

The Result is= 33

The declaration of the variables is not necessary at the start of the program. A variable can be declared in the program where it is needed. Consider the following example.

```
// variables declaration at the point of application
#include<conio.h>
#include<iostream.h>
int main()
{
cout << "Enter a number: ";
int x; // we need x starting here.
cin >> x;
cout << "Enter another number: ";
int y; // we don't need y until now
cin >> y;
cout << "The sum is: " << x + y << endl;
getch();
return 0;
}
```

Output of the Program

Enter a number: 4
Enter another number: 10
The sum is: 14

b. Initialization of variables

Assigning values to a variable during declaration time is called *initialization*. When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable.

Consider the following program to demonstrate the initialization of variables.

```
#include<conio.h>
#include <iostream.h>
int main()
{
int Vara=55; // initial value = 55
int Varb(22); // initial value = 22
int result; // initial value undetermined
result = Vara - Varb;
cout << "The Result is= "<<result;
getch();
return 0;
}
```

Output of the Program

The Result is= 33

3.2. 4 Fundamental data types in C++

While programming, we store the variables in computer's memory, but the computers have to know that what kind of data we want to store in them.

Data type tells the compiler that what types of data, means integer, character or floating point etc. the variable will store and what will be the range of the values.

The most commonly used data types in C++ are:

- Character
- Integer
- Floating point

- Boolean
- Unsigned

a. Character data type

The *char* keyword represents the character data type in C++. Any character belonging to the ASCII character set is called a character data type. The maximum size of *char* data type is 8 bits (1 byte). *Signed char* and *unsigned char* are its two distinct types. They occupy the same amount of memory space. Consider the following line of code:

char x;

Here, x is a character type variable that occupies 1 byte of memory. Variables declared as **char** data type are used to store character constants. Character constants are always represented with single quotes. For example:

```
char gender;
gender= 'M';
```

Here, gender is a character type variable that occupies 1 byte of memory and store character constant, M, which stands for Male.

b. Integer data types

The data types which store only integer numbers such as 100, -200 etc. are called integer data types. Its sub types are:

- int
- short int
- long

i. int data type

The keyword int stands for the integer data type in C++ and normally has two bytes of memory (16 bits). A 16 bit integer may fall in the range of -2^{15} to $2^{15}-1$, which means that it can store values in the range of -32768 to 32767.

ii. short int data type

short int is used to declare the short integer data type in C++. The maximum size of the *short int* data type is 2 bytes (16 bits). It may fall in the range -2^{15} to $2^{15}-1$, which means that it can store the values from –32768 to 32767.

iii. long int data type

long int stands for long integer data type and is used in C++ to store larger integer values. Its size is 4 bytes (32 bits) which means that it can stores values in the range of –2147483648 to 2147483647.

c. Floating point data type

Integers are only used for storing whole numbers, but sometimes we need to store very large numbers, or numbers with a decimal point. The data types which are used to store such type of data are called **floating point** data types. Variables of floating point types hold real numbers, such as 4.0, 2.5, 3.33, or 0.1226. There are three different types of floating point data types:

- float
- double
- long double

i. float data type

In C++, the *float* data type is used to declare floating point type of variables to store numbers with decimal point. It needs 4 bytes (32 bits) of memory to store values.

ii. double data type

double is a keyword to represent double precision floating point numbers in C++. *double* data type occupies 8 bytes (64 bits) of memory. The size of *double* data type is a rational number in the same range as long float and is stored in the form of floating point representation with binary mantissa and exponent.

iii. long double data type

In C++, the *long double* data type is used to declare *long double* type of variables to represent long double precision floating point numbers in C++. It needs 10 bytes (80 bits) of memory space in the RAM.

d. Boolean data type

bool is the keyword used to represent Boolean data type. It is capable of holding one of the two values: true (1) or false (0). It occupies 1 byte of memory.

bool flag;

e. Unsigned data type

The data types discussed so far are signed data types which means that one bit is reserved for the sign of the value (i.e. +/-). The unsigned numbers are whole numbers and always hold positive values starting from 0 till its maximum size. Here, the sign bit also used for the value which means no bit is reserved for holding a sign (+/-). The unsigned numbers may be classified into four types:

- unsigned char
- unsigned integer
- unsigned short integer
- unsigned long integer

A summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one is given in Table 3.2.

Data Type	Name	Description	Size	Range
character	char	Character or small integer	1byte	signed: -128 to 127
				unsigned: 0 to 255
Integer	short int (short)	Short Integer	2bytes	signed: -32768 to 32767
				unsigned: 0 to 65535
	int	Integer (16 bit system)	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
		Integer (32 bit system)	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
	long int (long)	Long integer	4bytes	signed: -2147483648 to 2147483647
				unsigned: 0 to 4294967295
Floating Point	float	Floating point number	4bytes	3.4×10^{-38} to 3.4×10^{38}
	double	Double precision floating point number	8bytes	1.7×10^{-308} to 1.7×10^{308}
	long double	Long double precision floating point number	10bytes	1.7×10^{-4932} to 1.7×10^{4932}
Boolean	bool	Boolean value. It can take one of two values: true or false.	1byte	true (1) or false (0)

Table 3.2: C++ Data Types

3.2.5 Constant qualifier

const keyword is used to define constant identifier whose values remain constant during the whole life of the program. *Const* identifier must be assigned with a value when declared, and then that value cannot be changed. Here is the segment of code used to define constant using *const* identifier keyword.

```
const int DollarRate = 86;
```


Computer Science (XII)

Declaring a variable as **const** prevents the programmers from unintentionally changing its value. Consider the following lines of code to change the value of variable DollarRate.

```
const int DollarRate = 86;
DollarRate = 123; // compiler error!
```

A compiler error message will be displayed by the compiler because the code at line 2 tries to change the value of the variable DollarRate from 86 to 123, which is declared as constant. It is an alternative to **#define** preprocessor directive which is used for defining constants in a program.

3.2.6 Type casting operator

The conversion of data from one type to another is called **type casting**. It has two types.

- Implicit type casting
- Explicit type casting

In **Implicit type casting**, the compiler automatically converts data from one type to another when they are used in expressions. When a value of one type is assigned to another type, the compiler implicitly converts that value into the value of the new type. For example:

```
double a= 3; // implicit casting to double value 3.0
int b= 3.14156; // implicit casting to integer value 3
```

In C++, there are several ways to convert one type to another but the simplest one is to precede the expression to be converted by the new type enclosed in parentheses ( ). Such type of conversion is called **explicit type casting**.

Consider the following segment of code:

```
int x;
float pi = 3.14;
x = (int) pi;
```

In the above code, the float number 3.14 is converted to an integer value 3 and the remainder is discarded. Here, the type casting operator is (**int**). Another way for explicit type casting is to precede the expression to be converted by the type and enclose the expression in parentheses. Consider the following line of code:

```
x = int ( pi );
```

3.3 Input/Output Handling

For input/output operations, C++ uses *streams* in sequential media such as the screen or the keyboard. A stream is an object where a C++ program inserts characters to or extracts the characters from stream.

All the standard input and output stream objects are declared within the header file **iostream** which is a part of the standard C++ library.

3.3.1 Standard output (cout)

The default standard output of a C++ program is the screen, and **cout** is the C++ stream object which defines it. **cout** is used in conjunction with the *insertion* operator, which is written as <<. Consider the following segment of code:

```
cout << "Output"; // prints Output on the screen
cout << 121; // prints number 121 on the screen
cout << var; // prints the content of 'var' on the screen
```


The insertion operator << inserts the data that follows it into the stream. In the above examples, it inserts the string "Output", the numerical constant 121 and the value of the variable var into the standard output stream **cout**.

The insertion operator << may be used more than once in a single statement. Consider the following line of code.

```
cout << "Welcome " << "to " << "the first C++ statement";
```

This statement prints the message "Welcome to the first C++ statement" on the screen as an output. The repetition of the insertion operator << is needed when more than one variables or combination of variables and constants are needed to print. Consider the following line of code.

```
cout << "Assalamoalaikum, I am a student of " << class << " Class";
```

If the value of the variable class1 is 12 then the above statement would be:

Assalamoalaikum, I am a student of 12 Class

3.3.2 Standard input (cin)

The standard input device used for entering input to the computer is keyboard. In C++, the stream *extraction* operator >> and object **cin** are used to handle the standard input. The operator >> must be followed by the variable that will store the data that is going to be extracted from the stream. Consider the following lines of code:

```
int marks;
cin >> marks;
```

Here, the first statement, *int* marks; declares a variable of type **int** called 'marks', and the second statement waits for an input from **cin** in order to store it in this integer variable.

The object **cin** can only process the input from the keyboard once the enter key is pressed. Therefore, even if you request a single character, the extraction from **cin** will not process the input until the user presses enter after the character has been introduced. Consider the following example to demonstrate **cin** and **cout** objects.

```
// program to demonstrate cin and cout objects
#include <iostream.h>
#include<conio.h>
int main()
{
int i;
float age;
cout << "Please enter your age: ";
cin >> age;
cout << "The age you entered is " << age <<" years";
getch();
return 0;
}
```

Output of the Program

```
Please enter your age:
16
The age you entered is 16 years
```

cin can also be used to request more than one input from the user. Consider the following line of code:

```
cin >> x >> y;
```

is equivalent to:

```
cin >> x;
cin >> y;
```


Here, in both the cases, the user need to enter two values, one for variable x and another one for variable y.

- cin and strings

cin can also be used to get strings with the extraction operator >> in the same way as it is used to get values for variables. Consider the following line of code:

```
cin >> TestStr;
```

cin object stops reading characters when it gets any blank space character, so in this case of entering strings, the object will get only one word. For example, if one wants to input a sentence using **cin** object, it will only accept the first word of the sentence and will discard the rest of the words.

In order to get entire line with spaces between the words, the function **gets** is used which is recommended by the programmers as compared to **cin**. Consider the following program demonstrating **cin** object for string.

```
// cin with strings
#include <iostream.h>
#include<conio.h>
int main()
{
char TestStr[60];
cout << " Enter your name please \n";
cin >>Test Str;
cout << "\nYour name is" << TestStr;
getch();
return 0;
}
```

Output of the Program

Enter your name please
Mohammad Khalid
Your name is Mohammad Khalid

3.3.3 gets(), puts() and getch() functions

C++ uses a wide range of standard Input/output functions. These functions are defined in standard library header files. The following sub-sections discuss some of the most commonly used I/O built-in functions.

a. gets() function

gets() is a standard library function defined in **stdio.h** header file which is used to get a string from the keyboard. Its general syntax is:

gets(string variable name);

It reads characters from stdin and stores them as a string into 'string variable' until a newline character or the EOF is reached. Consider the following example:

```
/* gets example */
#include <stdio.h>
#include <conio.h>
#include<iostream.h>
int main()
{
char Test Str [150];
cout<<"Insert your full address: ";
gets (TestStr);
cout<<"\nYour address is:"<<TestStr;
```



```
getch();
return 0;
}
```

Output of the Program

Insert your full address: *H.No. 10 Lalazar Colony*
Peshawar
Your address is: *H.No. 10 Lalazar Colony*
Peshawar

b. puts() function

puts() function writes a string to the screen and appends a newline character automatically at the string. The general syntax of puts () function is:

puts (string variable);

Consider the following example to demonstrate **gets()** and **puts()** functions:

```
#include <stdio.h>
#include<conio.h>
#include<iostream.h>
int main()
{
char TestStr [100];
cout<< " Enter a string\n";
gets(TestStr);
cout<< "You entered: ";
puts (TestStr);
getch();
return 0;
}
```

Output of the Program

Enter a string

Hello World

You entered: *Hello World*

c. getch() function

getch() function is a function that obtains the next available keystroke or character from the console. As a result of this function, nothing is echoed on the screen. When no keystroke is available, the function waits until a key is pressed. The value of the keystroke is returned by the function when a key is pressed. Its general syntax is:

getch( );

Consider the follow ing C++ program that demonstrates the use of **getch**() function.

```
/* getch() example */
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
int main()
{
char ch;
cout<< "Press any key\n";
ch = getch();
cout<<"You pressed: " << ch <<endl;
getch();
return 0;
}
```

Output of the Program

Press any key

S

You pressed: S

3.3.4. Escape sequences

C++ has some characters that have special meaning. These characters are called **escape sequences**. An escape sequence starts with a \ followed by a letter or number.

Table 3.3 shows the list of all escape sequences that are used in C++ along with their purpose.

Name	Symbol	Meaning
Alert	\a	Makes an alert, such as a beep
Backspace	\b	Moves the cursor back one space
Formfeed	\f	Moves the cursor to next logical page
Newline	\n	Moves cursor to next line
Carriage return	\r	Moves cursor to beginning of line
Horizontal tab	\t	Prints a horizontal tab
Vertical tab	\v	Prints a vertical tab
Single quote	\'	Prints a single quote
Double quote	\"	Prints a double quote
Backslash	\\	Prints a backslash
Question mark	\?	Prints a question mark
Octal/hex number	\(number)	Translates into char represented by octal/hex number

Table 3.3: Escape Sequences

The most common escape sequence is \n, which can be used to embed a newline in a string of text:

```
// \n escape sequence program
#include <iostream.h>
#include<conio.h>
int main()
{
cout<< "Welcome to \nthe College " <<endl;
getch();
return 0;
}
```

Output of the Program

Welcome to
the College

Consider another program for escape sequence character "\t".

```
// escape sequence \t program
#include <iostream.h>
#include<conio.h>
int main()
{
cout<< "First part\tSecond part";
getch();
return 0;
}
```

Output of the program

First part Second part

3.3.5 I/O handling functions

Most commonly used

- **getchar()** stands for get character and is used to get a single character from standard input device (keyboard). **getchar**() is defined in **stdio.h** header file. The general syntax for the use of getchar () function is:

```
int getchar ();
```

This function gets no parameters.

- **putchar()** is a standard output function defined in **stdio.h** header file and is used to display a single character, received as an argument, on the screen. Its general syntax is:

```
putchar (ch); // where ch is a char type variable holding a character constant
```

This putchar() display the character constant stored in 'ch' on the screen. Consider the following example:

```
// getchar() and putchar() example
#include <conio.h>
#include <stdio.h>
#include<iostream.h>
int main()
{
char ch;
cout<<"Enter character for gender\n";
ch=getchar();
cout<< "You have entered: ";
```

```
putchar (ch);
getch();
return 0;
}
```

Output of the Program

```
Enter character for gender
M
You have entered: M
```

This program reads a character from the keyboard using **getchar**() function and assign it to a character type variable 'ch' which is displayed on the screen by another standard library function **putchar**().

3.3.6 endl and setw Manipulator

Manipulators are used for formatting output. The data is manipulated by the programmer's choice of display. There are several manipulators available in C++. The most commonly used manipulators are discussed below:

a. endl manipulator

This manipulator has the same functionality as the '\n' newline character but with the difference of flushing the stream. Consider the following example:

```
// endl manipulator program
#include <iostream.h>
#include <conio.h>
int main()
{
cout << "Welcome" << endl;
cout << "to" << endl;
```



```
cout << "the College";
getch();
return 0;
}
```

Output of the program

```
Welcome
to
the College
```

b. setw manipulator

This manipulator sets the minimum field width on output. The syntax is:

```
setw(x)
```

Here, **setw** causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in **setw** manipulator. The header file that must be included while using **setw** manipulator is **<iomanip.h>**. Consider the following example:

```
//setw manipulator program
#include <iostream.h>
#include <iomanip.h>
#include<conio.h>
int main( )
{
int x1=2,x2= 4, x3=6;
cout<<setw(8)<<"Number"<<setw(20)<<"Square"<<endl
<< setw(8) << "2" << setw(20)<< x1*x1 << endl
```

```
<< setw(8) << "4" << setw(20)<< x2*x2 << endl
<< setw(8) << "6" << setw(20)<< x3*x3 << endl;
getch();
return 0;
}
```

Output of the program

Number	Square
2	4
4	16
6	36

3.4. Operators in C++

Operators are the symbols which perform operation on operands. The operation may be arithmetic or comparison of data.

z=x+y;

Here, x, y and z are operands and '+' and '=' are the operators.

3.4.1. Operators in C++

C ++ is rich in operators and supports a wide range of these operators that range from the simplest unary to the complex ternary. Operators are very important because without their use, expressions cannot be evaluated.

The following are different types of operators used in C++.

a. Assignment = operator

The assignment operator assigns a value to a variable. The symbol = is used as an assignment operator. For example:

VarA = 55;

This statement assigns the integer value 55 to the variable VarA. The part at the left of the assignment operator = is a variable whereas the right side can be a constant value, a variable, the result of an operation or any combination of these.

The assignment operation always takes place from right to left.

VarA = VarB;

This statement assigns the value of variable VarB to VarA.

Consider the following program to demonstrate the use of assignment operator.

```
#include <iostream.h>
#include <conio.h>
int main()
{
int VarA, VarB;
VarA = 100;
VarB = 400;
cout<<" VarA :"<<VarA;
cout<<endl;
cout<<" VarB :"<<VarB;
getch();
```

```
return 0;
}
```

Output of the Program

VarA :100
VarB :400

b. Arithmetic operators

There are five arithmetic operators used in C++. These are (+, -, *, /, %) that support the following arithmetic operations shown in Table 3.4.

+	addition
−	subtraction
*	multiplication
/	division
%	modulus

Table 3.4: Arithmetic Operators

The percentage sign % shown in the above table is used for modulus. Modulus is the operation that gives the remainder of a division of two values. For example, if we write:

```
remainder = 11 % 5;
```

The variable 'remainder' will contain the value 1, since 1 is the remainder from dividing 11 by 5. The following program demonstrates the use of these operators.


```
#include <iostream.h>
#include <conio.h>
int main()
{
int a=11, b=5;
int sum=a+ b;
int difference=a - b;
int product=a * b;
float division=a / b;
int remainder=a % b;
cout<< "sum ="<<sum<<endl;
cout<< "difference ="<< difference <<endl;
cout<< "product ="<< product <<endl;
cout<< "division ="<< division <<endl;
cout<< "remainder ="<< remainder;
getch();
return 0;
}
```

Output of the Program

sum =16
difference =6
product =55
division =2.0
remainder =1

c. Compound or arithmetic assignment operators

Compound assignment operators used in C++ are: (+=, -=, *=, /=, %=). These operators are used to modify the value of a variable by performing an operation on the value that is

currently stored in that variable. For example, if we want to add the value of a variable y to the value of another variable x then the following statement is used.

x+=y;

i.e. x=x+y;

The list of these compound assignment operators along with their use is given in the Table 3.5.

Operator	Symbol	Form	Operation
Addition assignment	+=	x += y	Add y to x
Subtraction assignment	-=	x -= y	Subtract y from x
Multiplication assignment	*=	x *= y	Multiply x by y
Division assignment	/=	x /= y	Divide x by y
Modulus assignment	%=	x %= y	Put the remainder of x / y in x

Table 3.5: Arithmetic Assignment Operators

Consider the following example to demonstrate the use of compound assignment operators.

```
#include <iostream.h>
#include <conio.h>
int main()
{
int VarA, VarB=300;
VarA = VarB;
VarA +=200; // equivalent to VarA=VarA+200
cout<< "Value after addition="<<VarA;
getch();
return 0;
}
```


Output of the Program

Value after addition=500

d. Increment and decrement operators

These operators are also called unary arithmetic operators. These operators are used to make expressions short. The increment operator ++ and the decrement operator -- increase or decrease the value stored in a variable by 1. For example the following statements are equivalent and all perform the same operation of increasing the value of the variable 'VarC' by 1.

```
VarC ++;
VarC +=1;
VarC = VarC +1;
```

There are two versions of each of these increment and decrement operators. These are: a prefix version and a postfix version.

Prefix and Postfix forms

In **prefix** form of the increment and decrement operators, the operators precede their operands, e.g. ++x, --z

Consider the following segment of code:

```
// Prefix Increment and Decrement Operators
int x = 5;
cout<< "increment in prefix form ="<<++x;
cout<< "decrement in prefix form ="<<--x;
```

Output of the Segment

increment in prefix form =6

decrement in prefix form =4

In **postfix** form of the increment and decrement operators, the operands precede their operators, e.g. x++, z--

Consider the following program to demonstrate the use of increment and decrement operators.

```
// increment and Decrement Operators
#include<iostream.h>
#include<conio.h>
int main()
{
int x = 25, y = 25;
cout<< x << " " << y <<endl;
cout<< ++x << " " << --y <<endl; // prefix
cout<< x << " " << y <<endl;
cout<< x++ << " " << y-- <<endl; // postfix
cout<< x << " " << y <<endl;
getch();
return 0;
}
```

Output of the program

25 25
26 24
26 24
26 24
27 23

e. Relational operators

Relational operators are used for the comparison between two expressions. These operators are: ==, !=, >, >=, <, <=. The result of these operators can be either true or false. The list of all relational operators used in C++ is given in Table 3.8.

Operator	Symbol	Form	Operation
Greater than	>	x > y	true if x is greater than y, false otherwise
Less than	<	x < y	true if x is less than y, false otherwise
Greater than or equals	>=	x >= y	true if x is greater than or equal to y, false otherwise
Less than or equals	<=	x <= y	true if x is less than or equal to y, false otherwise
Equality	==	x == y	true if x equals y, false otherwise
Inequality	!=	x != y	true if x does not equal y, false otherwise

Table 3.8: List of Relational Operators

Consider the following lines of code:

(7 == 5) // false.

(5 > 4) // true.

(3 != 2) // true.

(6 >= 6) // true.

(5 < 5) // false.

f. Logical operators

There are three logical operators used in C++. These are: !, &&, || that are shown in Table 3.9 along with their operations.

Operator	Symbol	Form	Operation
Logical NOT	!	!x	true if x is false, or false if x is true
Logical AND	&&	x && y	true if both x and y are true, false otherwise
Logical OR	||	x || y	true if either x or y are true, false otherwise

Table 3.9: List of Logical Operators

i. Logical NOT

The ! operator is the C++ unary operator that is used to perform the Boolean NOT operation. The output of this operator is the inverse of the value of its operand. It produces **false** if its operand is **true** and **true** if its operand is **false**. The effects of logical NOT operator is summarized in the truth Table 3.10.

X	!X
True	False
False	True

Table 3.10: Logical NOT Operator

The following lines of code demonstrate the working of NOT operator.

```
!(5 == 5) // evaluates to false because the expression (5 == 5) is true.
!(6 <= 4) // evaluates to true because (6 <= 4) would be false.
!true// evaluates to false
!false// evaluates to true.
```

ii. Logical AND

Logical && operator is a binary operator that needs two operands. It is used to test whether both conditions are true or not. If both the conditions are **true**, the logical &&returns **true**, otherwise, it returns **false**. Truth Table 3.11 shows the results of logical &&operator.

X	Y	X && Y
False	False	False
False	True	False
True	False	False
True	True	True

Table 3.11: Logical AND Operator

iii. Logical OR

The logical || operator is used to test whether either of two conditions is true. If either of the left operand or right operand evaluates to **true**, the logical ||operator returns **true**. Also, if both the operands are **true**, logical ||returns to **true** but if both are **false**it returns to **false** Truth Table 3.12 shows the operations of logical || operator.

X	Y	X || Y
False	False	False
False	True	True
True	False	True
True	True	True

Table 3.12: Logical OR Operator

g. Ternary (or conditional) operator (? :)

The conditional operator takes three operands and is therefore also called ternary operator. It evaluates an expression and returns a value if that expression is true and a different value if the expression is false. The general syntax used for conditional operator is:

condition ? result1: result2;

Here, if condition is true the expression will return *result1*, and if condition is false it will return *result2*. Consider the following lines of code to demonstrate the use of conditional operator.

```
7==5 ?4 : 3; // returns 3, since 7 is not equal to 5.
7==5+2 ?4 : 3; // returns 4, since 7 is equal to 5+2.
5>3 ?a : b; // returns the value of a, since 5 is greater than 3.
```

The following program demonstrates the use of conditional operator.

```
// conditional operator
#include <iostream.h>
#include<conio.h>
int main()
{
int x, y, z;
x=22;
y=77;
z = (x>y) ? x : y;
cout<< "The greater value is: "<<z ;
getch();
return 0;
}
```

Output of the Program

The greater value is: 77

3.4.2. Unary, Binary and Ternary operators

The set of C++ operators is divided into three main categories: unary, binary and ternary. This division is on the basis of number of operands upon which the operators operate. The following sections discuss them.

a. Unary operators

Unary operators are those operators which need a single operand. There are two arithmetic unary operators: unary plus + and unary -.

Consider the following examples:

```
x = -y;
a = +b;
```


The logical ! operator is also a unary operator. Consider the following examples:

```
!x;
!(x > y);
```

Increment ++ and decrement -- operators are also the exampies of unary operators.

b. Binary operators

i. Binary operators are those operators which need two operands. For example (+, -, *, /, %)

ii. The assignment operator = is also a binary operator as it requires two operands. The operands are: the receiving variable on the left hand side and the evaluated expression on the right hand side.

iii. The logical and relational operators (<, <=, ==, >, >=, !=, &&, ||) are binary operators as well, as they require two operands.

c. Ternary operator

Ternary operator needs three operands. There is only one ternary operator. This operator is conditional operator (? :)

3.4.3. Expression

An expression consists of operators and operands or it may be the combination of variables and/or constants linked with operators. For example:

```
x=a + b;
Z= 2+ z / y;
y > 4;
```

The expressions are categorized as below:

i. **Arithmetic Expressions** having numeric operands and arithmetic operators and give numeric results. For example

```
x = y + 25;
sum = x + y;
```

ii. **Relational Expressions** having operands and relational operators for comparison of data. For example

```
x > y;
z >=10;
```

iii. **Logical Expressions** for testing of two or more Boolean expressions. For example

```
(x > y) || (x = 10);
(n=5) && (m > n);
```

3.4.4 Precedence of operators

Precedence of operator means that which operand will be evaluated first and which one will be evaluated later if the expression is a complex one. For example, in the following expression % will be evaluated first and then addition.

```
a = 5 + 7 % 2 /
```

From highest to lowest the priority is given below.

1. The parentheses are evaluated first.
2. Multiplication, division and modulus operators are performed next, from left to right at same level.
3. Addition and subtraction operators are resolved last, from left to right with same priority.

The following table shows the list of all operators with their precedence level.

Level	Operator	Description	Associativity
1	++ --	Postfix increment/decrement	left-to-right
2	++ -- + - !	Prefix increment/decrement Unary plus/minus Logical negation	right-to-left
3	* / %	Multiplication/division/modulus	left-to-right
4	+ -	Addition/subtraction	left-to-right
5	< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
6	== !=	Relational is equal to/is not equal to	left-to-right
7	&&	Logical AND	left-to-right
8	||	Logical OR	left-to-right
9	?:	Ternary conditional	right-to-left
10	= += -= *= /= %=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus assignment	right-to-left

Table 3.13 Operators Precedence

3.4.5 Compound expression

Compound expression is an expression that involves more than one sub-expressions linked with operators. Usually two or more relational expressions are linked with logical operators to form compound Boolean expression.

Following are the examples of compound expressions.

```
X1= (- B + sqrt( B*B – 4*A*C))/(2*A);
( (b * b) - 4 * a * c ) > 0
(age <= 20) && (height >= 5)
(m > 5) || (j > 0)
```

Summary

- C++ is a programming language developed by Bjarne Stroustrup which derives most of its features from C language.
- Each C++ program makes use of header files which have definitions for C++ objects.
- Pre-processor directives are the directives that are used to include header files to the C++ program.
- C++ Comments are the explanatory statements that are usually added to the source program statements to make them understandable to the readers.
- A C++ program needs variables to hold data upon which a programmer performs operations.
- Constant are those components of C++ language whose values cannot be changed during the execution of a program.
- Data types are the qualifiers that tell the compiler what type of data will be stored in a particular variable.
- For the declaration of variables, one needs to select proper data type.
- C++ supports a wide range of data types, such as, int, float, double char etc.

- For getting input and producing output, C++ has a rich library of pre-defined and pre-compiled objects, cin and cout, and functions such as getch (), getchar (), gets, putch () and putchar ().
- Escape sequences are special characters that start with back slash symbol ('\'). The subsequent character after ('\') symbol has special meaning.
- Operators are those symbols which operate over the operands.
- C++ supports a wide range of operators to perform operations on operands.
- Arithmetic operators are those operators that are used in arithmetic expressions and perform arithmetic operations.
- Relational operators are used for comparing two values and return (1) if the condition is TRUE and (0) if FALSE.
- Unary operators take one operand; binary operators take two operands and ternary operator takes there operands.

Exercise

Q.1 Fill in the Blanks.

i. In C/C++ language, a hexadecimal number is represented by writing _____ symbol.
ii. (?) is the _____ operator used in C++.
iii. Output of the logical and relational operators is _____.
iv. Assigning values to a variable during declaration time is called _____.
v. Each C++ program has _____ function from where the execution of the program starts.

Q.2 Select the correct choice for the following Multiple Choice Questions.

i. What does the given expression evaluate to? E=6 + 5 * 4 % 3
   a. 1
   b. 8
   c. 11
   d. None of the above
ii. (true && true) || false
   a. true
   b. false
   c. both a and b
   d. none of the above
iii. Expression C=i++ causes
   a. Value of i assigned to C and then i incremented by 1
   b. to be incremented by 1 and then value of i assigned to C
   c. Value of i assigned to C
   d. to be incremented by 1

iv. Which of the following languages is a subset of C++ language?
   a. C language
   b. Java language
   c. B language
   d. C# language
v. In C++ language char ch='3' represents:
   a. A digit
   b. An integer constant
   c. A character constant
   d. A word

Q.3 Write TRUE/FALSE against the following statements.

i. Definition for the C++ object cout is stored in header file conio.h.
ii. In C++ a statement ends with ':' symbol.
iii. *int* **20sum** is the correct declaration of the variable **20sum**.
iv. In C++ the statement a=a+20 is the same as a+=20.
v. Size of *short int* data type is 4 bytes.

Q.4 Evaluate the following:

1) (false && true) || true
2) (false && true) || false || true
3) (5 > 6 || 4 > 3) && (7 > 8)
4) !(7 > 6 || 3 > 4)

Q.5 Pick those variables from the following list which are improperly declared and named. Also, describe the reason of its invalidity?

1) int sum;
2) int cats=5, dogs=5;

3) int my variable name;
4) int void = 5;
5) int nAngle;
6) int 3some;
7) int meters_of_pipe;
8) int length, width=5;

Q.6 Write a C++ program to get six subjects marks of a student and then calculate its total, average and percentage and display them on the screen.

Q.7 Write a C++ program to find out maximum value out of three integers using conditional operator.

Q.8 Write a C++ program to find out the roots of a quadratic equation.

Q.9 Write a C++ program to find out the area of a rectangle and display the result on the screen.

Q.10 Write a C++ program to get the age of a student from the user at run time and display it on the screen.

UNIT 4

CONTROL STRUCTURE

After the completion of Unit-4, Students will be able to:

- Explain the use of the following decision statements:
  - **if** statement
  - **if-else** statement
  - **switch** statement
- Know the concept of nested **if** statement
- Use **break** statement and **exit** function
- Explain the use of the following looping structure:
  - **for** loop
  - **while** loop
  - **do-while** loop
- Use **continue** statement
- Know the concept of nested loop

INTRODUCTION

C++ provides **control flow statements,** also called *flow control statements*, which allows the programmer to control the sequence of execution of statements of a program by the processor. The most commonly used control flow statements are decision, repetitions and compound statements.

4.1 Decision statements

Decision statement is a statement that causes the program to change the path of execution, sequence of execution, based on the value of an expression. It is also called conditional statement. Most commonly used decision support statements are: if, if-else and nested if-else statements. The switch also provides a mechanism for doing conditional branching.

4.1.1 Use of decision statements

In a C++ program, a programmer sometimes needs to execute one statement (or a set of statements) if a certain condition is true and another statement (or set of statements) if it is false. In a C++ program, this is only possible with the use of decision statements.
The following are different types of decision statements.

a. if statement

The most basic kind of conditional statement used in C++ is **if** statement. General syntax of an *if statement* is given below:

Single statement if structure

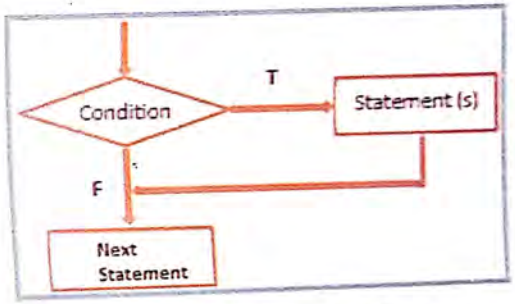
```
if (condition)
Statement;
```


Multiple statements if structure

```
if (condition)
{
Statement 1;
Statement 2;
.
.
Statement n;
}
```

If the condition is true, the statement after **if** will be executed and if the condition is false, the statement will be skipped. If we want to execute multiple statements with one **if** statement then the statements should be enclosed in a pair of curly braces { }.

Flowchart 4.1 shows working of **if statement.**



Flowchart 4.1: Working of If Structure

Consider the following program to demonstrate the use of **if** statement.

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

```
int main()
{
int X;
cout << "Enter a number: ";
cin >> X;
if (X %2==0)
cout << X << " is even number";
getch();
return 0;
}
```

Output of the Program

```
Enter a number:
12
12 is even number
```

b. if-else statement

In **if** construct, the statements in the body of **if** are executed only if the condition is true and nothing is executed if it is false. Sometimes the programmers need to execute some other part of a program even if the condition in **if** statement is false. For this purpose, *if-else statement* is used.

The general syntax of this statement is given below.

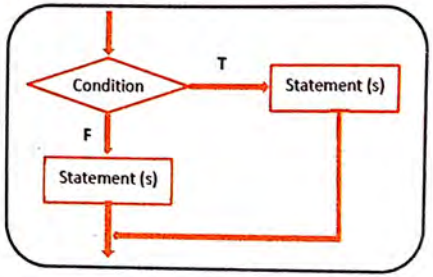
Single statement if-else structure

```
if (condition)
Statement;
else
Statement;
```

Multiple statements if-else structure

```
if (condition)
{
Statement 1;
Statement 2;
.
.
Statement n;
}
else
{
Statement 1;
Statement 2;
.
.
Statement n;
}
```

If the condition is true, the body of **if** is executed. If the condition is false, the body of *else* is executed. Flowchart 4.2 shows working of **if –else Structure**.



Flowchart 4.2: Working of If-else Structure

Consider the following program to demonstrate the use of *if-else statement*.

```
//if-else (even/odd) program
#include <iostream.h>
#include<conio.h>
int main()
{
int X;
cout << "Enter a number: ";
cin >> X;
if (X %2==0)
cout << X << "is even number";
else
cout << X << " is odd number";
getch();
return 0;
}
```

Output of the program

```
Enter a number:
9
9 is odd number
```


In order to execute multiple statements, a block should be used. Consider the following program:

```
//if-else (multiple statements) program
#include <iostream.h>
#include<conio.h>
int main()
{
int X;
cout << "Enter a number: ";
cin >> X;
if (X%2==0)
{
cout << "You have entered " << X << endl;
cout << X << "is an even number" << endl;
}
else
{
cout << "You have entered " << X << endl;
cout << X << " is an odd number" << endl;
}
getch();
return 0;
}
```

Output of the program

```
Enter a number:
08
You have entered 08
08 is an even number
```

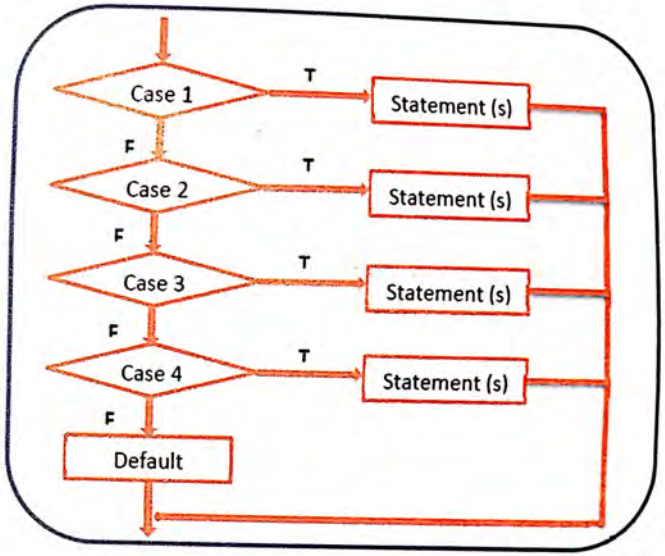
c. Switch-default statement

Switch statement is a better alternative of the nested if-else structure. The nested if-else structure is sometimes more difficult to understand to find that which if clause is associated with which else clause. The objective of switch statement is to check several possible constant values for an expression and then execute those which match the particular label or case constant value. Its general form is as follows:

```
switch (expression)
{
case constant1:
group of statements 1;
break;
case constant2:
group of statements 2;
break;
.
.
.
default:
default group of statements
}
```

switch evaluates the expression and if the value of the expression equal to constant1, it executes group of statements 1 until it encounters the **break** statement. When it reaches the **break** statement the program jumps to the end of the switch. If the value of the expression is not equal to constant1, it is matched with constant2. If it is equal to this constant2, this group of statements 2 is executed until the **break** keyword is reached, and then a jump will occur to the end of the switch. Finally, if the value of expression does not match with any of the constants then the program executes the statements after the

default keyword. The **default** clause is optional. Flowchart 4.3 shows working of **switch** Structure.



Flowchart 4.3: Working of switch-default Structure

Consider the following two code segments which have the same behavior.

switch example	if-else equivalent
// Switch program #include <iostream.h> #include<conio.h> int main() { int x; cout<<"Enter value for x"; cin>>x; switch (x) { case 1: cout << "x is 1"; break; case 2: cout << "x is 2"; break; default: cout << "value of x is neither 1 nor 2"; } getch(); return 0; }	*// if-else alternative of the switch program* #include <iostream.h> #include<conio.h> int main() { int x; cout<<"Enter value for x"<<end; cin>>x; if (x == 1) { cout << "x is 1"; }else if (x == 2) { cout << "x is 2"; }else { cout << " value of x is neither 1 nor 2"; } getch(); return 0; }

In the above program, if we did not include a break statement after the first group for **case** 1, the program will not automatically jump to the end of the switch and it would continue executing the rest of statements until it reaches either a break instruction or the end of the switch block.

switch statement can only be used to compare the value of the expression with the constants written with each **case** clause. Therefore, variables cannot be used with constants such as **case n** or ranges such as **case (1..3):** because they are not valid C++ constants. These constants be either integers or character type.

4.1.2 Nested if statement

It is possible to chain if-else statements together. When one if-else statement is used inside the body of another if-else structure then it forms a nested if ladder.

The following program demonstrates the use of nested if structure.

```
#include <iostream.h>
#include<conio.h>
int main()
{
int X;
cout << "Enter a number: ";
cin >> X;
if (X > 100)
cout << X << "is greater than 100" << endl;
else if (X < 50)
cout << X << "is less than 50" << endl;
else
cout << X << " is between 50 and 100" << endl;
getch();
return 0;
}
```

Output of the program

```
Enter a number:
96
96 is between 50 and 100
```

Consider another program demonstrating nested if-else structure by displaying whether an entered integer is multiple of 2 or 3 or is some other number.

```
#include <iostream.h>
#include<conio.h>
int main()
{
int n;
cout<<"Enter a number: ";
cin>>n;
if (n%2== 0)
cout << n<< " is multiple of 2";
else
if (n %3== 0)
cout << n<< " is multiple of 3";
else
cout << n<< " is neither multiple of 2 nor 3";
getch();
return 0;
}
```

Output of the program

```
Enter a number:
55
55 is neither multiple of 2 nor 3
```


Consider another example demonstrating nested if-else structure to display the grades of students. This program takes percentage marks as input and then using relational and logical operators find its corresponding grade.

```
#include <iostream.h>
#include<conio.h>
int main()
{
float per;
cout<<"Enter percentage marks: "<<endl;
cin>>per;
if (per >= 80)
{
cout << "Your grade is A+"<<endl;
cout<< "Excellent";
} else
{
if (per < 80 && per>=70)
{
cout << "Your grade is A"<<endl;
cout<< "Very good";
} else
{
if(per<70 && per>=60)
{
cout << "Your grade is B"<<endl;
cout<< "Good";
}else
cout<<"You are failed, try again";
```

```
}
}
getch();
return 0;
}
```

Output of the program

```
Enter percentage marks:
65.09
Your grade is B
Good
```

4.1.3 break statement and exit() function

Sometimes a programmer needs to make use of such statements in a program that are responsible for the jump from one statement of the program to another statement. The most commonly used jump statements of C++ language are: **goto, break,** and **continue.** Similarly, the execution of a running program is also sometime needed to be halt and to jump out of a program. A function used to accomplish this is **exit()**. The following sections discuss the use of **break** statement and **exit ()** function with the help of examples.

a. break statement

break statement is used to make jump from one part of a program to another. In C++, *break* statement is used in two situations: one in *switch-default* structure to properly end a *case* clause and another in loops structure to leave a loop. The use of *break* in *switch-default* structure has already been discussed in Section 4.1.3. Its use in loops will be discussed in Section 4.2 with the help of examples.

b. exit () function

This function tells the program to quit running immediately. This function is responsible for the immediate termination of a C++ program. It is defined in the *stdlib* header file. The *exit* function takes an integer, usually 0, as a parameter that is returned to the operating system as an exit code and tells it that the program has terminated normally.

Consider the following example for demonstrating the use of **exit** () function.

```
#include<conio.h>
#include <stdlib.h>
#include <iostream.h>
int main()
{
cout << "Statement before exit() function";
exit(0);
//terminate and return 0 to the operating system
// The following statements never execute
cout << "Statement after exit() function";
getch();
return 0;
}
```

Output of the program

```
Statement before exit() function
```

4.2 LOOPS

Loops are used to repeat a statement or group of statements a certain number of times until the condition is true. These are also called repetitive or iterative statements.

4.2.1 Types of loops

There are three types of loops.

- for
- while
- do-while

a. for loop

The most commonly used looping structure in C++ is **for** loop. **for** loop is ideal in situation when we know in advance the exact number of iteration. Its general syntax is:

Single statement for-loop

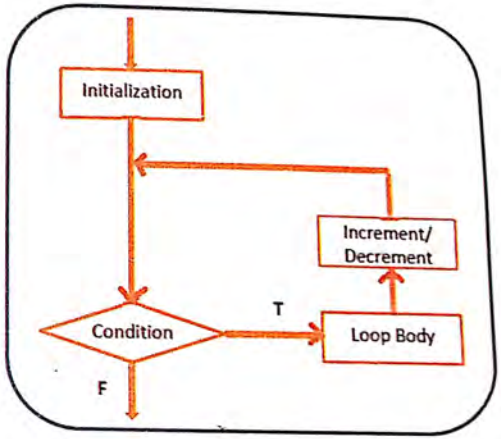
```
for (initialization; condition; increment/decrement)
statement;
```

Multiple statements for-loop

```
for (initialization; condition; increment/decrement)
{
Statement 1;
Statement 2;
.
.
Statement n;
}
```

In the above two general forms of *for* loop, the first one has only one statement in its body and has no need for braces { } while in the second form multiple statements are used so they are enclosed in the pair of braces { }.

The main function of **for** loop is to repeat statement(s) while condition remains true. It works in the following way shown in Flowchart 4.4.



Flowchart 4.4: Working of for Loop

1. **Initialization**: Generally it is an initial value setting for a counter variable. This is executed only once.
2. **Condition**: If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
3. **Loop Body**: It can either be a single statement or a block enclosed in braces { }, for multiple statements.
4. Finally, the loop counter is incremented/decremented and control goes back to step2.

Consider the following program to display the string PAKISTAN five times on the screen.

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

```
int main()
{
for (int i=1; i<6; i++)
cout << "PAKISTAN"<<endl;
getch();
return 0;
}
```

Output of the program

PAKISTAN
PAKISTAN
PAKISTAN
PAKISTAN
PAKISTAN

Here is an example of countdown using **for** loop:

```
// countdown using a for loop
#include <iostream.h>
#include<conio.h>
int main()
{
for (int n=10; n>0; n--)
cout << n << ", ";
getch();
return 0;
}
```


Output of the program

10, 9, 8, 7, 6, 5, 4, 3, 2, 1

The *initialization* and *increment/decrement* fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example, we could write: for (;n<10;) if we want to specify no initialization and no increase.

```
int n=0;
for ( ; n < 10; )
{
cout << n << ",";
n++;
}
```

Output of the code segment

0,1,2,3,4,5,6,7,8,9

Similarly, *for* loop can also be used as: for (;n<10;n++) if we want to include an increment/decrement expression but no initialization.

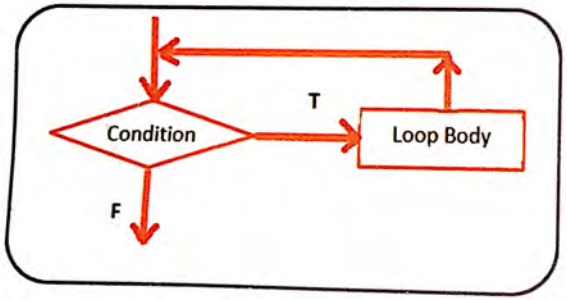
b. while loop

while loop is another structure that is used in computer programs to repeat a statement or a group of statements until the condition is true.

```
while (condition)
{
Statement 1;
Statement 2;
```

Statement n;
}

Flowchart 4.5 shows working of **while loop.**



Flowchart 4.5: Working of while Loop

Consider the following program to display the string PAKISTAN five times on the screen using **while** loop.

```
// displaying PAKISTAN 5 times using while loop
#include <iostream.h>
#include<conio.h>
int main()
{
int i=1;
while (i<6)
{
cout << "PAKISTAN"<<endl;
++i;
}
```



```
getch();
return 0;
}
```

Output of the program

PAKISTAN
PAKISTAN
PAKISTAN
PAKISTAN
PAKISTAN

Like **for** loop, the **while** loop has also three parts: *initialization, condition and increment/decrement.* The difference is that here the initialization is done before the **while** clause and the increment/decrement is done within the body of the **while loop.**

The second example of countdown using while-loop is given below.

```
// count down program using while loop
#include <iostream.h>
#include <conio.h>
int main()
{
int n;
cout << "Enter the starting number: ";
cin >> n;
while (n>0)
{
cout << n << ", ";
--n;
```

```
}
getch();
return 0;
}
```

Output of the program

Enter the starting number:
8
8, 7, 6, 5, 4, 3, 2, 1

Here, when the program starts the user enters a number to start the countdown. In this case, the number is 8 and thus the condition n>0 is true, therefore, the block within the braces { } is executed as long as n>0 and the output 8,7,6,5,4,3,2,1 is displayed on the screen.

Consider another program that prints numbers from 1 to 50 in such a way that there should be 10 digits per line.

```
#include <iostream.h>
#include <conio.h>
int main()
{
 int n = 1;
 while (n <= 50)
 {
  cout << n << " ";
  if (n % 10 == 0)
  cout << endl;
   n++;
```



```
}
getch();
return 0;
}
```

Output of the program

1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50

Consider another program using *while* loop to get a sentence from the user and count the number of characters in this sentence.

```
#include <iostream.h>
#include <conio.h>
int main()
{
char ch;
int NumberOfCharacters=0;
ch= getche();
while (ch!= '.')
{
++NumberOfCharacters;
ch= getche();
}
cout<< "The number of characters in the sentence are: "<<NumberOfCharacters;
getch();
```

```
return 0;
}
```

Output of the program

Welcome to the class.
The number of characters in the sentence are: 20

The main difference of the *while* loop and *for* is that in *while* loop it is not necessary to know the exact number of iterations of the body of the loop in advance as in *for loop*.

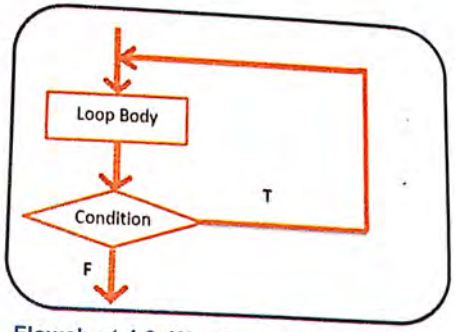
c. do- while loop

Sometime a programmer want to execute the loop at least once for some problems. To do this, C++ offers the facility of **do while** loop. The general format of a **do while** loop is:

```
do
{
Statement 1;
Statement 2;
.
.
Statement n;
}
while (condition);
```

The working of *do while* loop is the same as the *while* loop, except that the condition in *do while* loop is checked after the execution of the body of the loop. The execution of *do while* loop, guaranties the execution of the statements in its body, at least once, even if the condition is false.

Flowchart 4.6 shows working of **while loop.**



Flowchart 4.6: Working of do-while Loop

Consider the following program that prints the string PAKISTAN once although the condition is false in the start.

```
// displaying PAKISTAN at least once using do while loop
#include <iostream.h>
#include<conio.h>
int main()
{
int i=10;
{
cout << "PAKISTAN"<<endl;
++i;
} while (i<6);
getch();
return 0;
}
```

Output of the program

PAKISTAN

In the above program, the loop control variable 'i' is initially set to 10 and in the condition it is restricted to be less than 6 (which is violated in the start) but still it is executed once and have displayed the message PAKISTAN.

Consider another program that echoes any number you enter from the keyboard to the screen until you enter 0 using do-while loop.

```
#include <conio.h>
#include <iostream.h>
int main()
{
int x;
do {
cout << "Enter number (0 is for ending): ";
cin >> x;
cout << "You entered: " << x << "\n";
} while (x != 0);
getch();
return 0;
}
```

Output of the program

```
Enter number (0 is for ending): 12
You entered: 12
Enter number (0 is for ending): 16
You entered: 16
Enter number (0 is for ending): 0
You entered: 0
```


4.2.2 continue statement

The continue statement provides a convenient way to jump to the start of a next iteration by passing the remaining part of the loop for an iteration. For example, we are going to skip the number 5 in our countdown:

```
// use of continue statement in loop
#include <iostream.h>
#include <conio.h>
int main()
{
for (int n=10; n>0; n--)
{
if (n==5) continue;
cout << n << ", ";
}
getch();
return 0;
}
```

Output of the program

10, 9, 8, 7, 6, 4, 3, 2, 1

4.2.3 Nested loops

The use of a loop inside the body of another loop is called **nested** loop. The enclosing loop is called *outer* loop and the enclosed loop is called *inner* loop. The following example demonstrates the concept of nested loop.

```
// nested while loop program
#include <iostream.h>
```

```
#include <conio.h>
int main()
{
// Outer Loop between 1 and 5
int O=1;
while (O<=5)
{
int I = 1;
while (I <= O)
{
cout << I++;
}// end of inner loop
// print a newline at the end of each row
cout << endl;
O++;
}// end of outer loop
getch();
return 0;
}
```

Output of the program

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Consider another example:


```
// nested for loop
#include <iostream.h>
#include <conio.h>
int main()
{
for (int i=1; i<=5; i++) //outer loop
{
for (int j=1; j<=i; j++) //inner loop
cout << "$";
cout << endl;
}
getch();
return 0;
}
```

Output of the program

```
$
$$
$$$
$$$$
$$$$$
```

Summary

- A decision statement causes the control to pass to different parts of the program based on the value of an expression. Decision can be made in C++ in several ways.
- The most important decision statements in C++ are if, if-else and switch Statements.
- if statement uses a condition and based on the values of the condition, the body of if is executed. If the condition is true then body is executed otherwise nothing is executed.
- if -else statement selects either the body of if or else based on the result of the condition. If condition is true then body of if is executed and if false then body of else is executed.
- if-else statement can be nested within each other.
- switch statement is used to select from a number of choices and is a better alternative of nested if-else statement.
- Loops are the iterative statements used to repeat a statement or group of statements until a terminating condition is true.
- for loop is the most commonly used iterative statement in C++ which is used in the situation when one knows the number of iterations in advance.

- while loop is similar to if-else structure in which the condition is checked first and then the body is executed.
- In do while structure the body of the loop is executed at least once even if the condition is false.
- Sometimes, the programmers need to shift the control and make jump from one statement of the program to another statement or part. Commonly used jump statements of C++ language are: break and continue.

Exercise

Q.1 Fill in the Blanks.

i. An if statement can include multiple statements provided that they are _____.
ii. The switch statement controls program flow by executing a specific set of statements, depending on _____.
iii. When the value returned by a switch statement expression does not match a case label, the _____ statements within the label execute.
iv. You can exit a switch statement using a(n) _____ statement.
v. Each repetition of a looping statement is called a _____.

Q.2 Select the correct choice for the following Multiple Choice Questions.

i. Find out the error in following block of code.

```
if (x = 100)
cout << "x is 100";
```

a. 100 should be enclosed in quotations
b. There is no semicolon at the end of first line
c. Equals to operator mistake
d. Variable x should not be inside quotation

ii. Looping in a program means
a. Jumping to the specified branch of program
b. Repeat the specified lines of code
c. Both of above
d. None of above

iii. Which of the following is not a looping statement in C++?
a. while
b. until
c. do
d. for

iv. Which of the following is not a jump statement in C++?
a. break
b. goto
c. exit
d. switch

v. Which of the following is selection statement in C++?
a. break
b. goto
c. exit
d. switch

vi. The continue statement
a. resumes the program if it is hanged
b. resumes the program if break was applied
c. skips the rest of the loop in current iteration
d. all of above

vii. Consider the following two pieces of codes and choose the best answer

1.

```
switch (x) {
case 1:
cout <<"x is 1";
break;
case 2:
cout <<"x is 2";
break;
default:
cout <<"value of x unknown";
}
```

2.

```
if (x==1){
cout <<"x is 1";
}
else if (x==2){
cout << "x is 2";
}
else{
cout <<"value of x unknown";
}
```

a. Both of the above code fragments have the same behaviour
b. Both of the above code fragments produce different effects
c. The first code produces more results than second
d. The second code produces more results than first

viii. Observe the following block of code and determine what happens when x=2?

```
switch (x)
{
case 1:
case 2:
case 3:
cout<< "x is between 1 .. 3";
break;
default:
cout<<"x is not within the range, so need to say Thank You!";
}
```

a. Program jumps to the end of switch statement
b. The code after default will run since there is no task for x=2
c. Will display x is between 1 ... 3
d. None of above

ix. Which of the following is false for switch statement in C++?
a. It uses labels instead of blocks
b. we need to put break statement at the end of the group of statement
c. we can put range for case such as case 1..3
d. None of above

Q.3 Write TRUE/FALSE against the following statements.

i. Decision-making structures cannot be nested.
ii. Body of do while loop is executed at least once if the condition is false.
iii. When ***for*** statement begins executing, at the same time the initialization expression is also executed?
iv. ***switch*** statement is a better alternative of nested if-else structure.
v. Syntax of if-else statement is:

```
else {
}
if (condition) {
}
```

Q.4 Write a C++ program that takes two integers and a chararacter representing one of the following mathematical operations: +, -, /, or *. Use a switch statement to perform the appropriate mathematical operation on the integers, and display the result. If an invalid operator is entered then "Error" message should be displayed and the program should exit (use the exit() function).

Q.5 Write a program using *for* loop that prints even numbers from 0 to 20.

Q.6 Write a program using *while loop* that takes an integer for a variable nValue, and returns the sum of all the numbers from 1 to nValue.
Hint: For example, nValue=5 should return 15, which is 1 + 2 + 3 + 4 + 5.

Q.7 What is wrong with the following for loop?

```
// Print all numbers from 9 to 0
for (unsigned int nCount = 9; nCount >= 0; nCount--)
cout << nCount << " ";
```

Q.8 Write a C++ program to read the address of a person and exit when the user enters dot (.) from the keyboard.

Q.9 Write a C++ program to find out the area of a triangle and if any side is zero then display the message "There is no triangle".

Q.10 Write a C++ program to input a character from the keyboard and display the message after testing whether it is Vowel or Consonant.

UNIT 5

ARRAYS AND STRINGS

After the completion of Unit -5, Students will be able to:

- Explain the concept of an array
- Know how array elements are arranged in memory
- Explain the following array terminology
  - Name
  - Size
  - Index
- Explain how to define and initialize an array of different size and data types
- Explain how to access and write at an index in an array
- Explain how to traverse an array using all loop structures
- Use the **sizeof()** operator to find the size of an array
- Explain two dimensional array
- Explain how to define and initialize a two dimensional array of different size and data types
- Explain how to access and write at an index in a two dimensional array
- Explain strings
- Explain how to define and initialize a string
- Explain the most commonly used string functions

INTRODUCTION

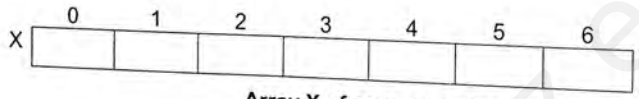
This unit is about arrays i.e. its definition, representation, accessing and traversing. There are two types of array data structures, one dimensional and two dimensional. The unit also focuses on strings and its most commonly used functions.

5.1 Introduction to array

We group our everyday data into conceptual units rather than storing in individual pieces. We collect the data items into conceptual units and assign names to them. For a single conceptual unit we have a single name. This conceptual unit of homogeneous data having the same name for all the data items is called array. The array is used to store many items of the same type as a single unit or group. Consider the examples of storing the scores of all the students of a class in an array or storing their names.

5.1.1 Concept of arrays

An array is a collection of data items of the same data types placed in contiguous memory locations that can individually be referenced by using an index to a unique identifier. Consider the example for storing 7 values of type *int* in an array without having to declare 7 different variables each one with a different variable name. Instead of that, using an array we can store 7 different values of the same data type with a unique variable name. Consider an array X that contains 7 integer values of type *int*. It can be represented as:



Array X of seven locations

These elements are numbered from 0 to 6 since in arrays the first index is always 0.

5.1.2 Representation of array

Array elements are stored in consecutive memory locations inside the computer memory. For example, if in the above array X we store the marks of seven students and the first element X[0] is stored at address 100, inside the computer memory, then X[1], X[2], X[3], X[4], X[5], X[6] will be stored at memory addresses 102,104,106, 108, 110, and 112 respectively. Here, the difference in the values of the memory addresses is two numbers because *an* ***int*** *data type occupies* 2 bytes of memory space. It is shown in figure 5.1.

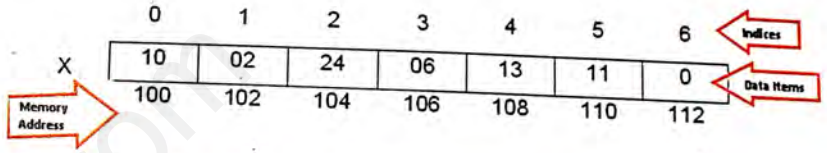


Figure 5.1: Memory Representation of 1-D Array for an int Data Type

Consider another example, shown in figure 5.2, of one dimensional array named SUBJ for storing the title of a subject whose length is 7 characters, such as, PHYSICS.

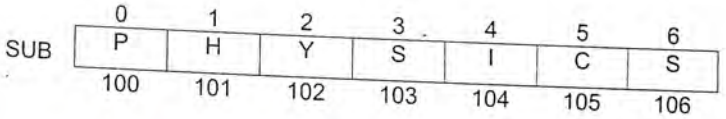


Figure 5.2: Memory Representation of 1-D Array for a char Data Type

5.1.3 Terminology used in Arrays

Arrays can be understood well by first knowing some commonly used terminologies such as name, size, and index.

a. Name of an array

The name of the array can be any valid identifier. Each location of the array has the same name with different index. In the above example, all the 7 locations of the array X has the same name X but different indices, X[0], X[1], X[2], X[3], X[4], X[5], and X[6].

b. Size of the array

The number of elements that can be stored in an array is called the size or length of that array.

For example:

float Z [10];

Here, size of the array Z is 10, which means that we can store 10 floating point values in this array.

c. Index

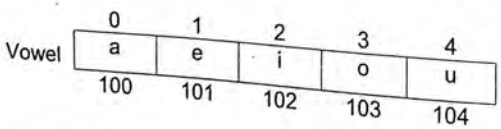
The location number of an item in an *array* is called index or subscript of that element. For example 0,1,2,3,4,5 and 6 are the indices of the elements of the array int X[7].

5.1.4 Definition and initialization of an array

As we have two types of arrays, one dimensional and two dimensional, both have their own ways of definition. Here, we are defining and initializing one dimensional array.

a. One dimensional array

A one-dimensional array or linear array can be represented as a single row of contiguous memory locations. Accessing of its elements involves a single subscript.



Like a regular variable, an array must be declared before it is used. A typical declaration for a one dimensional array in C++ is:

data type name [***size***];

Here, the data type is a valid data type (such as int, float...), name is a valid identifier and enclosed in square brackets [ ] is the size of the array which means that how many elements this array can accommodate.

For Example:

int x[7];
char vowel[5];
float temperature[7];

b. Initializing one dimensional array

Arrays can be initialized, like variables, when created. C++ provides a convenient way to initialize entire array via use of an initializer list.

The following example shows the initialization of an *integer* array.

```
// initializing array using initializer list method
#include <iostream.h>
#include <conio.h>
int main()
{
int A[5] = { 3, 2, 7, 5, 8 };
cout << A[0] << endl;
cout << A [1] << endl;
cout << A [2] << endl;
cout << A [3] << endl;
```

```
cout << A [4] << endl;
getch();
return 0;
}
```

Output of the program

3
2
7
5
8

If we do not initialize all of the elements in an array then the remaining elements will be initialized to 0.

Similarly, to initialize all the elements of an array to 0, we can define that array as:

```
int A[5]={0};
```

5.1.5 Accessing and writing at an index in an array

Accessing of array means to get into an index in an array and take its value for some operation while by writing at an index of an array is storing some value at a particular index of the array. These are the two important actions that should be taken while dealing with arrays.

The following section addresses them with examples.

Accessing an array

Array elements are accessed by using respective indices. The format for accessing is as follows:

Array name[index];

Looking at the example discussed in the above section, in which A has 5 elements, each of the element has been accessed as A[0], A[1], ..., A[4] using cout statement.

To access all elements of an array we use loops instead of individual reference to each element, For Example:

```
// accessing array using loop
#include <iostream.h>
#include <conio.h>
int main()
{
int A[5] = { 3, 2, 7, 5, 8 };
for(int i=0; i<=4; ++i)
cout << A[i] << endl;
getch();
return 0;
}
```

Output of the program

3
2
7
5
8

Here, **for** loop has been used to access the elements of the array one by one and print the result on the screen.

Writing at an index in an array

Accessing of an index of an array is followed by the writing operation, only in the case, if we want to store a new value at that index or to update the existing one. For example, if we want to add the values in the array **A**, we can individually access each array location and write the value using assignment operator or using input statement (cin) It is shown in the following program.

```
// writing at in index of an array
#include <iostream.h>
#include <conio.h>
int main()
{
int A[5];
A[0]=12;
A[1]=-20
cout<< "Enter number"<<endl;
cin >> A[2];
cout<< "Enter number"<<endl;
cin >> A[3];
cout<< "Enter number"<<endl;
cin >> A[4];
getch();
return 0;
}
```

Output of the program

```
Enter number
7
Enter number
5
Enter number
8
```

Loops can also be used for writing at all indexes of an array. Consider the following example having an array *Temperature* that accepts the values of weekly temperature from the users at run time and write them into their respective indexes.

```
// writing an array using loop
#include <iostream.h>
#include <conio.h>
int main()
{
float Temperature[7];
cout<< "Enter 7 days temperature:"<<endl;
for(int i=0; i<7; ++i)
cin >>Temperature[i]; // writing at indexes
cout<< "The last week temperature was:"<<endl;
for(int i=0; i<7; ++i)
cout<<Temperature[i] << ", "; // accessing the elements
getch();
return 0;
}
```

Output of the program

```
Enter 7 days temperature:
35.9 22.0 27.3 25.5 28.3 31.2 33.7
```

The last week temperature was:

35.9, 22.0, 27.3, 25.5, 28.3, 31.2, 33.7

5.1.6 Traversing an array

Accessing the elements of an array is called traversing of array.

```
// reading numbers into an array and then searching for an item
#include <iostream.h>
#include <conio.h>
int main()
{
int a[10]; // Declare an array of 10 integers
int n = 0;
cout<< "Enter 10 integer values"<<endl;
while (n<10)
{
cin >> a[n];
n++;
}
int item; // declare an item to search
cout<< "Enter an item to search"<<endl;
cin>>item;
for (int i=0; i<10; i++) //traverse array
{
if(item!=a[i]) // comparison
continue;
else
{
cout<< "Item "<<item<<" is found at location "<<i+1;
goto end;
}
}
cout << "Search unsuccessful: "<<item<< " is not found";
end:
getch();
return 0;
}
```

Consider another program of array traversal using *for* loop to display those values from the array which are odd.

```
#include <iostream.h>
#include <conio.h>
int main()
{
int A [] = {3, 2, 7, 5, 8};
for ( int i=0 ; i<5 ; i++ )
{
if (A[i] % 2 !=0)
cout<< A[i] << endl;
}
getch();
return 0;
}
```

Output of the program

```
3
7
5
```

5.1.7 sizeof() Operator

The **sizeof()** operator can be used with arrays to return the size allocated for the entire array. Consider the following program:

```
// size of an array example
#include <iostream.h>
#include <conio.h>
int main()
{
int A[] = { 0, 1, 2, 3, 4 };
cout << "Size of the array A is: " <<sizeof (A); // 5 elements * 2 bytes for each element
getch();
return 0;
}
```

Output of the program

```
Size of the array A is: 10
```

Using the **sizeof()** operator, we can also find out the number of elements in an array. Consider the following example:

```
/* to find the number of elements in array using
sizeof() operator*/
# include <iostream.h>
#include <conio.h>
int main()
{
int A[] = { 0, 1, 2, 3, 4 };
cout << "Size of the array ="<<sizeof(A)<<" bytes ";
int Total_Elements = sizeof(A) / sizeof(A[0]);
cout<<"\nTotal number of elements in the array A
are: "<<Total_Elements;
getch();
return 0;
}
```

Output of the program

```
Size of the array =10 bytes
Total number of elements in
the array A are: 05
```

5.2 Two-dimensional arrays

An array having more than one subscripts is called *multidimensional* array. A multidimensional array is an "array of arrays". It is not limited to two indices i.e., two dimensions, but can also have three-dimensions and four-dimensions etc.

5.2.1 Concept of two dimensional arrays

The most communally used multidimensional array is two dimensional arrays which is also called *bi-dimensional* array. It is used for storing of tabular data arranged in rows and columns. It can be represented by two indices i.e. row index and columns index.

All the elements of two dimensional array are of the same data type. Consider the following array:

```
int X[3][5];
```

Array X consisting of three rows and five columns. It can be represented as:

	0	1	2	3	4
0	12	-123	58	95	67
1	56	45	23	45	176
2	78	-56	90	55	74

Two dimensional array is also called *matrix* in mathematics and *table* in business applications.

5.2.2 Definition and initialization of two dimensional arrays

Like one-dimensional array, a two-dimensional array is defined and initialized before using it in a program. The following sections describe its definition and initialization with the help of examples.

a. Defining two dimensional array

The general syntax for the declaration of two dimensional arrays is given below.

Data Type **Array name [row index] [column index];**

In the above syntax, the *data type* is any valid data type, such as *int, float, long etc.,* which is followed by array name and two indices. The first index is for number of rows and the second index is for number of columns in the array.

A *float* type array, **tdArray**, consisting of 2 rows and 2 columns can be defined by using the following statement.

```
float tdArray [2][2];
```


b. Initializing two dimensional arrays

To initialize a two-dimensional array, it is easiest to use nested braces, with each set of numbers representing a row:

```
// two dimensional array initialization
int anArray[3][5] =
{
{ 1, 2, 3, 4, 5 }, // row 0
{ 6, 7, 8, 9, 10 }, // row 1
{ 11, 12, 13, 14, 15 } // row 2
};
```

If we want to initialize a *two-dimensional array* to 0 then it can be done as follows:

```
int anArray[3][5] = { 0 };
```

5.2 .3 Accessing and writing at an index in a two dimensional array

To operate on the elements of two dimensional arrays, the elements should be first accessed and then updated. Writing at an index of an array is also done when a new two dimensional array is declared. Accessing and writing at an index of a two dimensional array is similar to one dimensional array. The following sections discuss them with the help of examples.

a. Accessing two dimensional arrays

Accessing two dimensional arrays need two indices, one for the row and the other for the column. For example, if we have an array anArray[3][5] and want to access an element at second row and third column. Then it can be written as:

```
anArray[1][2];
```

Accessing elements of a two-dimensional array individually is very difficult and time consuming. A better solution to this is the application of loop. Normally, nested loops are used for this purpose: outer loop for the row, and inner loop for the columns. Since, two-dimensional arrays are typically accessed row by row, therefore, the row index is used as the outer loop and the column index is used as the inner loop.

Consider the following program used to access the elements of the array, ***anArray[3][5]***, initialized in the above section and display them on the screen in a matrix form.

```
// accessing two dimensional array using loop
#include <iostream.h>
#include <conio.h>
int main()
{
int anArray[3][5] =
{{ 1, 2, 3, 4, 5 },
{ 6, 7, 8, 9, 10 },
{ 11, 12, 13, 14, 15 }};
for ( int row=0; row<3; ++row)
{
 for (int col=0; col<5; ++col)
 cout<<anArray [row][col]<< ", ";
 cout<<endl;
 }
 getch();
 return 0;
 }
```

Output of the program

1, 2, 3, 4, 5
6, 7, 8, 9, 10
11, 12, 13, 14, 15

b. Writing at an index of two dimensional array

Loop is a better way of writing to the indices of a two dimensional array. Consider the following example:

```
// writing in a two dimensional array using loop
#include <iostream.h>
#include <conio.h>
int main()
{
int A[2][3];
cout<< "Kindly enter values for the array"<<endl;
for (int i = 0; i < 2; i++)
for (int j = 0; j < 3; j++)
cin>> A[i][j]; // writing to the array
cout<< "Values of the array are:"<<endl;
for (int i = 0; i < 2; i++)
{
for (int j = 0; j < 3; j++)
cout << A[i][j]<< "\t"; //accessing of the array
cout<<endl;
}
getch();
return 0;
}
```

Output of the program

2 -1 7 1 0 10
Values of the array are:
2 -1 7
1 0 10

5.3 Strings

5.3.1 What is String?

The sequence of characters enclosed in quotation marks and is used to handle non-numeric data i.e. names, addresses etc. are called strings.

5.3.2 Defining a C String

General syntax for defining string in C is given below:

data type name_of_string [size of string];

In the above general syntax, *data type*, is **char** data type and *name_of_string* is the name of the string. The size (number of characters) of the string is specified in square brackets, [ ]. The following statement defines a string with the name *str* having size 20.

char str [20];

For handling strings, the header file <*cstring*> is needed but in some implementations this library may be automatically included when the header file <*iostream*> is included. There are some limitations of strings in C. To overcome and remove these problems, C++ provides a class <*string*> that removes many of these problems. This class provides some typical string operations like *comparison*, *concatenation*, *find* and *replace*, and a function for obtaining *substrings*. The general syntax for defining strings in C++ is given below:

string str;

This statement defines a string name *str* of type *string*.

5.3.3 Initializing string

1. *Strings in* **C** can be initialized as arrays are initialized because *string* is a character array.

Consider the following example:

char greeting [6] = { 'H', 'e', 'l', 'l', 'o', '\0' };

The above statement declares a string named "*greeting*" with six elements of type *char* that are initialized with the characters that form the word "Hello" plus a null character '\0' at the end. Another method for initializing a string in C is to enclose the string in double quotes, "" as shown below:

char greeting [ ] = "Hello";

In both cases, the array of characters **greeting** is declared with a size of 6 elements of type *char*: the 5 characters that compose the word "Hello" plus a final null character ('\0') which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically.

Consider an example to ask for your name and then display a greeting message on the screen as shown below using string:

```
#include <iostream.h>
#include<conio.h>
int main()
{
char question[] = "Please, enter your first name: ";
char greeting[] = "Hello, ";
char yourname [80];
```

```
cout << question;
cin >> yourname;
cout << greeting << yourname << "!";
getch();
return 0;
}
```

Output of the program

```
Please, enter your first name:
Rahman
Hello, Rahman!
```

2. *Strings in* **C++** can be initialized both ways like in C and directly like ordinary variables with the only difference that the string should be enclosed in double quotes. Consider the following initialization examples:

```
string str1("Call me");
string str2 = "Send sms!";
string str3("OKay");
```

Here, in these examples three strings: *str1*, *str2* and *str3* are directly initialized to "Call me", "Send sms" and "OKay" respectively. Note that while using C++ string in programs, the class *string* must be included.

Consider the following program to demonstrate initialization of C++ string:

```
// C++ strings initialization program
#include <iostream.h>
#include<conio.h>
```



```
#include<string>
int main()
{
string str1("Call me");
string str2 = "Send SMS";
string str3("OKay");
cout<< "First string str1 is: "<<str1<<endl;
cout<< "Second string str2 is: "<<str2<<endl;
cout<< "Third string str3 is: "<<str3<<endl;
getch();
return 0;
}
```

Output of the program

```
First string str1 is: Call me
Second string str2 is: Send SMS
Third string str3 is: Okay
```

The following program demonstrates some more features of C++ string e.g. concatenation, copying and finding size of string.

```
#include <iostream.h>
#include<conio.h>
#include<string>
int main()
{
string str1="Hello";
string str2 = "World";
```

```
string str3, str4;
int len;
str3 = str1; // String copying
str4 = str1 + str2; // String concatenation
len = str4.size( ); // Finds string size
cout<<"str3=\t"<<str3<<endl;
cout<<"str4=\t"<<str4<<endl;
cout<<"length of str4 = \t"<<len<<endl;
getch();
}
```

Output of the program

```
str3=     Hello
str4=     HelloWorld
length of str4      = 10
```

5.3.4 Commonly used string functions

The header file *string* provides many string manipulation functions. These are discussed with the help of programs as follows.

a. Concatenation of strings

Strcat() function

Combining two strings into a single string is called string concatenation. *Strcat()* function is used for this purpose. The following program demonstrates the implementation of *strcat()* function for the concatenation of the two strings.

```
// strings concatenation program
#include <iostream.h>
```



```
#include <string.h>
#include<conio.h>
int main()
{
char FirstName[15] = "SSC ";
char LastName[15] = "Computer";
strcat(FirstName, LastName);
cout << FirstName;
getch();
return 0;
}
```

Output of the program

```
SSC Computer
```

b. Copying of strings

copy() function

copy() function is used to copy one string to another string. The general syntax of *copy* function is given below:

str.copy (string_name, number_of_characters, starting_pointer)

Here, the *string_name* is the name of the string to which the string is to be copied. *number_of_characters* represents the exacts number of characters to be copied and *starting_pointer* indicates the starting point from where to copy. Consider the following program:

```
// strings copying program using copy() function
#include <iostream.h>
```

```
#include <string>
#include<conio.h>
int main()
{
string str = "Welcome to C++ Strings";
char x[55];
cout << "str is: " << str << endl;
str.copy(x, 7, 0); // copy 7 characters of string str into string x, starting from position zero
cout << "The copied string is: " << x << endl;
getch();
return 0;
}
```

Output of the program

Welcome to C++ Strings
The copied string is: Welcome

strcpy() function

For copying string, another string function *strcpy()* can also be used. Consider the following program to demonstrate the use of *strcpy()*.

```
// copying string using strcpy() function
#include <iostream.h>
#include <string>
#include<conio.h>
main()
{
string SS;
char CC[17];
SS = "This is a string";
strcpy(CC,"This is a string");
cout << SS << endl; cout << CC << endl;
getch();
}
```

Output of the program

This is a string
This is a string

c. Finding a character or word in a strings

find() function

To find a particular word or a character in a string, *find ()* function is used. *find ()* function takes only one argument which is the desired character or word within the string. The following program demonstrates how *find()* function is used to search the word **interesting** and character **g** in the string "C++ is a very interesting programming language".

```
// finding a word or character using find() function
#include <iostream.h>
#include <string>
#include<conio.h>
main()
{
string str("C++ is a very interesting programming language");
int loc1, loc2;
loc1 = str.find ("interesting ");
cout << "Word interesting is found on position: " <<loc1+1;
loc2 = str.find('g');
cout << "\nFirst character 'g' found on position: "<< loc2;
```

```
getch();
return 0;
}
```

Output of the program

Word interesting is found on position: 15

First character 'g' found on position: 24

d. Finding length of a strings

length() function

To find the length of a string, *length* () function is used. Consider the following program to demonstrate working of the *length* () function.

```
#include <iostream.h>
#include <string>
#include<conio.h>
main()
{
string str("C++ is a very interesting programming language");
cout << "Length of the given string is: "<< str.length();
getch();
return 0;
}
```

Output of the program

Length of the given string is: 46

e. Swapping strings

swap() function

To swap (or interchange) values of two strings, *swap*() function is used. Consider the following program to demonstrate working of *swap* () function.

```
#include <iostream.h>
#include <string>
#include<conio.h>
int main()
{
string str1 = "F.Sc Part I";
string str2 = "F.Sc Part II";
cout << "str1 is: " << str1 << endl;
cout << "str2 is: " << str2 << endl;
str1.swap(str2);
cout << "str1 is: " << str1 << endl;
cout << "str2 is: " << str2 << endl;
getch();
return 0;
}
```

Output of the program

str1 is: F.Sc Part I
str2 is: F.Sc Part II
str1 is: F.Sc Part II
str2 is: F.Sc Part I

Summary

- A collection of homogeneous data items stored in consecutive memory locations is called array.
- An array can store data of any valid data type, e.g. integer, floating point and character. An integer array can only store integer data.
- There are two types of arrays: one dimensional and multi-dimensional. Two dimensional arrays is a type of multi-dimensional array.
- One dimensional array can be defined by using a single subscript.
- Two dimensional arrays can be represented with the help of two subscripts. It is also called matrix in mathematics and table in business application.
- One dimensional array can be defined by preceding the name of the array by a valid data type and followed by a single subscript.
- Two dimensional arrays can also be defined by preceding the name of the array by a valid data type, such as int, long and char etc., and followed by two subscripts.
- One dimensional and two dimensional arrays can be initialized in the same way during the declaration time.
- Arrays can be efficiently and easily processed with the help of looping structures.
- A sequence of characters enclosed in double quotes is called string.
- Different operations such as comparison, concatenation, copying, and finding length can be performed on strings.

Exercise

Q.1 Fill in the Blanks.

i. In C/C++ language, normally the indexing of an array starts from _____.
ii. The size of the array int x[10] is _____.
iii. char century [100][365][24][60[60] is an example of_____ array.
iv. An array of characters is called _____.
v. float x[12] defines one dimensional array of type _____.

Q.2 Select the correct choice for the following Multiple Choice Questions.

i. A collection of data items of the same data types placed in contiguous memory locations is called_____,
   a. Record
   b. Array
   c. Program
   d. File

ii. Observe the following statements and decide what they do.
   string mystring;
   getline(cin, mystring);
   a. reads a line of string from cin into mystring
   b. reads a line of string from mystring into cin
   c. cin can't be used this way
   d. none of the above

iii. Which of the header file must be included to use string stream?
   a. <iostream>
   b. <string>
   c. <sstring>
   d. <sstream>

iv. The location number of an item in an array is called _____.
   a. Data
   b. Value
   c. Index
   d. Constant

v. Strings are character arrays. The last index of it contains the null character_____.
   a. \n
   b. \t
   c. \0
   d. \1

Q.3 Write TRUE/FALSE against the following statements.

i. Array is a collection of heterogeneous type of data.
ii. Array can be initialized during declaration time.
iii. char vowel[5] declares an integer type array that holds the vowel characters of English language.
iv. Loop is the best tool for handling large arrays.
v. Array elements are stored in the memory where ever free space is available.

Q.4 Declare an array to hold the high temperature (to the nearest tenth of a degree) for each day of a year. Assign a value of 0 to each day.

Q.5 Write down a C++ program to find the multiplication of two matrices A[3][2] and B[2][3].

Q.6 Write down a C++ program to find the subtraction and addition of two matrices A[3][3] and B[3][3].

Q.7 Write a C++ program to find the transpose of a matrix A[3][3].

Q.8 Write a C++ program to read the temperature of the whole week in an array and then find the hottest day of the week.

Q.9 Write a C++ program to read ten alphabets of English from the keyboard into a character type array and then sort them in descending order.

Q.10 Write a C++ program to find sum of the values of a two dimensional array int Test[2][3] and display the result on the screen.

UNIT 6

FUNCTIONS

After the completion of Unit-6, Students will be able to:

- Explain the concept and types of functions
- Explain the advantages of using functions
- Explain the signature of functions (Name, Arguments, Return type)
- Explain:
  - Function prototype
  - Function definition
  - Function call
- Differentiate among local, global and static variables
- Differentiate between formal and actual parameters
- Know the concept of local and global functions
- Use inline functions
- Pass the arguments:
  - constants
  - By value
  - By reference
- Use default arguments
- Use **return** statement
- Define function overloading
- Know advantages of function overloading
- Understand the use of function overloading with:
  - Number of arguments
  - Data types of arguments
  - Return type

INTRODUCTION

A good programming technique is to modularize the programs into small segments of codes. These small segments are called modules or procedures and perform some specific tasks. In C++ language, these modules are usually called functions. This unit describes the basic concepts of functions, its parts, parameters and arguments. Here, the concept of function overloading has also been discussed with the help of examples.

6.1 Functions

In C++ program, when a set of related statements are grouped together, in a particular format, to accomplish a specific task is called *function*. *Function* is one of the main building blocks of any programming language. There are some functions whose work has been fixed by the developers of C++ programming language. These are called *built-in functions*. *Built-in functions* are specific in its functionalities; therefore, the programmers develop their own functions called *user-defined functions*. These functions are the main focus point of this unit.

6.1.1 Concept and types of functions

Function is an important feature of any programming language. It makes the work easier by writing the code once and executing it again and again. Mainly, functions are divided into two types built-in and user defined.

a. Concept of function

Whenever we want to perform a task again and again in a program such as calculating the square or cube for multiple integer values then it is too difficult to write the same code multiple times, say 5 times for 5 different integer values, in the same program.

b. Types of functions

C++ functions are categorized into two types namely *built-in* function and *user defined* functions. The following sections describe these types in detail.

i. Built-in Functions

C++ language provides a number of pre-compiled functions for some commonly used operations. These functions are called *built-in functions*. *Built-in* functions are part of the C++ language and are highly valuable, pre-tested and completely reliable. The C++ built-in functions are made for various assignments ranging from algebra, geometry, trigonometry, finance, text and graphics to some complex operations.

Consider the following example that requests the value of angle and calculate its *sine* and display it on the screen.

```
// built-in function(sin())program
#include <iostream.h>
#include<conio.h>
#include<math.h>
int main()
{
float angle;
cout << "Enter value for angle:"<<endl;
cin >> angle;
cout<<"The Sine of "<<angle<<" degree is =
"<<sin(angle);
getch();
return 0;
}
```

Output of the program

Enter value for angle:

45.0

The Sin of 45.0 degree is = 0.850904

Here, **sin()** is a built-in function defined in the **math.h** header file. The value inside the **sin()** is converted to its corresponding *sin* value and displayed on the screen. Similarly we can calculate cosine, tangent and tangent inverse by using cos(x), tan(x) and cot(x) built-in functions respectively.

Consider another example:

```
//built-in function abs( ) program
#include <math.h>
#include <iostream.h>
#include <conio.h>
int main()
{
int X;
cout<<"Enter value for X"<<endl;
cin>>X;
cout<< "Absolute value of " <<X << " is: " <<abs(X);
getch();
return 0;
}
```

Output of the program

Enter value for X

-12

Absolute value of -12 is: 12

In this example, the value entered by the user, from the keyboard, is taken into the *abs( )* function and the result is displayed on the screen. Like abs( ), some other functions that are included in <math.h> header file are pow(x,n), sqrt(x), etc.

The built-in functions are defined in the C++ library which should be included in the program before using these functions. These functions are used for the most commonly and frequently used operations and cannot be modified according to the users need. This feature of the functions is its main limitation. To overcome this problem, C++ provides a new set of functions that can be defined by the users itself. The following section describes this set of functions in detail.

ii . User-defined functions

The functions defined by the users, according to their needs, to perform their own tasks are called users-defined functions. These functions are used to overcome the limitations of build-in functions.

Defining user-defined function

A function must be defined first to use it in the program. The general syntax for defining a user-defined function is given below.

```
type name (parameter1, parameter2, ...)
{
Statements;
}
```

Where:

- ***Type*** is the data type specifier of the data returned by the function.
- ***Name*** is the identifier by which a function will be called.

- ***Parameters*** are the variables that receive arguments. Each parameter has its data type like variable. There can be as many parameters as needed but they should be separated by commas.
- ***Statements*** constitute the body of the function. It is a block of statements surrounded by braces { }.

Consider a function named *addition* to take two integer values, add them together and return the result to the calling program. The following code segment defines this function:

```
// user-defined function (addition) program
#include <iostream.h>
#include <conio.h>
int addition (int a, int b)
{
int sum;
sum=a+b;
return (sum);
}
int main()
{
int z;
z = addition (55,33);
cout << "The sum of a and b is= " << z;
getch();
return 0;
}
```

Output of the program

The sum of a and b is =88

Working of user-defined functions

In the above program, addition (int a, int b) is the example of user-defined function defined by the user to perform the addition of two integer values (55 and 33 in this case). These values are passed by the function call *addition (55,33)*. After performing addition, the function returns the sum (88) back to calling program (in this case, main) using the following statement.

return (sum);

The sum is assigned to the variable z, in *main*(), as shown below:

z = addition (55, 33);

The following line of code, in *main*, is used to display the result on the screen as shown in the above program.

cout << "The sum of a and b is= " << z;

6.1.2 Advantages of using functions

Using functions we can structure our programs in a more modularized way. The use of functions provides many benefits, including:

- ***Code reusability***: The code inside the function can be reused by calling it again and again in the program.
- ***Updating code***: It is much easier to change or update the code in a function, which needs to be done once.
- ***Readability and understandability***: It makes the code easier to read and understand.

6.1.3 Function signature

The signature of a function comprises of the parts given below.

- Name of the function
- The number, order and data types of the arguments
- Return type of function

Let us consider the *add* function to demonstrate function signature

```
double add(int x, double y)
{
return x+y;
}
```

The signature of the above mentioned function is:

double *add*(int x, double y)

The Signature of this *add* function comprises of:

- Name of the function: **add**
- The data types of the arguments: **int , double**
- Return type of the function: **double**

6.1.4 Components of functions

Each function consists of three components. These are:

- Function prototype/declaration
- Function definition
- Function call

a. Function prototype

A *function prototype* is that part of a function that tells the compiler the name of the function, the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters, and the order of parameters.

A function prototype is also called *function declaration* and is used in situation whenever the function is defined after the **main()** function. The function prototype ends with a semicolon (;).

The function prototype for a function named "*maximum*" to calculate maximum number out of three integers is given below:

int maximum(int a, int b, int c);

The compiler use function prototypes to validate function call. In a prototype, argument names are optional; however, the types are necessary as shown below:

int maximum(int , int , int);

Function prototype is used in C++ program only when the function is defined after the definition of *main()* function. If the function definition lies before the *main()* function then there is no need for the function prototype.

b. Function definition

A function definition specifies what a function does. It has two parts: a *declaration /header* and *function body* enclosed in { }. Function *declarator* is similar to *function* prototype with the only difference that it has the variables name for the parameters and without semicolon (;). Consider the following function *declarator*:

int maximum(int x, int y, int z)

The second part of function definition is the function body. It contains the actual statements that perform the intended task and is always enclosed in { }.

The whole function definition is shown below:

```
int maximum(int x. int y, int z) // function declarator
{
int max = x;
if ( y > max )
max = y;
if ( z > max )
max = z;
return max;
}
```

function body

c. Function call

A *function call* is a statement in the calling function (e.g. **main()** function) to execute the code of the function. Consider the following statement:

maximum(a, b, c);

The function call has the list of arguments that are passed to the *function declarator*. The arguments in this call are a, b and c. The following program shows different parts of the function i.e. prototype, definition and call.

```
// finding maximum out of three integers using function
#include <conio.h>
#include <iostream.h>
int maximum( int, int, int ); // function prototype
int main()
{
int a, b, c;
cout << "Enter three integers: ";
cin >> a >> b >> c;
cout << "Maximum is: " << maximum ( a, b, c ) << endl; // Function call
getch();
return 0;
}

int maximum( int x, int y, int z ) // function definition
{
int max = x;
if ( y > max )
max = y;
if ( z > max )
max = z;
return max;
}
```

Output of the program

```
Enter three integers: 4 -2 6
Maximum is: 6
```

6.1.5 Scope of variables

By the scope of a variable we mean that if a variable is defined in a program then in which parts of the same program this variable can be accessed. There are three scopes of variables.

- Local scope
- Global scope
- Static scope

a. Local/Automatic variables

The variables defined within a block are called local variables and its scope is local to block. These variables are not accessible from outside the block and thus their visibility remains only to that block. For example, in the following code segment, the variables a, b and c are local variables and cannot be used outside this boundary.

```
int main()
{
int a, b, c;
cout << "Enter three integers: ";
cin >> a >> b >> c;
// a, b and c are the local variables and cannot be used outside the {} block
cout << "Maximum is: " << maximum ( a, b, c ) << endl; getch();
return 0;
}
```

It is impossible to use the variables x, y or z defined in maximum( ) function directly within *main* function because these variables are local to the *maximum* function.

The following program makes use of a function *square()* that has its own local variable 'y' which is not accessible in the *main()* function.

```
// local variables (example) to calculate square of integers
#include <iostream.h>
#include<conio.h>
int square( int y); // function prototype
```



```
int main()
{
for ( int x = 1; x <= 10; x++ )
cout << square( x ) << " ";
cout << endl;
getch();
return 0;
}
// Function definition
int square( int y )
{
return y * y;
}
```

b. Global variables

The variables defined usually at the top of a program before the *main()* function are called global variables. The variables declared *globally* are visible from any point of the code. They are accessible from inside and outside of all the functions of the same program. In order to declare global variables we simply have to declare the variable outside any function or block; i.e. directly in the body of the program.

The following program describes local and global variables.

```
#include <iostream.h>
#include <conio.h>
int add(); /* function prototype */
int val1, val2, val3; /*These are global variables */
int add()
{
```

```
int sum; // here, sum is local variable
sum = val1 + val2 + val3;
return sum;
}
int main()
{
int total; //here, total is local variable
val1 = 10;
val2 = 20;
val3 = 30;
total = add();
cout<<"The sum of the three global variables="<<total;
getch();
return 0;
}
```

Output of the program

The sum of the three global variables=60

In this example, the variables *val1, val2, val3* are defined globally and can be accessed both in *main()* function and *add()* function without generating any error message. Here in the same program, the variables *sum* in function *add()* and *total* in function *main()* are local to the respective functions and are not visible outside the boundaries of these functions.

c. Static variables

Static variables are those variables which are preceded by the keyword **static** while declaring. For example:

```
static Static_Var;
```


Static variables are initialized once in the program and remains in memory until the end of the program. These variables have the capability to preserve information about the last value a function returned, until the program completely ends. Static variables are *local in scope* to their module or function in which they are defined, but *life is throughout the program.*

Consider the following example to demonstrate the use of static variables.

```
/* static and automatic variables program*/
#include <conio.h>
#include <iostream.h>
void demo(){
auto int Auto_Var = 0; // automatic variable
static int Static_Var = 0; // static variable
cout<<"Auto ="<<Auto_Var<< ","<< "Static="<<Static_Var<<endl;
++ Auto_Var; ++ Static_Var; }
int main()
{
int i=0;
while( i < 3 )
{ demo(); i++; }
getch();
return 0;
}
```

Output of the program

```
Auto = 0, Static = 0
Auto = 0, Static = 1
Auto = 0, Static = 2
```

Here, the automatic variable **Auto_Var** losses its values when the control goes out of the function body but the static variable **Static_Var** keeps its last value even if the control goes out of the function.

6.1.6 Formal parameters and actual parameters

In function prototypes and function calls, variables and values are written in the parenthesis of the functions. These are divided into two categories; ***formal parameters*** and ***actual parameters***.

a. Formal parameters

Formal parameters are those parameters which appear in function *declarator/header* and also in *function prototype*. These are also called *formal arguments* and are used to receive values for functions.

For example:

```
void foo(int x); // prototype -- x is a formal parameter
void foo(int x) // declarator -- x is a formal parameter
{
body
}
```

b. Actual parameters

Actual parameters are those parameters which appear in function calls and are used to send data to formal parameters. These are also called *actual arguments*. Consider the following function call:

```
foo(6); // 6 is the argument passed to parameter x
foo(y+1); // the value of (y+1) is the argument passed to parameter x
```


The actual parameters can be fixed values or variables holding values or expressions resulting in some values.

In the function **main,** in the following example, **rice, chicken** and **fruit** are actual parameters when used to call **calculate_bill**. On the other hand, the corresponding variables in calculate_bill (namely **diner1, diner2** and **diner3**, respectively) are formal parameters because they appear in the function definition and receive values passed in the function call.

Let's look at calculate_bill function in the following program:

```
/* formal and actual parameters program */
#include <stdio.h>
#include<conio.h>
#include <iostream.h>
int calculate_bill (int, int, int);
int main()
{
int bill;
int rice = 25;
int chicken = 32;
int fruit = 27;
bill = calculate_bill (rice, chicken, fruit);
cout<<"The total bill comes to "<<bill<< "Rs.";
getch();
return 0;
}
int calculate_bill (int diner1, int diner2, int diner3)
{
```

Actual Parameters

Formal Parameters

```
int total;
total = diner1 + diner2 + diner3;
return total;
}
```

Output of the Program

The total bill comes to 84 Rs.

Although, formal parameters are always variables but actual parameters may not be variables. Numbers, expressions, or even function calls can also be used as actual parameters. Here are some examples of valid actual parameters in the function call to calculate_bill:

```
bill = calculate_bill (25, 32, 27);
bill = calculate_bill (50+60, 25*2, 100-75);
bill = calculate_bill (rice, chicken, (int) sqrt(27));
```

6.1.7 Local and Global functions

The terms local and global functions specify the scope of functions within programs. Based on the scope of the functions, functions are categorized into two types; ***local functions*** and ***global functions***.

a. Local functions

Those functions which are defined inside the body of another function are called local function. Normally, we use *built-in* function inside the body of *main ()* function to perform our activities. Such declarations are termed as local.

b. Global functions

A function declared outside any function is called *global function*. A *global function* can be accessed from any part of the program. Normally, *user-defined* functions are considered as *global function* if they are defined before the *main ()* function and are thus accessible to every part of the program.

6.1.8 inline function

Using function calls in program, a lot of time is wasted in passing control from the calling function to the called function and returning control back to the calling function. This limitation can be overcome by the use of **inline** function.

C++ offers a way to combine the advantages of functions with the speed of code written in-place with the help of **inline functions**. In inline function, the function return type is preceded by the **inline** keyword which requests the compiler to treat the function as an inline and do not jump again and again to the called function. In the case of inline function, when the compiler compiles the code, all inline functions are expanded in-place, that is, the function call is replaced with a copy of the code of the function itself, which removes the function call overhead.

The disadvantage of the *inline function* is that it can make the compiled code quite larger, especially if the inline function is long and/or there are many calls to the inline function.

The format for the declaration of inline function is given in the following segment.

```
inline type name ( arguments ... )
{
Statements;
}
```

Calling an inline function is simple and is just like the call to any other function. In the call, the **inline** keyword is not needed to be included.

The following program demonstrates the use of inline function in a program.

```
/* use of inline function to find minimum out of two integers*/
#include<iostream.h>
#include<conio.h>
inline int min(int a, int b)
{
return a > b ? b : a;
}
int main()
{
cout <<"Minimum out of 25 and 26 is: "<<min(25, 26);
cout <<"\nMinimum out of 23 and 22 is:"<<min(23,22);
getch() ;
return 0;
}
```

Output of the program

Minimum out of 25 and 26 is: **25**
Minimum out of 23 and 22 is: **22**

6.2 Passing arguments and returning values

When we want to execute functions, we need to pass arguments (values) to a function. The result (values) produced within the function body is then returned back to the calling program. The following section describes different methods used for passing arguments (values) to functions.

6.2.1 Passing arguments

The following are the most commonly used methods of passing arguments to functions.

- Passing arguments by constants
- Passing arguments by values
- Passing arguments by reference

a. Passing arguments by constants

While calling a function, arguments are passed to the calling function. In passing constants as arguments, the actual constant values are passed instead of passing the variables.

Consider the following program:

```
// passing integer constant as argument.
#include<iostream.h>
#include<conio.h>
void display(int x)
{
cout << " x= " << x << endl;
}.
int main()
{
display(25); // function call
getch();
return 0;
}
```

Output of the program

x = 25

In the above program, the function call, *display (5)*, passes the constant argument (5) to the function.

Consider another program defining a function *displayGender ()* that takes a character constant, 'F' or 'M', as argument from the calling function:

```
// passing character constants as argument
#include<iostream.h>
#include<conio.h>
void displayGender(char ch)
{
cout << "The gender marker you passed is : " << ch << endl;
}
int main()
{
displayGender('F'); // function call
getch();
return 0;
}
```

Output of the program

The gender marker you passed is : F

b. Passing arguments by values

By default, arguments in C++ are passed by value. When arguments are **passed by value**, a copy of the argument value is passed to the function formal parameter.

Consider the following program to demonstrate the use of passing arguments by value.


```
// passing parameters by value
#include<iostream.h>
#include<conio.h>
void foo(int y)
{
cout << "y in foo() = " << y << endl;
y = 6;
cout << "y in foo() = " << y << endl;
} // y is destroyed here
int main()
{
int x = 5;
cout << "x in main() before call= " << x << endl;
foo(x); //argument passed by value
cout << "x in main() after call= " << x << endl;
getch();
return 0;
}
```

Output of the program

```
x in main() before call=5
y in foo() = 5
y in foo() = 6
x in main() after call=5
```

In this program, the original value of x is not changed before and after calling the functi
foo().

Consider another example that uses a function **addition** that gets arguments by values.

```
// passing parameters by value
#include <iostream.h>
#include <conio.h>
int addition (int a, int b)
{
int result;
result=a+b;
return (result);
}
int main()
{
int z, x=5, y=3;
z = addition (x, y); //arguments passed by value
cout << "The sum of both the values is =" << z;
getch();
return 0;
}
```

Output of the program

```
The sum of both the values is =8
```

c. Passing arguments by reference

In **pass by reference**, the reference to the function parameters are passed as arguments rather than value of variables. Using pass by reference, the value of arguments can be changed within the function.

In passing arguments by reference, the formal parameters should precede by the ampersand sign &. Consider the following example.

```
#include<iostream.h>
#include<conio.h>
void AddOne(int &y)
{
y++; // changing values in function
}
int main()
{
int x = 55;
cout<<"x in main() before call= " << x << endl;
AddOne(x); //passing arguments by reference
cout<<"x in main() after call = " << x << endl;
getch();
return 0;
}
```

Output of the program

```
x in main() before call= 55
x in main() after call = 56
```

In the above example, the value of x is passed by reference and changed inside the function **AddOne().** This change affects the values of x in **main()** because the formal argument **&y** in the function declarator is a reference to x and any change to y results in change in x.

Consider another example having a function named **duplicate** that receive arguments by reference:

```
#include <iostream.h>
#include<conio.h>
void duplicate (int& a, int& b, int& c)
{
a=a*2;
b=b*2;
c=c*2;
}
int main()
{
int x=1, y=3, z=7;
cout<<"values of x, y and z in main()before calling function\n";
cout << "x=" << x << ", y=" << y << ", z=" << z<<endl;
duplicate (x, y, z);
cout<<"values of x, y and z in main() after calling function\n";
cout << "x=" << x << ", y=" << y << ", z=" << z;
getch() ;
return 0;}
```

Output of the program

```
values of x, y and z in main()before calling function
x=1,y=3,z=7
values of x, y and z in main() after calling function
x=2, y=6, z=14
```


Sometimes we need a function to return multiple values. However, functions can only return one value. One way to return multiple values is the use of the method of passing arguments by reference.

Passing arguments by this method allow programmers to change the value of the arguments, which is sometimes useful. Similarly, it is a fast approach to passing and processing values in a function and also has the facility of returning multiple values from a function.

6.2.2 Default arguments

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do this, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead.

To demonstrate the concept of default arguments, consider the following general syntax.

Type function_name (parameter1=value, ...);

Here in this line of code, the *type* is any valid data type, *function_name* is the name of the function and *parameter1* is the parameter having a *default value* named *value* assigned to it in the prototype. In the example given below, parameter b has a default value 2 assigned to it in the declaration. Now, if, one value is passed in the function call then the default value of b will be taken as the value of parameter 'b'.

```
int divide (int a, int b=2)
```

The following program demonstrates the concept of default arguments.

```
#include <iostream.h>
#include<conio.h>
int divide (int a, int b=2)
{
int r;
r=a/b;
return (r);
}
int main()
{
cout<<"Result by providing one argument="
<< divide (12); // function call with one arguments
cout<< endl;
cout<<"Result by providing both the
arguments=" <<divide (20,4);
getch();
return 0;
}
```

Output of the program

```
Result by providing one argument=6
Result by providing both the arguments=5
```

As we can see in the body of the program there are two calls to function *divide()*. In the first one, we have only passed one argument, but the function **divide()** gets the second value from the default argument, int b=2, in the function prototype and thus results is 6.

In the second call, there are two parameters passed, so the default value for b is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5.

6.2.3 return statement

If the return type of a function is any valid data type such as *int, long, float, double, long double* or *char* then *return* statement should be used in the function body to return the result to the calling program.

The general syntax of the use of *return* statement is given below.

return (expression or variable holding results or constant);

Consider the following function:

```
// the use of return statement in the cube() function
int cube(int x)
{
int c;
c= x*x*x;
return c; // or return (c);
}
```

If a function returns a value by the use of *return* statement then the call to the function should be from a statement or an expression that utilizes it.
For example:

```
cout<<cube(x);
```

```
int y=cube(x);
```

6.3 Function overloading

6.3.1 Definition

In programming languages, two or more variables or functions with the same name cannot be used in a single program or block of code. **Function overloading** is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different number or types of parameters.

Consider the following example having two functions with the same name *display* used to perform slightly different tasks in over loaded form.

```
#include <iostream.h>
#include<conio.h>
void display (int a)
{
cout<<a;
}
void display()
{
cout<< "Welcome"<<endl;
}
int main()
{
display() ;
display(55) ;
getch();
return 0;
```



```
}
```

Output of the program

```
Welcome
55
```

Here, the function *display* is overloaded with two tasks: one for displaying an integer value and the other for displaying a welcome message. While calling the overloaded functions, the compiler decides at run time that which *display* function should be called. To determine which *display* to call is decided by looking at the number and types of the arguments or parameters of the functions. In this example, *void display ()* is called for display () and *void display (int a)* for *display (55)*.

Consider another program having two functions with the same name *multiply* that operate on integers and floating points numbers.

```
#include <iostream.h>
#include<conio.h>
int multiply (int a, int b)
{
return (a*b);
}
float multiply (float a, float b)
{
return (a*b);
}
int main()
{
cout << "Output of integers="<< multiply (5, 33)<<endl;
```

```
cout << "Output of floats="<< multiply (2.1, 3.0);
getch();
return 0;
}
```

Output of the program

Output of integers=165
Output of floats=6.3

In this case, we have defined two functions with the same name, ***multiply***, that accept two arguments. One accepts two arguments of type **int** and the second of type **float**. The compiler looks at the arguments and calls the respective function. Here, ***multiply*** *(int a, int b)* is called for ***multiply*** *(5, 33)* because the arguments match with the data types of the formal parameters. Similarly, ***multiply*** *(float a, float b)* is called for the call ***multiply*** *(2.1, 3.0)* because the arguments and formal parameters match each other in types.

6.3.2 Advantages of function overloading

- Function overloading can significantly reduce complexity of programs.
- Using function overloading, we can declare multiple functions with the same name that have slightly different purposes.
- As multiple functions have same name, therefore, remembering them is easier as compared to remembering more names.
- It increases the readability of programs.
- It exhibits the behavior of polymorphism.

6.3.3 Understanding function overloading concept

For the complete understanding of the concept of function overloading, one should know those features of the overloaded functions which disambiguate them and make them unique in a single program. These features are listed below:

- Number of arguments
- Data types of arguments
- Return type

a. Number of arguments

Functions can be overloaded if they have different numbers of parameters. More than one functions with the same name but different number of parameters can be used in a single program. In the calls to these functions, the compiler disambiguates the calls by looking at the number of arguments and formal parameters in the function decelerators. Consider the following example:

```
/* overloaded functions with different number of parameters */
#include <iostream.h>
#include <conio.h>
int Addition(int X, int Y)
{
return (X + Y);
}
int Addition(int X, int Y, int Z)
{
return (X + Y + Z);
}
int main()
{
int a=55, b=22, c=100;
cout << "Output of 2 integers="<<Addition (a,b);
cout << endl;
cout << "Output of 3 integers="<<Addition (a,b,c);
cout << endl;
getch();
```

```
return 0;
}
```

Output of the program

```
Output of 2 integers=77
Output of 3 integers=177
```

In this example, two functions have been used with the name *Addition*. One has two parameters and the second has three parameters. In the *main()* function there are two calls *Addition (a,b)* and *Addition (a,b,c)* which call functions *int Addition(int X, int Y)* and *int Addition(int X, int Y, int Z)* respectively by looking at the number of parameters.

b. Data types of arguments

One way to achieve function overloading is to define multiple functions with the same name but different types of parameters. Consider the following example:

```
/*overloaded functions print()with different types of
parameters*/
#include <iostream.h>
#include <conio.h>
void print (int X)
{
cout<< "X="<<X<<endl;
}
void print (float Y)
{
cout<< "Y="<<Y<<endl;
}
int main()
{
print(55);
print(33.66);
getch();
return 0;
}
```


Output of the program

```
X=55
Y=33.66
```

Here, two functions, *print (int X)* and *print (float Y)*, have been used with different types of parameters, *int* and *float*. In *main()* function, two calls are used *print(55)* and *print(33.66)*. The compiler looks at the type of arguments and calls the corresponding function.

c. Return type

The return type of a function is not considered when functions are overloaded. It means that if two or more functions with the same name have same function signatures but different return types then they are not considered as overloaded and the compiler will generate error message.

Consider the following case:

```
int display();
double display(); // This prototype generates error message
```

Consider another example:

```
int RandomNumber();
double RandomNumber(); // This prototype generates error message
```

Here, the compiler generates an error message of re-declaration for the second declaration at line number 2. It is because that both the declarations have same number of parameters (zero in this case) but only the return types are different which do not play any role in the overloading. If we want to do it, for then, these functions will need to be given different names.

Summary

- A function is a self-contained program that performs a specific task.
- C++ has two types of functions, built-in and user-defined. Built-in functions are specific in their activities and cannot be used for general type of tasks.
- Every C++ program comprise of one function called main (). It is the point from where the compiler starts the execution of every C++ program.
- A good C++ programmer writes programs that comprise of small functions.
- Functions are one of the main building blocks of C++ programs. It modularizes large programs into segments and thus increase the program readability and also provides the facility of code reusability.
- Each function has its own name and when the function is called the execution of the program branches to the body of that function. When the function is finished, execution returns to the area of the program code from which it was called, and the program continues on to the next line of code.
- Function prototype tells the compiler the name of the function, number, types and order of parameters.
- A function definition is a set of instructions enclosed in a block which performs the intended task of the function.
- Local/Automatic variables have local scope and can only be accessed in the block in which they are declared. These variables cannot be accessed from outside the block.
- Global variables have global access and their visibility is throughout the program in which they are defined.

- Static variables have scope of local variables and retain its values throughout the life of the program.
- The variables that appear in the function prototype and function declaration are called formal parameters.
- The variables in function calls that hold the values to be passed to the function are called actual parameters.
- inline functions are special functions with inline keyword that are used to minimize the problem of function call overhead with the expenses of memory wastage.
- C++ provides the facility of calling a function with fewer arguments with the help of default arguments.
- The phenomenon of using same function name for related but slightly different tasks is called function overloading.
- In overloaded function, the function signature, number, types and order of arguments should be different from each other.

Exercise

Q.1 Fill in the Blanks.

i. Normally a function can return _____ value.

ii. The only difference between the function declarator and function prototype is the use of _____.

iii. The scope of global variable declared in a C++ program, is _____.

iv. Using _____, we can declare multiple functions with the same name, having slightly different purpose.

v. If a function needs to return a value to the calling function then its return type should not use the keyword _____.

Q.2 Select the correct choice for the following Multiple Choice Questions.

i. The phenomenon of having two or more functions in a program with the same names but different number and types of parameters is known as:
   a. Inline Function
   b. Nested Function
   c. Function overloading
   d. Recursive Function

ii. We declare a function with _____ if it does not have any return type
   a. long
   b. double
   c. void
   d. int

iii. Arguments of a functions are separated with_____.
   a. Comma (,)
   b. Semicolon (;)
   c. Colon (:)
   d. None of these

iv. Variables inside parenthesis of functions declarations have _____ level access.
   a. Local
   b. Global
   c. Module
   d. Universal

v. Observe the following function declaration and choose the best answer:
   int divide (int a, int b = 2)
   a. Variable b is of integer type and will always have value 2
   b. Variable a and b are of int type and the initial value of both variables is 2
   c. Variable b is international scope and will have value 2
   d. Variable b will have value 2 if not specified when calling function

Q.3 Write TRUE/FALSE against the following statements.

i. A function cannot be overloaded only by its return type.

ii. A function can be overloaded with a different return type if it has all the parameters same.

iii. Inline functions minimize the size of the program.

iv. If we have a function: int sum (int x, int y), then it will return float value to the calling program.

v. Passing arguments by address allows the function to change the value of the argument.

Q.4 Write a program with a function that takes two *int* parameters, adds them together, and then returns the sum.

Q.5 Write a program with a function name "**mean**" to read in three integers from the keyboard to find the arithmetic mean.

Q.6 Write a C++ program having a function name **rectangle** to read the length and width of a rectangle from the keyboard and find the area of the rectangle. The result should be returned to the *main* program for displaying on the screen.

Q.7 Write a C++ program having two function names **area** and **perimeter** to find the area and perimeter of a square.

Q.8 Write a C++ program to read a number from the keyboard and then pass it to a function to determine whether it is prime or composite.

Q.10 Write a C++ program to get an integer number from keyboard in *main* program and pass it as an argument to a function where it calculate and display the table.

Q.11 Define function and differentiate between built-in and user-defined functions with the help of examples.

Q.12 How function prototype and declarator differ from each other? Explain with the help of examples.

Q.13 Define default arguments. Describe the advantages and disadvantages of default argument.

Q.14 What is meant by the term function overloading? How a function can be overloaded in C++? Describe it with the help of an example program.

Q.15 Describe the role of the following in function overloading.

- Data types of arguments
- Number of arguments
- Order of arguments

UNIT 7

POINTERS

After the completion of Unit-7, Students will be able to:

- Define pointers
- Understand memory addresses
- Know the use of reference operator &
- Know the use of dereference operator *
- Declare variables of pointer types
- Initialize the pointers

INTRODUCTION

When variables are declared, computer reserves space in its memory for some objects. The variable name refers to that memory space that is occupied by that object. This allows us to store the value of those variables in the space reserved. Computer refers to these spaces using their addresses. To refer to memory addresses, C++ provides the facility of pointers. The objective of this Unit is to understand how variables are stored in computer memory and how spaces are allocated to them. Sometimes, a programmer needs to refer to another memory location which can only be done by the use of **pointers**. *Pointer* is one of the most *powerful* tool to handle memory addresses. This Unit covers the concept of memory addresses in detail and explains the working of reference operator & and dereference operator * with the help of examples. It also focuses on how to declare pointer type variables and how to initialize them in a program.

7.1 Pointers

Pointer is an important and powerful feature of C++ language, in which a variable points to another variable by holding its address.

7.1.1 Pointer

A **pointer** is a variable that holds the address of another variable. In memory, each and every variable has an address assigned to it by the compiler and if a programmer wants to access that address, another variable called "pointer" is needed. For the declaration of pointer, an *asterisk* * symbol is used between the data type and the variable name.

Consider the following pointers declaration of the integer variable **marks**, floating point variable **percentage** and character type variable **name**:

```
int * marks;
float * percentage;
char * name;
```

7.1.2 Memory addresses

When writing a program, variables need to be declared. Declaration of variable is simple and its declaration tells the computer to reserve space in memory for this variable. The name of the variable refers to that memory space. This task is automatically performed by the operating system during runtime. When variables are declared, programmers store data in these variables. Computer refers to that space using an *address*. Therefore, everything a programmer declares has an address. It is analogous to the home address of some person. Using pointer variables, one can easily find out the address of a particular variable.

7.1.3 Reference operator &

As pointers are the variables which hold the addresses of other variables, therefore, while assigning addresses to them, a programmer needs a special type of operator called **address-of operator** that is denoted by ampersand **&** symbol. This provides address of a memory location.

To understand it, consider the following segment of code.

```
loat x = 5.5;
loat *fPointer;
Pointer;= &x; // assign address of x to fPointer;
```

Conceptually, the above lines of code can be pictorially represented as:

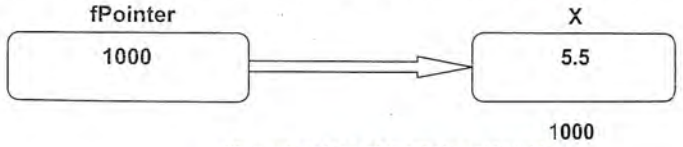


Figure 7.1: Use of the Pointer Variable

this pictorial representation, the variable x has a *float* value 5.5 that is stored at location 000. The pointer variable *fPointer* points to the variable x by holding its address 1000 as value. In this case, if we want to print the value of the variable x it will display 5.5 but if e print the value of *fPointer* it will display 1000.

onsider the following program:

```
* use of pointer in a program */
include <iostream.h>
include<conio.h>
t main()
```

```
{
float x = 5.5;
float *fPointer;
fPointer= &x;
cout << "The address of x= "<<&x << endl;
cout<< "The value of x= "<<x<<endl;
cout << "The value of fPointer = "<< fPointer;
getch();
return 0;
}
```

Output of the Program

The address of x= 1000
The value of x= 5.5
The value of fPointer =1000

This program gives some other output (address) on another computer depending upon the availability of the memory.

7.1.4 Dereference operator *

Pointers are used to store the address of another variable. If we want to store the value of the variable through the pointer, then we need a special type of operator called **dereference operator** denoted by **asterisk** *. To use dereference operator, precede the pointer variable with **asterisk** * symbol. This operator can be read as "value pointed by".

Consider the following program to demonstrate the use of **dereference operator** *.


```
/* use of dereference operator (*) in a program */
#include <iostream.h>
#include<conio.h>
int main()
{
int n = 100;
int *Pn; //defines a pointer to n
Pn=&n; //Pn stores the address of n
int valueN;
valueN=*Pn;
cout << "The address of n= "<<&n << endl;
cout<< "The value of n= "<<n<<endl;
cout << "The value of Pn = "<< Pn<<endl;
cout << "The value of *Pn = "<< (*Pn)<<endl;
cout << "The value of valueN = "<< valueN;
getch();
return 0;
}
```

Output of the Program

The address of n= 0x8fc5fff4
The value of n= 100
The value of Pn = 0x8fc5fff4
The value of (*Pn) = 100
The value of valueN= 100

In this example, the instructions at lines 10 and 14 make use of the **dereference operator** * and thus access the actual values of the original variable n which is pointed out by the pointer Pn. In pointers, the ampersand operator & is the **reference** operator

and can be read as "address of" and * is the **dereference** operator that can be read as "value pointed by". These operators are complementary of each other and have opposite meanings. A variable referenced with & can be dereferenced with *.

7.1.5 Declaring variables of pointer types

The declaration of pointer is simple and is similar to the declaration of a regular variable with a minor difference of the use of an *asterisk* * symbol between the data type and the variable name. Consider the following general format:

Data type * nameVariable;

Here, *data type* is the type of the value of that variable to which this pointer will point. This *date type* is not the type of the pointer itself but the type of the data the pointer points to. To understand it, consider the following examples of pointers declaration.

*int** totalMarks;
char *Name;
*float** percentage;

In this example, three pointers *totalMarks*, *Name* and *percentage* are declared. Each one is intended to point to a different data type. All these pointers occupy space in memory. The first pointer points to an **int**, the second to a **char** and the last one to a **float**.
It is notable that the asterisk * symbol used between the data type and the name of the variable is the indication for the pointer declaration and is not used for multiplication. There are three ways to place the asterisk. These are: placing next to the data type, the variable name, or in the middle. All these variations are shown in the above declarations of variables *totalMarks*, *Name* and *percentage*.

A pointer of type *int* can only points to a variable of type *int* and cannot to some other type of variable. Same is the case for other data types as well, but, there is a special type of pointer named **void pointer** or generic pointer that can point to an objects of any data type. Its declaration is similar to the declaration of normal pointers with the only difference of the use of *void* keyword as the pointer's type. Consider the following statement of declaration of the *void* pointer:

void *pointerVoid; // pointerVoid is a void pointer

As, a *void* pointer can point to objects of any data type, therefore, consider the following statements.

```
int X;
float Y;
void *pointerVoid;
pointerVoid = &X; // valid
pointerVoid = &Y; // valid
```

In this segment of code, *pointerVoid*, stores the address of both *integer* variable X and *floating point* variable Y. The compiler decides at run time that the address of which type of variable should be assigned to this pointer.

7.1.6 Pointer initialization

Assigning values to pointers at declaration time is called pointer initialization. As we know that the values of pointers are the addresses of other variables, therefore, sometimes when we declare pointers we may want to explicitly specify to which variables they will point.

Consider the following segment of code to understand the concept of pointer initialization:

***float* Temperature;**
***float* *PTemperature = &Temperature;**

Here, *PTemperature* is a pointer variable to a floating point variable. As this pointer is created with the statement '*float *PTemperature*', immediately the address of a float variable '*Temperature*' is assigned to it. The behavior of the above code is being equivalent to the following code.

***float* Temperature;**
***float* *PTemperature;**
PTemperature = &Temperature;

It should be considered that at the moment of declaring a pointer, the asterisk * indicates that it is a pointer variable and not the **dereference operator.**

Consider the following program to explain the concept of pointer initialization.

```
/* pointer initialization program */
#include <iostream.h>
#include<conio.h>
int main()
{
float Temperature;
float *PTemperature = &Temperature;
cout << "The address of Temperature is = "
<<&Temperature << endl;
cout<< "The value of *PTemperature is = "
<< PTemperature <<endl;
getch();
```



```
return 0;
}
```

Output of the Program

The address of Temperature is = 0x8f98fff2
The value of PTemperature is = 0x8f98fff2

Here, the pointer *PTemperature* is initialized with the address of the variable *Temperature*.

Sometimes we need to initialize a pointer to zero. Such pointers are called null pointers and they do not point to anything. Null pointers can be defined by assigning address 0 to them.

Consider the following initialization to demonstrate null pointer:

```
int *NPtr; //defines Null pointer
NPtr = 0; // this assigns address 0 to NPtr
```

The use of NULL pointers is mostly done in dynamic memory allocation.

Summary

- A pointer is a variable that points to or refers to another variable. That is, if we have a pointer variable of type "pointer to int" it might point to the int variable.
- Some tasks in C++ are easier to do with pointers, such as accessing the addresses of variables indirectly and operating their values.
- Like normal variables, Pointer variables are also declared in programs before using them. To declare a pointer variable, the data type of the variable is followed by * symbol.
- While using pointers in C++ programs, address-of-operator denoted by ampersand & symbol is used to assign the address of variables to the pointer variable.
- In C++ program, if we want to use the value of the variable to which pointer points a special type of operator called dereference operator, denoted by asterisk *; is used with pointers.
- In C++ programs, pointers are initialized to addresses of other variables like the initialization of ordinary variables.

Exercise

Q.1 Fill in the Blanks.

i. Pointers are special type of variable that store _____ of other variables.
ii. If we have a statement int n; and we want to define a pointer to n then its data type must be of the type _____.
iii. Assigning values to a pointer during its declaration time is called _____.
iv. & is the _____ operator.
v. * is the _____ operator which can be read as "value pointed by" operator.

Q.2 Select the correct choice for the following Multiple Choice Questions.

i. If we have the statement int *Ptr; then to what Ptr point?
   a. Points to an integer type variable
   b. Points to a character type variable
   c. Points to a floating point type variable
   d. None of above
ii. A pointer is:
   a. A keyword used to create variables
   b. A variable that stores address of an instruction
   c. A variable that stores address of another variable
   d. All of the above
iii. The operator used to get value at address stored in a pointer variable is
   a. *
   b. &
   c. &&
   d. ||
iv. Which of the statements is correct about the following segment code?

```
int i=10;
int *j=&i;
```

   a. j and i are pointers to an int
   b. i is a pointer to an int and stores address of j
   c. j is a pointer to an int and stores address of i
   d. j is a pointer to a pointer to an int and stores address of i

v. In pointers, dereference operator * is used to:
   a. Address the value of the pointer variable.
   b. Points to the value stored in the variable pointed by the pointer variable
   c. Both a and b
   d. None of the above

Q.3 Write TRUE/FALSE against the following statements.

i. The statement int *ptr; defines an ordinary C++ variable.
ii. The following lines of codes assign the value of the variable Temp to the pointer variable PTemp.

```
float Temp;
float *PTemp = &Temp;
```

iii. Pointer variables can be initialized by addresses of the variables to which they refer.
iv. To assign addresses to pointers, a programmer needs a special type of operator called address-of-operator &.
v. When a variable is declared in C++, the compiler reserves location for this variable in computer memory to store data.

Q.4 Define the term pointer. Describe the advantages of using pointer variables.

Q.5 What is the difference between the dereference operator * and reference operator &? Explain with the help of some lines of code.

Q.6 What is meant by the term pointer initialization? Write a simple program to illustrate this concept.

Q.7 How the declaration of a pointer variable is different from the declaration of a variable.

UNIT 8

OBJECTS AND CLASSES

After the completion of Unit-8, Students will be able to:

- Define class and object
- Know the members of class:
  - Data
  - Functions
- iii. Understand an access specifiers:
  - Private
  - Public
- Know the concept of data hiding
- Define constructor and destructor
  - Default constructor/destructor
  - Users-defined constructor
  - Constructors overloading
- Declare objects to access
  - Data members
  - Member functions
- Understand the concept of the following with daily life examples:
  - Inheritance
  - Polymorphism

INTRODUCTION

In structural or traditional programming, the data and actions are considered as two separate entities which cannot provide a very intuitive representation of real world objects and phenomenon. To eliminate this limitation, Object-Oriented Programming (OOP) concept was introduced which combines data and actions together in one package called class. This allows programmers to write programs in more modular fashion to make them easier to understand. This modularization also provides a higher degree of code-reusability. Object-oriented programming also brings several other useful concepts into the field of programming which are: data hiding, inheritance, encapsulation, abstraction, and polymorphism. This Unit is focused on these features of OOP.

8.1 Classes

Class is one of the main features of Object Oriented Programming language. It combines both data and functions into units or packages and makes the programs modularized.

8.1.1 Class and object

a. Class

A **class** consists of both *attributes* of real world objects and *functions* that perform operations on these attributes. The attributes are called **data members** and the operations are called **member functions**. It is an important feature of object oriented programming (OOP) and is used to hold both data and functions, of real world objects, tightly into a single package.

Classes are generally declared using the keyword **class**, with the following format:

```
class class_name
{
access_specifier_1:
member1;
```



```
access_specifier_2: // Body of the class
member2;
...
};
```

Here, **class_name** is a valid identifier for the class which is followed by the **body of the class** that is enclosed in pair of braces. The body of the class consists of:

- Data members
- Member functions

Data members and member functions are used with the *access specifiers*. An access specifier is one of the three types:

- private
- public
- protected

A C++ class ends with a semicolon (;). Consider the following example to understand the concept of **class**.

```
// Class declaration example
class CRectangle
{
private:
int x, y; // declaration of data members
public:
void set_values (int a, int b); // declaration of member function
int area (); // declaration of member function
};
```

The above statements declare a class named **CRectangle**. This class contains four members:

Two data members x and y of type **int** with **private** access specifier and two member functions with **public** access; **set_values ()** and **area ()**. In this example, only declarations for the member functions has given and not their definitions.

b. Object

A variable of type class is called **object**. In C++, when we declare a variable of type class, we call it *instantiating* the class and the variable itself is called an **instance** or **object** of that class. **Object** is of great importance in OOP, because, a class cannot be used without creating its **object**. The general syntax for creating an object of a class is given below.

Class_name Object_name;

Thus, for the above class **CRectangle**, the object can be created as follows:

CRectangle R1;

We can also create more than one objects in a single statement like:

CRectangle R1, R2, R3;

Here, **R1, R2, R3** are the objects of the same class **CRectangle** that share the same data member and member functions. The definition of a class does not occupy any memory. It only defines what the class looks like. In order to use a class, a variable of that class type must be declared. When an object is created then memory is set aside for all the data members and member functions of that class.

The following program demonstrates the concept of **class** in C++ and creation of object.

```
#include <iostream.h>
#include <conio.h>
class Test
{
```



```
public:
void print()
{
cout<< "Welcome to the class";
}
}; //End of the class Test
int main()
{
Test t1; // Object creation
t1.print();//call to print() function
getch();
return 0;
}
```

Output of the program

Welcome to the class

Accessing Members of a Class

Data members and member functions of a class can be accessed in the **main** program (i.e. **main** function) by using the object of that class, and dot operator (.) followed by the name of the member i.e. object member.

For example, in the above program, the member function **print()** is accessed by using the following statement:

t1.print ();

Similarly a data member can be accessed as object_name data member with private data member and public functions.

The following program implement the class *CRectangle* discussed above.

```
#include <iostream.h>
#include <conio.h>
class CRectangle
{
private:
int X, Y;
public:
void set_values (int a,int b)
{
X=a;
Y=b;
}
int area()
{
cout<< "area of the rectangle="<<(X*Y);
}
}; //End of class
int main()
{
 CRectangle R1; // Object creation
 R1.set_values(44,22);//call to set_values function
 R1.area(); // call to area function
 getch();
 return 0;
 }
```

Output of the program

Area of the rectangle=968

8.1.2 Members of a class

Class has two members; **data members** and **member functions**.

a. Data member

The attributes of a class are called **data members**. Mostly, data members are declared under the **private** access specifier to make them private to the class in which they are used. Data members can also be used under **public** access specifier.
For ech object of a class, a separate copy of the data members is created in memory.

b. Member function

The functions declared or defined inside the body of the class are called **member functions**. These member functions are usually written under the public access specifier to operate on the data members of the class. Member functions can also be written in the *private* area of the class but the practice is to write them in the *public* area.

```
// Example of a class with its members as public
#include<iostream.h>
#include<conio.h>
class Date
{
public:
int Month;
int Day;
int Year;
void SetDate (int nMonth, int nDay, int nYear)
{
Month = nMonth;
Day = nDay;
Year = nYear;
}
void ShowDate()
```

```
{
cout<<"Month of birth:"<<Month<<endl;
cout<<"Day of birth:"<<Day<<endl;
cout<<"Year of birth:"<<Year;
}
};
int main()
{
Date d1; // Object creation
d1.SetDate(01,01,2012);//call to SetDate function
d1.ShowDate(); // call to ShowDate function
getch();
return 0;
}
```

Output of the program

Month of birth: 01
Day of birth: 01
Year of birth: 2012

8.1.3 Access specifiers

Access specifiers are the determiners that determine which member of a class is accessible in which part of the class/program. The most commonly used access specifiers in C++ are:

- private access specifier
- public access specifier

a. private access specifier

private access specifier tells the compiler that the members defined in a class by preceding this specifier are accessible only within the class and its friend function. ***private*** members are not accessible outside the class.

A program explaining the use of *private* data members within and outside the class is given below.

```
#include<iostream.h>
#include<conio.h>
class Date
{
private:
int Day;
int Month;
int Year;
public:
void SetDate(int nDay, int nMonth, int nYear)
{
Day = nDay;
Month=nMonth;
Year = nYear;
cout<< "Today's date is: "<<Day<< "/"<<Month<< "/"<<Year;
}
};
int main()
{
Date cDate;
//cDate.Day = 10;
//cDate.Month = 01;
//cDate.Year = 2012;
cDate.SetDate(15, 01, 2012); //Valid
getch();
return 0;
}
```

Okay: because private members are accessible within the class

Invalid: because private members cannot be accessible from outside the class

b. public access specifier

***public** members* are accessible from anywhere where the object is visible i.e. within the class, in the derived classes and in the **main** function.

Consider the following program to demonstrate the visibility of *public* members of a class.

```
// public access specifier example 1
#include<iostream.h>
#include<conio.h>
class PublicTest
{
public:
int X=10; // Public data member
void showPublic() // Public member function
{
cout<< "X= "<<X;
}
};
int main()
{
PublicTest PT1; //PT1 is an object of class PublicTest
PT1.X; //access private data members
PT1.showPublic(); // access public member function
getch();
return 0;
}
```

In the above program, the class *PublicTest* is defined which comprise of one data member X and one member function *showPublic()* in its *public* part. The data member X

is accessed within the member function *showPublic()* and outside the class, i.e. in *main()* function, without generating any error message. Similarly, the member function *showPublic()* is also accessed in *main()* function.

If no access specifier is mentioned in a class for a member then it will combined as private by default.

The following program demonstrates private and public access specifier.

```
#include<iostream.h>
#include<conio.h>
class Access
{
int A; // private by default
void GetA() // private by default
{
cout<<"I am private member accessible only through public member function"<<endl;
}
public:
int B; // public
void GetB()
{
GetA();
cout<<"I am B in public"<<endl;
}
};
int main()
{
Access cAccess; //cAccess is an object of the class Access
```

```
//cAccess.A = 2; // WRONG because A is private data member
//cAccess.GetA(); //WRONG because GetA() is private member function
cAccess.B=10; // Okay because B is public data member
cAccess.GetB(); // Okay because GetB() is public member function
getch();
return 0;
}
```

The above program explains the scope (or accessibility) of the *private* and *public* data members and member functions in detail. Also, comments are added with each line of code in *main()* program for the explanation of its accessibility level.

The output of the program is:

```
I am private member accessible only through public member function
I am B in public
```

8.1.4 Data hiding

Data Hiding is an important concept of C++ programming language in which the members of a class are protected against illegal access from outside the class. In C language, **struct** is used but it cannot hide data of its data members. Whereas in C++, we got the facility of hiding data using different access levels: private, protected, public in classes.

Private data members and member functions can only be accessed by the members of the same class defining them and cannot be accessed from outside the class. Similarly, **protected** members (data members and member functions) of a class can be accessed only by the class defining them and its derived classes. By using *friend function*, the *private* and *protected* members (data members and member functions) of a class can also be accessed. Except these, *private* and *protected* members are not accessible

anywhere else and thus provide the facility of data hiding. The **public** access specifier has global access and can be accessed within the same class, derived classes and *main()* program.

The following table shows the level of hiding members of a class.

Access	Public	Protected	Private
Access of members (data, functions) in the same class	Yes	Yes	Yes
Access of members (data, functions) in the derived classes	Yes	Yes	No
Access of members (data, functions) from outside the class i.e. from main()	Yes	No	No

Table 8.1: Access Specifiers and Data Hiding

The concept of data hiding, shown in table 8.1, will be demonstrated after the introduction of inheritance.

8.1.5 Constructor and Destructor

Constructors and destructors are special member functions within the class with the same name as that of the class. The following sections discuss them in detail with the help of examples.

a. Constructor

A **constructor** is a special kind of member function that is executed automatically when an object of the class is instantiated. Constructors are typically used to initialize data members of the class to appropriate default values, or to allow the user to easily initialize those member variables to whatever values are desired.

constructors have specific rules for naming:

- Constructors have the same name as that of the class
- Constructors have no return type (not even void)

Consider the following simple program to explain the concept of constructor in a class.

```
#include <iostream.h>
#include <conio.h>
class ConstTest
{
public:
ConstTest () // Constructor
{
cout<< "I am Constructor";
}
};
int main()
{
ConstTest CT; // CT is an object to the class ConstTest
getch();
return 0;
}
```

Output of the program

I am Constructor

In the above example, a class *ConstTest* is defined which have the constructor *ConstTest()* in its *public* part. In *main()* program, there is no explicit call to this constructor and it is automatically called when the object CT of this class is created. It should be noted that *Constructors* are always called at the moment when objects are created. They cannot be called explicitly as member functions are called.

Consider another program to implement the class **CRectangle** including a constructor:

```
#include <iostream.h>
#include <conio.h>
class CRectangle
{
int width, height;
public:
CRectangle (int a, int b) // constructor
{
width = a;
height = b;
}
int area()
{
return (width*height);
}
};
int main()
{
CRectangle r1 (13,14);
CRectangle r2 (15,16);
cout << "Area of Rectangle 1: " << r1.area() << endl;
cout << " Area of Rectangle 2: " << r2.area() << endl;
getch();
return 0;
}
```

Output of the program

Area of Rectangle 1: 182

Area of Rectangle 2: 240

In the above example, the constructor *CRectangle* (*int* a, *int* b) is defined in the class *CRectangle* that initializes the values of width and height with the parameters that are passed to it when objects, r1 and r2, are created with the statement *CRectangle r1 (13,14)* and *CRectangle r1 (13,14)*.

i. Default constructor

If we do not declare any constructor in a class definition, the compiler assumes that a class has a default constructor with no arguments.

```
// default constructor
class DConstructor
{
public:
int a,b,c;
void multiply (int n, int m)
{
a=n;
b=m;
c=a*b;
}
};
```

The compiler assumes that the class *DConstructor* has a default constructor, so the object of this class can be declared by specifying no arguments as given below:

DConstructor DC;

Here, DC is an object to the class *DConstructor* with no arguments. Default constructors are also called *implicit* constructors.

ii. User-defined constructor/ Explicit constructor

When users define constructors in class for their own purpose, especially for initialization of variables, then such types of constructors are called **user defined constructors.** When constructor is declared for a class, the compiler no longer provides an *implicit default* constructor. So, the objects should be declared according to the constructor prototypes that have been defined for the class.

```
// user-defined ( parameterized) constructor example
#include <iostream.h>
#include <conio.h>
class CExample
{
public:
int a,b,c;
CExample (int n, int m)
{
a=n;
b=m;
}
int multiply ()
{
c=a*b;
return c;
}
};
int main()
{
CExample CObj (50,10);
int result=CObj.multiply();
```

```
cout<<"Multiplication Results is: "<<result;
getch();
return 0;
}
```

Output of the program

Multiplication Result is: 500

Here, a constructor *CExample* with two parameters of type *int* has been declared that initializes *private* variables a and b. When the object declaration is done as follows:

CExample CObj (5,10);

The following declaration is not correct, because, we have declared the class to have an *explicit constructor (user-defined constructor)* to which the values should be passed.

CExample CObj;

A user-defined constructor can be either parameterized or non-parameterized. A *parameterized constructor* is the one which accepts arguments while the object is declared and *nonparameterized constructor* is the one whose object declaration is done without specifying arguments. The program given above defines *parameterized constructor*. The following program explains *non-parameterized* constructor.

```
#include <iostream.h>
#include <conio.h>
class CNonParameter
{
public:
CNonParameter ()
{
cout<< "I am non-parameterized constructor";
}
```



```
};
int main()
{
CNonParameter NP;
getch();
return 0;
}
```

Output of the program

I am non-parameterized constructor

In this example, the constructor *CNonParameter*() has no parameters and thus the object creation statement, *CNonParameter NP;*, passes no arguments.

iii. Constructor overloading

Using more than one constructors in the same class is called **constructor overloading.** Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. For an overloaded function, the compiler calls the function whose parameters match the arguments used in the function call. In the case of constructor overloading, that constructor is executed whose parameters match the arguments passed to the object declaration. Consider the following program.

```
// constructors overloading
#include <iostream.h>
#include <conio.h>
class CRectangle
{
int width, length;
```

```
public:
CRectangle()
{
width = 5;
length = 15;
}
CRectangle (int a, int b)
{
width = a;
length = b;
}
int area()
{
return (width*length);
}
};
int main()
{
CRectangle r1 (5,14);
CRectangle r2;
cout << "Area of first rectangle: " << r1.area() << endl;
cout << "Area of second rectangle: " << r2.area() << endl;
getch();
return 0;
}
```

Output of the program

```
Area of first rectangle: 70
Area of second rectangle: 75
```


In the above example, the first object r1 passes the arguments (5,14) to the second parameterized constructor of the class and initializes the variables width with 5 and length with 15 and thus the result calculated is 70. The second object r2 calls the non-constructor and thus the output generated is 75.

b. Destructors

Destructors are special member functions that are executed when an object is destroyed. Destructor fulfills the opposite functionality of constructor and thus de-allocates the memory allocated to an object during its creation. They are called automatically when objects are destroyed.

Like constructors, destructors have the following specific features:

- Destructor has the same name as that of the class, preceded by a tilde symbol ~.
- Destructor cannot take arguments.
- Destructor has no return type.

The second feature given above, implies that only one destructor may exist per class, as there is no way to overload destructors since they cannot be differentiated from each other based on arguments.

Consider the following program to explain the concept of both constructor and destructor.

```
#include <iostream.h>
#include <conio.h>
class A
{
public:
A () //constructor
{
cout<<"I am constructor"<<endl;
```

```
}
~A () // destructor
{
cout<<"I am destructor";
}
};
int main()
{
A a1;
getch();
return 0;
}
```

Output of the program

```
I am constructor
I am destructor
```

In this example, ~A () is a destructor. When the program runs and the object a1 is created, constructor A() is called displaying the message "I am constructor". But, when the control goes out of the program and the object is destroyed, the destructor ~A () is called to de-allocate the memory reserved by the object of the class and a message, "I am destructor", is displayed to verify that the destructor is executed.

8.1. 6 Inheritance and Polymorphism

Object Oriented Programming (OOP) languages have the features of code reusability and polymorphism. Code reusability is the key feature among all other features of OOP which can be achieved by the use of inheritance. Similarly, C++ has the ability to use function name and operator for multiple tasks.

a. Inheritance

A key feature of C++ classes in which new classes are created from existing classes is called inheritance. Inheritance uses the concept of parent and child class. A **Parent Class** is the class from which others classes are derived. It is also called **base class**. A **Sub Class** is a class which inherits features from the base class. A sub-class is also called **child class** or **derived class**. A few examples of inheritance from daily life are given below:

- Consider an apple and a banana. Although an apple and banana are different fruits, both have common feature that they are fruits. Because, apples and bananas are fruits, anything that is true for fruits is also true for apples and bananas. For example, all fruits have a name, a flavor, and are tangible objects. Thus, apples and bananas also have a name, a flavor, and are tangible objects. Apples and bananas inherit these properties from the concept of fruit because they are fruit. Apples and bananas then define some of the properties in different ways as bananas shape is different from apple, also apples have seeds but bananas have no seeds, which make them different from each other.

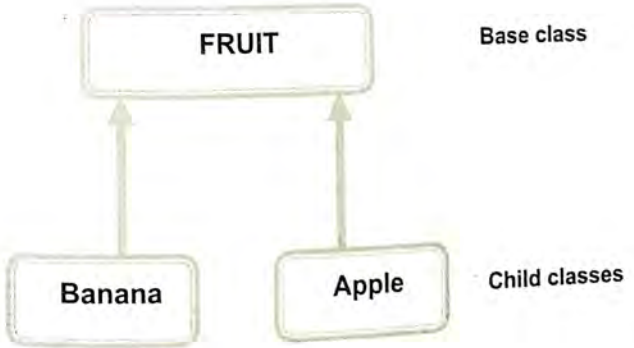


Figure 8.2: Inheritance of Fruit Class

- Here is another example of shape class which has a series of child classes that describe shapes Rectangle and Triangle. They have certain common properties, such as both can be described by means of their sides but they have their own characteristics i.e. triangle has three sides and rectangle has four sides. Square is more special case with four sides of equal length.

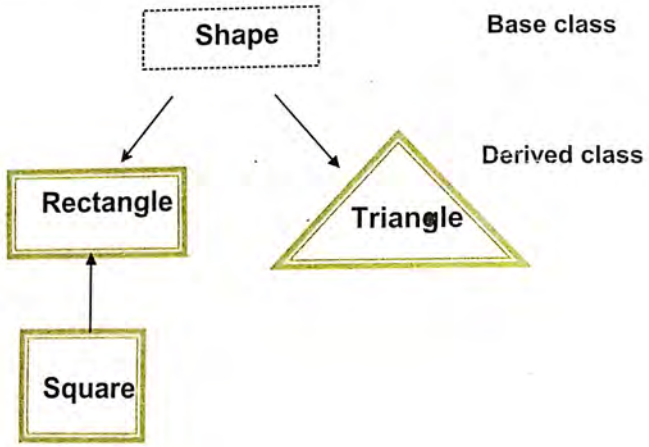


Figure 8.2: Inheritance of Shape Class

- We can also represent patients into indoor and outdoor patients in which both have some common features like name, father name, age, address and disease but indoor patient have 'Ward No' and 'Bed No' as its unique features and the outdoor patient have 'Next date of visit' as its unique feature. It can be represented as follows:

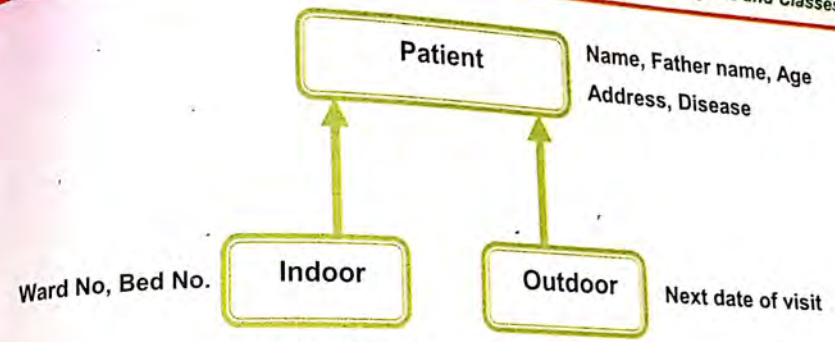


Figure 8.3: Inheritance of Patient Class

In order to derive the child class *Indoor* from the base class *patient*, the following statement is used.

```
class Indoor: public Patient
{
};
```

The **public** access specifier is used to derive a class from its parent class. The keyword **public** may be replaced by any one of the other access specifiers **protected** and **private**.

The following lines of code give the general syntax of using inheritance.

```
class A // base class
{
};
class B: public A //class B is derived from class A
{
};
```

A complete C++ program for the above sketch is given below.

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

```
class A
{
public:
void showa()
{
cout<<"I am in base class A"<<endl;
}
};
class B: public A
{
public:
void showb()
{
cout<<"I am in derived class B"<<endl;
}
};
int main()
{
B b1; // object of the derived class
b1.showa(); // object of B calling function of base class A
b1.showb(); // object of B calling own functiom
getch();
return 0;
}
```

Output of the program

I am in base class A
I am in derived class B

In this example, we have two classes **A** and **B**. Class B is *publically* derived from class A and has therefore right to access the member function of class A. In **main** (), we have created only object **b1** for class B and have called the member functions **showa**() of base class A and **showb**() of derived class B that have produced the output as shown above.

In inheritance, sometimes a class is derived from more than one parent classes and the phenomenon is called **multiple inheritance**. In this case the derived class has properties of base classes as well as its own.

```
class A
{ };
class B
{ };
class c: public A, public B
{ };
```

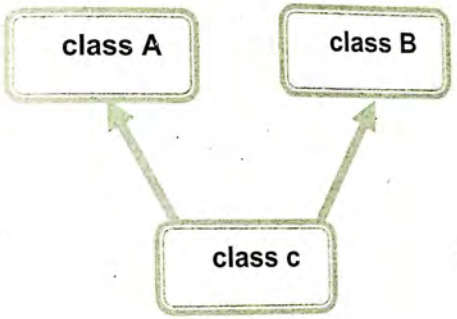


Figure 8.4: Multiple Inheritance

b. Polymorphism

Polymorphism is the ability to use an operator or function in multiple ways. Polymorphism gives different meanings or functionality to the operators or functions. Poly, refers to many, signifies that there are many uses of these operators and functions.

In C++, polymorphism can be achieved by one of the following concepts.

- Function overloading
- Operator overloading
- Virtual functions

Function overloading is the concept of using same function name for related but slightly different purposes in a single program. We have discussed it in detail in unit 6 with the help of programming examples.

Consider the following simple examples to understand the concept of polymorphism with respect to operator overloading.

6 + 10;

The above statement refers to integer addition with the help of + operator. The same + operator can also be used for addition of two floating point numbers or two strings (concatenation) as shown below.

7.15 + 3.78
"Computer" + "Training"

Polymorphism is a powerful feature of every Object Oriented Programming (OOP) language, especially of C++. A single operator '+' behaves differently in different contexts such as *integer* and *float* addition and *strings* concatenation. This concept is known as **operator overloading.**

A **virtual function** or virtual method is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature. This concept is a very important part of the polymorphism portion of object-oriented programming (OOP).

Summary

- A class consists of attributes called data members and function called member function.
- A variable of type class is called object. In C++, when variables of type class are declared they create objects.
- The attributes of a real world objects are called data members.
- The functions declared or defined inside the body of the class are called member functions.
- Access specifiers determine that which member of a class is accessible by which member of the class/program.
- private access specifier tells the compiler that the members defined in a class by preceding this specifier are accessible only within the class and its friend function.
- public members are accessible from anywhere where the object is visible.
- Data Hiding is the concept in which the members of a class are protected against illegal access from outside the class.
- A constructor is a special kind of class member function that is executed when an object of the class is instantiated.
- Destructors are special member functions having the same name as that of the class with a tiled symbol ~ and executed when an object is destroyed.
- To access members of a class, dot operator (.) is used with the member name.
- The phenomenon of creating new classes from existing classes is called inheritance.
- Polymorphism is the ability to use an operator or function in multiple ways.

Exercise

Q.1 Fill in the Blanks.

i. _____ is a member of function that is called automatically when an object is created..

ii. The ability to reuse code already defined for the new purpose, is referred to as_____.

iii. In OOP the main advantage of inheritance include_____.

iv. An object of a class can be defined as " a variable of type _____"

v. Function overloading is used to achieve _____.

Q.2 Select the correct choice for the following Multiple Choice Questions.

i. A constructor is called whenever_____.
   a. An object is destroyed
   b. An object is created
   c. A class is declared
   d. A class is used

ii. Destructor is used for _____.
   a. Initializing the values of data members in an object
   b. Initializing arrays
   c. Freeing memory allocated to the object of the class when it was created
   d. Creating an object

iii. The name of destructor is always preceded by the symbol _____
   a. +
   b. %
   c. -
   d. ~

iv. Constructors are usually used for _____.
   a. Constructing programs
   b. Running classes
   c. Initializing objects
   d. All of the above

v. Inheritance is used to_____.
   a. Increase the size of a program
   b. Make the program simpler
   c. Provide the facility of code reusability
   d. Provide the facility of data hiding

Q.3 Write TRUE/FALSE against the following statements.

i. C++ is a pure object-oriented programming language.

ii. One of the main features of object oriented programming language is data hiding.

iii. If you do not declare a constructor in a class, the compiler will furnish a default constructor for you automatically.

iv. A constructor is a function that is called when an instance of a class is deleted.

v. Function overloading is an example of polymorphism.

Q.4 Write a C++ program implementing a class with the name **Circle** having two functions with the names: **GetRadius** and **CalArea.** The functions should get the value for the radius from the user and then calculate the area of the circle and display the results.

Q.5 Write a C++ program implementing inheritance between **Employee** (base class) and **Manager** (derived class).

Q.6 Diagrammatically represent the following scenario of inheritance:

- We have **Shape** as a base class and **Circle, Rectangle, Triangle** and **Square** as its derived classes.

Q.7 Write a C++ program implementing a class with the name **ConstDest** having constructor and destructor functions in its body.

Q.8 Write a C++ program implementing a class with the name **Time.** This class should have a constructor to initialize the time, **hours, minutes** and **seconds**. The class should have another function named **ToSecond()** to convert and display the time into seconds.

Q.9 What is object? How objects are created to access members of a class?

Q.10 Explain inheritance and polymorphism with examples.

UNIT 9

FILE HANDLING

After the completion of Unit-9, Students will be able to:

- Know the binary and text file
- Open the file in different modes
- Know the concept of
  - bof()
  - eof()
- Define stream
- Use the following stream
  - Single character
  - String

INTRODUCTION

A **file** is a collection of related records that are permanently stored in secondary storage. File handling is an important feature of C++ language. This unit describes different types of files such as text files and binary files and describes how different operations like opening, reading, writing and closing are performed on files.

9.1 File handling

The combinations of characters, words and records are called **files.** The operations on these files are the major focus point of file handling. Suppose we have a file on the disk and want to open it then reading from or writing into the file before closing it will be termed as **handling files**. The basic steps involved in file handling are:

- Opening file
- Reading and writing a file
- Closing file

9.1.1 Types of files

C++ divides files into two different types based on how they store data. These are:

- Text files
- Binary files

Text files are those files that store data in text format which is readable by humans. Example of text file is the C++ source program which can be read by human.

Binary files store data in binary format which is not readable by the humans but readable by the computers. Binary files are directly processed by the computers. Binary files are normally used to store large data files. They take less space to store data as compared to text files. For example, an integer having length of six digits will take six bytes to accommodate in a text file while in binary files they will take only two bytes. The best example of binary file is the *object file* of a C++ source file that is generated by the compiler of C++.

9.1.2 Opening file

To open a file, the function **open ()** is used whose syntax is:

myFile.open(filename);

Here, **myFile** is an internal variable, actually an object, used to handle the file whose name is written in the parenthesis. To declare this variable the following statement is used:

Ifstream myFile;

The dot . operator is used between the variable *myFile* and *open* function. *myFile* is an object of *ifstream* while *open*() is a function of *ifstream*. The argument for *open* function is the name of the file on the disk which should be enclosed in double quotes. The file name can be simple file name like "sale.txt". It can also fully qualify path name like "C:\myprogs\sale.txt".

a. Modes of opening a file

While opening a file, we tell the compiler what we want to do with it i.e. we want to read the file or write into the file or want to modify it. In order to open a file in any desired mode the member function **open** () should take mode as an argument along with the file name. Its general syntax is shown below:

open (filename, mode);

Here, file name representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
ios::trunk	If the file opened for output operations already exists then its previous contents are deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR |. For example, if we want to open the file **test.bin** in binary mode to add data we could do it by the following call to member function **open**().

```
Ofstream myfile;
myfile.open ("test.bin", ios::out | ios::app | ios::binary);
```

Each one of the **open**() member functions of the classes *of stream*, *ifstream* and *fstream* has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

For *ifstream* and *ofstream* classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open() member function.

Let us see an example program that creates a text file "**example1.txt**" and writes a line of text in it.

```
//opening a file and writing into it
#include <iostream.h>
#include <fstream.h>
#include <conio.h>
int main()
{
Ofstream myfile;
```



```
myfile.open ("example1.txt");
myfile<< "This is a program that tells you how to write to a file.\n";
myfile.close();
getch();
return 0;
}
```

This code creates a file called *example1.txt* and inserts a sentence into it in the same way as we do with *cout*, but using the file stream *myfile* instead. Figure 9.1 shows the output of the above program in the form of a text file that is created in the same working directory.

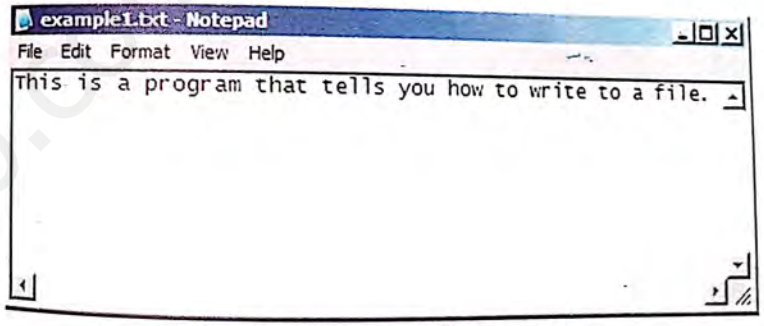


Figure 9.1: Opening and Writing into a Text File

b. Reading input file

To read a word from the file we can write as:

```
myFile>> c;
```

So, the first word of the file will be read in c, where c is a character array. It is similar as we used with **cin**. There are certain limitations to this. It can read just one word at one

time. It means, on encountering a space, it will stop reading further. Therefore, we have to use it repeatedly to read the complete file. We can also read multiple words at a time as:

myFile>> c1 >> c2 >> c3;

The first word will be read in c1, second in c2 and third in c3. Before reading the file, we should know some information regarding the structure of the file. If we have a file of an employee, we should know that the first word is employee's name, second word is salary etc,so that we can read the first word in a **string** and second word in an **int** variable.

Consider the following input file "*inputfile.txt*", shown in figure 9.2, which is read and displayed on the screen.

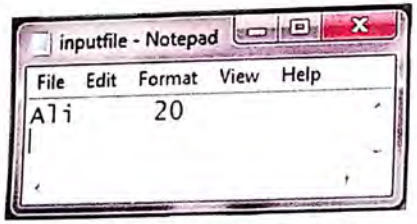


Figure 9.2: Input File to be Read

```
//reading input file program
#include <iostream.h>
#include <fstream.h>
#include <conio.h>
int main()
{
Ifstream myfile("c:\\inputfile.txt");
Char ch [20];
int m;
myfile>>ch>>m;
cout<<ch<<"\t"<<m;
myfile.close();
getch();
return 0;
}
```

The output of the above program is shown in figure 9.3:

Figure 9.3: Output of Reading Program

Let us write another simple program, to read from a file '**myfile.txt**' that is in the current directory, and print it on the screen. "myfile.txt" contains employee's name, salary and department in which they are employed. Figure 9.4 shows **myfile.txt** which is followed by the complete program to describe how it is opened, read and closed.

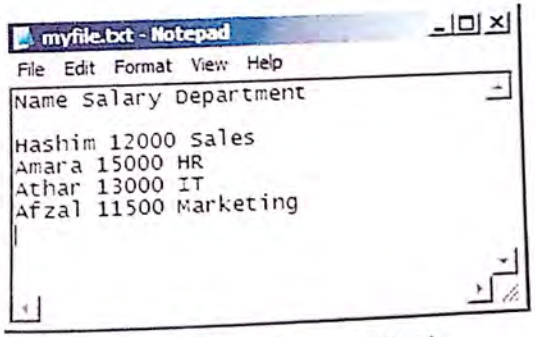


Figure 9.4: Input File to be Read

```
#include <iostream.h>
#include <fstream.h>
#include <conio.h>
main()
{
Char name[50]; // used to read name of employee from file
Char sal[10]; // used to read salary of employee from file
Char dept[30]; // used to read dept of employee from file
Ifstream inFile; // Handle for the input file
Charin putFileName[] = "myfile.txt"; // file name, this file is in the current directory
inFile.open(inputFileName); // Opening the file
// checking that file is successfully opened or not
if (!inFile)
{
cout<< "Can't open input file named " <<inputFileName<<endl;
exit(1);
}
// Reading the complete file word by word and printing on screen
while (!inFile.eof())
{
inFile>> name >>sal>>dept;
cout<< name << "\t" <<sal<< " \t" <<dept<<endl;
}
inFile.close();
getch();
return 0;
}
```

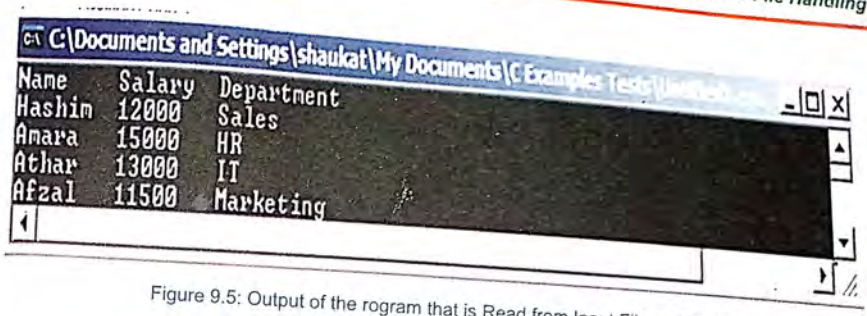



Figure 9.5: Output of the rogram that is Read from Input File myFile.txt

c. Closing file

Once we have read the file, it must be closed. It is the responsibility of the programmer to close the file. We can close the file by using the following statement:

myFile.close();

The function **close**() does not require any argument, as we are going to close the file associated with **myFile**. Once we close the file, no file will be associated with **myfile**.

d. Opening files in binary mode

To open a file in binary mode, we need to set the file mode to **ios::binary**. Suppose we have a binary file named as "**test.dat**". To open this file in binary mode, we write the statement as:

ofstream myFile;
myFile.open ("test.dat", ios::binary);

In order to write the data to a binary file, "**write**" method is used. This method is a member function of **ofstream** or **fstream** class.

9.1.3 bof() and eof()

C++ provides special functions **bof()** and **eof()** that are used to set the pointers to the beginning of a file and at the end of a file respectively. The following sections explain them with the help of proper examples.

a. bof() – beginning-of-file

The **bof()** is a pointer which returns true if the current position of the pointer is at the beginning of the input file stream, and false otherwise. It means that it tells the compiler whether the cursor is at the beginning of file or not.

myFile.close();

b. eof() – end-of-file

The **eof()** is a pointer which returns true when there are no more data to be read from an input file stream, and false otherwise. It means that this function checks whether control has reached to the end of file or not. This function is very useful in the case when we do not know the exact number of records in a file.

Rules for using eof( )

- Always test for the end-of-file condition before processing data.
- Use a **while** loop for getting data from an input file stream. A **for** loop is desirable only when you know the exact number of data items in the file

```
// reading a text file using eof() function
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
int main()
{
char ch [50];
ifstream flag ("c:\\TestRecord.txt");
```



```
cout<<"Output is"<<endl;
while(!flag.eof())
{
flag>>ch;
cout<<ch<<endl;
}
flag.close();
getch();
return 0;
}
```

Output of the program

```
Output is
Ali      20
Anwar 15
Zainab 17
Zonash21
```

In the above program, the input file *TestRecord* is loaded and the *eof ()* function detects the end of file. The program reads the file record by record into the string variable *ch* which is then displayed on the screen.

9.1.4 Stream

A **stream** can be thought of as a sequence of bytes of infinite length that is used as a buffer to hold data to be processed. In C++ a **stream** is a sequence of bytes associated with a file. Most of the times streams are used to assure a good and secure flow of data between an application and file.

In C++ there are two types of streams, ***input stream*** and ***output stream***.

Input streams take any sequence of bytes from an input device such as a keyboard, a file, or a network, while **output streams** are used to hold output for a particular data consumer, such as a monitor, a file, or a printer. When writing data to an output device, the device may not be ready to accept that data e.g. the printer may still be warming up when the program writes data to its output stream. The data will reside in the output stream until the printer starts consuming it.

The act of taking data from the input devices is called *extraction* and the operator used for this purpose is called *stream extraction operator* >>. Similarly, the act of displaying data on the screen is called *insertion* and the operator used for this purpose is called *stream insertion operator* <<.

Sometimes, both operations **input/output** may need to be used at the same time. **Standard streams** in C++ consist of four predefined standard stream objects. These are: **cin, cout, cerr** and **clog.** C++ language handles files using these streams objects. For this purpose, the header file <**fstream.h**> is needed to be included.

If the file is only used for reading purpose then the header file **ifstream** (input file stream) is needed to be included. Similarly, if one wants to write in some file then header file **ofstream** (output file stream) is needed. One can read, write and manipulate the same file by using the header file **fstream**.

Consider the following example of input and output using the standard streams:

```
// input/output stream example
#include <conio.h>
#include <iostream.h>
#include <process.h>
int main()
{
float height;
cout<< "Enter your height: " <<endl; // output stream
cin>> height; // input stream
if (height <= 0)
{
cerr<< "You entered an invalid height!" <<endl;
exit(1);
}
cout<< "Your height is " << height << " feet" <<endl;
getch();
return 0;
}
```

Output of the program

```
Enter your height:5.8
Your height is 5.8 feet
```


9.1.5 Use of the Streams

The data can be read from and written to files with the help of single character stream and string stream.

a. Single character stream

Using single character stream, the data can be read from and written to files character by character.

i. Reading files character by character

The function *get()* is used to read data character by character from files.

Consider the following program that reads the data one character at a time from the file "*characters file.txt*",shown in figure 9.7, and display them on the screen.

```
//character stream (read) program
#include <conio.h>
#include <iostream.h>
```

```
#include <fstream.h>
int main()
{
char ch;
ifstream reads("c:\\charactersfile.txt");
while(!reads.eof())
{
read.get(ch);
cout<<ch<<endl;
}
read.close();
getch();
return 0;
}
```

Output of the program

```
r
e
a
d

t
h
e

f
i
l
e

p
l
e
a
s
e
.
```

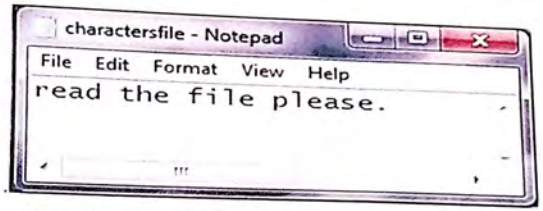



Figure 9.7: Input File to be Read Character by Character

ii. Writing files character by character

Similarly, using single character stream, data can be written to files character by character by using the function *put ()*. Consider the following program that writes data character by character to an empty file "*characterswrite.txt*".

```
//character stream ( write) program
#include <conio.h>
#include <iostream.h>
#include <fstream.h>
int main()
{
char ch;
ofstream writes("d:\\characterswrite.txt");
for (int i=0; i<=20; ++i)
{
cin>>ch;
cout<<writes.put(ch);
}
writes.close();
getch();
return 0;
}
```

When the above program is executed, characters are entered one by one from the keyboard.The characters entered above are written to the "*characterswrite*" file stored at secondary storage at location *d-drive* of the hard disk. The output file generated is shown in figure 9.8.

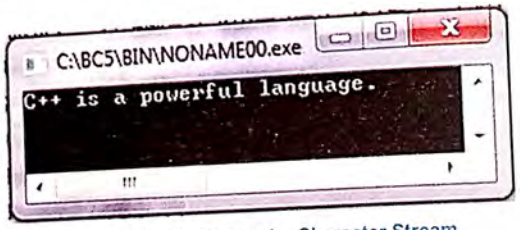


Figure 9.8: File Written by Character Stream

b. String stream

A **string stream** is a stream which reads input from or writes output to an associated string.Consider the text file "*stringread*" shown in figure 9.9.

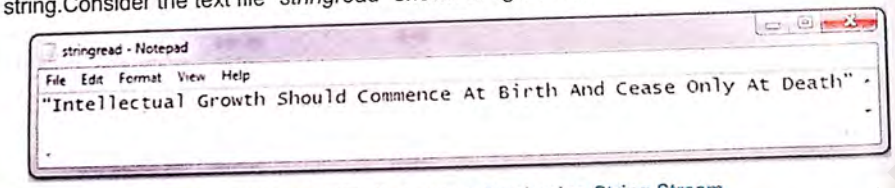


Figure 9.9: Input File that will be Read using String Stream

The following program reads this file (*stringread.txt*) into string (*str*) using *getline()* function and display the result on the screen as shown in figure 9.10.

```
//string stream (read) program
#include <conio.h>
#include <iostream.h>
#include <fstream.h>
#include <string>
int main()
{
char str[20];
```



```
ifstream reads("d:\\stringread.txt");
while(!reads.eof())
{
reads.getline(str,21);
cout<<str<<endl;
}
reads.close();
getch();
return 0;
}
```

Figure 9.12: File Read by String Stream

Summary

- The combinations of characters, words and records is called files.
- The process of opening, reading from and writing into files is termed as handling files.
- There are two types of files: text files and binary files.
- Those files that store data in text format and are readable by human are called text file.
- Binary files are those files that store data in binary format which is not readable by the human but readable by the computers. Binary files are directly processed by the computers
- To open a file, the function open() is used.
- There are different modes of opening files i.e. binary mode, input mode and output mode.
- The bof( ) is a pointer which is true if the current position of the pointer is at the beginning of the input file stream, and false otherwise.
- The eof( ) is a pointer which returns true when there are no more data to be read from an input file stream, and false otherwise.
- A stream can be thought of a sequence of bytes of varying length that is used as a buffer to hold data that is waiting to be processed

Exercise

Q.1 Fill in the Blanks.

i. In C++ language, for handling of files, the header file _____ is used.
ii. The function _____ is used for closing a function.
iii. In the statement, open ("xyz.txt", ios::out); _____ file is opened in output mode.
iv. If all output operations are to be performed at the end of the file then _____should be used in the open function.
v. A _____is "a continuous flow or succession of bytes a stream of characters."

Q.2 Select the correct choice for the following Multiple Choice Questions.

i. eof stands for _____.
   a. errors-on-files
   b. end-of-file
   c. exit-of-file
   d. none of the above
ii. In file handling open() function is used _____.
   a. to open C++ compiler
   b. to open a file
   c. both a and b
   d. none of the above
iii. Default mode parameter for ofstream is_____.
   a. ios::out
   b. ios::in
   c. ios::binary
   d. both a and b

iv. A text file has the extension_____.
   a. .doc
   b. .docx
   c. .txt
   d. .bin

v. ios::binary is used as an argument to open() function for_____.
   a. opening file for input operation.
   b. opening file for output operation.
   c. opening file in binary mode.
   d. none of the above.

Q.3 Write TRUE/FALSE against the following statements.

i. A string stream is a stream which reads input from or writes output to an associated string.
ii. The eof( )is a pointer which returns false when there are no more data to be read from an input file stream, and true otherwise.
iii. To open a file in binary mode, we need to set the file mode to ios::binary.
iv. close() Function is used in C++ file handling for opening a new file.
v. Those files that store data in text format and are readable by human are called text file

Q.4 Define file. Describe different types of file.

Q.5 Describe different types of operations performed on the files.

Q.6 Define streams. Describe input and output streams in detail.

Q.7 What is meant by the term mode of file opening? Describe different modes of opening file.

Q.8 What is file handling? Explain it in detail.

Bibliography

1. *Basic Computer: Operating System Functions.* [Cited 2011 Jun 10]; Available from: http://www .comptechdoc.org/basic/basictutl.
2. Stalling, W., *Operating Systems: Internals and Design Principles.* Sixth ed. 2009: Macmillan, New York.
3. *Operating System.* Webopedia (N.D) [cited; Available from: http://www .webopedia.comrERMIO/{}perating system html.
4. Silberschatz, Galvin, and Gagne, *Operating System Concepts.* Seventh ed: Addison-Wesley Inc.
5. Ritchie, C., *Operating System: Incorporating Unix and MS-DOS.* 2nd ed. 1997: BPB Publication, Delhi. 232.
6. *Common Operating Systems: Microsoft DOS.* (N.D) [cited 2011 May, 20th] Available from: http://www.wiley.com/college/busin/cmis/oakroan/outjine/chap05/misc/dos.htm.
7. *DOS (Disk Operating System).* 2004-04-03 [SB] Last Updated: 2010-05-14 [cited 2011 April, 20th]; Available from: http://www.operating- system.org/betriebssystem/ englishbs-msdos.htm.
8. *Unix Tutorials: Introduction to the UNIX Operating System* 19 October 2001 [cited 2011 April 17]; Available from: http://www.ee.surrey.ac.ukffeaching/Unixlunixintro.html.
9. *History of Mac OS,* in *Wikipedia.* 2008.
10. *Common Operating Systems: Macintosh System 7.5 (Apple).* [cited 2011 March 28]; Available from: http://www.wiley.com/college/businlcmis/oakman/outline/chap05/misc/mac.htm.
11. *Harvey M. Deitel, Paul J. Deitel, and D.R. Choffues, Operating Systems (3rd Edition). 3rd ed. 2004.*
12. *Multi Programming Operating System.* June 2, 2011 [cited 2011 Jun 10]; Available from: http //easy2te.ach. netloperating-.systemlmulti-programming-operating-system.
13. Stankovic, I.A., Real-Time Computing. 1992, Department of Computer Science, University ofMassacusetts.
14. Fry, G. *Real Time Systems.* [cited 2011 May 10]; Available from: http://cs-people. bu.eduJ gfry/realtime.html.
15. *Physical Oceanographic Real-Time System (PORTS®).* [Cited 2011 April 8];

Available from: http://tidesandcurrents.noaa.gov/ports.html.
16. Tanenbaum, A.S., *Modern Operating Systems.* 3rd ed. 2008 Prentice-Hall.
17. Blaise Barney and L. Livermore, *Introduction to Parallel Computing,* National Laboratory.
18. Alecu, F., *Operating System for Parallel Processing.* Studies in Informatics and Control, 2004. 13(2).
19. Tannenbaum and M.v. Steen, *Distributed Systems.* 2001: Prentice Hall.
20. *Operating Systems Services and Functions.* UzEE November 4, 2007 [cited 2011 March 2]; Available from: http://uzeeinc.wordpress.coml2007111/04/operating-systems-services-and- functions/
21. *Muhammad, R..B. Operating Systems Lecture Notes: System Components. [Cited 2011 Jun 01]; Available from:* http://www.personal.kent.edu/~rmuhamma/OpSystems/os.htmL
22. *What Does an Operating System Do?* 2009 [cited 2011 May 7]; Available from: http://computerspot.netlwhat -does-an-operating-system-do/.
23. *File management system,* in *Webopedia,* Webopedia.
24. Bhat, P.C.P. *Operating Systems/Input Output (10) management Lecture Notes* Volume,
25. Dean, T., Comptia Network+ 2009 in Depth. 2009. 767.
26. *Operating System Notes: Threads.* [Cited 2011 May 21]; Available from: http://www.personal.kent.edu/~rrnuhamma/OpSystemslMyos/threads.htm.
27. Joydip Kanjilal, Operating Systems: Coneeptsand Terminologies, 31 Jul 2006, [cited 12, Dee 2011]
28. Aggarwal, K.K., *Software Engineering:* Revised Edition ed. 2001: New age international publisher. 494.
29. Goldsmith, R.F. *Software development life cycle phases, iterations, explained step by step.* Available from: www.SearchSoftwareQuality.com.
30. Gary B. Shelly, Thomas J. Cashman, and H.J. Rosenblatt, *System Analysis and Design,* Eight ed. 2009.
31. Ankur Patel, Sagar Padaliya, and D. Pande. *SDLC Methodologies.* [cited 2011 Jun 12]; Available from: http://community.mis.temple.edu/fox20/about/.
32. bijayani. *SDLC.* 2010 [cited 2011 Jun, 11]; Available from: http://bijayani.wordpress.com/2010/02115/sdlc/.
33. Pressman, R., *Software Engineering: A Practitioner's Approach.* seventh ed. 2010: McGraw-Hill Higher Education.

34. Hemphill, M. *MIS for digital age: Systems Development Lifecycle.* February 4, 2004 [cited 2011 Jun, 02]; Available from: http://exonous.typepad.comimis/2004/02/systems develop.html.
35. Sommerville, *Software Enginering.* Eight ed. 2008: Addision Wesely Publisher Limited 864.
36. Wasson, C.S., *System analysis, design, and development: concepts, principles and Practices.1005* John Wiley & Sons Inc. 818.
37. Glossary of Computer Systems Software Development Terminology *(8/95)*
38. "What is C++." [cited May 28, 2011]; 2005, Available from: http://www.hitmill.comiprogrammingl cpp/whatiscpp .html.
39. Stair, and Ralph M t et al., *Principles of Information Systems.* Sixth ed.: Thomson Learning, Inc, 2003.
40. *Program.* [cited May 28, 2011]; Available from: http://www.webopedia.com/TERMIP/program.html.
41. *"C++ : Documentation: C++ Language Tutorial." [Cited May 2. 2011J: Available from:* http://www.cpluspius.com.
42. Alex, "The C++ Tutorial.", [cited April, 2011]; available from: http://www.leamcpp.com/. *2007.*
43. "Types of Constants in C++." [cited May 19, 2011]; Available from: http://www.adiwebs.com/types-of-constants-in-cl.
44. Sripriya, R., "C++ Manipulators." 2007. [last updated lith Nov 2008]. [cited 2011 29th May]; Available from: http://www exforsvs.com/tutorials/c-plus-plus/c- manipulators.html.
45. "C++ lessons and Topics: FunctionX Tutorial", available from: http://www.functionx.wm/, Last updated: Monday, June I3, 201 L
46. Robert Lafore, "Object-Oriented Programming in C++", 4th Edition, Dee 29, 2001.
47. Nicolai M. Josuttis, "Object-Oriented Programming in C++", December 30, 2002.
48. Paul Deitel and Harvey M. Deitel, "C++ How to Program", 7th Edition. Aug 16, 2009.
49. Paul Deitel and Harvey Deitel, "C++ How to Program (8th Edition)". Mar 25, 2011.
50. "String in C++", [cited 04, Jan 2012], available from: http://anaturb.netiC/stringexapm.htm
51. Bjame Stroustrup's C++ Glossary, Modified, February 19, 2007, available from: http://www?.research.att.coml~bs/ glossarv.html

Answers

Unit 1

Q.1 Fill in the Blanks.

i	ii	iii	iv	v
Multiprocessing	Multiple Tasks	Multiprogramming	Exit	UNIX

Q.2 Multiple Choice Questions.

i	ii	iii	iv	v	vi	vii	viii	ix	x
b	c	c	a	b	b	d	a	b	a

Q.3 TRUE/FALSE statements.

i	ii	iii	iv	v	vi	vii	viii	ix	x
T	T	T	T	F	F	F	F	F	T

Unit 2

Q.1 Fill in the Blanks.

i	ii	iii	iv	v
Functional	Deployment	System Analyst	Programmer	Project management

Q.2 Multiple Choice Questions.

i	ii	iii	iv	v	vi	vii
c	d	a	a	b	a	d

Q.3 TRUE/FALSE statements.

i	ii	iii	iv	v
T	F	T	F	T

Unit 3

Q.1 Fill in the Blanks.

i	ii	iii	iv	v
0x	Conditional operator	True/False	Initialization	main( )

Q.2 Multiple Choice Questions.

i	ii	iii	iv	v
b	a	a	a	c

Q.3 TRUE/FALSE statements.

i	ii	iii	iv	v
F	F	F	T	F

Unit 4

Q.1 Fill in the Blanks.

i	ii	iii	iv	v
Enclosed within a block	The value returned by a conditional expression	Default	Exit	iteration

Q.2 Multiple Choice Questions.

i	ii	iii	iv	v	vi	vii	viii	ix	x
c	b	e	b	d	d	c	a	c	c

Q.3 TRUE/FALSE statements.

i	ii	iii	iv	v
F	T	T	T	F

Unit 5

Q.1 Fill in the Blanks.

i	ii	iii	iv	v
0	10	Multidimensional	String	float

Q.2 Multiple Choice Questions.

i	ii	iii	iv	v
b	a	d	c	c

Q.3 TRUE/FALSE statements.

i	ii	iii	iv	v
F	T	F	T	F

Unit 6

Q.1 Fill in the Blanks.

i	ii	iii	iv	v
One	Semicolon (;)	Throughout the program	Function overloading	Void

Q.2 Multiple Choice Questions.

i	ii	iii	iv	v
c	c	a	a	d

Q.3 TRUE/FALSE statements.

i	ii	iii	iv	v
T	F	F	F	T

Unit 7

Q.1 Fill in the Blanks.

i	ii	iii	iv	v
Address	int	Initialization	Reference	Dereference

Q.2 Multiple Choice Questions.

i	ii	iii	iv	v
a	c	a	c	b

Q.3 TRUE/FALSE statements.

i	ii	iii	iv	v
F	F	T	T	T

Unit 8

Q.1 Fill in the Blanks.

i	ii	iii	iv	v
Constructor	Inheritance	Code reusability	Class	Polymorphism

Q.2 Multiple Choice Questions.

i	ii	iii	iv	v
b	c	d	c	c

Q.3 TRUE/FALSE statements.

i	ii	iii	iv	v
F	T	T	F	T

Unit 9

Q.1 Fill in the Blanks.

i	ii	iii	iv	v
<fstream.h>	close	xyz.txt	Ios::app	stream

Q.2 Multiple Choice Questions.

i	ii	iii	iv	v
b	b	a	c	c

Q.3 TRUE/FALSE statements.

i	ii	iii	iv	v
T	F	T	F	T

Glossary

Actual parameters
Actual parameters are those parameters which appear in function calls.

Address
A memory location within computer that is used to store value.

Algorithm
A finite set of well-defined rules for the solution of a problem in a finite number of steps.

Allocate
To assign a resource to a process for performing a specific task.

Alphanumeric
A character set that contains letters, digits, and usually other characters such as punctuation marks.

Analysis phase
In this phase, the in-charge of the project team must decide if the project should go ahead with the available resources or not.

Analyst
Analyst is a profession al in the field of software development that studies the problems, plans solutions for them, recommends software systems, and coordinates development to meet business or other requirements.

Argument
A value passed to a function from the calling program, "main()" function.

Argument passing
The process of passing values to the formal parameters mentioned in the function declator.

Array
An n-dimensional ordered set of data items identified by a single name and one or more indices, so that each element of the set is individually addressable; e.g., a matrix, or table.

Assignment operator
The operator denoted by the symbol (=) which is used to assign values to variables.

Auxiliary storage
Storage devices other than main memory e.g., disks and tapes.

Base class
A class from which other classes are derived. Also called parent class or super class.

Batch
A group of records or data processing jobs brought together for processing or transmission.

Batch processing
A feature of operating system in which more than one records (a batch) are accumulated and processed after a regular interval of time.

Batch processing system
It is that type of operating system which collects jobs in batches before being processed by the CPU.

Binary file
Binary files store data in binary format which is not readable by the human being but readable by the computers.

Binary operator
An operator that operates on two operands at a time, such as +, /, &&, etc.

Black box testing
It is that type of testing which tests all possible combinations of end-user actions. It focuses on validation tests, boundary conditions, destructive testing, and security-related testing that helps to identify the vagueness and contradiction in functional specifications.

Block
A group of statements enclosed in curly braces, {}.

Block comment
These types of comments are represented by the symbols /* ..... */

Blocked (or waiting) State
A process is said to be in blocked state if it is waiting for some event to happen such as an Input/output completion before it can proceed.

bof()
The bof( ) is a pointer which returns TRUE if the current position of the pointer is at the start of the input file stream, and FALSE otherwise.

Bool
The built-in data type that can have either true or false value.

Break statement
It is used to make jump from one part of a program to another.

Bug
A fault in a program which causes the program to perform in an unintended or unanticipated manner.

Built-in functions
These are pre-compiled functions for some commonly used operations.

Built-in type
A data type directly provided by C++, such as int, double, and char.

Byte
A unit of memory that can hold a character of the C++ representation character set. The size of a byte is equal to 8 bits.

C++
An object-oriented high-level programming language.

C++ statement
A C++ statement is a simple or compound expression that can actually produce some effect.

Call-by-reference
Calling a function in such a way that to pass a reference to the function rather than a value

Call-by-value
A type of method used to pass arguments to a function in which a copy of the arguments is passed.

Char
A built-in data type used to declare variables of character type to store character constants.

cin
Stands for console input. It is used to get values form the users at run time.

Class
A feature of object oriented programming languages used to divide large size programs into smaller units which have their own data members and member functions.

Coding
The transforming of logic and data from design specifications (design descriptions) into a programming language.

Comment
The explanation added with statements. These are block comment /* ... */ and line comment //.

Compilation
Translating a program expressed in source language into object code.

Compiler
A computer program that translates programs expressed in a high-level language into their machine language equivalents.

Compound assignment operators
These operators are (+=, -=, *=, /=, %=).

Compound expression
Compound expression is an expression that involves more than one operator.

Compound statement
Sequence of statements enclosed in curly braces { ... }.

Computer language
A language designed to enable humans to communicate with computers e.g. C, C++, Java etc.

Computer program
A set of computer instructions written in any computer language that perform a specific task is called computer program.

Concatenation of strings
Joining two strings into a single string is called string concatenation.

Const
A key word used to declare constant identifier whose values cannot be changed during program execution.

Constant
Those objects in C++ whose values cannot be changed during program execution i.e. integer constant, character constants and string constant.

Constructor
A constructor is a special kind of class member function that is executed when an object of the class is instantiated.

Constructor
A special member function with the same name as its class and used to initialize objects of its class.

Constructor overloading
Using more than one constructors in the same class is called constructor overloading.

Context Switch
This refers to switching the CPU from one process to another by saving the state of the old process in the stack and executing the new process.

Continue statement
The continue statement provides a convenient way to jump back to the top of a loop earlier than normal, which can be used to bypass the remainder of the loop for an iteration.

Cout
A C++ object used to print values and strings on the screen.

Customer
Customer is an individual or an organization that is a current or potential buyer or user of the software product.

Data hiding
Data hiding is an important concept of C++ programming language in which the members of a class are protected against illegal access from outside the class.

Data member
Member of a class that can hold a value. Usually data members are called variables. A data member can be a static member or a non-static member.

Data type
Data type is the property of variables which tells the compiler that what type of data the variables will store.

Debugging
Determining the exact nature and location of a program errors, and fixing them.

Decision statement
It is a statement that causes the program to change the path of execution, sequence of execution, based on the value of an expression.

Declaration
An introduction of a name of a variable or function into a scope by specifying its type.

Default argument
A value specified for an argument in a function declaration, to be used if a call of the function doesn't specify a value for that argument.

Default constructor
Constructor requiring no arguments. Used for default initialization.

Derived class
A class that inherits features from one or more base classes.

Design
The process of defining the architecture, components, interfaces, and other characteristics of a system or component.

Design phase
That phase of software development life cycle during which the designs for architecture, software components, interfaces, and data are created, documented, and verified to satisfy requirements.

Desktop
Desktop is the essential component of a graphical user interface of an operating that is used for the quick and easy access of data and other computer resources.

Destructor
Member of a class used to clean up before deleting an object. Its name is its class' name prefixed by '~'.

Developer
A person, or group, that designs and/or builds ocuments configures the hardware software of computerized systems.

Device manager
The module of operating system used to manage input/output devices is called Input/output manager or device manager.

Disk
A storage device that includes one or more flat, circular plates with magnetic or optical surfaces on which information is stored.

Dispatch
The process of allocating CPU to a ready job, waiting in ready queue, is called dispatching.

Distributed operating
A distributed operating system is an operating system that manages a group of independent computers and makes them appear to the users as a single computer.

Documentation
The aids provided for the understanding of the structure and intended uses of an information system or its components, such as flowcharts, textual material, and user manuals.

DOS
DOS is the abbreviation for Disk Operating System which was the first operating system used for personal computers.

Double
Double-precision floating-point number.

Driver
Drivers are the software which are placed between the hardware layer and the application or operating system. These software make the hardware ready to be used by the application software and users.

Economic feasibility
Refers to the cost of the project.

Embedded software
Software that is part of a larger system and performs some of the requirements of that system; e.g., software used in an aircraft.

End user
A person that uses an information system for the purpose of data processing.

eof()
When users process files, they need to check for the end of file, eof, pointer to know whether the end of file has reached or not. If the end of file has reached, the function returns true and false otherwise.

exit() function
This function tells the program to quit running immediately.

Expression
Combination of operators and operands.

Feasibility study
Analysis and evaluation of a proposed project or system, to determine, whether it is technically, economically and operationally feasible within the estimated cost and time is called feasibility study.

File
A collection of related data (records) that is stored on secondary storage with a unique name.

File and I/O Management
This is the module of the Operating System that is responsible for creating and/or deleting files in the file system and managing the input and output of data in the file system.

File System
A File System refers to the arrangement in a secondary storage that is done by the Operating System for the purpose of data storage and retrieval.

Floating-point type
A floating point type can be either float, double, or long double. It is typically represented as a mantissa and an exponent.

Flow control statements
It is also called flow control statements, which allow the programmer to change the flow of sequence of execution of statements of a program by the processor.

Flowchart
A flowchart is a type of diagram that represents an algorithm or a process.

Formal parameters
These are those parameters which appear in function declarator or function prototype.

for-statement
For-statement is an iteration statement specifying an initializer, an iteration condition, an increment/decrement operation, and a controlled statement.

fstream
A file stream for input and output.

Function
A self-contained program that is used to perform a specific task and be invoked/called from a calling program by providing its arguments is called a function.

Function call
A function call is a statement in the calling function (e.g. main() function) to execute the code of the function.

Function declaration
The declaration of a function tells the compiler about the name, argument types, and return type of the function.

Function definition
Function declarator that includes the full body of the function is called function definition i.e. function declarator plus function body.

Function overloading
is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different number or types of parameters.

Function prototype
A function prototype is that part of a function that tells the compiler the name of the function, the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters, and the order in which these parameters are expected. It does not contain the function body.

Functional requirement
Are those requirements of a software system which describe a function of a software system or of its component. It includes calculations, technical details, data manipulation and processing and other specific functionality that define what a system is supposed to accomplish.

getch ()
This is a standard library function defined in the header file <conio.h> and is used to wait for the user to input a character from the keyboard.

Global function
A function declared outside any function is called global function.

Global variable
The variables defined at the top of a program before the main() function are called global variables.

Graphical User Interface (GUI)
Graphical User Interface is the interface with graphical components; such as icon, menus, buttons and lists etc. that provide easy access to the information stored on the computers.

Header file
Files that hold declarations for pre-defined C++ objects are called header files. Thus, a header file acts as an interface between separately compiled parts of a program.

Hierarchical file system
A file system in which a directory can logically contain other directories.

if-statement
Statement selecting between two alternatives based on a condition. In if construct, the statements in the body of if are executed only if the condition is true and nothing is executed if it is false.

ifstream
A file stream for input defined inside the stream fstream.

Implementation
The process of translating a design into hardware components, software components, or both

Inheritance
The feature of C++ language in which new classes are derived from -existing classes is called inheritance.

Initialization
Assigning values to C++ variables and objects during its declaration time is called Initialization.

inline function
These are special function declared in the programs with the inline keyword to exploit the advantage of faster execution time by overcoming the problem of function call overhead.

Input stream
Input streams take any sequence of bytes from an input device such as a keyboard, a file, or a network.

Integer type
It is a group of data type used in C++ that occurs in one of its forms: short, int, or long. It defines integer variables that store integer constants.

iostream
It is a standard library file (header) or stream that can be used for both input and output in C++ programs.

Job
A unit of work for an operating system.

Kernel
This is the core of Operating System that is responsible for Memory and Processor Management in the system.

Line comment
Comment that starts with symbol (//) and ends with end-of-line.

Local function
A function defined within a function. Not supported by C++. Most often, the use of a local function is a sign that a function is too large.

Local variable
The variables defined within a block are called local variables.

Logical operator
There are three logical operators used in C++. These are (!, &&, !!).

long double
A float type of data that is used to define variables to store extended-precision floating point numbers.

long int
Integer data type that occupies 4 bytes of memory and is used to store large integer values.

Loop
An iterative statement that executes a statement or group of statements zero or more times is called loop. Examples are for-loop, while-loop and a do-while loop.

Macintosh
It is a series of operating systems having graphical user interfaces that have been developed by Apple Inc.

main program
A component of a C++ program from where the execution of the C++ program starts and calls other functions from.

Maintenance
Activities such as adjusting, cleaning, modifying, repairing equipment to assure performance in accordance with requirements.

Management
The process of organizing, coordinating and controlling the activities of a software development by the managers and executives in accordance with certain standard procedures is called management.

Member function
A function declared in the scope of a class to operate on data members.

Memory Management
This is the module of Operating System that manages memory in a system by deciding when to allocate and de-allocate memory to a process and how much to allocate.

Memory manager
A module of operating system responsible for the coordination of different types of computer memories by tracking which memory is available, which is to be allocated or de-allocated and how to move data among them.

Multiple inheritance
The concept of using more than one immediate base classes for a derived class is called multiple inheritance.

Multiprocessing
The simultaneous execution of two or more computer programs or sequences of instructions on multiple processors.

Multi-programming
A mode of operation in which two or more programs are executed in an interleaved manner by a single CPU.

Multitasking
Multitasking is the logical extension of multiprogramming in which multiple tasks are executed by a single CPU.

Multitasking operating system
Those operating systems which perform multiple tasks simultaneous on a single processor in such a way that creates the impression of parallel executions.

Multithreading
Multithreading is the ability of operating system that have multiple threads active in memory at the same time and the processor is switched quickly among them for processing.

Multi-user
Multi-user is the ability of an operating system that makes it possible for several users to login at the same time on a single computer.

Nested if-else
The use of one if statement in the body of another if statement is called nested if structure.

Nested loops
The use of a loop inside the body of another loop is called nested loop.

Non parameterized constructor
It is a [illegible] those object declaration is done without specifying arguments.

Object
A variable of type class is called object. Object has attributes and functions which are combined together in a class construct.

Object oriented language
A programming language that allows the user to express a program in terms of objects and messages between those objects. Examples include C++, and Java etc.

ofstream
It is a file stream used for output that is defined in the stream "fstream".

One dimentional array
A one-dimensional array (or single dimension array) is a type of linear array which can be represented by a single subscript.

Operating System
System software that controls the execution of programs, and that provides services such as resource allocation, scheduling, input/output control, and data management.

Operational feasibility
In this type of feasibility it is determined that whether the proposed system will be used effectively after it has been developed or not.

Operator
The symbols used to perform operations over the operands. Examples are +, *, and &.

Output stream
are used to hold output for a particular data consumer, such as a monitor, a file, or a printer.

Overloading
Having more than one functions with the same name in the same scope or having more than one operator with the same name in the same scope.

Parallel processing
The ability to process "chunks" or blocks of a program simultaneously. To do this, the computer has to have several CPU units and the software to coordinate them.

Parameter
A value or reference passed to a function, or program that serves as input or controls actions.

Parameter
A variable declared in a function prototype or declarator for representing formal argument.

Peripheral devices
Any device which is attached to a computer system or LAN. Devices such as printers, disks and tape drives which are often attached to computers or LANs.

Pointer
A pointer is a variable that holds the address of another variable.

Polymorphism
Polymorphism is the ability to use an operator or function in different ways. Polymorphism gives different meanings or functions to the operators or functions.

Precedence of operator
means that which operand will be evaluated first and which one will be evaluated later if the expression is a complex one.

Preprocessor
The part of a C++ implementation that removes comments, performs macro substitution and #includes.

Private
An access control keyword used in classes. A member defined as private can only be accessed within the class and by the friends of the class.

Private access specifier
It tells the compiler that the members defined in a class by preceding this specifier are accessible only within the class and its friend function.

Private member
A member defined in the scope of private access specifier and accessible only from its own class is called private member.

Process
A job under execution is called process.

Process Control Block
A process control block (PCB) is a data structure that contains information related to a running process.

Process States
A status of a process at a particular time in which it may be is called process state. The process can be in one of the three states: Ready, Running or Suspended.

Processor Management
This relates to the Operating System's activity of managing the processor in the system, i.e. allocating and de-allocating of the processor to the process.

Programmer
Programmer is a technical person that writes computer programs in computer programming languages to develop software.

Programming language
A language used to express computer programs.

Project manager
is a professional in the field of project management responsible for planning, execution, and closing of any project.

Pseudo code
Pseudo code is an outline of a program, written in a form that can easily be converted into real programming statements.

Public
Access control keyword that is used to define public member (s) of a class is called public.

Public member
A member of a class defined by using the keyword public which is then accessible throughout the program is called public member.

Queue
A list formed by users programs (jobs) in a system waiting for their turns to process by the processor.

RAM
Random Access Memory: The memory that is available on a computer for storing data and programs that are currently being processed.

Ready state
A process state in which all resources except the processor are available.

Real Time Operating Systems
A Real Time Operating System (RTOS) is one in which the response to an event takes place in real time (in other words, as and when it is required). A typical example of a RTOS is the operating system used for flight control.

Register
A fast memory like data storage element which is part of the CPU or other control unit.

Relational operators
Are used for the comparison between two expressions. These operators are (==,=, >, >=, <, <=

Requirement validation
Concern with examining the requirements to certify that they meet the intentions of the stakeholders or not.

Requirements analysis
The process of studying user needs to arrive at a definition of a system, hardware, or software requirements.

Requirements phase
The phase of software development life cycle during which the requirements, such as functional and performance capabilities for a software product, are defined and documented.

ROM
Read Only Memory: Stored permanent systems instructions which are never changed; ROM is generally installed by the manufacturer as part of the system.

S

Schedule feasibility
It means that a project can be implemented in an acceptable time frame or not.

Scope
A region within a C++ program in which a C++ object or variable or function is visible is called its scope. It is usually delimited by curly braces { ... }.

Scope of a variable
By the scope of a variable we mean that if a variable is defined somewhere in a program then in which parts of the same program this variable can be accessed.

Secondary storage
Online peripheral data storage where data is permanently stored. Examples are magnetic disk (hard disk), DVD disks and CD disks etc.

Secondary storage management
The function of operating system to manage secondary storage devices and handle proper flow of data among primary and secondary storage devices is called secondary storage management.

Short
It is an integer type holding 2 bytes of memory and used to define integer variables.

Size of array
The number of elements that can be stored in an array is called the size or length of that array.

Software maintenance
Maintenance to a software system includes correcting software errors, adapting software to a new environment, or making enhancements to software i.e. adaptive maintenance, corrective maintenance, and perfective maintenance.

Source code
The human readable version of the list of instructions (program) that cause a computer to perform a task.

Stakeholders
Those entities which are either within the organization or outside of the organization that sponsor a project, or have an interest or have the intention to get it after its successful completion, or may have a positive or negative influence in the project completion are called stakeholders.

Standard stream
Standard streams in C++ consist of four predefined standard stream objects. These are cin, cout, cerr and clog.

Statement
In C++, a line of code ending with a semicolon symbol (;) is called statement.

Statement terminator
Each C++ statement is terminated by a symbol named semicolon (;) which is called statement terminator.

Static variable
A variable defined by using the keyword static is called static variable.

String
A sequence of character is called string.

String stream
A string stream is a stream which reads input from or writes output to an associated string.

switch-statement
Switch statement is a better alternative of the nested if-else structure which selects among many alternatives based on an integer value specified in switch expression.

Syst
It can be defined as a set of interrelated components having a clearly defined boundary that work together to achieve a common set of objectives.

System analysis
A systematic investigation of a real or planned system to determine the functions of the system and how they relate to each other and to any other system.

System design
A process of defining the hardware and software architecture, components, modules, interfaces, and data for a system to satisfy specified requirements.

System Development Life Cycle (SDLC)
SDLC is a step wise process of creating computer systems. It is also known as information system development or application development.

System support
Keeping a system in its proper working condition is called maintenance.

T

Technical feasibility
Refers to the technical resources needed for the development of the proposed software system.

Ternary operator
An operator taking three operands, such as ? :).

Tester
Also called software tester who is a computer programmer having specialty in testing the computer programs using different testing techniques.

Testing
The execution of a program to find its errors is called testing.

Text file
Text files are those files that store data in text format which is readable by human and printable by the printers to make its hard copy.

Thread
A thread is a light weight process. It is the smallest unit of CPU utilization and is also the path of execution within a process.

Time sharing
Time sharing operating systems are those operating systems which slice processor (CPU) time into small fixed time slices and assign the processes to the CPU for that time slice to be executed.

Traversing of array
Accessing the elements of an array only once for the purpose of an operation is called traversing of array.

Two dimensional array
An array having more than one subscripts, i.e. X[i] [j]..[n], is called multidimensional array.

Unary operator
An operator taking one operand, such as, NOT operator (!), increment operator (++) or decrement (--) is called unary operator.

UNIX
Unix is a multitasking and multi-user operating system that was developed at Bell Labs in 1969.

User-defined type function
A function defined by the users for their own task is called user-defined function.

Variable
A name, label, identifier whose value may be changed many times during processing.

Virtual Memory
Virtual Memory refers to the concept whereby a process with a larger size than available memory can be loaded and executed by loading the process in parts.

Void
A keyword used to indicate an absence of information.

Waiting state
A waiting state is the blocked state of a process in which the process either waiting for its turn to be assigned to the CPU for execution or an external event to occur.

While-statement
An iterative statement that presents its condition "at the top" and is usually used for situations in which we do not know the exact number of iterations in advance is called while-statement or loop.

White-box testing
It is a procedure of testing software that tests internal structures or workings of an application.

Window
Windows operating systems are the most commonly used operating systems that are based on Graphical User Interfaces (GUI).

Index

Annexure 1

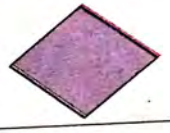
List of Figures

Annexure 2

List of Tables

Annexure 3

List of Symbols

Symbol	Name	Use
	Oval	Start/ End of Program
	Flow Line	Direction of Flow
	Parallelogram	Input or Output
	Rectangle	Processing
	Diamond	Decision Making

Annexure 4

List of Commands

Command	Action
CD	changes the current directory
COPY	copies a file
DEL	deletes a file
DIR	lists directory contents
EDIT	starts an editor to create or edit plain text files
FORMAT	formats a disk to accept DOS files
HELP	displays information about a command
MKDIR	creates a new directory
RD	removes a directory
REN	renames a file
TYPE	displays contents of a file on the screen

Annexure 5

List of Built-in Functions

Function	Header File	Purpose
getch()	*conio.h*	Used to get the character input from the keyboard
getche()	*conio.h*	Used to get character input form the keyboard and echo it on the screen
gets()	*stdio.h*	Stands for get string and is used to get a string input
puts()	*stdio.h*	Used to display the string
getchar()	*stdio.h*	Stands for get character it is used to input as single character from the keyboard. This function receives no argument.
putchar()	*stdio.h*	Standard output function used to display the character, on the output screen, received as an argument.
strcat()	*string.h*	Stands for String concatenation. It combines two strings into a single string.
copy()	*string.h*	This function copy one string into another string.
strcpy()	*string.h*	It is also used to copy one string into another string
find()	*string.h*	To find a particular word or a character in a string
length()	*string.h*	Used to find the length of a string.
swap()	*string.h*	To swap the values of two strings.
sin()	*math.h*	Used to find the sin value of an angle.
cos()	*math.h*	Used to find the cos value of an angle.
tan()	*math.h*	Used to ding the tangent value of an anbgle.
cot()	*math.h*	Used to find the cotangent value of an angle.
abs()	*math.h*	Used to find the absolute value of a variable or expression.
pow()	*math.h*	Used to find the power of a number. It takes two arguments first shows the number and second shows the power to which the number is raised.
sqrt()	*math.h*	Used to find the square root of an number received as an argument.
open()	*fstream.h*	Used to open a file.
close()	*fstream.h*	Used to close a file.
bof()	*fstream.h*	used to set the pointers to the beginning of a file
eof()	*fstream.h*	used to set the pointers at the end of a file

nnexure 6

Complete list of ASCII Codes

ASCII code	symbol	Description
0	NULL	(Null character)
1	SOH	(Start of Header)
2	STX	(Start of Text)
3	ETX	(End of Text)
4	EOT	(End of Transmission)
5	ENQ	(Enquiry)
6	ACK	(Acknowledgement)
7	BEL	(Bell)
8	BS	(Backspace)
9	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowledgement)
22	SYN	(Synchronous idle)
23	ETB	(End of transmission block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
32		(space)
33	!	(exclamation mark)
34	"	(Quotation mark)
35	#	(Number sign)
36	$	(Dollar sign)
37	%	(Percent sign)
38	&	(Ampersand)
39	'	(Apostrophe)
40	(	(round brackets or parentheses)
41	)	(round brackets or parentheses)
42	*	(Asterisk)
43	+	(Plus sign)
44	,	(Comma)
45	-	(Hyphen)
46	.	(Full stop , dot)
47	/	(Slash)
48	0	(number zero)
49	1	(number one)
50	2	(number two)
51	3	(number three)
52	4	(number four)
53	5	(number five)
54	6	(number six)
55	7	(number seven)
56	8	(number eight)
57	9	(number nine)
58	:	(Colon)
59	;	(Semicolon)
60	<	(Less-than sign)
61	=	(Equals sign)
62	>	(Greater-than sign ; Inequality)
63	?	(Question mark)
64	@	(At sign)
65	A	(Capital A)
66	B	(Capital B)
67	C	(Capital C)
68	D	(Capital D)
69	E	(Capital E)
70	F	(Capital F)
71	G	(Capital G)
72	H	(Capital H)
73	I	(Capital I)
74	J	(Capital J)
75	K	(Capital K)
76	L	(Capital L)
77	M	(Capital M)
78	N	(Capital N)
79	O	(Capital O)
80	P	(Capital P)
81	Q	(Capital Q)
82	R	(Capital R)
83	S	(Capital S)
84	T	(Capital T)
85	U	(Capital U)
86	V	(Capital V)
87	W	(Capital W)
88	X	(Capital X)
89	Y	(Capital Y)
90	Z	(Capital Z)
91	[	(square brackets or box brackets)

ASCII code	symbol	Description
92	\	(Backslash)
93	]	(square brackets or box brackets)
94	^	(Caret or circumflex accent)
95	_	(underscore, under strike, underbar or low line)
96	`	(Grave accent)
97	a	(Lowercase a)
98	b	(Lowercase b)
99	c	(Lowercase c)
100	d	(Lowercase d)
101	e	(Lowercase e)
102	f	(Lowercase f)
103	g	(Lowercase g)
104	h	(Lowercase h)
105	i	(Lowercase i)
106	j	(Lowercase j)
107	k	(Lowercase k)
108	l	(Lowercase l)
109	m	(Lowercase m)
110	n	(Lowercase n)
111	o	(Lowercase o)
112	p	(Lowercase p)
113	q	(Lowercase q)
114	r	(Lowercase r)
115	s	(Lowercase s)
116	t	(Lowercase t)
117	u	(Lowercase u)
118	v	(Lowercase v)
119	w	(Lowercase w)
120	x	(Lowercase x)
121	y	(Lowercase y)
122	z	(Lowercase z)
123	{	(curly brackets or braces)
124	|	(vertical-bar, vbar, vertical line or vertical slash)
125	}	(curly brackets or braces)
126	~	(Tilde ; swung dash)
127	DEL	(Delete)
128	Ç	(Majuscule C-cedilla)
129	ü	(letter "u" with umlaut or diaeresis ; "u-umlaut")
130	é	(letter "e" with acute accent or "e-acute")
131	â	(letter "a" with circumflex accent or "a-circumflex")
132	ä	(letter "a" with umlaut or diaeresis ; "a-umlaut")
133	à	(letter "a" with grave accent)
134	å	(letter "a" with a ring)
135	ç	(Minuscule c-cedilla)
136	ê	(letter "e" with circumflex accent or "e-circumflex")
137	ë	(letter "e" with umlaut or diaeresis ; "e-umlaut")
138	è	(letter "e" with grave accent)
139	ï	(letter "i" with umlaut or diaeresis ; "i-umlaut")
140	î	(letter "i" with circumflex accent or "i-circumflex")

ASCII code	symbol	Description
141	ì	(letter "i" with grave accent)
142	Ä	(letter "A" with umlaut or diaeresis ; "A-umlaut")
143	Å	(letter "A" with a ring)
144	É	(Capital letter "E" with acute accent or "E-acute")
145	æ	(Latin diphthong "ae")
146	Æ	(Latin diphthong "AE")
147	ô	(letter "o" with circumflex accent or "o-circumflex")
148	ö	(letter "o" with umlaut or diaeresis ; "o-umlaut")
149	ò	(letter "o" with grave accent)
150	û	(letter "u" with circumflex accent or "u-circumflex")
151	ù	(letter "u" with grave accent)
152	ÿ	(letter "y" with diaeresis)
153	Ö	(letter "O" with umlaut or diaeresis ; "O-umlaut")
154	Ü	(letter "U" with umlaut or diaeresis ; "U-umlaut")
155	ø	(slashed zero or empty set)
156	£	(Pound sign ; symboi for the pound sterling)
157	Ø	(slashed zero or empty set)
158	×	(multiplication sign)
159	ƒ	(function sign ; f with hook sign ; florin sign)
160	á	(letter "a" with acute accent or "a-acute")
161	í	(letter "i" with acute accent or "i-acute")
162	ó	(letter "o" with acute accent or "o-acute")
163	ú	(letter "u" with acute accent or "u-acute")
164	ñ	(letter "n" with tilde ; enye)
165	Ñ	(letter "N" with tilde ; enye)
166	ª	(feminine ordinal indicator)
167	º	(masculine ordinal indicator)
168	¿	(Inverted question marks)
169	®	(Registered trademark symbol)
170	¬	(Logical negation symbol)
171	½	(One half)
172	¼	(Quarter or one fourth)
173	¡	(Inverted exclamation marks)
174	«	(Guillemets or angle quotes)
175	»	(Guillemets or angle quotes)
176	▒	
177	▓	
178	▓	
179	│	(Box drawing character)
180	┤	(Box drawing character)
181	Á	(Capital letter "A" with acute accent or "A-acute")
182	Â	(letter "A" with circumflex accent or "A-circumflex")
183	À	(letter "A" with grave accent)
184	©	(Copyright symbol)
185	╣	(Box drawing character)
186	║	(Box drawing character)
187	╗	(Box drawing character)
188	╝	(Box drawing character)
189	¢	(Cent symbol)
190	¥	(YEN and YUAN sign)
191	┐	(Box drawing character)
192	└	(Box drawing character)
193	┴	(Box drawing character)
194	┬	(Box drawing character)
195	├	(Box drawing character)
196	─	(Box drawing character)
197	┼	(Box drawing character)
198	ã	(letter "a" with tilde or "a-tilde")

ASCII code	symbol	Description
199	Ã	(letter "A" with tilde or "A-tilde")
200	╚	(Box drawing character)
201	╔	(Box drawing character)
202	╩	(Box drawing character)
203	╦	(Box drawing character)
204	╠	(Box drawing character)
205	═	(Box drawing character)
206	╬	(generic currency sign)
207	¤	(lowercase "eth")
208	ð	(Capital letter "Eth")
209	Ð	(letter "E" with circumflex accent or "E-circumflex")
210	Ê	(letter "E" with umlaut or diaeresis ; "E-umlaut")
211	Ë	(letter "E" with grave accent)
212	È	(lowercase dot less i)
213	ı	(letter "I" with acute accent or "I-acute")
214	Í	(Capital letter "I" with circumflex accent or "I-circumflex")
215	Î	(letter "I" with umlaut or diaeresis ; "I-umlaut")
216	Ï	(Box drawing character)
217	┘	(Box drawing character)
218	┌	(Block)
219	█	
220	▄	(vertical broken bar)
221	¦	(letter "I" with grave accent)
222	Ì	
223	▀	(Capital letter "O" with acute accent or "O-acute")
224	Ó	(letter "Eszett" ; "scharfes S" or "sharp S")
225	ß	(letter "O" with circumflex accent or "O-circumflex")
226	Ô	(letter "O" with grave accent)
227	Ò	(letter "o" with tilde or "o-tilde")
228	õ	(letter "O" with tilde or "O-tilde")
229	Õ	(Lowercase letter "Mu" ; micro sign or micron)
230	µ	(capital letter "Thorn")
231	þ	(lowercase letter "thorn")
232	Þ	(Capital letter "U" with acute accent or "U-acute")
233	Ú	(letter "U" with circumflex accent or "U-circumflex")
234	Û	(letter "U" with grave accent)
235	Ù	(letter "y" with acute accent)
236	ý	(Capital letter "Y" with acute accent)
237	Ý	(macron symbol)
238	¯	(Acute accent)
239	´	(Hyphen)
240		(Plus-minus sign)
241	±	(underline or underscore)
242	‗	(three quarters)
243	¾	(paragraph sign or pilcrow)
244	¶	(Section sign)
245	§	(The division sign ; Obelus)
246	÷	(cedilla)
247	¸	(degree symbol)
248	°	(Diaeresis)
249	¨	(Interpunct or space dot)
250	·	(superscript one)
251	¹	(cube or superscript three)
252	³	(Square or superscript two)
253	²	(black square)
254	■	(non-breaking space or no-break space)
255	nbsp	

About Authors

1. **RAHMAN ALI** earned M.Phil (MS) degree in Computer Science from the Department of Computer Science, University of Peshawar in 2009. He earned his Master (M.Sc) degree in Computer Science from Hazara University, Mansehra in 2005. He did his B.Sc and F.Sc from Govt. Post Graduate lehanzeb College, Saidu Sharif, Swat in 2002 and 1999 respectively. He did SSC from Govt. High School Kishawara, Swat in 1997. He also holds bachelor degree in Education (B.Ed), earned from University of Peshawar in 2006.

He served Govt. Degree College, Kabal, Swat as a lecturer in Computer Science selected from September 2007 to August 2009. He also served Institute of IT, University of Science & Technology, Bannu as a lecturer in Computer Science from August 2009 to December 2009.

Rahman Ali is serving is a Lecturer in Computer Science at Quaid-e-Azam College of Commerce, University of Peshawar since December 2009.

Ali has his MS major in Artificial Intelligence (AI) and worked in Natural Language Processing (NLP). He has worked in anaphora, ambiguity and ellipsis resolution, Part-of- Speech (POS) tagging, natural language interfaces to databases, corpus linguistics, and machine translation and transliteration. He has published his research work in well reputed journals and conferences.

Mohammad Khalid is Head of Computer Science Department at OPF Boys College, Islamabad. Throughout his educational career he has got first-class first division. He did his M.Sc in Computer Science from the University of Peshawar in 1996 and earned his B.Ed degree in Science from Institute of Education and Research, University of Peshawar in 1997. He has been instructor and teacher in Computer Science for the last seventeen years. He has vast teaching experience from Secondary to Masters Level. He has attended many national and International level seminars and meetings for the promotion of teaching Computer Science at different levels.

He has been member of the National Curriculum development team sicne 2007. He has contributed a great deal in writing the Curriculum of Computer Education from Grades VI-VIII and Curriculum of Computer Science from Grades IX-XII.

As an author he has written a textbook of Computer Science for grade IX for KPK Textbook Board and a textbook of Computer Education for grade VII for National Books Foundation.

Keeping in view his experience and expertise in the subject, this book will prove to be an asset both for the students and the teachers.

مومن تو آپس میں بھائی بھائی ہیں۔
پس اپنے دو بھائیوں میں صلح کرا دیا کرو۔
اور اللہ سے ڈرتے رہو تاکہ تم پر رحم کیا جائے۔
(سورۃ الحجرات: ۱۰)

مومن تو آپس میں بھائی بھائی ہیں۔
پس اپنے دو بھائیوں میں صلح کرا دیا کرو۔
اور اللہ سے ڈرتے رہو تاکہ تم پر رحم کیا جائے۔
(سورۃ الحجرات: ۱۰)

قومی ترانہ

پاک سَر زمین شاد باد　　کشورِ حَسین شاد باد
تو نشانِ عَزم عالی شان　　ارضِ پاکستان
مرکزِ یقین شاد باد

پاک سَر زمین کا نظام　　قُوت اُخوت عوام
قوم، ملک، سلطنت　　پائندہ، تابِندہ باد
شاد بَاد منزلِ مُراد

پرچمِ سِتارہ و ہلال　　رَہبرِ ترقی و کمال
ترجُمان ماضی شانِ حال　　جانِ اِستقبال
سایۂ خدائے ذُوالجلال

LEADING BOOKS PUBLISHER
JAMRUD ROAD, PESHAWAR

Printer	Price	Quantity	Code No.
Creative Printers & Publishers	184.00	8,962	PSP/CTV-44-228-1819(O)