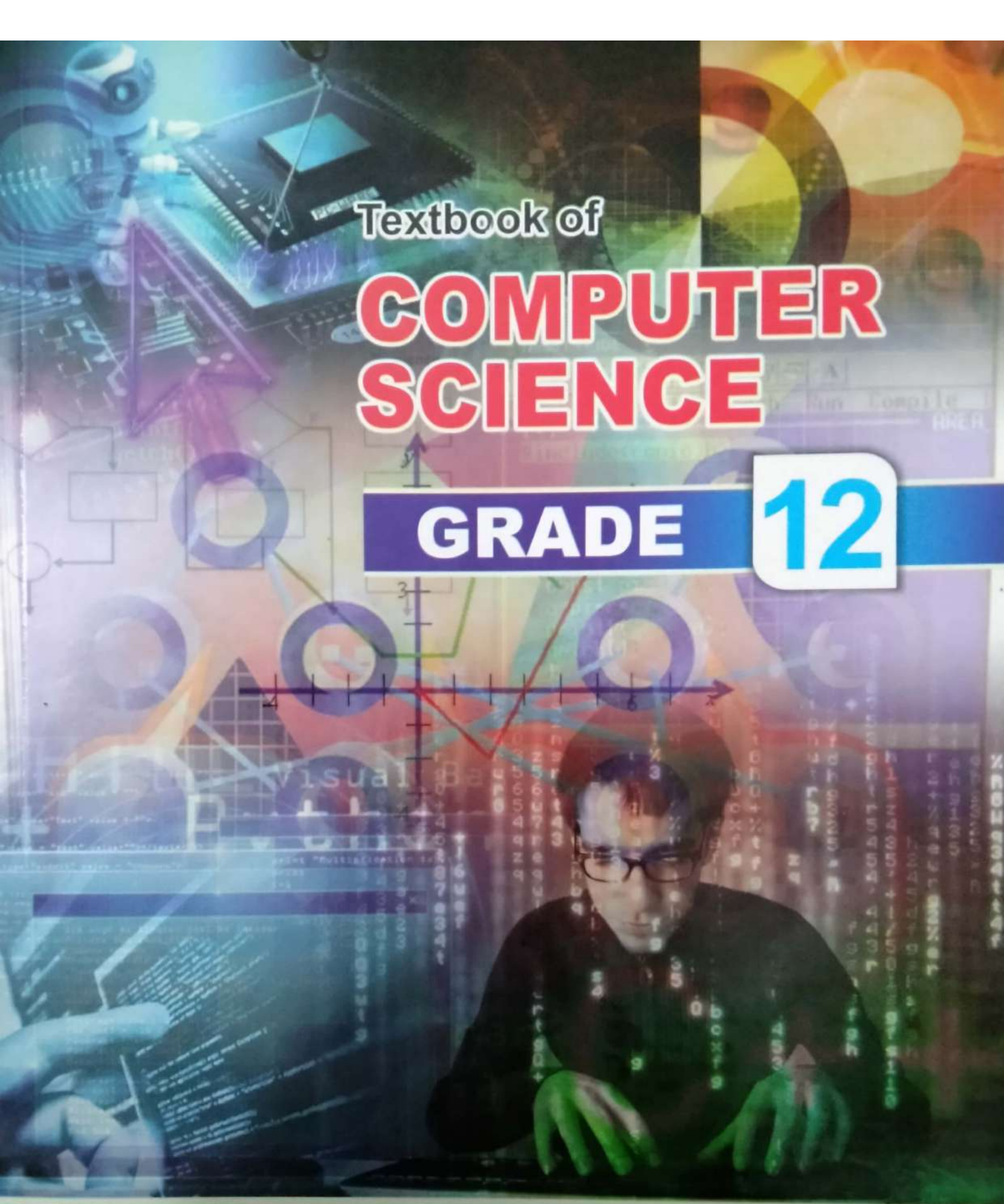


Textbook of

# COMPUTER SCIENCE

GRADE

12



National Book Foundation  
as  
Federal Textbook Board  
Islamabad

# COMPUTER SCIENCE

Grade

12

MDCAT BY FUTURE DOCTORS-TOUSEEF AHMAD-03499815886



National Book Foundation  
as  
Federal Textbook Board  
Islamabad

## OUR MOTTO

• Standards • Outcomes • Access • Style

© 2020 National Book Foundation as Federal Textbook Board, Islamabad.  
All rights reserved. This volume may not be reproduced in whole or in part in any form (abridged, photo copy, electronic etc.) without prior written permission from the publisher.

Textbook of  
Computer Science Grade - 12



Authors	:	Mr. Mohammad Sajjad Heder Mr. Mohammad Khalid
Designer	:	Hafiz Rafiuddin (Late), Shahzad Ahmad
Desk Officer	:	Dr. Zulfiqar Ali Cheema (EO) CD & TP Wing
Management of	:	Mr. Ishtiaq Ahmed Malik, Secretary, NBF
Incharge Textbooks	:	Muhammad Rafique, Assistant Director, NBF
First Edition	:	2018 Qty: 24000
2nd Print	:	May 2019 Qty: 10000
3rd Print	:	Sep. 2020 Qty: 3000
4th Print	:	July 2020 Qty: 15000
Price	:	Rs. 170/-
Code	:	STE: 575
ISBN	:	978-969-37-1102-8
Printer	:	S.I Printers, Rawalpindi

for Information about other National Book Foundation Publications,  
visit our Web site <http://www.nbf.org.pk>, Phone: 92-51-9261124, 9261125  
Email us at: [nbftextbooks@gmail.com](mailto:nbftextbooks@gmail.com) / [books@nbf.org.pk](mailto:books@nbf.org.pk)

# PREFACE

The textbook of COMPUTER SCIENCE for GRADE - 12 has been developed according to the National Curriculum 2009. This book consists of nine chapters. It is based on C Programming Language, Computer Logic and Gates, and HTML.

Computers are changing our world and driving our economy. We are living in a world controlled by computer software. Students must learn how to design and write computer programs to solve problems related with any field. No matter what students' future plan may be, learning computer programming is essential for them. It enables them to succeed in the technology-driven world of 21<sup>st</sup> century. Programming allows putting ideas into reality in any industry. It provides skills about how to make use of programming language to solve various computational problems. There will be high demand of programmers in tomorrow's world to build useful applications and websites.

This textbook is developed with due care using simple language, supporting pictures to make it understandable by secondary school students. Keen efforts have been made to minimize the errors and omissions. This will surely enhance the confidence and comprehension of the students and general readers in the subject. Yet there is always room for improvement and any suggestion with regard to the quality of work will be warmly welcomed at our end.

Quality of Standards and Actualization of Style is our motto. With these elaborations, this series of new development is presented for use. As there is always room for improvement, criticism and suggestions are always welcome to make the subsequent editions of the textbook more interesting, informative and useful for the students. After research and necessary changes, the book is being published again.

**National Book Foundation**

# TABLE OF CONTENTS

<b>CHAPTER 1: OPERATING SYSTEM.....</b>	<b>08</b>
1.1 Introduction .....	09
1.1.1 Operating System.....	09
1.1.2 Commonly Used Operating Systems .....	09
1.1.3 Types of Operating System .....	13
1.1.4 Single-User and Multi-User Operating Systems .....	14
1.2 Operating System Functions.....	15
1.3 Process Management .....	17
1.3.1 Process Definition.....	17
1.3.2 Various States of a Process.....	18
1.3.3 Thread and Process.....	18
1.3.4 Multithreading .....	19
1.3.5 Multitasking.....	19
1.3.6 Multiprogramming .....	20
<b>CHAPTER 2: SYSTEM DEVELOPMENT LIFE CYCLE .....</b>	<b>24</b>
2.1 System Development Life Cycle .....	25
2.1.1 A System .....	25
2.1.2 System development life cycle (SDLC) and its importance .....	25
2.1.3 Objectives of SDLC .....	25
2.1.4 Stakeholders of SDLC .....	26
2.1.5 SDLC Phases / Steps .....	26
1. Defining Phase .....	27
2. Planning Phase.....	27
3. Feasibility.....	27
4. Analysis Phase .....	28
5. Requirement Engineering .....	28
6. Design Phase .....	29
7. Construction / Coding .....	31
8. Testing / Verification .....	31
9. Deployment / Implementation .....	32
10. Maintenance / Support.....	33
2.1.6 Personal involved in SDLC and their Role .....	33
<b>CHAPTER 3: OBJECT ORIENTED PROGRAMMING IN C++ .....</b>	<b>40</b>
3.1 Introduction .....	41
3.1.1 Computer Program .....	41
3.1.2 Header File and Reserved Words.....	41
3.1.3 Structure of A C++ Program .....	42
3.1.4 Statement Terminator.....	43
3.1.5 Comments in C++ Program .....	44
3.2 C++ Constants and Variables .....	45
3.2.1 Constants and Variables.....	45
3.2.2 Rules for Specifying Variable Names.....	45
3.2.3 C++Data Types .....	46
3.2.4 The Const Qualifier.....	47
3.2.5 Variable Declaration and Initialization .....	47

3.2.6	Type Casting .....	47
3.3	<b>Input / Output Handling.....</b>	<b>48</b>
3.3.1	The cout Statement .....	49
3.3.2	The cin Statement.....	49
3.3.3	The getch(), gets() and puts() Functions .....	50
3.3.4	The Escape Sequence .....	51
3.3.5	Commonly Used Escape Sequences.....	52
3.3.6	Programming with I/O Handling Functions .....	53
3.3.7	The manipulators endl and Setw.....	54
3.4	<b>Operators in C++ .....</b>	<b>57</b>
3.4.1	Types of Operators .....	57
3.4.2	Unary, Binary and Ternary Operators .....	64
3.4.3	Order of Precedence of Operators .....	64
3.4.4	Expressions in C++.....	65
<b>CHAPTER 4: CONTROL STRUCTURES .....</b>		<b>70</b>
4.1	<b>Decision Structures .....</b>	<b>71</b>
4.1.1	The If Statement.....	71
4.1.2	The If-Else Statement.....	73
4.1.3	The Else-If Statement.....	74
4.1.4	The Switch Statement.....	77
4.1.5	Difference Between If-Else If and Switch Statements .....	80
4.2	<b>LOOPS .....</b>	<b>81</b>
4.2.1	The For Loop.....	82
4.2.2	The While Loop .....	84
4.2.3	The Do While Loop.....	86
4.2.4	Difference Between While Loop and Do-While Loop.....	87
4.2.5	The Break and Continue Statements .....	87
4.2.6	The Exit Function.....	89
4.2.7	Nested Loop .....	89
<b>CHAPTER 5: ARRAYS AND STRINGS.....</b>		<b>96</b>
5.1	<b>Introduction to Arrays.....</b>	<b>97</b>
5.1.1	Concept of an Array.....	97
5.1.2	Declaring and Array.....	98
5.1.3	Initialization of Array .....	98
5.1.4	Using Arrays in Programs .....	98
5.1.5	The sizeof() Function .....	102
5.2	<b>Two Dimensional Arrays.....</b>	<b>103</b>
5.2.1	Introduction to Two Dimensional Arrays.....	103
5.2.2	Defining and Initializing A Two Dimensional Array .....	103
5.2.3	Accessing and Writing in a Two Dimensional Array .....	105
5.3	<b>STRINGS .....</b>	<b>107</b>
5.3.1	Introduction to Strings.....	107
5.3.2	Defining a String .....	107
5.3.3	Initializing Strings.....	108
5.4	<b>Commonly Used Strings Functions.....</b>	<b>108</b>
<b>CHAPTER 6: FUNCTIONS .....</b>		<b>116</b>
6.1	<b>Functions.....</b>	<b>117</b>

6.1.1	Types of Functions .....	117
6.1.2	Advantages of Functions .....	120
6.1.3	Function Signature.....	120
6.1.4	Function Components.....	120
6.1.5	Scope of Variables in Functions.....	122
6.1.6	Parameters.....	125
6.1.7	Local and Global Functions .....	126
6.2	<b>Passing Arguments and Returning Values .....</b>	<b>129</b>
6.2.1	Passing Arguments.....	129
6.2.2	Default Arguments .....	134
6.2.3	Return Statement.....	135
6.3	<b>Function Overloading .....</b>	<b>136</b>
6.3.1	Advantages of Function Overloading .....	136
6.3.2	Use of Function Overloading .....	137
<b>CHAPTER 7: POINTERS .....</b>		<b>142</b>
7.1	<b>POINTERS.....</b>	<b>142</b>
7.1.1	Pointer Variable .....	142
7.1.2	Memory Addresses.....	143
7.1.3	Reference Operator (&) .....	143
7.1.4	Dereference Operator (*) .....	144
7.1.5	Declaring Variables of Pointer Types.....	145
7.1.6	Pointer Initialization.....	146
<b>CHAPTER 8: OBJECTS AND CLASSES.....</b>		<b>150</b>
8.1	<b>Classes and Objects .....</b>	<b>151</b>
8.1.1	Class and Object .....	151
8.1.2	Member of a Class.....	153
8.1.3	Access Specifiers .....	155
8.1.4	Data Hiding .....	159
8.1.5	Constructor and Destructor .....	159
8.1.6	Declaration of Objects for Accessing Members of a Class .....	165
8.1.7	Inheritance and Polymorphism .....	167
<b>CHAPTER 9: FILE HANDLING .....</b>		<b>174</b>
9.1	<b>File Handling.....</b>	<b>174</b>
9.1.1	Types of Files .....	175
9.1.2	Opening File .....	176
9.1.3	bof() and eof().....	181
9.1.4	Stream.....	182
9.1.5	Types of Streams.....	182





# 1

## OPERATING SYSTEM



After completing this lesson, you will be able to:

- Define operating system
- Describe commonly used operating systems
- 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
- Define Single-user and Multi-user Operating Systems
- Describe the following main functions of operating system
  - Process Management
  - Memory Management
  - File Management
  - I/O System Management
  - Secondary Storage Management
  - Network Management
  - Protection System
  - Command Interpreter
- Define Process
- Describe new, running, waiting/blocked, ready and terminated states of a process
- Differentiate between the following.
  - Thread and process
  - Multi-threading and multitasking
  - Multitasking and multiprogramming.



## 1.1 INTRODUCTION

An **operating system** is the most important software that runs on a computer. It manages the computer's memory and processes, as well as all of its software and hardware. It also allows users to communicate with the computer. Without an operating system, a computer user cannot run any program on the computer. It automatically loads in RAM when the computer is turned on. Operating systems exist from the very first computer generation and keep evolving with time.

### 1.1.1 OPERATING SYSTEM

An Operating System (OS) is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware.

Every desktop computer, tablet, and smartphone includes an operating system that provides basic functionality for the device. Common desktop operating systems include Windows, OS X, and Linux.

Operating system performs the following tasks.

- Loads application/system software into main memory and executes it.
- Controls the operation of main memory and external storage devices.
- Manages files and folders on storage devices such as hard disk, USB flash drive, etc.
- Manages the operations of all the input/output devices.
- Allows multitasking to handle several tasks at the same time such as running a spreadsheet software and a word-processor simultaneously.
- Performs network operations which enable a number of users to communicate with each other in a network environment and share computer resources such as CPU, main memory, hard disk, printer, Internet, etc.
- Detects hardware failures.
- Provides security through username and password.

### 1.1.2 COMMONLY USED OPERATING SYSTEMS

The commonly used operating systems are DOS, WINDOWS, Macintosh's OS X/OS2 and UNIX/LINUX.



#### Teacher Point

Before starting the chapter, the students could be encouraged to explain what they understand about Operating system.

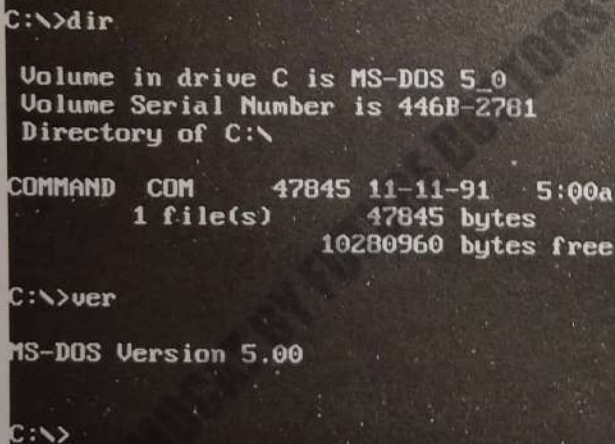
## DOS

DOS stands for Disk Operating System. It was developed in 1970s when microcomputer was introduced. It was called Disk Operating System because the entire operating system was stored on a single floppy disk. It had text-based (also known as command-line) user interface. The user had to type commands to interact with the computer.

The following are some DOS commands.

RENAME	For renaming a file
CD	For changing directory (called <b>folder</b> in Windows)
DIR	To display directories and files in a directory
DEL	To delete one or more files
COPY	To copy files from one drive/directory to another
FORMAT	To format a disk

The user had to learn the basic commands to operate the computer effectively. DOS was not a user-friendly operating system. DOS commands were difficult to learn, memorize and use for novice computer users. DOS had been used successfully on microcomputers for many years but it was replaced by a more user-friendly operating system called Windows in the early 1990s. DOS interface is shown in Figure 1.1.



```
C:\>dir

Volume in drive C is MS-DOS 5_0
Volume Serial Number is 446B-2781
Directory of C:\

COMMAND  COM      47845 11-11-91  5:00a
          1 file(s)      47845 bytes
                        10280960 bytes free

C:\>ver

MS-DOS Version 5.00

C:\>
```

Figure 1.1 DOS Interface



### Teacher Point

Explain the importance of Operating System for a computer.

## WINDOWS OPERATING SYSTEM

Windows operating system was developed in mid 1980s by Microsoft Corporation. It provides a Graphical User Interface (GUI) which is user-friendly. The user does not have to memorize commands like DOS. It allows user to give commands to computer through icons, menus, and buttons etc. Today, it is the most commonly used operating system on PCs and laptop computers all over the world. Microsoft has released many versions of Windows over the years to enhance its user interface in computer technology. Some popular versions of Windows in the past were Windows 95, Windows 98, Windows Millennium, Windows XP and Windows Vista etc. Windows 10 interface is shown in Figure 1.2.



Figure 1.2 Windows 10 Interface

## Mac OS

Mac OS is a series of operating systems developed by Apple Corporation. Mostly it is installed on all the Apple computers. The latest version is known as OS X. It is the tenth major release of the Mac operating systems. It is a more secure operating system compared to Windows. Mac hardware and software works together very well with minimum flaws. Mac computer is of high quality but more expensive than IBM compatible computers. A large variety of application software is easily available for Windows operating system whereas the OS X has very limited application software. The OS X is not a widely used operating system like the Windows. Mac OS X interface is shown in Figure 1.3.



## Teacher Point

Teacher may assist his/her students in the installation of common operating systems in computer and other devices.



Figure 1.3 Mac OS X Interface

## UNIX

UNIX operating system was developed in early 1970s at Bell Laboratories research center by Ken Thompson and Dennis Ritchie. It was developed in C language. It provides greater processing power and better security than Windows operating system. Computers running UNIX operating system rarely have malware attack. It is available for a wide range of computer systems from microcomputers to mainframes. It is less popular on microcomputers on which Windows is pre-installed when they are sold. UNIX OS/390 interface is shown in Figure 1.4.

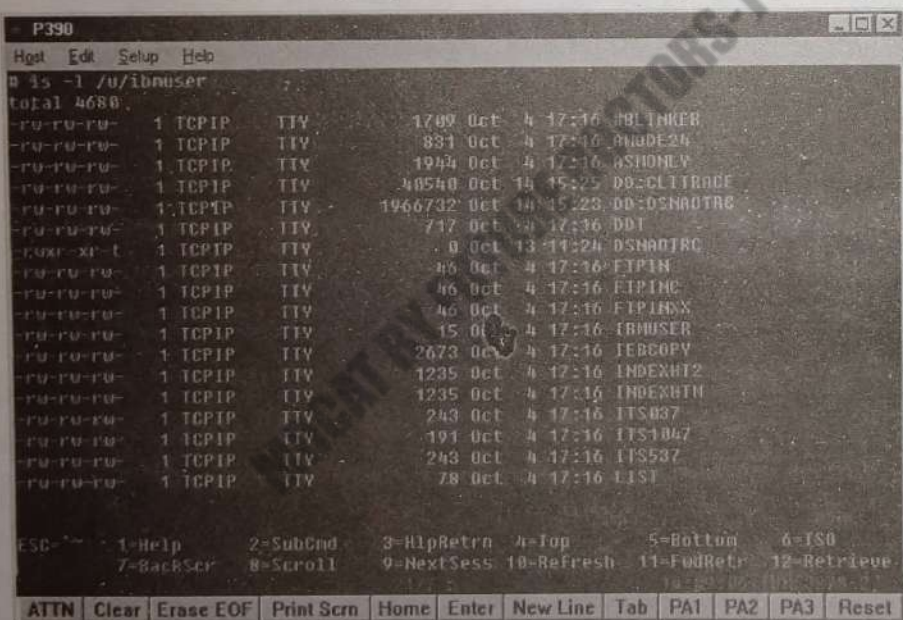


Figure 1.4 UNIX OS/390 Interface



## Teacher Point

Teacher could also introduce the operating systems like iOS and Android.



### 1.1.3 TYPES OF OPERATING SYSTEMS

The following are the important types of operating systems that are commonly used on various computer systems.

#### Batch Processing Operating System

A batch processing operating system is a software that groups together same type of jobs in batches and automatically executes them one by one. It performs the same type of task on all the jobs in a batch in the sequence in which they appear. It provides an easy and efficient way of processing the same type of jobs. For example, at the end of month, banks print statement for each account holder. A batch processing system can easily and efficiently print each account holder's statement one by one.

#### Multiprogramming Operating System

A multiprogramming operating system is a software that loads one or more programs in main memory and executes them using a single CPU (Central Processing Unit). In fact, the CPU executes only one program at a time while other programs are waiting in queue. In multiprogramming system when one program is busy with input/output operation, the CPU executes another program that is in queue. In this way, multiprogramming operating system uses the CPU time and other resources of computer to improve the performance of computer.

#### Multitasking Operating System

A multitasking operating system is a software that performs multiple tasks at the same time on a computer that has a single CPU. The CPU executes only one program at a time but it rapidly switches between multiple programs and it appears as if all the users' programs are being executed at the same time.

#### Time-sharing Operating System

A time-sharing operating system is a software that shares the CPU time between multiple programs that are loaded in main memory. A time-sharing operating system gives a very short period of CPU time to each program one by one. This short period of time is called time slice or quantum. Since the CPU is switched between the programs at extremely fast speed, all the users get the impression of having their own CPU. It is used in mini and mainframe computers that support large number of users in big organization such as airline, bank, university, etc.

#### Real-time Operating System

A real-time operating system is a software that runs real-time applications that must process data as soon as it comes and provides immediate response. Real-time operating system executes special applications within specified time with high reliability. It is commonly used in space research programs, real-time traffic control and to control industrial processes such as oil refining.

#### Multiprocessor Operating System

A multiprocessor operating system is a software that controls the operations of two or more CPUs within a single computer system. All the CPUs of computer share the same main memory and input/output devices. Multiprocessing operating systems are used to obtain very

high speed to process large amount of data. It executes a single program using many CPUs at the same time to improve processing speed. Computers that support multiprocessing have sophisticated architecture which is difficult to design.

### Parallel Processing Operating System

A parallel processing operating system is a software that executes programs developed in a parallel programming language. It uses many processors at the same time. In a parallel processing system, the task of a program that requires many calculations is divided into many smaller tasks and these are processed by multiple processors at the same time. Parallel processing operating systems are used in supercomputers that have thousands processors.

### Distributed Operating System

A distributed operating system is a software that manages the operation of a distributed system. A distributed system allows execution of application software on different computers in a network. In a distributed system, user programs may run on any computer in the network and access data on any other computer. The users of distributed system do not know on which computer their programs are running. Distributed operating system automatically balances the load on different computers in the network and provides fast execution of application software.

### Embedded Operating System

An embedded operating system is a built-in operating system which is embedded in the hardware of the device. It controls the operation of devices such as microwave oven, TV, camera, washing machine, games, etc. It runs automatically when the device is turned on and performs specific task.

## 1.1.4 SINGLE-USER AND MULTI-USER OPERATING SYSTEMS

Operating systems are divided into single-user and multi-user operating systems based on the number of users they can support.

### Single-user Operating System

The operating system that allows only one person to operate the computer at a time is known as single-user operating system. Commonly used single-user operating systems are DOS and Windows (starting versions upto 1995).

### Multi-user Operating System

The operating system that allows many users on different terminals or microcomputers to use the resources of single central computer (server) in a network is known as multi-user operating system. It is used on servers in business and offices where many users have to access the same application software and other resources. Some examples of multi-user operating systems are UNIX, Linux, Windows 2000 onward and Max OS X.



### Teacher Point

Students may be taken to some organizations like Electric supply companies, Sui gas companies, Airlines, etc. to show the working of different types of operating systems, like Batch processing, Multiprogramming, Time sharing and Real time O.S. etc.



## 1.2 OPERATING SYSTEM FUNCTIONS

The following are the functions performed by operating system.

- Process Management
- Memory Management
- File Management
- I/O Management
- Secondary Storage Management
- Network Management
- Protection System
- Command-interpreter

### Process Management

A process is a program in execution. Process management is the part of operating system that manages allocation of computer resources (like CPU time) to various processes in main memory. Process management actually describes the state and resource ownership of each process.

**Example:** In this example there are three processes **A**, **B** and **C** ready for execution. The OS will manage the CPU time as follows.

**Process A** has CPU cycle ( $t_a = 5$  milli sec)

**Process B** has CPU cycle ( $t_b = 2$  milli sec)

**Process C** has CPU cycle ( $t_c = 1$  milli sec)

Case 1: When the 3 processes become ready in the order of ABC, the total execution time will be:

$$\tau = (5 + 7 + 8)/3 = 6.67 \text{ milli sec}$$

Case 2: When the 3 processes become ready in the order of BCA, the total execution time will be:

$$\tau = (2 + 3 + 8)/3 = 4.33 \text{ milli sec}$$

In the above example, in Case2, the OS is managing the processes more efficiently. The execution time in Case 2 is less as compare to Case 1.

### Memory Management

Memory management is the part of operating system that controls and manages the operation of main memory during the operation of computer. It allocates space to programs that are loaded in main memory for execution. It keeps track of freed memory when a program is closed and updates the memory status.



**Example:** In this example the OS is managing memory for two processes P0 and P1. P1 is being loaded (swap in) and P0 is being taken out (swap out) from the main memory (RAM). The whole process is shown in Figure 1.5.

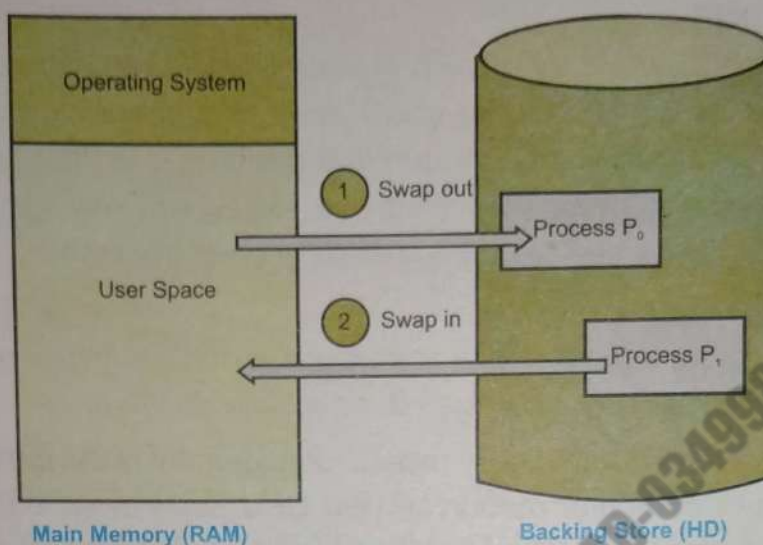


Figure 1.5 Memory Management

## File Management

File management is the part of operating system that manages files and folders on storage devices such as hard disk, USB flash drive and DVD. It allows computer user to perform operations such as creating, copying, moving, renaming, deleting, and searching files and folders. It also allows the user to perform read, write, open and close operations on files and folders. Figure 1.6 shows the management of files in various folders by OS.



Figure 1.5 File Management



### I/O Management

I/O management is the part of operating system that controls all the input/output operations during program execution. It manages all the input/output operations of input/output and storage devices. Efficient I/O management improves the performance of computer.

**Example:** There are three programs A, B and C which are using the printer. Now the OS will decide which program to use the printer first. A queue will be set by the OS and each program will get the printer by its turn.

### Secondary Storage Management

Secondary storage management is the part of operating system that manages free space and storage allocation of user programs and data on secondary storage devices.

**Example:** Program 'A' is ready to be stored in Harddisk. Now OS will look for any free space in the Harddisk and assign proper address to it. If space is not available, OS will prompt the user to empty some space.

### Network Management

Network management is the part of network operating system that monitors and manages the resources of a network. It allows to create user groups and assigns privileges to them. It shares the network resources among users and detects and fixes network problems.

### Protection System

Protection system is the part of operating system that ensures that each resource of computer is used according to the privileges given to users by the system administrator. It creates account for each user and gives privileges to prevent misuse of the system. It provides password to all the users to maintain network security.

### Command-Interpreter

Command-Interpreter is the part of operating system that provides interface between user and the computer system. It is a file in operating system that reads and executes user commands entered as text through keyboard. For example, Windows operating system uses the **cmd.exe** file as command-interpreter.

## 1.3 PROCESS MANAGEMENT

Process management is an important task of operating system. It allocates systems resources to various processes so that they can run efficiently.

### 1.3.1 PROCESS

A process is a program in execution. For example, when we write a program in C++ and compile it, the compiler creates a binary code. The original code and Binary code, both are programs. When we actually run the binary code, it becomes a process. Process is a part of program under execution that is scheduled and controlled by operating system. When a program is loaded in memory for execution, it becomes a process. A program is an executable code that is stored in disk as a text file whereas a process is a dynamic instance of a program during its execution in RAM. It represents basic unit of work. It uses various resources of computer such as CPU time, files, I/O devices, memory, etc.



### 1.3.2 VARIOUS STATES OF A PROCESS

There are five states of a process which are new, ready, running, waiting and terminated as shown in Figure 1.5.

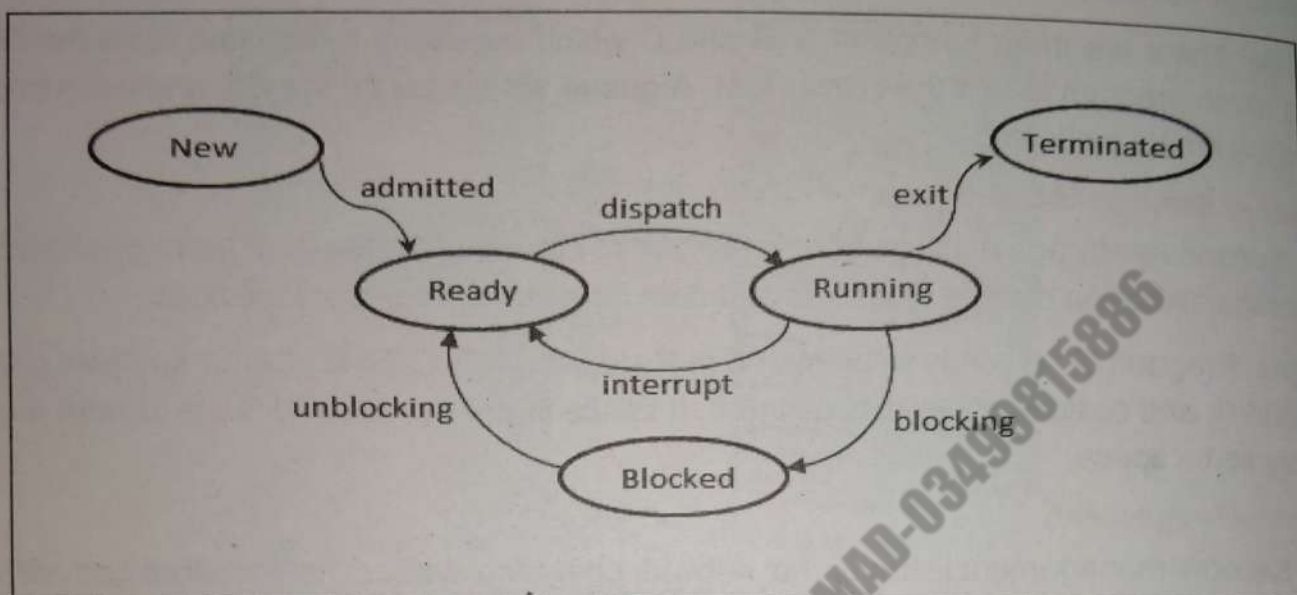


Figure 1.5 States of a Process

#### New State

This is the first state of a process when it is created. Any new operation or service that is requested by a program for execution by the processor is known as new state of process.

#### Ready State

A process is said to be in ready state when it is ready for execution but it is waiting to be assigned to the processor by the operating system.

#### Running State

A process is said to be in running state when it is being executed by the processor. A process is assigned to a processor for execution by operating system.

#### Blocked State/Waiting State

A process is in blocked or waiting state when it is not under execution. It is waiting for a resource to become available.

#### Terminated State

A process is in terminated state when it completes its execution.

### 1.3.3 THREAD AND PROCESS

In programming, there are two basic units of execution: processes and threads. They both execute a series of instructions. A **Process** is an instance of a program that is being executed. A process may be made up of multiple threads. A **Thread** is a basic ordered sequence of instructions within a process that can be executed independently. The threads are made of and



exist within a process; every process has at least one thread. Multiple threads can also exist in a process and share resources.

### Comparison between Process and Thread

	Process	Thread
1	An executing instance of a program is called a process.	A thread is a subset of the process.
2	It has its own copy of the data segment of the parent process.	It has direct access to the data segment of its process.
3	Any change in the process does not affect other processes.	Any change in the thread may affect the behavior of the other threads of the process.
4	Processes run in separate memory spaces.	Threads run in shared memory spaces.
5	Process is controlled by the operating system.	Threads are controlled by programmer in a program.
6	Processes are independent.	Threads are dependent.

### 1.3.4 MULTITHREADING

The process of executing multiple threads simultaneously is known as multithreading. Multithreading is an execution method of a program that allows a single process to run multiple threads at the same time. Multithreading allows multiple threads to exist within a single process and these threads can execute independently. The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time.

Some examples of multithreading are:

- A user is typing a paragraph on MS word. But in background one more thread is running and checking the spelling mistakes. As soon as user is doing a typing work the other thread notifies the user about the spelling mistakes.
- Web servers use multithreading all the time, every request is handled by a different thread.

### 1.3.5 MULTITASKING

Multitasking is the function of operating system that loads multiple (programs, processes, tasks, threads) in main memory and executes them at the same time by rapidly switching the



### Teacher Point

Teacher may also use presentations or animations or videos to explain the concepts of operating systems.

CPU among them. The operating system is able to keep track of where the users are in these tasks and go from one to the other without losing information. Each running task takes only a fair quantum of the CPU time.

### 1.3.6 MULTIPROGRAMMING

In multiprogramming many programs are loaded in memory but the CPU only executes one program at a time. Other programs wait until the previous program is executed out or blocked.

For example, when a user loads program 1 (say MS-Word) and program 2 (say C-language compiler). The CPU is able to execute only one program i.e. MS-Word or C-language compiler.

The advantage of multiprogramming is that it saves user's time in loading the programs to main memory and runs the programs quickly. The only drawback is, the system requires more main memory as it is occupied by many programs. Sometimes bigger programs cannot fully load in main memory and thus programs run slowly.

### 1.3.7 MULTIPROCESSING

Multiprocessing is the ability of an operating system to execute more than one process simultaneously on a multi-processor machine (having more than one CPUs). In this, a computer uses more than one CPU at a time.



#### Key Points

- An Operating System (OS) is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware.
- A batch processing operating system is a software that groups together same type of jobs in batches and automatically executes them one by one.
- A multiprogramming operating system is a software that loads one or more programs in main memory and executes them using a single CPU (Central Processing Unit).
- A multitasking operating system is a software that performs multiple tasks at the same time on a computer that has a single CPU.
- A time-sharing operating system is a software that shares the CPU time between multiple programs that are loaded in main memory. A time-sharing operating system gives a very short period of CPU time to each program one by one.
- A real-time operating system is a software that runs real-time applications that must process data as soon as it comes and provides immediate response.
- A multiprocessor operating system is a software that controls the operations of two or more CPUs within a single computer system. All the CPUs of computer share the same main memory and input/output devices.
- A parallel processing operating system is a software that executes programs developed in a parallel programming language. It uses many processors at the same time.



- A distributed operating system is a software that manages the operation of a distributed system. A distributed system allows execution of application software on different computers in a network.
- An embedded operating system is a built-in operating system which is embedded in the hardware of the device.
- The operating system that allows only one person to operate the computer at a time is known as single-user operating system.
- The operating system that allows many users on different terminals or microcomputers to use the resources of single central computer (server) in a network is known as multi-user operating system.
- Process management is the part of operating system that manages allocation of computer resources such as CPU to various programs in main memory.
- Memory management is the part of operating system that controls and manages the operation of main memory during the operation of computer.
- File management is the part of operating system that manages files and folders on storage devices such as hard disk, USB flash drive and DVD.
- I/O management is the part of operating system that controls all the input/output operations during program execution. It manages all the input/output operations of input/output and storage devices.
- Secondary storage management is the part of operating system that manages free space and storage allocation of user programs and data on secondary storage devices.
- Network management is the part of network operating system that monitors and manages the resources of a network.
- Protection system is the part of operating system that ensures that each resource of computer is used according to the privileges given to users by the system administrator.
- Command-Interpreter is the part of operating system that provides interface between user and the computer system. It is a file in operating system that reads and executes user commands entered as text through keyboard.
- Process is a part of program under execution that is scheduled and controlled by operating system.

**Exercise****Q1. Select the best answer for the following MCQs.**

- i. In which operating system, same types of jobs are grouped together and executed one by one?
  - a) Multiprogramming operating system
  - b) Batch processing operating system
  - c) Real-time operating system
  - d) Time-sharing operating system

- ii. In \_\_\_\_\_ operating system CPU is rapidly switched between programs so that all the programs are executed at the same time.
- a) Multiprogramming operating system
  - b) Batch processing operating system
  - c) Real-time operating system
  - d) Time-sharing operating system
- iii. Which operating system runs applications with very precise timing and provides immediate response to avoid safety hazards?
- a) Real-time operating system
  - b) Multitasking operating system
  - c) Multiprocessing operating system
  - d) Distributed operating system
- iv. \_\_\_\_\_ operating system divides a task into many subtasks and processes them independently using many processors.
- a) Real-time operating system
  - b) Distributed operating system
  - c) Parallel processing operating system
  - d) Multitasking operating system
- v. Which operating system is used in home appliances?
- a) Time-sharing operating system
  - b) Distributed operating system
  - c) Parallel processing operating system
  - d) Embedded operating system
- vi. Which of the following manages allocation of computer resources during program execution?
- a) Memory management
  - b) Process management
  - c) I/O management
  - d) File management
- vii. Which of the following creates user groups and assigns privileges to them?
- a) Process management
  - b) I/O management
  - c) File management
  - d) Network management
- viii. In which state, a process is waiting to be assigned to the processor by the operating system scheduler?
- a) New state
  - b) Ready state
  - c) Waiting state
  - d) Running state



**Q2. Write short answers of the following questions.**

- i. What is the purpose of operating system in a computer?
- ii. What is Graphical User Interface (GUI)?
- iii. Mention three advantages of UNIX operating system.
- iv. Differentiate between multiprogramming and time-sharing operating systems.
- v. Why multiprocessing operating systems have been developed?
- vi. Differentiate between single-user and multiuser operating system.
- vii. Why memory management is required in a computer?
- viii. Why protection system is required in a computer?
- ix. What is a thread?
- x. Differentiate between multiprogramming and multithreading by giving one example of each.

**Q3. Write long answers of the following questions.**

- i. Mention the tasks performed by operating system.
- ii. Compare DOS with Windows operating system.
- iii. Describe the following types of operating systems.
  - a. Real-time operating system
  - b. Parallel processing operating system
  - c. Embedded operating system
- iv. Define the following terms.
  - a. File management
  - b. I/O management
  - c. Network management
  - d. Command-Interpreter
- v. Describe the five states of process with diagram.



**Lab Activities**

1. Find out which type of operating systems are installed in your computer lab.
2. Observe the installation procedure of common types of operating system through animation/video.
3. If you have visited any organization to see the working of different types of operating systems then prepare a report on it.



# 2

## SYSTEM DEVELOPMENT LIFE CYCLE



After completing this lesson, you 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 SDLC:
  - Management in SDLC
  - Project manager
  - System analyst
  - Programmer
  - Software tester
  - Customer



## 2.1 SYSTEM DEVELOPMENT LIFE CYCLE

The **Systems Development Life Cycle (SDLC)**, in Software Engineering, is the process of creating or altering information systems. In other words these are the models and methodologies that experts use to develop these systems. Software engineering is an engineering approach for software development. In software engineering the SDLC concept reinforces many kinds of software development techniques. These techniques form the framework for planning and controlling the creation of an information system.

### 2.1.1 A System

A system is a set of components (hardware and software) for collecting, creating, storing, processing, and distributing information.

A system can be developed by applying a set of methods, procedures and routines in a proper sequence to carry out some specific task.

### 2.1.2 System development life cycle (SDLC) and its importance

System developing life cycle (SDLC) is a problem-solving process through which a series of steps or phases helps to produce a new information processing system or software.

#### Importance of System Development Life Cycle

The basic purpose of System development life cycle is to develop a system in a systematic way in the perfect manner.

- It delivers quality software which meet the system requirements.
- It ensures that the requirements for the development of the software/system are well defined and subsequently satisfied.
- It delivers cost-effective system.
- It maximizes the productivity.

### 2.1.3 Objectives of SDLC

A systems development lifecycle (SDLC) has three primary objectives:

- Ensure that high quality systems are delivered
- Provide strong management controls over the projects
- Maximize the productivity

These objectives are summarized as follows:



#### Teacher Point

Before starting the chapter, the students could be asked few questions about the term "System" to explain what they understand about it.

- 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 identify the potential project risks in advance so that proper planning should be done in early.

### 2.1.4 Stakeholders of SDLC

**Stakeholders** of SDLC are those entities or groups which are either within the organization or outside of the organization that sponsor, plan, develop or use a project. Stakeholders may be users, managers and developers. 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.

### 2.1.5 SDLC PHASES/STEPS

The following are phases/steps in SDLC. These phases are shown in Figure 2.1.

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

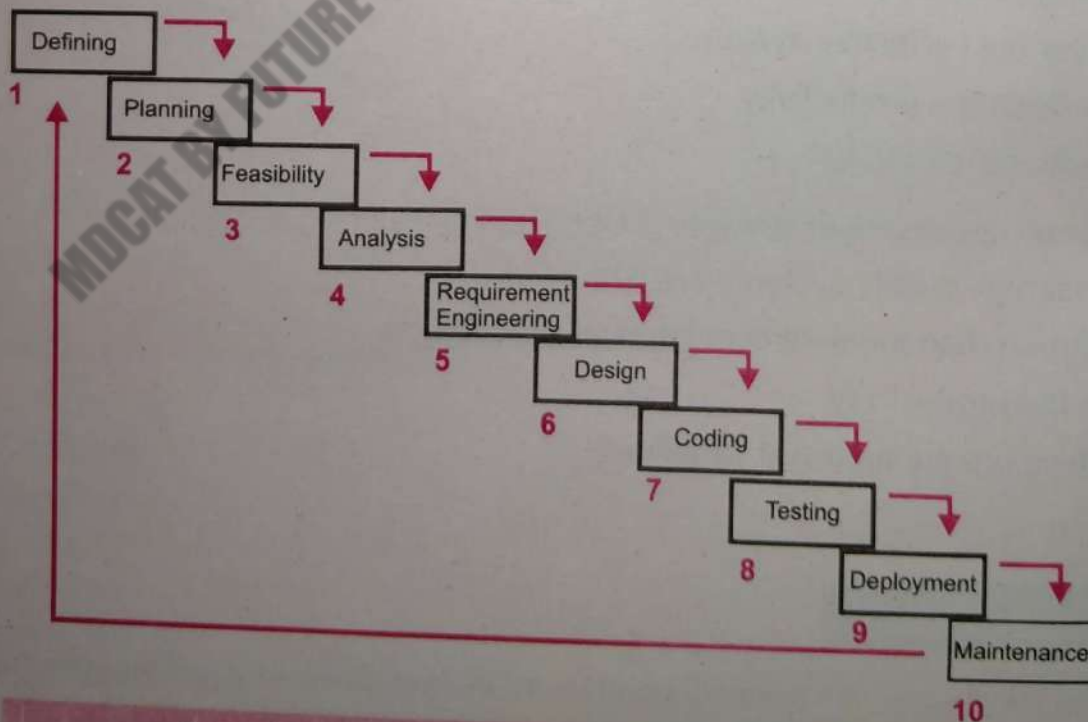


Figure 2.1: Phases of System Development Life Cycle



## 1. DEFINING PHASE

In this phase the problem to be solved or system to be developed is clearly defined. All the requirements are documented and approved from the customer or the company which consists of all the product requirements to be designed and developed during the development life cycle.

### Example: Students' Examination System Development

**Defining the problem:** A Students' Examination System is needed to be developed that covers all the aspects from Examination taking to the Students' results generations.

## 2. PLANNING PHASE

During the planning phase, the objective of the project is determined and the requirements to produce the product are considered. An estimate of resources, such as personnel and costs, is prepared, along with a concept for the new product. All of the information is analyzed to see if there is an alternative solution to creating a new product. If there is no other viable alternative, the information is assembled into a project plan and presented to management for approval.

**Example:** In the **Students' Examination System Development** project planning will be made to set the ultimate goals and an estimate of resources, such as personnel and costs, is prepared.

## 3. FEASIBILITY

Feasibility study is used to assess the strengths and weaknesses of a proposed software/system and present directions of activities which will improve a project and achieve desired results. The nature and components of feasibility studies depend primarily on the areas in which analyzed projects are implemented.

Feasibility study is the analysis and evaluation of a proposed project/system, to determine, whether it is technically, financially/economically, legally and operationally feasible within the estimated cost and time. Feasibility study is one of the important steps in SDLC. It is divided into the following types/forms.

- Technical feasibility
- Economic feasibility
- Operational feasibility
- Legal feasibility
- Schedule feasibility

**Example:** The **Students' Examination System Development** project is assessed for all types of feasibilities and presented to management for final approval.

#### 4. ANALYSIS PHASE

During the analysis phase the project team determines the end-user requirements. Often this is done with the assistance of client focus groups, which provide an explanation of their needs and what their expectations are for the new system and how it will perform.

In this phase, the in-charge of the project team must decide whether 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 or 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 etc.?

**Example:** The **Students' Examination System Development** project is analysed for development. The project team will visit the School/College to study the existing system and will suggest the possible improvements.

#### 5. REQUIREMENT ENGINEERING

It is the process of determining user expectations for a new or modified system/software. Requirements engineering is a set of activities used to identify and communicate the purpose of a software system, and the framework in which it will be used. Requirement engineering consists of the following steps.

- Requirement gathering
- Requirement validation
- Requirements management

##### i. Requirement Gathering

Requirement gathering is usually the first part of any software/system development process. In this, meetings with the customers are arranged, the market requirements and features that are in demand are analyzed.

These requirements are of two types:

- Functional requirements
- Non-Functional Requirements

**Functional requirements:** Functional requirements specify the software functionality that the developers must build into the product to enable users to accomplish their tasks.

**Non-functional requirements:** Non-functional requirements specify criteria for the judgment of the operations of a system. It describes that how well the system performs its duties.

##### ii. Requirements Validation

Requirement validation is concerned with examining the requirements to certify that they meet the intentions of 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 and accurate.

### iii. Requirements Management

Requirements management is performed to ensure that the software continues to meet the expectations of the acquirer and users. Requirements management needs to gather new requirements that arise from changing expectations, new regulations, or other sources of change.

**Example:** In the **Students' Examination System Development** project, the development team will gather the necessary information by 'Interviewing the users', 'Giving questionnaire' or 'by reading existing documents'.

## 6. DESIGN PHASE

The design phase is the "architectural" phase of system design. The flow of data processing is developed into charts, and the project team determines the most logical design and structure for data flow and storage. For the user interface, the project team designs mock-up screen layouts that the developers use to write the code for the actual interface.

The design phase normally consists of two different structures. These are:

- Algorithms
- Flow chart

### i. Algorithms

An algorithm is a specific step by step procedure for carrying out the solution of a problem.

**Example:** In the **Students' Examination System Development** project the following algorithm will find the result of a student on percentage marks.

1. **Start**
2. **Read Marks**
3. **If Marks  $\geq 40$  Then Print "Pass" Else Print "Fail"**
4. **End**

### 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 symbols and their order by arrows connecting the symbols. This diagrammatic representation of the algorithm gives a step-by-step solution to a given problem. The operations are represented in these symbols, 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 2.2.

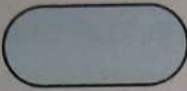

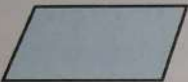
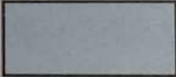

Symbol	Name	Function
	Start / Stop	An oval represents Start / Stop of flowchart
	Arrows	A line with arrow head connects the symbols and shows the direction of flow
	Input / Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Figure 2.2 Flowchart Symbols

**Example:** In the **Students' Examination System Development** project, the Flowchart for the above algorithm will be as follows.

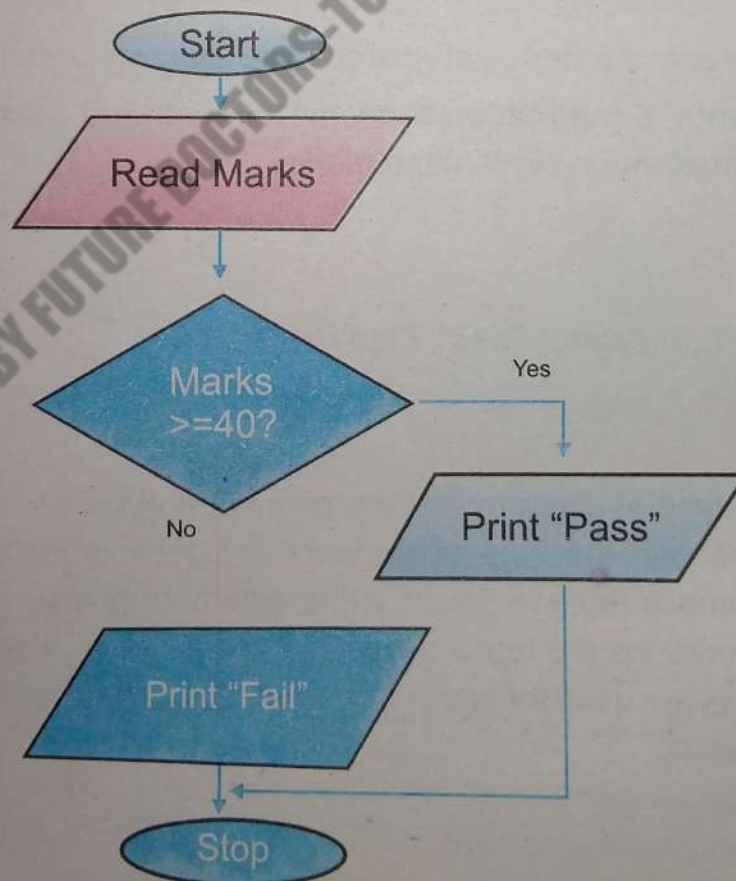


Figure 2.3 Flowchart



## 7. CONSTRUCTION OR CODING OR DEVELOPMENT PHASE

During the construction phase developers execute the plans laid out in the design phase. The developers design the database, generate the code for the data flow process and design the actual user interface screens. During the construction phase, test data is prepared and processed as many times as necessary to refine the code. This code is written in programming languages. Coding is also called computer programming.

**Example:** In the **Students' Examination System Development** project, the coding style of the C++ programming language is shown as follows.

```
/* 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 >=40.0)
cout<< "Pass";
else
cout<< "Fail";
getch();
return 0;
}
```

## 8. TESTING OR VERIFICATION PHASE

During the test phase all aspects of the system are tested for functionality and performance. The execution of a programming modules to find errors is called testing. Here, the bugs are identified in the programmed modules. The process of removing bugs/errors from the program is called debugging.

The main purpose of testing/verification of the system is to determine that whether it meets its required results or not. 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.

**Example:** In the **Students' Examination System Development** project the above programming module is tested for errors. The result of the program module is given as follows.

**Test 1:**

Enter Percentage of the student

70.0

Pass

**Test 2:**

Enter Percentage of the student

20.0

Fail

## 9. 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.

The main activities that are involved during deployment/implementation phase are:

- Installation and activation of the hardware 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.

### Deployment/Implementation Methods

The following four methods/techniques are used for system deployment/implementation. (Figure 2.4)

- Direct Implementation:** This method involves the old system being completely dropped and the new system being completely implemented at the same time. The old system is no longer available.
- Parallel:** The parallel method of implementation involves operating both systems together for a period. This allows any major problems with the new system to be encountered and corrected without the loss of data.
- Phased:** The phased method of implementing from an old system to a new system involves a gradual introduction of the new system. The old system is progressively discarded.



- iv. **Pilot:** With the Pilot method of implementation, the new system is installed for a small number of users. These users learn, use and evaluate the new system. Once the new system is deemed to be performing satisfactorily then the system is installed and used by all.

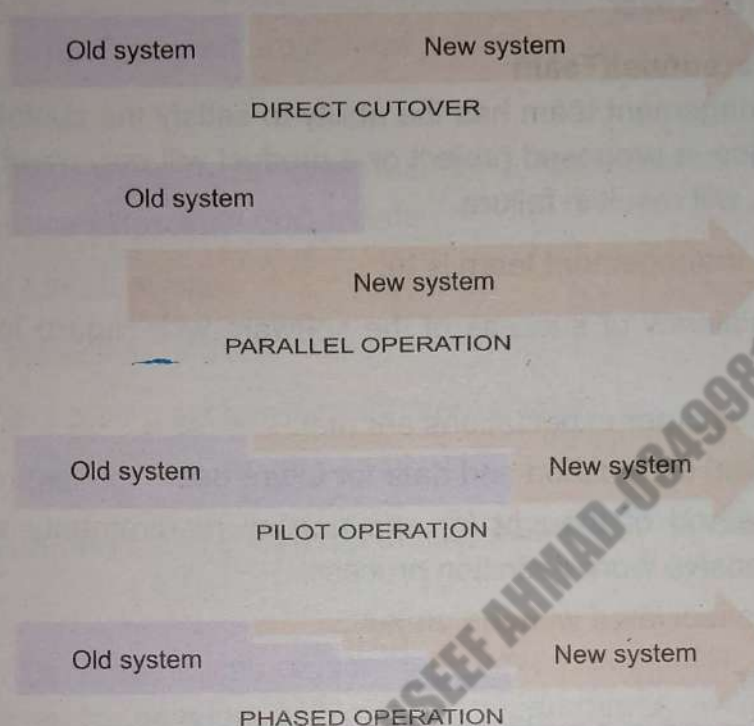


Figure 2.4 Deployment/Implementation

## 10. MAINTENANCE / SUPPORT

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 Personnel involved in SDLC and their Role

SDLC activities are performed by different groups of people and individuals called personnel. These personnel are professionals in performing their particular jobs. These include:

- Management Personnel
- Project Manager



### Teacher Point

Take one system, (other than the computer system), and explain how all the steps of SDLC could be applied to develop this system.

- System Analyst
- Programmer
- Software Tester
- Customer or End user

#### a. Management Personnel/Team

A strong management team 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 roles of a good management team 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.

#### b. Project Manager

A project manager is a professional 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/She is also expected to be familiar with the whole software development life cycle process. The key roles of a project manager are:

- Developing the project plan
- Managing the project budget
- Managing the project stakeholders
- Managing the project team
- Managing the project risk
- Managing the project schedule
- Managing the project conflicts



#### Teacher Point

Teacher may also use presentations or animations or videos to explain the steps of SDLC. Take any computer system as example.



### 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. The general roles and responsibilities of an analyst are defined below.

- i. Plan a system flow.
- ii. Interact with customers to learn and document requirements that are then used to produce business requirements documents.
- iii. Define technical requirements.
- iv. Interact with designers to understand software limitations.
- v. Help programmers during system development phase.
- vi. Manage system testing.
- vii. Document requirements and contribute to user manuals.

### 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. The responsibilities of a programmer includes:

- i. Writing, testing, and maintaining the instructions of computer programs.
- ii. Updating, modifying and expanding existing programs.
- iii. Testing the code by running to ensure its correctness.
- iv. 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.



### Teacher Point

Teacher should give some home assignments to the students at the end of the chapter.



f. Customer or End user

A Customer is an individual or a company which buys and uses the system. 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.



Key Points

- The Systems Development Life Cycle (**SDLC**), is the process of creating or altering information systems, and the models and methodologies that experts use to develop these systems.
- The term '**System**' is a Greek word meaning 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.
- The basic purpose of System development life cycle is to develop a system in a systematic way in the perfect manner.
- A systems development life cycle (SDLC) has three primary objectives: ensure that high quality systems are delivered, provide strong management controls over the projects, and maximize the productivity.
- **Stakeholders** of SDLC are those entities or groups which are either within the organization or outside of the organization that sponsor, plan, develop or use a project.
- The **planning phase** of SDLC determines the objective of the project and considers the requirements to produce the product.
- **Feasibility study**, in SDLC, is used to assess the strengths and weaknesses of a proposed project and present directions of activities which will improve a project and achieve desired results.
- During the **analysis phase** of SDLC the project team determines the end-user requirements.
- In SDLC, **requirements engineering**, also called requirements analysis is the process of determining user expectations for a new or modified system/software.
- The **design phase**, in SDLC, is the "architectural" phase of system design. The flow of data processing is developed into charts, and the project team determines the most logical design and structure for data flow and storage.



- An **algorithm** is a specific set of instructions for carrying out a procedure or solving a problem.
- A **flowchart** is a type of diagram that represents an algorithm or a process.
- During the **construction phase** of SDLC, developers execute the plans laid out in the design phase.
- The **test phase** of SDLC is used to test the functionality and performance of the system/program.
- **Software deployment**, in SDLC, is a set of activities that are used to make the software system available for use.
- In SDLC, keeping a system in its proper working condition is called **maintenance**.
- '**Management**' is the organization, coordination and controlling the activities of a software development by the managers and executives in accordance with certain standard procedures.
- A **project manager** is a professional responsible for planning, execution, and closing of any project.
- 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.
- A **programmer** is a technical person that writes computer programs in computer programming languages to develop software.
- A **software tester** is a computer programmer having specialty in testing the computer programs using different testing techniques.
- A **customer** is an individual or an organization that is a current or potential buyer or user of the software product.



## Exercise

**Q1. Select the best answer for the following MCQs.**

- 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) All of the above
  - d) User's requests for additional features

**Q2. Write short answers of the following questions.**

- i. What is a system?
- ii. Name different phases of SDLC.
- iii. What are the objectives of SDLC?
- iv. Give some activities of planning phase.
- v. Differentiate between functional and non-functional requirements.



- vi. Design phase is considered as the "architectural" phase of SDLC. Give reasons.
- vii. Explain flowchart symbols.
- viii. What is the purpose of Testing/verification phase of SDLC?
- ix. Give main activities of the Implementation phase.

### Q3. Write long answers of the following questions.

- i. Define software development life cycle (SDLC). What are its objectives?
- ii. What is system? Where exactly the Testing activities begin in SDLC?
- iii. Why software development life cycle is important for the development of software?
- iv. Who are stakeholders of SDLC? Describe their responsibilities.
- v. What is feasibility study? Explain its different types.
- vi. What is the purpose of Requirement Engineering phase? Explain its various steps in detail.
- vii. Which personnel are involved in SDLC? Explain their role briefly.



### Lab Activities

Prepare a chart to show all the phases of SDLC.



# 3

## OBJECT ORIENTED PROGRAMMING IN C++



After completing this lesson, you will be able to:

- Define program
- Define header file and reserved words
- Describe the structure of C++ program
- Know the use of statement terminator
- Explain the purpose of comments and their syntax
- Explain the difference between constant and variable
- Explain the rules for specifying variable names
- Know data types used in C++
- Define constant qualifier – const
- Explain the process of declaring and initializing variables
- Use type casting
- Explain the use of cout statement
- Explain the use of cin statement
- Define getch(), gets() and puts() functions
- Define escape sequence
- Use of escape sequence in programs
- Use I/O handling functions
- Use manipulators endl and setw
- Define operators and know their use in programs
- Use unary, binary and ternary operators in programs
- Define expression
- Define and explain the order of precedence of operators
- Define and explain compound expression



## 3.1 INTRODUCTION

C++ is a general purpose programming language that supports various computer programming models such as object-oriented programming (oops) and generic programming. It was created by Bjarne Stroustrup in early 1980s at Bell Laboratories. Its main purpose was to make writing good programs easier and more pleasant for the individual programmer.

C++ supports modern programming techniques. It is commonly used for developing high performance commercial software, games and graphics related programs.

### 3.1.1 COMPUTER PROGRAM

A computer program is a set of instructions that performs a specific task when executed by a computer. It tells the computer what to do. Everything a computer does is controlled by computer program. For example, Microsoft Word is a program that allows computer users to create documents and Skype is a program used to make calls free of charge to other people who are on Skype. Generally, programs are written in English oriented high level languages such as Visual Basic, Pascal, Java, C++, etc. A computer can only understand instruction in machine language which consists of 0s and 1s. Therefore, a program written in a high level language must be translated into machine language before execution. This task is achieved by software known as **compiler**. A program written in a high language is called **source code** and its equivalent program in machine language is called **object code**.

### 3.1.2 HEADER FILE AND RESERVED WORDS

#### Header File

The C++ contains many header files. Header files contain information that is required by the program in which these are used. It has .h extension. Some examples of header files are *iostream.h*, *conio.h* and *math.h*. These files are included in the standard library of C++ compiler.

**Preprocessor directive** is used to include a header file at the beginning of program. Preprocessor directive is not a normal program instruction to be executed by the CPU. It is a code for the compiler to include a header file.

The syntax of preprocessor directive to include a header file in a program is:

```
# include <name of header file>
```

It begins with a # sign, followed by the word *include* and then the name of header file is written within angled brackets.

For example, to include the header file *iostream.h* in a program, the code is

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



#### Teacher Point

Before starting the chapter, the students could be asked few questions about the "Programming Language" to explain what they understand about it.

The effect of this line is to copy all the contents of *iostream.h* header file into the program and make them available for use. Almost all the C++ programs perform input/output operations. Therefore, generally this header file is included in the programs.

The following is another example for including *math.h* header file in a program.

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

The *math.h* header file contains code for performing mathematical functions such as finding the square root of a number.

### Reserved Words

Reserved words are special words which are reserved by a programming language for specific purpose in program. These cannot be used as a variable names. Some examples of reserved words of C++ are *if*, *void*, *break*, *while*, *case* and *char*. All the reserved words are written in lower-case letters. There are about 80 reserved words in C++ but this may vary depending on the version being used.

The reserved words of C++ are:

and	and_eq	asm	auto	bitand
bitor	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	not	not_eq	operator
or	or_eq	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	xor	xor_eq		

### 3.1.3 STRUCTURE OF C++ PROGRAM

A C++ program has the following structure.

preprocessor directives



#### Teacher Point

Teacher should recommend a suitable C++ compiler to the students. All the parts of the compiler editor should also be explained.



```
void main()
{
    body of main() function
}
```

A C++ program starts with preprocessor directives, followed by the line **void main()** function and then the body of the *main()* function is written within curly brackets ({ and }). Body of *main()* function consists of executable statements. These statements perform a specific task when the program is executed by the CPU. There is no restriction on the number of statement that can be written in the body of *main()* function.

To understand the structure of program, consider the following program.

```
#include <iostream.h>
void main()
{
    cout<<"Information Technology";
}
```

### Explanation:

**#include<iostream.h>**

This is the first line of the program and it is a preprocessor directive. This program prints a message on the screen. Therefore, *iostream.h* header file is included which contains the code for performing input/output operations.

**void main()**

This line identifies the main function of the C++ program. All C++ programs must have *main()* function. If a program consists of more than one function, the location of *main()* does not matter. In C++ program the *main()* function is always executed first.

**cout<<"Information Technology";**

This is a C++ statement. The meaning of this statement is to print the message "Information Technology" on the screen. This statement is written within the curly brackets and is called body of the *main()* function.

### 3.1.4 STATEMENT TERMINATOR (;)

In C++, semicolon is a statement terminator. It marks the end of a statement. All the C++ statements must end with a semicolon.

The following statement was used in the previous program, notice that it ends with a semicolon.



### Teacher Point

Demonstrate how a program can be written, saved, compiled and executed in C++.



```
cout<<"Information Technology";
```

If ';' is missing the compiler will give syntax error number and also the message that ';' is missing. The error number and message may vary depending on the compiler used.

### 3.1.5 COMMENTS IN C++ PROGRAM

All the programming languages allow comments in programs. Comments are explanatory statements that help the reader in understanding source code. Comments can be entered at any location in the program. Comments are ignored during program execution which means they are not executable statements.

There are two types of comments in C++. These are single-line comments and multiple-line comments.

#### The Single-line Comments (//)

The single-line comments start with // (double slash) and continue until the end of the line.

The following program demonstrates the use of single-line comments.

```
// This is a very simple C++ program.
#include <iostream.h>
void main()
{
    cout<<"Information Technology"; // It prints a message on the screen.
}
```

#### The Multiple-line Comments (/\* and \*/)

This type of comment is used for entering multiple line comments in a program. The /\* is used at the beginning of comments and \*/ ends it.

The following program demonstrates the use of multiple-line comments.

```
/* This is my first C++ program.
   It demonstrates the
   structure of C++ program.
*/
#include <iostream.h>
void main()
{
    cout<<"Information Technology";
}
```

The /\* and \*/ type comments can also be used to enter comments on a single line as shown below.

```
/* This is my first C++ program. */
```



## 3.2 C++ CONSTANTS AND VARIABLES

Constants and variables are used in programs to perform calculations of various types. Therefore, it is important to understand how they are used in computer programs.

### 3.2.1 CONSTANTS AND VARIABLES

#### Constant

In computer programming, a constant is a value that does not change during execution of program. A constant can be a number, a character or a character string. A character string is a sequence of any number of characters.

Some examples of constants are 42, 7.25, 's' and "Computer" respectively. In C++, a single character constant is written within single quotes and a string constant within double quotes.

#### Variable

A variable is a name of memory location where data is stored. Variables are used in computer programs to store values of different data types. The data stored in a variable may change during program execution.

### 3.2.2 RULES FOR SPECIFYING VARIABLE NAMES

The following are the rules for naming a variable.

- i. The first character of a variable name must be alphabet or underscore.
- ii. The characters allowed in a variable name are:
  - Underscore ( \_ )
  - Digits (0 to 9)
  - Upper-case letters (A to Z)
  - Lower-case letters (a to z)

An upper-case letter is considered different from a lower-case letter. For example, the variable *SUM* is different from *Sum* or *sum*. The underscore is generally used to improve readability. For example, the variable **overtime** may also be written as **over\_time**.

- iii. Special symbols such as \$, @, %, #, etc. are not allowed.
- iv. Blank space or comma is not allowed.
- v. Reserved words of C++ are not allowed to be used as a variable name.



#### Teacher Point

Teacher should also explain few more related programs according to the chapter topics.

### 3.2.3 C++ DATA TYPES

Data types are declarations of variables for storing various types of data. Data types have different storage capacities. In C++ programs data type of a variable must be defined before assigning it a value.

The data types used in C++ programming are integer, floating-point, double precision and character.

#### Integer

It is a data type that is used to define numeric variables to store whole numbers, such as, -3, 0, 367, +2081, etc. Integers represent values that are counted, such as number of students in a class. The short form of integer is *int*. Numbers that have fractional part, such as, 3.84 cannot be stored in an integer variable.

The following table shows the integer types, the number of bytes it takes in memory to store the value and the range of numbers it can store. It is for 32-bit word size.

Integer Type	No. of Bytes	Range of Numbers
<b>int</b>	4 bytes	-2147483648 to 2147483647
<b>unsigned int</b>	4 bytes	0 to 4294967295
<b>short int</b>	2 bytes	-32768 to 32767
<b>unsigned short int</b>	2 bytes	0 to 65535
<b>long int</b>	4 bytes	-2147483648 to 2147483647
<b>unsigned long int</b>	4 bytes	0 to 4294967295

#### Floating-point

It is a data type that is used to define variables that can store numbers that have fractional part such as 3.75, -2.1, 388.80, etc. These numbers are also known as real numbers. The short form of floating-point is *float*.

The following table shows the floating-point types, the number of bytes it takes in memory to store the value and the range of real numbers it can store.

Float Type	No. of Bytes	Range of Numbers
<b>float</b>	4 bytes	$-3.4^{38}$ to $3.4^{38}$ (with 6 digits of precision)
<b>double</b>	8 bytes	$-1.7^{308}$ to $1.7^{308}$ (with 15 digits of precision)

The *float* type variables might occupy different number of bytes and their range might also be different depending on the computer and the compiler being used.

#### Character

It is a data type that is used to define variables that can store only a single character. One byte of memory is set aside in memory to store a single character. The short form of character is *char*. A variable of type *char* can store a lower-case letter, an upper-case letter, a digit or a special character. Some examples of characters that can be stored in a variable of type *char* are 'a', '+', '%' and '5'. Note that characters are written within single quotation marks.



### 3.2.4 THE CONSTANT QUALIFIER (const)

In C++ programming language, *const* defines a variable whose value cannot be changed throughout the program. When the *const* qualifier is used with a variable, it no longer remains a variable because its value will not be changed. A variable defined with the *const* qualifier must be assigned some value.

It has the following syntax.

```
const data_type variable = constant;
```

For example,

```
const int AGE = 34;
```

```
const float LENGTH = 7.5;
```

The variables of type *const* are generally written in upper-case letters.

### 3.2.5 VARIABLE DECLARATION AND INITIALIZATION

In C++, all the variables that are going to be used in a program must be declared before use. Declaring a variable means specifying the data type of a variable. It allows the compiler to decide how many bytes should be set aside in memory for storage of value that is going to be assigned to the variable in the program.

The following are some examples of declaring variables.

```
int x,y,z;
```

```
float length,breadth,sum;
```

```
char ch;
```

Here the variables, *x*, *y* and *z* are declared as of type *int*, *length* and *breadth* as of type *float* and *ch* as of type *char*.

A variable may be initialized at the beginning of a program when it is declared. Initializing a variable means assigning it an initial value.

The following are some examples of initializing variables in declaration statements.

```
int x=4,y=5,z;
```

```
float length=12.5,breadth=15.25,sum=0.0;
```

```
char ch='a';
```

In the first declaration statement, the variable *x* is initialized to integer constant 4 and *y* to 5. The second statement initializes *length* to 12.5, *breadth* to 15.25 and *sum* to 0.0. The last statement initialized the character variable *ch* to 'a'.

### 3.2.6 TYPE CASTING

Type casting is used in C++ to convert data type from one type to another. There are two types of type casting, implicit and explicit type casting.

#### Implicit Type Casting

Implicit type casting automatically converts a data type to another. This is explained by the following example.

Suppose variable *q* is declared as of type *float* and the following calculation is to be performed.

```
q = 15/6;
```

When this integer division is performed, the result will also become an integer value 2 which will be implicitly converted to floating-point value 2.0 and assigned to the variable *q*.

If one or both of the integer constants are converted to floating-point constant (14.0 or 6.0), this will perform division using floating-point mathematics. In this case, the result produced will be 2.5 and it will be assigned to *q*.

### Explicit Type Casting

In explicit type casting, a special operator is used to convert one data type into another.

The general form for conversion is

(type) expression

Here, expression can be an arithmetic expression or a variable. This is explained by the following example.

Suppose, *a* and *b* are variables of type *int* and *q* is of type *float*. The integer value 15 is stored in *a* and 6 in *b* and the following division is to be performed.

```
q = a/b;
```

When this division is performed, integer math will be used and the result produced will be 2 which will be assigned to the variable *q*.

To obtain correct result, type casting should be used to convert at least one of the operands to type *float* as shown below.

```
q = (float)a/b;
```

Now, first the value stored in *a* will be converted to type *float* and then the division will be performed. The floating-point math will be used and the correct result 2.5 will be produced which will be assigned to *q*.

Similarly, the *int* type can also be used to convert a floating-point value stored in a floating-point variable into integer type by truncating the fractional part of the number.

## 3.3 INPUT/OUTPUT HANDLING

In computer programming, providing data into a program from outside source is known as input and output means to display some data on screen or save in a file on a storage device.

In C++, input/output (I/O) is performed by using streams. A stream is a sequence of data that flows in and out of the program. The ***cin*** and ***cout*** are the standard statements that use the *iostream.h* header file for performing I/O operations. Therefore, it must be included at the beginning of program.

The functions *getch()*, *gets()* and *puts()* are also used for handling input/output operations.



### 3.3.1 THE **cout** STATEMENT

The *cout* statement is used to output text or values on the screen.

The following is the syntax of *cout* statement.

**cout<<character string/variable;**

The keyword *cout* is used with insertion operator on its right side which is two less than signs (<<), followed by a character string or a variable. The insertion operator displays the contents of the character string or the value stored in the variable on the screen. It directs the output to the standard output device which is monitor. Note that the statement ends with semicolon. The insertion operator may be used more than once in a single statement.

The following program demonstrates the use of **cout** statement.

```
#include <iostream.h>           // this header file is used for I/O functions
void main()
{
    int a;
    a=12;
    cout<<"The value stored in a is "<<a;
}
```

The output of the program will be

**The value stored in a is 12**

The first output statement will print the text and the second will print the value stored in variable *a*. The output of the two *cout* statements will appear on the same line.

### 3.3.2 THE **cin** STATEMENT

The *cin* statement is used to input data from the keyboard and assign it to one or more variables.

The following is the syntax of *cin* statement.

**cin>>variable;**

The keyword *cin* is used with the extraction operator on the right side which is two greater than signs (>>), followed by a variable. When this input statement is executed, it causes the program to wait for the user to input data. The data entered by the user is assigned to the variable.

The following program demonstrates the use of *cin* statement.

```
#include <iostream.h>
void main()
{
    int n;
```

```

    cout<<"Enter an integer:";
    cin>>n;
    cout<<"The number you typed is "<<n;
}

```

When this program is executed, the first *cout* statement prompts the user to enter an integer. The *cin* statement causes the typed number to be assigned to variable *n*. The last statement prints the number stored in *n* on the screen.

The execution of program is shown below.

Enter an integer:7

The number you typed is 7

More than one extraction operator can be used in a single *cin* statement to input more than one data values as shown below.

```
cin>>a>>b;
```

The values for variables *a* and *b* should be input with space between them.

### 3.3.3 THE *getch()*, *getche()*, *gets()* and *puts()* FUNCTIONS

These three functions are also used for handling I/O operations.

#### The *getch()* Function

In some situations in programming, it is required to read a single character the instant it is typed without waiting for Enter key to be pressed. For example, in a game we might want an object to move each time we press one of the arrow keys. It would be awkward to press the Enter key each time we pressed an arrow key.

The *getch()* function is used for this purpose. The *get* means it gets something from an input device and *ch* means it gets a character. This function uses the *conio.h* header file. Therefore, it must be included in the program.

The following program demonstrates the use of *getch()* function to read a character from the keyboard and displays it on the screen.

```

#include<iostream.h>
#include<conio.h>           // this header file is used for console (screen) input/output
void main()
{
    char ch;
    cout<<"Enter a character:";
    ch=getch;
    cout<<"The character you typed is "<<ch;
}

```

The execution of the program is shown below with the input character *a*.

Enter a character:



The character you typed is a

Note that, when this function is used, the input character *a* is read and stored in variable *ch* but it is not displayed on the screen.

### The *getche()* Function

If the user wants the typed character to be displayed on the screen then another similar function *getche()* is to be used. In this function the letter *e* is added which means to echo or display the input character on the screen.

When executing a program, some C++ compilers display the output for a very short time and immediately returns to the editing window. The user is not able to see the program output. Therefore, very often, *getch()* function is used at the end of the program so that the user is able to see the program output and has to enter any character to return to the editing window. Since the character entered is not used in program, there is no need to store it in a variable. This is shown below.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    char ch;
    cout<<"Enter a character:";
    ch=getche();
    cout<<"The character you typed is "<<ch;
}
```

The execution of the program is shown below with the input character 'h'.

**Enter a character:**

**The character you typed is h**

Note that, when this function is used, the input character 'h' is read and stored in variable *ch* and displayed on the screen.

### The *gets()* and *puts()* Functions

These functions are used to handle input/output of character strings. The *gets()* function is used to input a string from the keyboard and the *puts()* is used to display it on the screen. In C++ strings are handled as arrays. These functions will be explained in Chapter 5 in which arrays are discussed.

### 3.3.4 THE ESCAPE SEQUENCE

Escape sequences are special characters used to control the output look on output devices. These characters are usually not printed. These are used inside the output statement.

An escape sequence begins with a backslash (\) followed by a code character. These are called 'escape sequences' because the backslash causes an 'escape' from the normal way characters are interpreted in C++ programming language. Escape sequences are used for

special purposes in programming such as to begin printing on the next line, to issue a tab, to print special characters, etc.

### 3.3.5 COMMONLY USED ESCAPE SEQUENCES

The commonly used escape sequences are `\a`, `\b`, `\n`, `\r`, `\t`, `\\`, `'` and `"`.

Suppose we want to print the following message on the screen.

```
cout<< "There are many versions of";
```

```
cout<< "Windows operating system.";
```

When the above two statements are executed, the output will appear on a single line as shown below.

There are many versions of Windows operating system.

If it is desired to display the output in two lines then `\n` escape sequence can be used in various ways to move cursor to the beginning of next line.

```
cout<< "\nThere are many versions of";
```

```
cout<< "\nWindows operating system.";
```

The same output can also be obtained by the following two statements.

```
cout<< "\nThere are many versions of\n";
```

```
cout<< "Windows operating system.";
```

A single output statement can also be used.

```
cout<< "\nThere are many versions of\nWindows operating system.";
```

Similarly, the `\t` escape sequence can also be used to tab over eight characters as shown in the following statement.

```
cout<< "C++\tis\ta\thigh\tlevel\tlanguage";
```

The output of this statement will be

C++ is a high level language

A list of commonly used escape sequences is given in the following table with their meanings.

Escape Sequence	Meaning
<code>\a</code>	Produces alert (beep) sound
<code>\b</code>	Moves cursor backward by one position
<code>\n</code>	Moves cursor to the beginning of next line
<code>\r</code>	Moves cursor to the beginning of current line
<code>\t</code>	Moves cursor to the next horizontal tabular position
<code>\\</code>	Produces a backslash
<code>'</code>	Produces a single quote
<code>"</code>	Produces a double quote



The following statement uses the `\` escape sequence to print the word "Pakistan" in double quotation marks.

```
cout<<"This will print the word \"Pakistan\" in double quotation marks.";
```

The escape sequences `\` and `\\` are also used in the same way to print a single quote or a backslash.

### 3.3.6 PROGRAMMING WITH I/O HANDLING FUNCTIONS

**Program 1:** The following program reads three integers and prints their sum and product.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int x,y,z,sum,prod;
    cout<<"\nEnter first number:";
    cin>>x;
    cout<<"\nEnter second number:";
    cin>>y;
    cout<<"\nEnter third number:";
    cin>>z;
    sum=x+y+z;
    prod=x*y*z;
    cout<<"\nSum="<<sum;
    cout<<"\nProduct="<<prod;
    getch();
}
```

The execution of the program is shown below.

Enter first number:3

Enter second number:4

Enter third number:5

Sum=12

Product=60

When this program is executed, it prompts the user to enter three numbers. The user enters the numbers, 3,4 and 5 separated by space and presses the Enter key. The program calculates the sum and product and displays on the screen.

**Program 2:** The following program reads the base and height of a triangle and prints area using floating-point values.

```

#include<iostream.h>
#include<conio.h>
void main()
{
    float base,height,area;
    cout<< "\nEnter the base:";
    cin>>base;
    cout<< "\nEnter the height:";
    cin>>height;
    area=(base*height)/2;
    cout<< "\nThe area of triangle is "<<area;
    getch();
}

```

In this program, two separate input statements are used for reading the values for variables *base* and *height*. The execution of program is shown is below.

Enter the base:4.5;

Enter the height 6.2;

The area of triangle is 13.95

### 3.3.7 THE MANIPULATORS *endl* AND *setw*

A manipulator is a command in C++ that is used for formatted output. It modifies the output in various ways. There are many manipulators available in C++. The most commonly used manipulators are *endl* and *setw*. The *iomanip.h* header file should be included when manipulators are used in a program. Only the *endl* manipulator can be used without using *iomanip.h* file.

#### The *endl* Manipulator

The *endl* manipulator has the same function as the *\n* escape sequence. It causes a linefeed in the *cout* statement so that the subsequent text is displayed on the next line.

The following program demonstrates the use of *endl* manipulator.

```

#include<iostream.h>
#include<conio.h>
#include<iomanip.h> // this header file is used for manipulators
void main()
{
    cout<< "\nI am a student."<<endl;
    cout<< "I was born in 2001.";
}

```



```
    getch();  
}
```

**The output of the program will be:**

I am a student.

I was born in 2001.

A single *cout* statement can also be used as shown below to obtain the same output.

```
cout<< "I am a student."<<endl<< "I was born in 2001.";
```

### The *setw* Manipulator

The *setw* manipulator is used in output statement to set the minimum field width.

It has the general form:

***setw*(*n*)**

Here, *n* is an integer value that causes the number or text that follows to be printed within a field width of *n* characters. The number or text is right justified within the set field width. It is commonly used in programs to align numbers or text on output.

The following program demonstrates the use of *setw* manipulator in a program.

```
#include<iostream.h>  
#include<conio.h>  
#include<iomanip.h>  
void main()  
{  
    int price1=8540,price2=325,price3=27800;  
    cout<< "Product" <<setw(10)<< "Price"<<endl;  
    cout<< "Hard disk" <<setw(10)<<price1<<endl;  
    cout<< "Mouse" <<setw(10)<<price2<<endl;  
    cout<< "Computer" <<setw(10)<<price3<<endl;  
    getch();  
}
```

This program will print the values following the *setw* manipulator within a field width of 10 characters and they will be right justified.

**The output of the program is given below.**

Product	Price
Hard disk	8540
Mouse	325
Computer	27800

The same program can also be written using a single *cout* statement as shown in the next program.

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main()
{
    int price1=8540,price2=325,price3=27800;
    cout<< "Product" <<setw(10)<<"Price"<<endl
    << "Hard disk" <<setw(10)<<price1<<endl
    << "Mouse" <<setw(10)<<price2<<endl
    << "Computer" <<setw(10)<<price3<<endl;
    getch();
}
```

If *setw* manipulator is not used, the output will be:

Product	Price
Hard disk	8540
Mouse	325
Computer	27800

The commonly used header files are listed in the following table with their purpose.

Header file	Purpose
<b>iostream.h</b>	Provides basic input/output operations such as cin and cout.
<b>conio.h</b>	Stands for console input/output. It manages input/output on console based applications. Console applications take input from keyboard and display output on monitor.
<b>math.h</b>	Provides basic mathematical operations such as sqrt(), pow(), etc.
<b>string.h</b>	Provides several functions to manipulate strings such as strcmp (compare string), strcpy (copy string), etc.
<b>iomanip.h</b>	Provides manipulator function such as setw(), endl, setprecision(), etc.
<b>time.h</b>	Provides date and time operations



## 3.4 OPERATORS IN C++

An operator is a symbol that tells the computer to perform a specific operation. For example the '+' operator is used to add two numbers.

### 3.4.1 TYPES OF OPERATORS

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

- Assignment operators
- Arithmetic operators
- Arithmetic assignment operators
- Increment/Decrement operators
- Relational operators
- Logical operators
- Ternary operator

#### Assignment Operator

Assignment operator is equal sign (=). It is used to assign value of an expression to a variable. It has the general form

variable = expression;

where expression may be a constant, another variable to which a value has previously been assigned or a formula to be evaluated. For example:

$z = x + y;$

When this statement is executed, the values stored in variables x and y will be added and the resulting answer will be stored in variable z.

#### Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations that include addition, subtraction, multiplication, division and to find the remainder of integer division.

The types of arithmetic operators used in C++ programming are described with their operations in the table given below.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operator

The modulo operator (%) gives the remainder after division of one number by another. For example,

$a = 20 \% 6;$



will give the result 2 which will be assigned to the variable *a* because 6 will divide 20 by 3 with a remainder of 2.

Some more examples of modulo operator are given below.

- 13 % 4 will give the result 1
- 15 % 4 will give the result 3
- 10 % 5 will give the result 0
- 4 % 5 will give the result 4

### Arithmetic Assignment Operators

In addition to equal (=) assignment operator, there are a number of assignment operators unique to C++ which are known as arithmetic assignment operators. These include +=, -=, \*=, /= and %=.

Suppose *op* represents an arithmetic operator. Then, the arithmetic assignment operator has the following general form to assign value of an expression to a variable.

**variable op = expression;**

For example:

*a* += *b*;

It is equivalent to:

*a* = *a* + *b*;

The effect is exactly the same but the expression is more compact. The arithmetic assignment operators are described in the following table.

Operator	Operation	Example
+=	Adds the right side operand to the left side operand and assigns the result to the left side operand	<i>a</i> += <i>b</i> It is equivalent to <i>a</i> = <i>a</i> + <i>b</i>
-=	Subtracts the right side operand from the left side operand and assigns the result to the left side operand	<i>k</i> -= <i>n</i> It is equivalent to <i>k</i> = <i>k</i> - <i>n</i>
*=	Multiplies the right side operand with the left side operand and assigns the result to the left side operand	<i>prod</i> *= <i>n</i> It is equivalent to <i>prod</i> = <i>prod</i> * <i>n</i>
/=	Divides the left side operand by the right side operand and assigns the result to the left side operand	<i>n</i> /= <i>m</i> It is equivalent to <i>n</i> = <i>n</i> / <i>m</i>
%=	Takes modulus and assigns the result to the left side operand	<i>x</i> %= <i>y</i> It is equivalent to <i>x</i> = <i>x</i> % <i>y</i>

### Increment and Decrement Operators

The increment operator is ++ and it is used to add one to the value stored in a variable. The decrement operator is -- and it subtracts one from the value stored in a variable. The purpose of using these operators is simply to shorten the expression.

**Examples:**

`++x` and `x++` are both equivalent to `x = x + 1`

`--x` and `x--` are both equivalent to `x = x - 1`

When increment or decrement operator is written before the variable, it is known as *prefix* and when it is written after the variable, it is known as *postfix*.

In certain situations, `++x` and `x++` have different effect. This is because `++x` increments `x` before using its value whereas `x++` increments `x` after its value is used.

As an example, suppose `x` has the value 3. The statement

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

will first increment `x` and then assigns the value 4 to `y`. But the statement

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

will first assign the value 3 to `y` and then increments `x` to 4. In both cases `x` is assigned the value 4 but `y` is assigned different value.

The same rule applies to `--x` and `x--` as well.

**Example of prefix increment:**

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

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

```
void main()
```

```
{
```

```
int x,y;
```

```
y=10;
```

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

```
cout<<"x: "<<x;
```

```
cout<<"y: "<<y;
```

```
getch();
```

```
}
```

**Output:**

```
x: 11
```

```
y: 11
```

**Example of postfix increment:**

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

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

```
void main()
```

```

{
    int x,y;
    y=10;
    x=y++;
    cout<<"x: "<<x;
    cout<<"y: "<<y;
    getch();
}

```

**Output:**

x: 10

y: 11

**Relational Operators**

Relational operators are used to compare two values of the same type. These operators are very helpful in computer programming when a flow of program is based on a condition. After evaluation of a relational expression, the result produced is **True** or **False**.

Six types of relational operators are available in C++ language. These are described in the following table.

Operator	Definition	Example
==	equal to	a==b
!=	not equal to	n!=m
<	less than	a<b+c
>	greater than	z>y
<=	less than or equal to	z<=(x+y)/2
>=	greater than or equal to	f>=a+12

The following are some examples of relational operators.

x &gt; y

x == y

z &gt; x + y

z &lt;= x + y

x!=10

If x has the value 12 and y has the value 7, then the condition in first line will become true since 12 is greater than 7.

The condition of second expression will become false since 12 is not equal to 7.

In the third expression, if  $z$  has the value 15, then the condition will become false since 15 is not greater than the sum of  $x$  and  $y$  which is 19.

In the fourth expression, 15 is less than 19. Therefore, the condition will become true.

In the last expression if  $x$  is any number other than 10 then the expression will be true. It will only be false when  $x$  is equal to 10.

**Note:** In C++ language true is represented by the integer 1 and false is represented by the integer 0.

### Logical Operators

Logical operators are used in programming when it is required to take some action based on more than one condition. When two or more conditions are combined, it is called compound condition.

There are three types of logical operators. These are described below.

Operator	Definition
&&	AND
	OR
!	NOT

#### • Logical AND (&&) Operator

It is used to form compound condition in which two relational expressions are evaluated. One relational expression is to the left and the other to the right of the operator. If both of the relational expressions (conditions) are true then the compound condition is considered true otherwise it is false.

The following is a compound condition that uses the && operator.

$(x \geq 1) \ \&\& \ (y \leq 10)$

When this compound condition is evaluated, it will produce the result true if  $x$  is greater than or equal to 1 and  $y$  is less than or equal to 10. In other words, the result will be true if both conditions are true that is  $x$  is in between 1 and 10 otherwise it will be false.

The following compound condition will check whether the character stored in character variable  $ch$  is a lowercase letter or not.

$(ch \geq 'a') \ \&\& \ (ch \leq 'z')$

The following truth table shows all the possible results of AND (&&) logical operator for two expressions.

Expression-1	Expression-2	Expression-1 && Expression-2
False	False	False
False	True	False
True	False	False
True	True	True



### • Logical OR (||) Operator

Logical OR operator is also used to form a compound condition. Just like the logical AND operator, one relational expression is to the left and the other to the right of the OR operator. The compound condition is true if either of the conditions is true or both are true. It is considered false only if both of the conditions are false.

The following is an example of compound condition that uses || operator.

$(x > y) \parallel (z == 8)$

This compound condition will produce the result true, if one of the conditions is true, that is, if  $x$  is greater than  $y$  or  $z$  is equal to 8. The result will only be false when both of the conditions are false, that is,  $x$  is not greater than  $y$  and  $z$  is not equal to 8.

The following truth table shows all the possible results of OR (||) logical operator for two expressions.

Expression-1	Expression-2	Expression-1    Expression-2
False	False	False
False	True	True
True	False	True
True	True	True

### Logical NOT (!) Operator

The logical NOT operator is used with a single expression (condition) and evaluates to true if the expression is false and vice versa. In other words, it reverses the result of a single expression.

For example, the expression

$!(z < 10)$

will be true if  $z$  is not less than 10. In other words, the condition will be true if  $z$  is greater than or equal to 10. Some programmers may prefer to write the above expression as given below which is easy to understand and has the same effect.

$(z \geq 10)$

The following truth table shows all the possible results of NOT (!) logical operator for one expression.

Expression	! Expression
False	True
True	False

### Ternary Operator (? :) / Conditional Operator

The  $? :$  operator is known as ternary or conditional operator. It returns one of two values depending on the result of a condition. Therefore, it is also known as conditional operator. It is very useful in situation where the programmer needs to choose one of two options depending on a single condition.



The general form of ternary operator is

Condition? Expression1 : Expression2;

The condition is evaluated. If it is true then Expression1 is evaluated otherwise Expression2 is evaluated.

The following is an example of using ternary operator.

$(x > y) ? x + y : x - y$

When this code is executed, the condition  $x > y$  is evaluated. If the result is true then the value  $x + y$  is evaluated otherwise the value  $x - y$  is evaluated. It allows to execute different code based on the result of condition  $x > y$ .

The result of ternary operator can be assigned to a variable as shown below.

$k = (x > y) ? x + y : x - y;$

This is demonstrated in the following program.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int x,y,k;
    x=15; y=10;
    k=(x>y)? x + y : x - y;
    cout<< "The value of k is "<<k<<endl;
    getch();
}
```

The value of  $x$  is greater than  $y$ . Therefore, the condition  $(x > y)$  is true and  $k$  will be assigned the sum of  $x$  and  $y$ .

The output of the program will be

The value of  $k$  is 25

The ternary operator also allows to output text as shown below.

$(x > y) ? \text{cout} << \text{"x is greater than y"} : \text{cout} << \text{"x is smaller than y"};$

This is demonstrated in the following program.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int x,y,k;
    x=3; y=10;
    (x>y)? cout<< "x is greater than y"<<endl;
```



```

        cout<< "x is smaller than y"<<endl;
    getch();
}

```

In this program the value of x is smaller than y. Therefore, the condition (x>y) is false and the second output statement will be executed.

The output of the program will be

x is smaller than y

### 3.4.2 UNARY, BINARY AND TERNARY OPERATORS

There are three types of operators in C++ which are unary, binary and ternary operators.

The operator that works on a single operand is known as unary operators. Unary operators are -, ++, -- and the logical operator !.

Some examples of unary operators are

```
a = -b;
```

```
k++;
```

```
- -x;
```

The operators that work on two operands are known as binary operators. Binary operators are -, +, \*, /, % and logical operators && and ||.

Some examples of binary operators are

```
a = b + c;
```

```
z = x * y;
```

```
k = d % e;
```

The conditional operator (? :) is known as ternary operator since it has three parts. These three parts are a condition and two expressions. The condition is evaluated and based on its result one of the two expressions is executed.

### 3.4.3 ORDER OF PRECEDENCE OF OPERATORS

Order of precedence of operators describes the rules according to which operations are to be performed in an expression.

In the following table, the operator that has the highest precedence is written at the top and the one with the lowest precedence is written at the bottom.

Precedence	Operator	Description
1.	*, /, %	Multiplication, Division and Remainder
2.	+, -	Addition and Subtraction
3.	<, <=, >, >=	Relational Operators



4.	=, !=	Equal to and Not Equal to
5.	!	Logical NOT
6.	&&	Logical AND
7.		Logical OR
8.	=, *=, /=, %=, +=, -=	Assignment Operators

For example, in the expression

$$7 + 3 * 2$$

multiplication operator has higher precedence than the addition operator and thus will be evaluated first.

Parentheses are used in expression to change the order of evaluation of operators specified by operator precedence.

**For example**, in the above example, if it is required to perform addition before multiplication then parentheses can be used as shown below.

$$(7 + 3) * 2$$

When two operators of same precedence occur in an expression then they are evaluated from left to right as they occur.

When an expression contains arithmetic, relational and logical operators, the arithmetic operators are evaluated first, relational operators next and logical are evaluated last. Assignment operators are always applied at the end.

### 3.4.4 EXPRESSIONS IN C++

An expression is a combination of constants, variables and operators. Constants and variables are operands and operators tell the computer what types of action to perform on the operands.

There are three types of expressions in C++. These are arithmetic, relational and compound or logical expression.

An expression that contains constants, variables and arithmetic operators is called arithmetic expression.

For example, in the expression

$$a + 2 * b$$

$a$ ,  $b$  and 2 are operands and  $+$  and  $*$  are arithmetic operators. When this expression is evaluated, the values stored in variables  $a$  and  $b$  are substituted and the result produces another value.

The following are some more examples of arithmetic expressions.

$$5 * (a - b)$$



### Teacher Point

Teacher should give some home assignments to the students at the end of the chapter.



$$(a + b)/(a - b) + a*b$$

$$3*a + 8 - b + (b + c)/2$$

An expression that contains a relational operator to compare values of same type is called relational expression. Relational expressions are used in programming to create conditions based on which computer takes different path during program execution.

An expression that combines two or more conditions using logical operators, && or || is called compound expression.

The following is an example of compound expression.

```
((ch>='a')&&(ch<='z'))||((ch>='A')&&(ch<='Z'))
```

The first compound condition ((ch>='a')&&(ch<='z')) checks whether the character stored in variable ch is a lower-case letter or not and the second compound condition ((ch>='A')&&(ch<='Z')) checks whether it is an upper-case letter or not. If one of the two compound conditions is true, the compound expression will be true because the two compound conditions are combined with an || (OR) operator.



### Key Points

- A computer program is a set of instructions to perform a specific task.
- Header files contain information that is required by the program in which they are used. It has .h extension.
- Reserved words are special words which are reserved by a programming language for specific purpose in program.
- In C++, semicolon is a statement terminator. It marks the end of a statement. All the C++ statements must end with a semicolon.
- Comments are explanatory statements that help the reader in understanding source code.
- A constant is a value that does not change during execution of program.
- A variable is a name of memory location where data is stored. Variables are used in computer programs to store values of different data types.
- Escape sequences are special characters used to control printing on the output device. These characters are not printed. These are used inside the output statement.
- Order of precedence of operators describes the rules according to which operations are to be performed in an expression.
- An operator is a symbol that tells the computer to perform a specific computing task.
- Assignment operator is equal sign (=). It is used to assign value of an expression to a variable.
- Arithmetic operators are used to perform arithmetic operations that include addition, subtraction, multiplication, division and to find the remainder of integer division.



- Relational operators are used to compare two values of the same type.
- Logical operators are used in programming when it is required to take some action based on more than one condition.
- An expression is a combination of constants, variables and operators. Constants and variables are operands and operators tell the computer what types of action to perform on the operands.
- An expression that combines two or more conditions using logical operators, && or || is called compound expression.



### Exercise

#### Q1. Select the best answer for the following MCQs.

- Which of the following is ignored during program execution?
  - Reserved word
  - Constant
  - Comment
  - const qualifier
- What is the range of unsigned short integer?
  - 2147483648 to 2147483647
  - 0 to 4294967295
  - 32768 to 32767
  - 0 to 65535
- In C++ the expression  $\text{sum} = \text{sum} + n$  can also be written as:
  - $\text{sum} += n$
  - $\text{sum} = n++$
  - $\text{sum} = +n$
  - $n += \text{sum}$
- Which of the following is equal to operator?
  - $+=$
  - $==$
  - $=$
  - $+=$
- Which of the following is an arithmetic operator?
  - &&
  - %
  - $<=$
  - $++$
- Which of the following operators is used to form compound condition?
  - Arithmetic operator
  - Assignment operator
  - Relational operator
  - Logical operator
- The number of bytes reserved for a variable of data type 'float' is:
  - 2
  - 4
  - 6
  - 8
- How cursor is moved to the next tabular position for printing data?
  - By using reserved word
  - By using manipulator
  - By using escape sequence
  - By using header file

#### Q2. Write short answers of the following questions.

- Define reserved words and give three examples.

- ii. What is the purpose of using header file in program?
- iii. State whether the following variable names are valid or invalid. State the reason for invalid variable names.
- |          |           |               |           |
|----------|-----------|---------------|-----------|
| a) a123  | b) _abcd  | c) 5hml       | d) tot.al |
| e) f3ss1 | f) c\$avg | g) net_weight | h) cout   |
- iv. Why escape sequence is used? Give three examples with explanation.
- v. Differentiate between relational and logical operators.
- vi. What will be the output of the following statements?
- a) `cout<< "17/3 is equal to "<<17/3;`  
 b) `cout<< "10.0/4 is equal to "<<10.0/4;`  
 c) `cout<< "40/5%3*7 is equal to "<<(40/5%3*7);`
- vii. Evaluate the following integer expressions.
- |               |                   |                |
|---------------|-------------------|----------------|
| a) $3+4*5$    | b) $4*5/10+8$     | c) $3*(2+7*4)$ |
| d) $20-2/6+3$ | e) $(20-2)/(6+3)$ | f) $25\%7$     |
- viii. Evaluate the following expressions that have integer and floating-point data types.
- |                    |                      |                 |
|--------------------|----------------------|-----------------|
| a) $10.0+15/2+4.3$ | b) $10.0+15.0/2+4.3$ | c) $4/6*3.0 +6$ |
|--------------------|----------------------|-----------------|
- ix. What will be the output of the following program?

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int num1,num2,total;
    num1=13;
    num2=20;
    total=num1+num2;
    cout<< "The total of "<<num1<< " and "
    <<num2<< " is "<<total<<endl;
    getch();
}
```

- x. What will be the output of the following program?

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n;
```

```
n=10;
cout<< "The initial value of n is "<<n<<endl;
n++;
cout<< "The value of n is now "<<n<<endl;
n++; n++;
cout<< "The value of n is now "<<n<<endl;
n--;
cout<< "The value of n is now "<<n<<endl;
getch();
}
```

**Q3. Write long answers of the following questions.**

- Define variable and write the rules for specifying variable names.
- Why type casting is used? Explain the types of type casting with an example of each type.
- Define *endl* and *setw* manipulators and give an example of each.
- What is meant by precedence of operators? Write the operators with the highest precedence at the top and the lowest at the bottom.

**Lab Activities**

- Practice all the Example programs given in the chapter.
- Write a program that reads four integers and prints their sum, product and average.
- Write a program that reads length and breadth of a rectangle and prints its area.
- Write a program that reads temperature in Fahrenheit and prints its equivalent temperature in Celsius using the following formula.

$$c = 5/9(f - 32)$$



# 4

## CONTROL STRUCTURES



After completing this lesson, you will be able to:

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

### INTRODUCTION

Control structures are a very important concept in computer programming. Control structures allow programmers to control the flow of program execution. Three types of control structures are used in programming, sequential, conditional and repetition structures.

Sequential structure refers to execution of instructions one by one in the sequence in which they appear in the program from top to bottom till the last instruction.

Conditional structure, also known as decision making structure, refers to execution of instructions based on a condition. If a condition is met a specific set of instructions are executed otherwise control is transferred to some other part of the program.

Iteration structure, also known as loop, refers to execution of same set of instructions several times till a condition is met.

In this unit, students will learn the decision making and loop control structures available in C++ and how to implement them in programs to solve various problems.

## 4.1 DECISION CONTROL STRUCTURES

Programs are written to solve user problems. Various decisions have to be made in programs depending on the nature of problem. Decision making structures are control structures that are used in programming to make decisions. They allow programs to execute a specific statement or a set of statements based on one or more conditions.

The decision making statements available in C++ language are *if*, *if-else*, *else-if* and *switch* statements.

### 4.1.1 IF STATEMENT

The *if* statement is used to execute a block of statements based on a condition.

The following is general form of *if* statement.

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

#### Explanation:

- The condition is evaluated.
- If the condition is true, the block of statements within the braces, following the condition is executed.
- If the condition is false, the block of statements is skipped and control is transferred to the next statement after the closing brace.
- If there is a single statement to be executed then braces are not required.

**Program 1:** The following program will read two numbers. If the first number is a positive number then it will print the sum and product of the two numbers.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int x,y,sum,prod;
    cout<<"\nEnter first number:";
    cin>>x;
```



#### Teacher Point

Explain the concept of decision control structure with some simple examples.

```

cout<<"\nEnter second number:";
cin>>y;
if (x>0)
{
    sum=x+y;
    prod=x*y;
    cout<<"\nSum="<<sum<<endl;
    cout<<"\nProduct="<<prod;
}
getch();
}

```

This program prompts the user to enter two numbers which are stored in variables *x* and *y*. If the first number (*x*) is greater than zero then the sum and product of the numbers are calculated and stored in variables *sum* and *prod* and their values are printed.

The following is the execution of the program.

```

Enter first number:3
Enter second number:4
Sum=7
Product=12

```

**Program 2:** The following program reads marks and prints the message "You have passed" if marks are greater than or equal to 33.

```

#include<iostream.h>
#include<conio.h>
void main()
{
    int marks;
    cout<<"\nEnter the marks:";
    cin>>marks;
    if (marks >= 33)
        cout<<"You have passed";
    getch();
}

```

In this program there is a single statement to be executed when the condition is true. Therefore, braces are not used.

The following is the execution of the program.

```

Enter the marks:48
You have passed

```



### 4.1.2 IF-ELSE STATEMENT

The *if-else* statement allows making decision between two courses of action based on a condition.

The following is the general form of *if-else* statement.

```
if (condition)
{
    Block of statements-1           // if the condition is true
}
else
{
    Block of statements-2           // if the condition is false
}
```

#### Explanation:

- The condition is evaluated.
- If the result of evaluation is true, the first block of statements-1 is executed, the second block of statements-2 is skipped and then control is transferred to the next statement.
- If the condition evaluates to false, the first block of statements-1 is skipped and the second block of statements-2, following the keyword *else* is executed.
- If there is a single statement to be executed whether the condition is true or false then braces are not required.

**Program 3:** The following program reads a number and prints whether it is even or odd number.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n,r;
    cout<<"\nEnter a number:";
    cin>>n;
    r = n% 2;           // find the remainder
    if (r == 0)
        cout<<n<<" is even number";
    else
        cout<<n<<" is odd number";
    getch();
}
```

When this program is executed, it prompts the user to enter a number. The number is stored in variable  $n$ . The modulo operator (%) gives the remainder after division of the entered number by 2 and it is stored in variable  $r$ . If the remainder  $r$ , is equal to 0 then the program will print,  $n$  is even otherwise it is odd.

Braces are not used in *if-e/se* statement in this program because a single statement is to be executed whether the condition is true or false.

The following is the execution of the program.

Enter a number:15

15 is odd number

### 4.1.3 ELSE-IF STATEMENT

The *else-if* statement is used in situation where a decision is to be made from several alternatives based on various conditions.

The following is the general form of *else-if* statement.

```
if (condition-1)
{
    Block of statements-1
}
else if (condition-2)
{
    Block of statements-2
}
.
.
.
else
{
    Block of statements-n
}
```

#### Explanation:

- The condition-1 is evaluated.
- If it is true, the block of statements-1 is executed and control is transferred to the next statement.
- If the condition-1 is false then condition-2 is evaluated. If it is true then the block of statements-2 following condition-2 is executed.

- In this manner conditions are evaluated one by one. When any condition is true, the block of statements following that condition is executed, rest of the code is skipped and control is transferred to the next statement.
- If none of the conditions is true then the last block of statements following the keyword *else* is executed. The *else* block is optional.
- If a single statement is to be executed instead of a block of statements then braces are not required.

**Program 4:** The following program reads a number and prints the message whether it is positive number, negative number or it is equal to zero.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n;
    cout<<"\nEnter a number:";
    cin>>n;
    if (n > 0)
        cout<<n<<" is positive number";
    else if (n < 0)
        cout<<n<<" is negative number";
    else
        cout<<n<<" is equal to zero";
    getch();
}
```

The following is the execution of the program.

Enter a number:-6

-6 is negative number

**Program 5:** The following program prints grade based on the marks obtained according to the given scheme.

Marks	Grade
$80 \leq \text{Marks} \leq 100$	'A'
$70 \leq \text{Marks} \leq 79$	'B'
$60 \leq \text{Marks} \leq 69$	'C'
$50 \leq \text{Marks} \leq 59$	'D'
$0 \leq \text{Marks} \leq 49$	'F'

```

#include<iostream.h>
#include<conio.h>
void main()
{
    int marks;
    char grade;
    cout<<"\nEnter the marks (max 100):";
    cin>>marks;
    if (marks >= 80)
        grade = 'A';
    else if (marks >=70)
        grade = 'B'
    else if (marks >=60)
        grade = 'C';
    else if (marks >=50)
        grade = 'D';
    else
        grade = 'F';
    cout<< "\n Your grade is <<grade;
    getch();
}

```

The following is the execution of the program.

Enter the marks:74

Your grade is B

#### 4.1.4. SWITCH STATEMENT

The *switch* statement is a control statement that is used in programming when a single block of statements is to be selected among many choices. It is very similar to *else-if* statement.

The following is the general form of *switch* statement.

```

switch (expression/variable)
{
    case constant-1: block of statements
                    break;
    case constant-2: block of statements
                    break;
    case constant-3: block of statements
                    break;
}

```

```

    default:    block of statements
                break;
}

```

**Explanation:**

- The expression within the brackets is evaluated.
- The result of the expression or the value of variable is compared in sequence with the constant values given after the keyword *case*.
- If result matches any constant value then the block of statements following that case is executed and control exits from the body of the *switch* statement and goes to the first statement following the end of the *switch* statement.
- If none of the constant values after the *case* keyword match with the result of expression or the value of variable then the block of statements following the keyword *default* is executed. Its use is optional.
- In switch statement, it is allowed to use a variable within the parenthesis instead of an expression based on which block of statements following a *case* can be executed.
- The purpose of *break* statement is to exit the body of the *switch* statement.

**Program 6:** The following program reads an integer between 1 to 7 that represent a day of week starting from Monday. It prints the name of the day based on the value of day.

```

#include<iostream.h>
#include<conio.h>
void main()
{
    int a;
    cout<<"\nEnter an integer(1-7):";
    cin>>a;
    switch(a)
    {
        case 1:    cout<<"\nMonday";
                   break;
        case 2:    cout<<"\nTuesday";
                   break;
        case 3:    cout<<"\nWednesday";
                   break;
    }
}

```

```

        case 4:    cout<<"\nThursday";
                   break;
        case 5:    cout<<"\nFriday";
                   break;
        case 6:    cout<<"\nSaturday";
                   break;
        case 7:    cout<<"\nSunday";
                   break;
        default:   cout<<"\nNot a valid day";
    }
    getch();
}

```

The following is the execution of the program.

Enter an integer:6

Saturday

**Program 7:** The following program prompts the user to enter a lower-case letter and determines whether it is a vowel or a consonant.

```

#include<iostream.h>
#include<conio.h>
void main()
{
    char ch;
    cout<<"\nEnter a lower-case letter:";
    cin>>ch;
    switch(ch)
    {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':    cout<<"\nYou entered a vowel.";
                    break;
        default:    cout<<"\nYou entered a consonant.";
    }
    getch();
}

```

When this program is executed, it prompts the user to enter a lower-case letter which is stored in the character variable *ch*. If the user enters a lower-case vowel, the message "You entered a vowel." is printed otherwise the default message "You entered a consonant." is printed. The program only checks for the vowels ( 'a','e','i','o','u') using the switch statement. In the absence of statements after the keyword **case**, the control goes right through one case to the case below and this makes it easy for several values of the switch variable to execute the same code.

Enter a lower-case letter: g

You entered a consonant

### Comparison between Decision Control Structures

**if Statement:** It is used when a single block of statements is to be executed if the condition is true. If the condition is false then the block of statements is skipped and the program execution continues from the statement after **if** block.

**if-else Statement:** It is used when one block of statements is to be executed if the condition is true. If the condition is false then the block of statements following **if** are skipped and the other block of statements following **else** is executed. It selects one block of statements from two options.

**Else-if Statement:** It selects a block of statements from multiple options. When it is executed, a block of statements is selected for execution based on a condition starting from the first condition. If all the conditions are false then it executes the block of statements after **else** if it exists otherwise program execution continues from the statement after **if** block.

**switch Statement:** It is a multiple selection statement that is similar to **else-if** statement. It is used to select a block of statements based on the value of a variable or result of an expression. It compares the value of a variable or the result of an expression that is within the parenthesis after the keyword **switch**, with values specified in each **case**. If there is a match, the block of statements following that **case** is executed. If there is no match, the **default** block of statements is executed if it exists otherwise the control is transferred to the statement after **switch** block.

#### 4.1.5 DIFFERENCE BETWEEN IF-ELSE IF AND SWITCH STATEMENTS

	ELSE-IF	SWITCH
1	Which statement will be executed depend upon the output of the relational expression inside if statement.	Which statement will be executed is decided by user.
2	Else-if statement uses multiple decision statements for multiple choices.	Switch statement uses single expression for multiple choices.

	ELSE-IF	SWITCH
3	Else-if statement test for equality as well as for logical expression.	Switch statement test only for equality.
4	Else-if statement evaluates integer, character, or logical type expressions.	Switch statement evaluates only character or integer value.
5	Either if statement will be executed or else statement is executed.	Switch statement execute one case after another till a break statement is appeared or the end of switch statement is reached.
6	If the condition inside if statements is false, then by default the else statement is executed.	If the condition inside switch statements does not match with any of cases, for that instance the default() statements is executed.

#### 4.1.6 NESTED IF STATEMENT

An *if* statement inside another *if* statement is known as nested *if* statement. The C++ language allows to nest *if*, *if-else*, or *else-if* inside another *if*, *if-else* or *else-if* statement.

**Programs 8:** The following program demonstrates the use of nested *if-else* statement inside another *if-else* statement.

```
#include<iostream.h>
#include<conio.h>
void main()
{   i
    int marks;
    cout<<"\nEnter the marks:";
    cin>>marks;
    if((marks>=0)&&(marks<=100))    // outer if-else statement
    {                               // nested if-else statement
        if(marks>=33)
            cout>>"You are Pass";
        else
            cout>>"You are Fail"
    }
    else
        cout>>"Invalid Data";
    getch();
}
```



When this program is executed, the user is prompted to enter marks. The first *if-else* statement checks whether the marks entered are in the range of 0 to 100. If it is true then the nested *if-else* statement checks whether the user passed or failed and prints the appropriate message. If the marks entered are out of range then the message "Invalid data" is printed.

The following is the execution of the program when marks are in the range of 0 to 100.

Enter the marks:24

You failed

The following is the execution of the program when marks are not in the range of 0 to 100.

Enter the marks:113

Invalid data

## 4.2 LOOPS OR REPETITION CONTROL STRUCTURE

A loop is a control structure that repeatedly executes a sequence of statements until condition is true. Loops are also called repetition control structures in C++. There are three types of loops in C++ which are *for*, *while* and *do while*.

### 4.2.1 FOR LOOP

A *for* loop is used to execute one or more statements for a specific number of times. It is also known as counter loop.

The following is the general form of *for* loop.

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

```
{
```

```
    Block of statements
```

```
}
```

**Explanation:**

- A variable, known as loop counter or loop variable, is assigned an initial value in the initialization part of the loop. For example,  $a=1$  or  $b=50$ .
- The condition which is a relational expression such as  $a < 5$  is evaluated. If the condition is true then the block of statements within the braces is executed.
- After the execution of the statements, control is transferred to the increment/decrement part of the loop. This part consists of an assignment statement such as  $a=a+1$  or  $a=a-1$  that increments or decrements the loop variable.
- The condition is again evaluated. If it is true then the body of the loop is again executed. This process goes on till the loop condition becomes false. When the loop condition becomes false, the loop terminates and control is transferred to the next statement after the block of statements.

- If only a single statement is to be executed in for loop then braces are not required.

**Program 9:** The following program prints four times the output of statements that are in the for loop.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int k;
    for (k=1;k<5;k++)    // loop executes four times till k is less than 5
    {
        cout<< "\nI am a student."<<endl;
        cout<< "I was born in 2001."<<endl;
    }
    getch();
}
```

When this program is executed, the loop counter ( $k$ ) is initialized to 1. The loop condition  $k < 5$  is checked. Since, it is true, the loop executes and displays the output of two statements that are within the curly brackets. The loop counter is incremented by 1. The condition is checked and the loop statements are again executed. Each time the loop is executed, the counter  $k$  is incremented by 1. The loop continues to execute till  $k$  is less than 5. When  $k$  becomes 5, it terminates and control is transferred to the next line. In this program  $k++$  is used which is same as  $k=k+1$ .

The following is the execution of the program.

I am a student.

I was born in 2001.

I am a student.

I was born in 2001.

I am a student.

I was born in 2001.

I am a student.

I was born in 2001.



**Program 10:** The following program uses for loop to print all the positive odd numbers that are less than twenty (1 3 5 7 9 11 13 15 17 19) on a single line.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int k;
    for (k=1;k<20;k=k+2;)
        cout<< k << " ";
    getch();
}
```

The loop variable  $k$  is initialized to 1, condition is checked and its value is printed. After printing the value of  $k$ , control is transferred to the increment part of the loop. The value of  $k$  is incremented by 2 and then printed again. The process continues as long  $k$  is less than 20. The loop contains a single statement so braces are not used.

#### 4.2.2 WHILE LOOP

A **while** loop is a sentinel loop statement. In this loop the condition is checked at the beginning of the loop. The body of the loop executes as long as the condition remains true. The control can exit a loop in two ways, when the condition becomes false or using break statement.

The following is the general form of *while* loop.

```
while (condition)
{
    Block of statements
}
```

#### Explanation:

- The condition which is a relational expression such as  $k < 10$ , is evaluated.
- If the condition evaluates to true, the block of statements within the braces is executed.
- After the execution of statements, control is transferred back to the beginning of the loop and the condition is again evaluated. If it is true then the body of the loop is executed again.



#### Teacher Point

Use examples to explain the concept of loops.

- This process goes on till the condition becomes false. When the loop condition becomes false, the loop terminates and control is transferred to the next statement.
- If the body of the loop consists of a single statement then braces are not required.

**Program 11:** The following program prints the sum of all the positive numbers up to 15 using a *while* loop. (sum = 1 + 2 + 3 + 4 + ... + 15)

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int k, sum;
    sum=0;
    k=1;           // initialization of loop variable
    while (k<=15)
    {
        sum=sum+k;
        k=k+1;     // increment for loop continuation
    }
    cout<< "Sum="<<sum;<<endl;
    getch();
}
```

The variable *sum* is initialized to 0 and the loop variable *k* to 1. Each time the loop is executed, the value of *k* is added to *sum* and it is incremented by 1. This process continues till the condition *k*≤15 is true. When *k* becomes 16, the loop terminates.

The following is the output of the program.

Sum=120

**Program 12:** The following program uses *while* loop to repeatedly prompt the user to enter a number and prints its square. The loop terminates when the user enters 0 and the program prints the message "Goodbye".

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n;
    cout<< "Enter a number:";
    cin>>n;
```



```
while (n!=0 )      // != is not equal to relational operator
{
    cout<<n*n<<endl;
    cout<< "\nEnter a number (0 to quit):";
    cin>>n;
}
cout<< "Goodbye!";
getch();
}
```

In this program, the condition  $n \neq 0$  is true till the user enters 0.

The following is the execution of the program.

```
Enter a number:3
9
Enter a number (0 to quit):7
49
Enter a number (0 to quit):10
100
Enter a number (0 to quit):-8
64
Enter a number (0 to quit):0
Goodbye!
```

### 4.2.3 DO-WHILE LOOP

A **do-while** loop is very similar to *while* loop, except that the loop condition is checked at the end of the loop. Therefore, the body of the loop is executed at least once.

The following is the general form of *do while* loop.

```
do
{
    Block of statements
}
while (condition);
```

#### Explanation:

- The block of statements following the key word *do* is executed.
- The condition at the end of the loop is evaluated. If the condition is true, control is transferred back to the beginning of the loop.

- The loop is executed once again and then the condition is also checked again. This process continues till the condition becomes false.
- When the condition becomes false, control is transferred to the next statement.
- If the body of the loop consists of a single statement then braces are not required.

**Program 13:** The following program prompts the user to enter two numbers and prints their product. After printing the product, it asks the user whether he/she wants to continue printing product of another set of two numbers. If the user wants to continue, he/she enters the character 'y' otherwise the character 'n'.

```
#include<iostream.h>
void main()
{
    int a,b,prod;
    char ch;
    do
    {
        cout<< "\nEnter two numbers:";
        cin>>a>>b;
        prod=a*b;
        cout<< "Product="<<prod<<endl;
        cout<< "Do you want to continue? (y/n):";
        cin>>ch;
    }
    while (ch!='n');
}
```

The following is the execution of the program.

Enter two numbers:3,4

Product=12

Do you want to continue? (y/n):y

Enter two numbers:7,8

Product=56

Do you want to continue? (y/n):y

Enter two numbers:10,12

Product=120

Do you want to continue? (y/n):n



### 4.2.4 DIFFERENCE BETWEEN WHILE LOOP AND DO-WHILE LOOP

	WHILE LOOP	DO-WHILE
1	While loop is pre-tested loop as the condition is checked in the start.	Do-While loop is post-tested loop as the condition is checked in the end.
2	It is called entry controlled loop.	It is called exit controlled loop.
3	The body statements never executes if the condition is false at the beginning.	The body statements execute at least once even if the condition is false.
4	There is no semicolon at the end of while statement.	There is semicolon at the end of do-while statement.

### 4.2.5 THE BREAK AND CONTINUE STATEMENTS

#### The Break Statement

The *break* statement has two usages.

It is used to terminate a *case* in *switch* statement and program execution continues from the next statement following the *switch* statement. The use of *break* statement in a *switch* statement was demonstrated in the previous section.

The *break* statement is also used to terminate a loop when it is encountered inside a loop and program execution continues from the next statement following the loop.

**Program 13:** The following program demonstrates the use of *break* statement inside a *for* loop.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int k,count=0;
    for(k=1;k<10;k++)
    {
        cout<<k<<" ";
        count=count+1;
        if(k==4)
            break;
    }
    cout<<"\nThe loop executed "<<count<<"times."<<endl;
    getch();
}
```

The output of the program will be:



1 2 3 4

The loop is executed 4 times.

In this program when the value of loop variable  $k$  becomes 4, the loop terminates and program execution resumes from the next statement following the loop.

### The Continue Statement

The continue statement is used inside a loop. When it is encountered, it transfers control to the beginning of the loop, skipping the remaining statements.

**Program 14:** The following program demonstrates the use of *continue* statement. It prints all the positive numbers less than 15, skipping those that are greater than 5 and less than 10 i.e. the numbers 6, 7, 8, and 9 are not printed.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n;
    for(n=1;n<15;n++)
    {
        if((n>5)&&(n<10))
            continue;           // if n is greater than 5 and less than 10 then skip
        cout<<n<<" ";         // printing the value of n and continue the for loop
    }
    getch();
}
```

The output of the program will be:

1 2 3 4 5 10 11 12 13 14

### 4.2.6 EXIT() FUNCTION

The *exit()* function is used to terminate a C++ program before its normal termination and exit to the operating system. It requires the standard library header file *stdlib.h*.

The following is the general form of *exit* function.

`exit (value);`



### Teacher Point

Teacher may also use flowcharts to explain different control structures.



Here, value is an integer value or integer variable. It is known as exit code. The exit code "0" exits a program without any error and exit code "1" indicates that an error must have occurred. It helps the programmer in debugging the program.

Consider the following *if-else* statement.

```
if(n>0)
    cout<<n<<" is a positive number";
else
    exit(0);
```

In the above *if-else* statement, the control function *exit* is used to terminate the execution of *if-else* statement if the value of *n* is not greater than 0 and return to operating system.

### 4.2.7 NESTED LOOP

A loop inside another loop is known as nested loop. The C++ language allows to nest a *for*, *while* or *do while* loop inside another *for*, *while* or *do while* loop.

**Program 15:** The following program demonstrates the use of nested *for* loop inside another *for* loop.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int i,j;

    for(i=1;i<5;i++)    // outer loop
    {
        cout<<"\n";    // transfers printing to next line
        for(j=1;j<11;j++)    // inner loop
            cout<<"*";
    }
    getch();
}
```

In this program, *i* is the outer loop variable and *j* is the nested loop variable. When this program is executed, the outer loop will execute till the value of *i* is less than 5 which means it will execute 4 times. The variable *i* will be initialized to 1 and the statement *cout<<"\n";* will



### Teacher Point

Teacher should also explain few more related programs according to the chapter topics.

transfer the control to next line. The nested loop will execute and print 10 asterisks (\*) on a single line as shown below.

\*\*\*\*\*

The variable  $i$  will be incremented by 1 and the nested loop will again print another line of 10 asterisks. This process will continue till  $i$  is less than 5. Hence, this program will print 4 lines of 10 asterisks.

The following is the output of the program.

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

**Program 16:** The following program demonstrates the use of nested *for* loop inside *while* loop for printing the following pattern.

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

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n,k;
    n=1;
    while(n<=5)           // outer loop
    {
        cout<< "\n";      // transfers printing to next line
        for(k=1;k<=n;k++)  // inner loop
            cout<<k<< " ";
        n++;
    }
    getch();
}
```



### Teacher Point

Teacher should give some home assignments to the students at the end of the chapter.



In this program  $n$  is the outer loop variable and  $k$  is the loop variable of nested *for* loop. When the program execution begins,  $n$  is initialized to 1. The `cout<< "n";` statement instructs to start printing at the beginning of next line. The nested loop executes once since the value of  $n$  is equal to 1 and prints 1 at the beginning of the line. The value of  $n$  is incremented by 1 and becomes 2. The nested *for* loop now executes twice and prints the numbers 1 and 2 on the same line with space in between. The process continues as long the value of  $n$  is less than 6.



### Key Points

- Decision making structures are control structures that are used in programming to make decisions. They allow programs to execute a specific statement or a set of statements based on one or more conditions.
- The *if* statement is used to execute a block of statements based on a condition.
- The *if-else* statement allows making decision between two courses of action based on a condition.
- The *else-if* statement is used in situation where a decision is to be made from several alternatives based on various conditions.
- The *switch* statement is a control statement that is used in programming when a block of statements is to be selected among many choices.
- An *if* statement inside another *if* statement is known as nested *if* statement.
- A loop is a control structure that repeatedly executes a sequence of statements until a condition is reached.
- A *for* loop is used to execute one or more statements for a specific number of times.
- A *while* loop is used when the number of times the loop will execute may not be known in advance.
- The *do while* loop is very similar to *while* loop, except that the loop condition is checked at the end of the loop. Therefore, the body of the loop is executed at least once.



### Exercise

**Q1. Select the best answer for the following MCQs.**

- How is a single-statement *for* loop terminated?
  - with a colon
  - with a right brace
  - with a right bracket
  - with a semicolon
- How is a multiple-statement *for* loop terminated?
  - with a colon
  - with a right brace
  - with a right bracket
  - with a semicolon

- iii. Which of the following can be used to replace ternary operator?
- a) if statement
  - b) if-else statement
  - c) else-if statement
  - d) switch statement
- iv. Which of the following can be used to replace switch statement?
- a) if-else statement
  - b) break statement
  - c) else-if statement
  - d) while loop
- v. A while loop is more appropriate to use than a for loop when:
- a) the body of the loop is to be executed at least once
  - b) the termination condition occurs unexpectedly
  - c) the program executes at least once
  - d) when the number of iterations are known in advance
- vi. In which situation a do while loop is more appropriate to use?
- a) when the body of the loop is to be executed at least once
  - b) when the loop terminates unexpectedly
  - c) when the program executes at least once
  - d) when the number of loop iterations are known in advance
- vii. In which situation a for loop is more appropriate to use?
- a) when the body of the loop is to be executed at least once
  - b) when the loop terminates unexpectedly
  - c) when the program executes at least once
  - d) when the number of loop iterations are known in advance
- viii. Which of the following transfers control to the beginning of the loop, skipping the remaining statements?
- a) exit function
  - b) continue statement
  - c) break statement
  - d) nested loop

## Q2. Write short answers of the following questions.

- i. Why control statements are used in C++ programs?
- ii. Differentiate between *if-else* and *else-if* statements.
- iii. Differentiate between *for* and *while* loop.
- iv. What is the usage of *break* and *continue* statements in C++ programs.
- v. What is the purpose of exit function?
- vi. What is nested loop? Give one example.
- vii. Write the following code using while loop.

```
sum=0;
for (k=20; k<100; k=k+2)
    sum=sum+k;
cout<<"\nSum= "<<sum;
```

viii. Write the following code using switch statement, to produce the same output.

```
if(choice == 1)
    cout<<"\nSum= "<<x+y;
else if(choice == 2)
    cout<<"\nProduct= "<<x*y;
else
    cout<<"\nAverage= "<<(x+y)/2;
```

ix. What will be the output of the following code?

```
int k,sum;
k=1; sum=0;
while(k<10)
{
    sum=sum+k;
    cout<<k<<sum;
    k=k+2;
}
```

x. What will be the output of the following code?

```
int a,b,c;
a=0; b=1; c=2;
a=b+c;
b=++a;
c=b++;
cout<<a<<b<<c;
```

xi. Write the nested loops that will print the following patterns.

a) 1 2 3 4 5  
1 2 3 4  
1 2 3  
1 2  
1

b) 1 2 3 4 5  
2 3 4 5  
3 4 5  
4 5  
5

c) \*  
\* \*  
\* \* \*  
\* \* \* \*  
\* \* \* \* \*

**Q3. Write long answers of the following questions.**

- What is decision control structure? Explain all types of if statements with syntax and examples.
- Explain the purpose of switch statement with syntax and one example.
- What is looping control structure? Explain all types of looping statements with syntax and examples.



## Lab Activities

- Practice all the programs given in the chapter.
- Write a program that reads a number and prints its square if the number is greater than 10 otherwise prints its cube.
- Write a program that reads an integer and prints whether it is odd or even number.
- Write a program that reads three numbers and prints the largest one.
- Write a program that reads a letter and prints whether it is a lowercase or uppercase letter.
- Write a program that reads an integer and prints its multiplicative table up to 20.
- Sameer works in a firm as a programmer and is getting a monthly pay. Write a program to take his basic pay as input through the keyboard and calculate his net pay. His net pay is to be calculated as given below.

$$\text{net pay} = \text{basic pay} + \text{house rent}$$

Sameer gets his house rent based on his basic pay according to the scheme given below.

Basic Pay	House Rent
Basic Pay < 30000	30% of Basic Pay
Basic Pay ≥ 30000 and ≤ 50000	35% of Basic Pay
Basic Pay > 50000	40% of Basic Pay

Write a program to calculate the Net Pay of Sameer.

- Write a program that will produce a table of equivalent temperatures in both Fahrenheit and Celsius with an increment of 5 from 50 to 100 as given below.

$$C = \frac{5}{9}(F - 32)$$

Fahrenheit	Celsius
50	9.90
55	12.65
.	.
.	.
.	.
100	37.40

- Write a program that prints the sum of the following sequence using for loop.  

$$\text{sum} = 30 + 33 + 36 + 39 + \dots + 60$$
- Write the above program using while loop.
- Write a program that prints all the positive odd numbers up to 50 skipping those that are divisible by 5 using continue statement.
- Write a program that reads an integer and prints its factorial.



xiii. Write a program that reads the coefficients,  $a$ ,  $b$  and  $c$  of the quadratic equation

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

and prints the real solutions of  $x$ , using the following formula.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Note that if  $b^2 - 4ac = 0$  then there is only one real solution. If it is greater than zero then there are two real solutions and if it is less than zero then print the message "No Real Solutions".

MDCAT BY FUTURE DOCTORS-TOUSEEF AHMAD-03499815886



# 5

## ARRAYS AND STRINGS



After completing this lesson, you will be able to:

- Explain the concept of array
- Know how array elements are arranged in memory
- Explain the terms related with array
- Explain how to define and initialize array
- Explain how to access and write at an index in array
- Explain how to traverse an array using all loop structures
- Use the sizeof() function to find the size of array
- Explain the concept of two dimensional array
- Explain how to define and initialize two dimensional array
- Explain how to access and write at an index in two dimensional array
- Explain what strings are
- Explain how to define string
- Explain the techniques of initializing string
- Explain the most commonly used string functions

### INTRODUCTION

This unit describes a new type of data structure known as array which provides a convenient way to manipulate a collection of same type of data. Array is very commonly used in computer programming as it provides simple solutions to many problems when working with long lists of same type of data, such as numbers. For example, array can be used to sort a list of numbers or to find their total and average.

### 5.1 INTRODUCTION TO ARRAYS

Array allows programmer to use a single variable name to represent a collection of same type of data. This reduces the program size, provides an easy way of handling list of numbers or strings and makes computer programming task simple and easy.

### 5.1.1 CONCEPT OF AN ARRAY

An array is a collection of same type of elements stored in contiguous memory locations. For example, if we want to store marks of six subjects in computer memory, we have to declare six variables, one for each subject. Instead of using six variables we can declare one array variable called *marks* that can store marks of six subjects. This array could be represented as shown in Fig.5-1.

Array marks

45	marks[0]
67	marks[1]
50	marks[2]
79	marks[3]
58	marks[4]
36	marks[5]

Fig.5-1 An array having six elements

Array consists of contiguous memory locations and each cell represents an element of the array. In the array named *marks*, each element of the array represents marks of a subject. The number within the square brackets is called index and it is used to access a specific element of the array. The first element of the array always has the index 0. Therefore, the index of the last element is one less than the size of the array. In the marks array, there are 6 elements so the indexes are from 0 to 5. Index of array is always an integer value.

### 5.1.2 DECLARING AN ARRAY

To declare an array in C++, the type of the elements, the name of array and the number of elements it is required to store need to be mentioned in the declaration statement.

The following is the general form of declaration of array.

**datatype arrayname [arraysize];**

Here, *datatype* is a valid data type (such as integer, float, etc.), *arrayname* is the name of the array which is a valid variable name and *arraysize* is enclosed within square brackets and it specifies the number of elements that can be stored in the array. This type of array is known as one dimensional array.

For example, the following statement declares an array called marks of type integer that can store marks of six subjects.



#### Teacher Point

Explain the concept of arrays with some simple examples.

```
int marks[6];
```

The following are some more examples of array declaration.

```
int a[10], b[15];
```

```
float weight[8];
```

Here, array *a* and *b* are integer arrays, *a* can store 10 values whereas *b* can store 15 values. Array *weight* is declared as floating-point array and it can store 8 weights.

### 5.1.3 INITIALIZATION OF ARRAY

An array can be initialized in declaration statement. The following statement declares the *marks* array as integer of size 6 and assigns marks to each element of the array.

```
int marks[6]={45, 67, 50, 79, 58, 36};
```

When initializing an array, it is not necessary to mention the array size in the declaration statement since the compiler can find out the array size by counting the values in the curly brackets. Therefore, the above statement can also be written as

```
int marks[] = {45, 67, 50, 79, 58, 36};
```

This statement is equal to the following six statements.

```
marks[0]=45;
```

```
marks[1]=67;
```

```
marks[2]=50;
```

```
marks[3]=79;
```

```
marks[4]=58;
```

```
marks[5]=36;
```

### 5.1.4 USING ARRAYS IN PROGRAMS

The following programs demonstrate how arrays can provide simple solutions to problems.

**Program 1:** The following program reads 5 integer values in array *a* and finds their total and average.

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

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

```
void main()
```

```
{
```

```
int a[5], k, total;
```

```
float avg;
```

```
for(k=0;k<5;k++)
```

```
{
```



```
    cout<< "Enter a number:";
    cin>> a[k];
}
total=0;
for(k=0;k<5;k++)
    total=total+a[k];
avg=total/5;
cout<< "\nTotal="<<total<<endl;
cout<< "Average="<<avg<<endl;
getch();
}
```

When this program executes, the statements in the first *for* loop are executed 5 times. In each iteration, a number is assigned to the array elements. The first number typed is stored in array element `a[0]`, the second in `a[1]` and so on. The second *for* loop adds all the numbers to the variable `total`, one by one. After finding the total, average is calculated by dividing the total by 5.

**The following is the execution of the program.**

```
Enter a number:8
Enter a number:13
Enter a number:20
Enter a number:5
Enter a number:9
Total=55
Average=11
```

**Program 2:** The following program reads ten integers and prints the biggest.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int arr[10], k, max;
    k=0;
    cout<< "Enter the numbers, one on each line:\n";
    while(k<10)
    {
        cin>> arr[k];
        k=k+1;
    }
    max=arr[0];
```

```

k=1;
while(k<10)
    if(arr[k]>max)
        max=arr[k];
cout<< "The biggest number is " <<max;
getch();
}

```

This program declares an integer array *arr* of size 10. The variable *k* is loop variable and the variable *max* is used to store the biggest number.

The first while loop stores 10 numbers in the array. The first element of the array is assigned to the variable *max* to assume that it is the biggest number. The second while loop compares the second element of the array with *max*, if it is bigger than *max* then *max* is assigned the second element. In this manner each element of the array from the second element to the last is compared with *max*. Whenever an element is bigger than the current value of *max*, *max* is assigned the value of that element.

**The following is the execution of the program.**

Enter the numbers, one on each line:

38  
45  
78  
345  
80  
127  
27  
46  
9  
133

The biggest number is 345

**Program 3:** The following program reads marks of *n* students and prints the number of students passed. The marks are in the range of 0 to 100 and passing marks are 33.

```

#include<iostream.h>
#include<conio.h>
void main()
{
int marks[50],n,k,pass;
cout<< "Enter the number of students (Max 50):"
cin>>n;
cout<< "Enter the marks, one on each line:\n";
for(k=0;k<n;k++)

```

```
cin>> marks[k];
    pass=0;
for(k=0;k<n;k++)
    if(marks[k]>32)
        pass=pass+1;

cout<< "Number of students passed= " <<pass;
getch();
}
```

The declaration statement declares array marks of 50 elements of type integers, n is the number of students, k is loop variable and pass is a counter that stores the number of students passed in the examination.

When the program is executed, it prompts the user to enter the number of students and stores it in variable n. The array size is 50. Therefore, the number of students should not be more than 50.

The first for loop will fill the array marks with n marks. The user will enter n marks, one on each line one by one.

The counter pass is initialized to 0. The second loop will traverse the array and in each iteration it will check the marks whether they are greater than 32 or not. If the marks are greater than 32, the counter pass will be incremented by 1.

The last output statement will print the number of students passed in the examination.

**The following is the execution of the program.**

Enter the number of students (Max 50):10

55  
78  
43  
25  
80  
19  
30  
72  
38  
12

Number of students passed=6

### 5.1.5 THE sizeof() FUNCTION

The sizeof() function provides the number of bytes occupied to store values for data type named within the parenthesis. It is used to determine the amount of storage reserved for int, float, double, char, etc. data types.

**Program 4:** The following program prints the number of bytes occupied by int, float, double and char data types.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    cout<< "Data Type          Bytes"
    << "\n-----"          <-----"
    << "\nint                "<<sizeof(int)
    << "\nfloat              "<<sizeof(float)
    << "\ndouble             "<<sizeof(double)
    << "\nchar               "<<sizeof(char)
    <<endl;
    getch();
}
```

The output of this program may differ depending on the compiler used. Most of the C++ compilers will produce the following output.

Datatype	Bytes
-----	-----
int	4
float	4
double	8
char	1

The same function can also be used to find the number of bytes reserved for an array. For example, an array is declared as *int a[10]*. The number of bytes reserved for this array can be determined by writing the array name in the parenthesis as *sizeof(a)*. This will return 40 because each element of integer array occupies 4 bytes and there are 10 elements in the array. Similarly the number of bytes occupied by arrays of other data types can also be found.

## 5.2. TWO DIMENSIONAL ARRAYS

### 5.2.1 INTRODUCTION TO TWO DIMENSIONAL ARRAYS

A two dimensional array uses a single variable name to represent a collection of same type of data that is in the form of a table or matrix. It has two dimensions i.e. vertical and horizontal dimensions. Vertical dimension represents rows and horizontal dimension represents columns. Two dimensional array provides an easy way of handling data that is stored in the form of a table.



A two dimensional array *a*, that has three rows and five columns is shown below.

	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[0][3]</code>	<code>a[0][4]</code>
Row 1	<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[1][3]</code>	<code>a[1][4]</code>
Row 2	<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[2][3]</code>	<code>a[2][4]</code>

Fig. 5-2 A two dimensional array

Each cell in the table represents an element of the array in the form of `a[row][col]`, where *row* and *col* written within square brackets are indices. The first index *row*, represents the row number and second index *col*, represents the column number that uniquely identify each element of *a*. The indices *row* and *col* are always integer values and start from zero. For example `a[1][3]` will identify the element in second row and fourth column.

### 5.2.2 DEFINING AND INITIALIZING A TWO DIMENSIONAL ARRAY

#### Defining a Two Dimensional Array

To define a two dimensional array in C++, the type of the elements, the name of the array and the number of elements it is required to store in rows and columns is mentioned in the declaration statement.

The following is the general form of declaration of two dimensional array.

**`datatype arrayname[rowsize][columnsize];`**

Here, *datatype* is a valid data type (such as integer, float, etc.), *arrayname* is the name of array and *rowsize* and *columnsize* enclosed within square brackets specify the number of rows and columns in the two dimensional array.

For example, the following statement declares an array called *a*, that can store marks of 3 students of 5 tests.

**`int a[3][5];`**

The *rowsize* which is 3 represents the number of students and *columnsize* which is 5 represents the number of tests. In C++, indices always start from zero. Therefore, the student number will be in the range of 0 to 2 and test number will be in the range of 0 to 4.

The following statement declares a two dimensional array that stores floating-point values.

**`float height[8][10];`**

Here, *float* specifies that the elements of array are of data type float, *height* is the name of the array and it contains 8 rows and 10 columns. Total number of elements it can store are 80. The row index will be in the range of 0 to 7 and column index in the range of 0 to 9.

### Initializing a Two Dimensional Array

Just like one dimensional array, two dimensional array can also be initialized in declaration statement. The following statement declares and initializes a two dimensional array called *a* that has 3 rows and 4 columns and assigns values to it.

```
int a[3][4]=
{
    {45, 66, 39, 72},    //Elements of first row
    {87, 50, 56, 63},    //Elements of second row
    {44, 23, 58, 88}     //Elements of third row
};
```

Here nested braces are used for assigning values to each row of the array. This statement can be written on a single line as:

```
int a[3][4]={45, 66, 39, 72},{87, 50, 56, 63},{44, 23, 58, 88};
```

The above statement can also be written using a single set of braces as shown below.

```
int a[3][4]={45, 66, 39, 72, 87, 50, 56, 63, 44, 23, 58, 88};
```

The following statement initializes a two dimensional array of type float with 5 rows and 2 columns.

```
float weight[5][2]=
{
    {3.40, 2.75},    //Elements of first row
    {4.75, 3.28},    //Elements of second row
    {8.10, 6.22},    //Elements of third row
    {4.71, 3.92},    //Elements of fourth row
    {1.43, 7.25}     //Elements of fifth row
};
```

### 5.2.3 ACCESSING AND FILLING A TWO DIMENSIONAL ARRAY

Data can be written in any element of a two dimensional array as if it was a normal variable by specifying the index of row and column. For example the following assignment statement will store the value 15 in the element at row 2 and column 4 of two dimensional array named *k*.

```
k[1][3]=15;
```

Two dimensional arrays are generally accessed row by row using nested loop. Therefore, the row index is used as outer loop variable and column index as inner loop variable.

**Note:** It may also be accessed column by column.



The following program demonstrates how two dimensional array elements are accessed using nested loop.

**Program 5:** Program that declares a two dimensional integer array of 3 rows and 4 columns, initializes it with the data 30, 20, 55, 206, 78, 81, 25, 90, 3, 48, 67, 104 and finds the total of all the values.

```
#include<iostream.h>
#include<conio.h>
void main()
{   int i, j, total, k[3][4]={30, 20, 55, 206}, {78, 81, 25, 90}, {3, 48, 67, 104} };
    total=0;
    for(i=0; i<3; i++)                // i represents row number of array k
        for(j=0; j<4; j++)            // j represents column number of array k
            total=total+k[i][j];
    cout<< "Total="<<total<<endl;
    getch();
}
```

The variable total is initialized to 0. When the inner loop is executed for the first time, it finds the total of all the element of row 0 by adding the values 30, 20, 55, and 206 one by one in variable total. The elements of row 1 and 2 are also added in total in the same way when the inner loop is executed for the second and third time.

The output of the program is shown below.

Total=807

**Program 6:** The following program will multiply each element of the array by 2 that has 3 rows and 4 column and display it in the form of matrix. The array is initialized to data values from 1 to 12 with 3 rows and 4 columns.

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main()
{   int i, j, k[3][4]={ {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
                                //Each set of curly brackets initializes 4 values of each row
    for(i=0; i<3; i++)          //Index i represents row number
        for(j=0; j<4; j++)      //Index j represents column number
            cout<<setw(5)<< k[i][j]*2; //Multiply each element with 2 and output
    cout<<endl;                 //After outputting a row move pointer to next row
}
```

```
    getch();
```

```
}
```

In this program the `setw()` function is used to print each data value right justified in a width of 5. This will properly align the data in the output.

The output of the program is given below.

```
    2  4  6  8
   10 12 14 16
   18 20 22 24
```

**Program 7:** The following program prompts the user to enter 12 integers in a two dimensional array that has 3 rows and 4 columns and it finds the largest number.

```
#include<iostream.h>
#include<conio.h>
void main()
{   int i, j, a[3][4], max;
    cout<< "Enter 12 integers separated by space"<<endl;
    for(i=0; i<3; i++)           //Nested loop allows user to enter 12 enters
        for(j=0; j<4; j++)
            cin>>a[i][j];
    max=a[0][0];                 // the first value of array
    for(i=1; i<3; i++)
        for(j=1; j<4; j++)
            if(a[i][j]>max)       //Compare each element of array with max one by one
                max=a[i][j];     //If max is smaller than compared element then replace it
    cout<< "\nThe largest number is "<<max<<endl;
    getch();
}
```

When this program is executed the first nested loop prompts the user to enter 12 integers and stores it in the array `a`. The variable `max` is initialized to 0 and it is used to store the largest number. The second nested loop compares each element of the array one by one with the value stored in `max`. Whenever the value of any array element is found to be greater than the current value of `max` then that value is stored in `max` and the previous value is over written.

The execution of the program is shown below. *(This output is for the given data)*

```
23 44 66 70
12 69 87 45
21 34 74 53
```

The largest number is 87



## 5.3. STRINGS

### 5.3.1 INTRODUCTION

String is a sequence of characters. In C++, character string is stored in a one dimensional array of *char* data type. Each element of character string holds one character. All the strings end with a special character, known as null character and it is represented by '\0'. The null character is automatically appended at the end of string. String is most commonly used item in computer programming to represent name, address, object, book title, etc.

### 5.3.2 A STRING

To define a string in C++, the data type *char*, the name of string and the number of characters it is required to store is mentioned in the declaration statement.

The following is the general form of declaration of string.

```
char stringname[stringsize];
```

The *char* keyword lets the compiler know that a variable of type character is declared. The *stringname* is the name of the string variable. It follows the same name rules of other type of variables. The *stringsize* enclosed within square brackets specifies the number of characters that can be stored in the string.

**Note:** Since the null character is appended at the end of string, if a string has *n* characters then the *stringsize* should be at least *n*+1.

**For example,** the following statement declares a string called *weekday* that can hold maximum 10 characters including one for the null character.

```
char weekday[10];
```

As another example the following statement declares a string called *studentname* that can hold 20 characters including one for the null character.

```
char studentname[20];
```

### 5.3.3 INITIALIZING STRINGS

Just like arrays of integer and floating-point numbers, strings arrays can also be initialized in the declaration statement.

**For example,** the following statement declares and initializes the string variable *weekday* to Sunday.

```
char weekday[10] = {'S', 'u', 'n', 'd', 'a', 'y'};
```

The next statement provides another easy way for the same declaration and initialization.

```
char weekday[10] = "Sunday";
```



### Teacher Point

Use examples to explain the concept of strings.



It allows to declare and initialize the string variable by including the weekday within double quotes. When a string variable is initialized in this way as a whole, the curly brackets are not required.

The following diagram shows how the string variable *weekday* is represented in computer memory.

Index	0	1	2	3	4	5	6	7	8	9
Variable	S	u	n	d	a	y	\0			

Fig.5-2 Memory presentation of character string

The index starts from zero. The compiler automatically places the null character (`\0`) after the last character. The remaining three characters are not defined.

Another way used to initialize a string variable is to type the contents within the curly brackets but without mentioning its size by leaving the square brackets empty. This is shown in the following example.

```
char city[] = "Karachi";
```

In this statement *city* is a string variable that holds 8 characters. Although, "Karachi" has 7 characters, the null character is automatically appended to the end of the string which makes it a string of size 8.

## 5.4 COMMONLY USED STRING FUNCTIONS

To use strings in computer programs, it is essential to learn how string functions are used. The header file `<string.h>` is used when string functions are used in the program. C++ supports a large number of string handling functions in the standard library `<string.h>`.

The most commonly used string functions are described as follows.

### **cin.get() Function**

This function is used to read a string from the keyboard that may contain blank spaces.

The general form of `cin.get()` function is:

```
cin.get(strvar, strsize);
```

It has two arguments. First argument *strvar*, is the name of the string variable and the second argument *strsize*, is the maximum size of the string or character array.

**Program 8:** The following program demonstrates the use of `cin.get()` function.

```
#include<iostream.h>
#include<conio.h>
#include<string.h> // header file to use string functions
void main()
{
    char str[50];
    cout<< "Enter a string:";
    cin.get(str,50)
```



```

    cout<< "You typed:"<<str<<endl;
    getch();
}

```

The following is the execution of the program.

Enter a string: Information Technology

You typed: Information Technology

A string can also be read using *cin* statement but it has some limitation. This is shown in the following program.

**Program 9:** The following program reads a string using *cin* statement and displays it on the screen.

```

#include<iostream.h>
#include<conio.h>
void main()
{   char str[50];
    cout<< "Enter a string:";
    cin>>str;
    cout<< "You typed:"<<str<<endl;
    getch();
}

```

The following is the execution of the program.

Enter a string: Information Technology

You types: Information

In the output only the first word *Information* is printed. The reason for this is that the *cin* statement considers a space as terminating character. The *cin* statement reads strings that consist of a single word. Anything typed after a space is ignored.

### strcpy() Function

The *strcpy()* functions is used to copy contents of a string variable or string constant to another string variable.

The general form of *strcpy()* function is:

```
strcpy(string2,string1);
```

It has two arguments, *string1* and *string2* which are string variables. When it is executed, contents of *string1* will be copied to *string2*.

For example,

```

char string1[10]= "ISLAMABAD", string2[10], string3[10];
strcpy(string2,string1);

```

```
cout<<"string2="<<string2<<endl;
strcpy(string3, "PAKISTAN");
cout<<"string3="<<string3<<endl;
```

The output of the above code will be:

```
string2=ISLAMABAD
string3=PAKISTAN
```

### strcat() Function

The strcat() function is used for concatenation or joining of two strings.

The general form of strcat() function is:

```
strcat(string1,string2);
```

When this function is executed, it will append (concatenate) string2 onto the end of string1.

For example,

```
char string1[10], string2[10];
strcpy(string1,"HOME");
strcpy(string2,"WORK");
strcat(string1,string2);
cout<< "string1=",<<string1<<endl;
```

When the above code is executed, the first strcpy() function will copy the string "HOME" to string variable string1 and the second strcpy() function will copy the string "WORK" to string variable string2. The strcat() function will append the string2 onto the end of string1. Therefore, the output will be:

```
string1=HOMEWORK;
```

### strlen() Function

The strlen() function is used to return the length (the number of characters) of a string.

The general form of strlen() function is:

```
strlen(string);
```

Here, string is a string variable.

For example,

```
char string[10]="COMPUTER";
cout<<"The number of characters in the string are "<<strlen(string)<<endl;
```

The output of the above code will be:



### Teacher Point

Teacher should also explain few more related programs according to the chapter topics.



The number of characters in the string are 8

**Program 10:** The following program demonstrate the use of `strlen()` function. It prints the number of character in each city name.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
{   char city1="LAHORE", city2= "ISLAMABAD",
    city3= "KARACHI";
    cout<<"Characters in city1 are: "<< strlen(city1)<<endl;
    cout<<"Characters in city2 are: "<< strlen(city2)<<endl;
    cout<<"Characters in city3 are: "<< strlen(city3)<<endl;
    getch();
}
```

The output of the program will be:

Characters in city1 are: 6

Characters in city2 are: 9

Characters in city3 are: 7

### strcmp() Function

The `strcmp()` function compares two strings and returns an integer value based on the result of comparison. This comparison is based on ASCII codes of characters.

The general form of `strcmp()` functions is:

```
strcmp(string1,string2);
```

When it is executed, it will compare the first characters of `string1` and `string2`. If they are the same, it will compare the second pair of characters. The comparison will continue until the characters differ or a terminating null-character is reached. During comparison, A is considered less than B and B is considered less than C and so on.

The function will return the following integer values based on the result of comparison.

- i) It will return 0 if `string1` and `string2` are the same.
- ii) It will return 1 if `string1` is greater than `string2`.
- iii) It will return -1 if `string1` is less than `string2`.

**Program 11:** The following program demonstrate the use of `strcmp()` function.

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



### Teacher Point

Teacher should give some home assignments to the students at the end of the chapter.

```

#include<conio.h>
#include<string.h>
void main()
{ char string1="MANGO", string2= "MANGO",
  string3= "POTATO", string4="ORANGE";
  int x, y, z;
  x=strcmp(string1,string2);
  cout<<"string1 and string2 are equal, x="<<x<<endl;
  y=strcmp(string3,string1);
  cout<<"string3 is greater than string1, y="<<y<<endl;
  z=strcmp(string1,string4);
  cout<<"string1 is less than string4, z="<<z<<endl;
  getch();
}

```

The output of the program will be:

```

string1 and string2 are equal, x=0
string3 is greater than string1, y=1
string1 is less than string4, z=-1

```



### Key Points

- An array is a collection of same type of elements stored in contiguous memory locations.
- A two dimensional array uses a single variable name to represent a collection of same type of data that is in the form of a table or matrix.
- The `cin.get()` function is used to read a string from the keyboard that may contain blank spaces.
- The `strcpy()` functions is used to copy contents of a string variable or string constant to another string variable.
- The `strcat()` function is used to append a string onto the end of another string.
- The `strlen()` function is used to return the length (the number of characters) of a string.
- The `strcmp()` function compares two strings and returns an integer value based on the result of comparison.

**Exercise****Q1. Select the best answer for the following MCQs.**

- i. Which of the following is correct declaration of array?  
a) `int arr;`                      b) `int arr{10};`                      c) `int arr[10];`                      d) `int arr(10);`
- ii. What is the index number of the last element of an array with 5 elements?  
a) 5                      b) 4                      c) 0                      d) 6
- iii. What is an array?  
a) An array is a series of elements.  
b) An array is a series of elements of the same type placed in noncontiguous memory locations.  
c) An array is a series of elements of the same type placed in contiguous memory locations.  
d) None of the above.
- iv. Which of the following identifies the first element in array named temp?  
a) `temp[0]`                      b) `temp[1]`                      c) `temp(1)`                      d) `temp(0)`
- v. Which of the following identifies the last element of array declared as `int a[10][10];`?  
a) `a[10][10]`                      b) `a[9][10]`                      c) `a[11][11]`                      d) `a[9][9]`
- vi. Given the following:

```
int k[3][5]=  
    {{3,10,12,27,12},  
     {21,20,18,25,1}  
     {15,16,17,44,4}};
```

What is in `k[1][3]`?

- a) 12                      b) 18                      c) 25                      d) 15

- vii. Given the following:

```
int arr[3][4]=  
    {{12,0,5,10},  
     {7,8,19,30},  
     {33,1,2,22}};
```

Which of the following statements will replace 19 with 50?

- a) `arr[2][3]=50;`                      b) `arr[1][2]=50;`                      c) `arr[2][1]=50;`                      d) `arr[19]=50;`

- viii. Which of the following functions is used to append a string onto the end of another string?

- a) `strcpy()`                      b) `strcat()`                      c) `strlen()`                      d) `strcmp()`

**Q2. Write short answers of the following questions.**

- i. Define array and give its advantages in programming.
- ii. Differentiate between one dimensional and two dimensional array.
- iii. What is the purpose of `sizeof()` function? Give one example.
- iv. Declare an array named x, that has 3 rows and 5 columns and assign it values from 1 to 15 in the declaration statement.

- v. Define string and explain how it is stored in computer memory.
- vi. What is the advantage of using `cin.get()` function over `cin` statement for reading a string.
- vii. Differentiate between `strcpy()` and `strcmp()` functions.
- viii. Differentiate between `strlen()` and `strcat()` functions.

**Q3. Write long answers of the following questions.**

- i. What is an array? Explain one dimensional array in detail with one example.
- ii. Explain two dimensional array in detail with one example.
- iii. What are strings? How strings are defined in C++? Give examples.
- iv. Explain the purpose of the following string functions.  
`strcpy()`, `strcat()`, `strlen()`, `strcmp()`



### Lab Activities

- i. Practice all the programs given in the chapter.
- ii. Write a program that reads ten numbers in an array and prints them in reverse order.
- iii. Write a program that reads ten numbers and print the smallest along with its index.
- iv. For the given array:

```
int arr[15]={4,8,5,1,3,5,0,12,5,7,3,15,8,4,11};
```

Write a program that prints the number of times the number 5 appears in the array.

- v. For the given array:

```
int a[3][2]={{6,3},{7,8},{4,5}};
```

Write a program that displays all the elements in the form of a matrix as shown below and finds its sum.

```
6  3
```

```
7  8
```

```
4  5
```

- vi. For the given array:

```
int arr[3][4]=
```

```
{4,18,-16,11},
```

```
{-5,10,-2,12},
```

```
{15,-3,17,18}};
```

Write a program to find the sum of positive number.

- vii. For the given array:

```
int a[2][4]={{14,8,11,10},{15,12,20,3}},
```

```
b[2][4]={2,3,4,7},{6,7,8,9}};
```



Write a program that adds the two arrays and produces the following output.

Sum of two arrays is:

16 11 15 17

21 19 28 12

viii. Input data from keyboard in a two dimensional array  $x$ , that has  $r$  rows and  $c$  columns and print the sum of each row in the following format.

sum of row 1 =

sum of row 2 =

•  
•  
•

sum of row  $r$  =

ix. Write a program that reads a string, copies it into another string and then prints both strings.

x. Write a program that reads 2 strings of size 20 and perform the following operations.

a) Print both strings with their length.

b) Concatenate the second string onto the end of first string and print it.

xi. Write a program that reads 3 strings and prints the smallest.



# 6

## FUNCTIONS



After completing this lesson, you 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:

- By 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



## 6.1 FUNCTIONS

A **function** is a group of statements that together perform a task. Every C++ program has at least one function, which is `main()`. Additional functions can be defined in the program. Each function performs a specific task.

Functions are one of the main building blocks of any programming language. Functions are used to provide modularity to a program. Creating an application using function makes it easier to understand, edit, check errors etc. Function makes the work easier by writing the code once and executing it again and again as many times as required. C++ functions are divided into two types i.e. library or built-in functions and user defined functions.

### 6.1.1 Types of Functions

C++ functions are categorized into the following two types.

- Library or built-in or System defined functions
- User defined functions

#### Library or Built-in Functions

Library or built-in or system defined functions are the pre-defined functions in C++. Programmers can use these functions by invoking/calling them directly; they do not need to write code for them. C++ provides a series of library or pre-defined functions for various commonly used operations of algebra, geometry, trigonometry, finance, graphics, etc. These functions are part of the C++ language and are completely reliable. Header file **<cmath>** must be included in the program to use library or built-in functions. Some examples of commonly used library or built-in functions are `abs()`, `div()`, `getchar()`, `log()`, `pow()`, `putchar()`, `puts()`, `sqrt()`, `strcmp()`, `time()`, etc. The following program indicates the use of built-in function **`sqrt()`**. It requests a number and calculates its square root using **`sqrt()`** function.

*// Program to find square root of a number using sqrt() built-in function*

```
#include <iostream>
```

```
#include <cmath> // It is required to use built-in functions
```

```
int main()
```

```
{
```

```
    double number, squareroot;
```

```
    cout << "Enter a number: ";
```

```
    cin >> number;
```

```
    // sqrt() is a library function to calculate square root
```



### Teacher Point

Explain the concept of functions with some simple examples.

```

squareroot = sqrt(number);
cout << "Square root of " << number << " = " << squareroot;
return 0;
}

```

### Output of the program

Enter a number: 25

Square root of 25 = 5.0

In the above program, `sqrt()` library function is used to calculate the square root of a number. The code `#include <cmath>` in the above program is a header file. The function definition of `sqrt()` is present in the `cmath` header file. This header file contains definition of all the mathematical functions.

### User-defined Functions

C++ allows programmers to define their own functions. The functions defined by the programmers, according to their needs, are called **Users-defined** functions. If any function is not available as built-in function, users can define their own functions and use them in the same way as built-in functions are used.

#### Working of User-defined Function

Consider the Figure 6.1 for the working of user-defined function in a C++ Program.

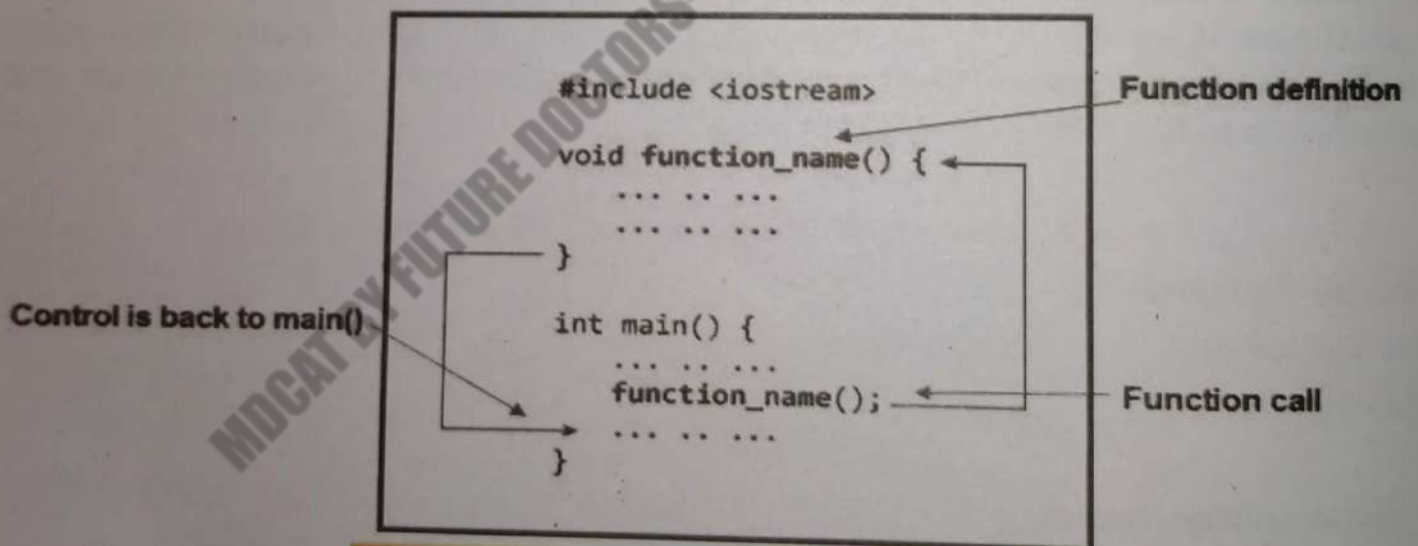


Figure 6.1 Working of User-defined function

- When a program begins running, the system calls the `main()` function, that is, the system starts executing codes from `main()` function.
- When a function is called inside `main()` by its name, it moves to `void function_name()` and all codes inside the function are executed.

- After completion of the function code, the control moves back to the `main()` function where the code after the call to the `function_name()` is executed.

### 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:

```
return-type function-name (parameters)
{
    // function-body
}
```

#### Explanation:

**Return-type:** suggests what the function will return. It can be `int`, `char`, some pointer or even a class object. There can be functions which does not return anything, they are mentioned with `void`.

**Function Name :** is the name of the function, using the function name it is called.

**Parameters:** are variables to hold values of arguments passed while function is called. A function may or may not contain parameter list.

**Function body :** is the part where the code statements of the function are written.

**Example:** The following program contains a function named `sum` with no return type and having parameters `x` and `y` with `integer` datatype.

```
void sum(int x, int y)    // Function definition
{
    int z;
    z = x + y;
    cout << z;
}
```

```
int main()                // main() program
{
    int a = 10;
    int b = 20;
    sum (a, b);            // Function sum is called/used by using variables
    sum (15,30);          // Function sum is called/used by using values/constants
}
```

Here, `a` and `b` are sent as arguments, and `x` and `y` are parameters which will hold values of `a` and `b` to perform required operation inside function. Also 15 and 30 are used directly as values/constants.

#### Output of the program:

30

45

### 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 advantages.

1. Using functions a program can be defined in logical blocks. It will make the code clear and easy to understand.
2. Use of function avoids typing same pieces of code multiple times. User can call a function to execute same lines of code multiple times without re-writing it.
3. Individual functions can be easily tested.
4. In case of any modification in the code user can modify only the function without changing the structure of the program.
5. It is much easier to change or update the code in a function, which needs to be done once.

### 6.1.3 Function Signature

The signature of a function comprises of the parts given below.

- Name of the function
- Arguments: The number, order and data types of the arguments
- Return type of function

Let us consider the *add* function to demonstrate function signature

*//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 Function Components

Each function consists of three components. These are:

- Function prototype/declaration
- Function definition



- Function call

### Function prototype/declaration

Function prototype/declaration, is done to tell the compiler about the existence of the function. Function's return type, its name and parameter list is mentioned. A function prototype/declaration is used before the **main()** function. It ends with a semicolon (;). 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.

**Example:** In the following program the portion in red indicates the function\_prototype/declaration.

```
#include <iostream>
using namespace std;
int sum (int x, int y); // Function prototype/declaration
int main()
{
    int a = 10;
    int b = 20;
    int c = sum (a, b);
    cout << c;
}
int sum (int x, int y) // Function definition
{
    return (x + y);
}
```

### Function definition

A function definition specifies what a function does. It has two parts: a **header** and **function body** enclosed in { }. Function *header* is similar to *function* prototype with the only difference that it has the variables name for the parameters and no semicolon (;).

**Example:** In the following program the portion in red indicates the function\_definition.

```
#include <iostream>
using namespace std;
int sum (int x, int y); // Function prototype/declaration
int main()
{
    int a = 10;
    int b = 20;
    int c = sum (a, b);
    cout << c;
}
int sum (int x, int y) // Function definition
{
```

```
return (x + y);
}
```

### 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 program.

**Example:** In the following program the portion in red indicates the function call.

```
#include <iostream>
using namespace std;
int sum (int x, int y); // Function prototype/declaration
int main()
{
    int a = 10;
    int b = 20;
    int c = sum (a, b); // Function call
    cout << c;
}
int sum (int x, int y) // Function definition
{
    return (x + y);
}
```

### 6.1.5 Scope of Variables in Functions

By the scope of a variable we mean that in which parts of the same program the variable can be accessed. There are three scopes of variables.

- Local scope
- Global scope
- Static scope

#### Local/Automatic variables

Variables defined within a block of a function are called local variables. Their scope is local to the block of function only. 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 the main function. Following is the example using local variables:

```
#include <iostream>
using namespace std;
int main () {
    // Local variable declaration:
    int a, b;
```



```
int c;  
// actual initialization
```

```
a = 10;  
b = 20;  
c = a + b;  
cout << c;  
return 0;
```

**Output of the program:**

30

### Global variables

Variables defined at the top of a program before the `main()` function are called global variables. These variables are accessible by 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; that means, directly in the body of the program.

The following program describes local and global variables.

*// use of global variables*

```
#include <iostream.h>  
#include <conio.h>  
int addition(); // function prototype  
int a,b,c; // Global variables  
int addition()  
{  
    int sum; // Local variable  
    sum = a + b + c;  
    return sum;  
}  
int main() // Start of main()  
{  
    int total; // Local variable  
    a = 5;  
    b = 7;  
    c = 3;  
    total = addition();  
    cout<<"Sum of the three global variables="<<total;  
    getch();  
    return 0;
```

}

**Output of the program**

Sum of the three global variables=15

In this example, the variables *a*, *b* and *c* are defined globally and have therefore been accessed both in *main()* function and *addition()* function.

**Static variables**

*Static variables* are those variables which are preceded by the keyword **static** while declaring them. The general form to declare static variable is:

**static datatype variable name;**

For example:

**static int abc;**

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. Static variables are *local in scope* to their module or function in which they are defined.

Consider the following example to demonstrate the use of static variables.

**// static variables program**

```
#include <conio.h>
#include <iostream.h>
void demo(); // function prototype
void demo()
{
    auto int var1 = 0; // automatic/local variable
    static int var2 = 0; // static variable
    cout<<"Automatic/Local variable = "<<var1<<" "<<"Static variable="<<var2<<endl;
    ++ var1; ++ var2;
}
int main() // start of main() program
{
    int i=0;
    while( i < 3 )
    { demo(); i++; }
    getch();
    return 0;
}
```



### Output of the program

Automatic/Local variable = 0, Static variable = 0

Automatic/Local variable = 0, Static variable = 1

Automatic/Local variable = 0, Static variable = 2

Here, the automatic variable **var1** loses its values when the control goes out of the function body but the static variable **var2** keeps its last value.

### 6.1.6 Parameters

In function prototype and function call, the variables and values, (written in the parenthesis) are called parameters. There are two types of parameters; **formal parameters** and **actual parameters**.

#### Formal parameters

*Formal parameters* are those parameters which appear in function declaration/header and also in function prototype. These are also called *arguments*.

**For example:**

```
void foo(int x); // prototype/declaration -- x is a formal parameter
```

```
void foo(int x) // definition -- x is a formal parameter
```

```
{
```

```
    Statements;
```

```
}
```

#### Actual parameters

*Actual parameters* are those parameters which appear in function calls. These are also called *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, variables holding values or expressions resulting in some values.

**Example:** The following program illustrates the concept of actual and formal parameters.

In the function **main()**, in the following program, **candies**, **toys** and **cups** are actual parameters when used to call function **total\_bill()**. On the other hand, the corresponding variables in **total\_bill** (namely **item1**, **item2** and **item3**, respectively) are formal parameters because they appear in the function definition and receive values passed in the function call.

*// Program to illustrate formal and actual parameters*

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

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

```

#include <iostream.h>
int main ();
int total_bill (int, int, int);
int main()
{
    int bill;
    int candies = 125;
    int toys = 300;
    int cups = 100;
    bill = total_bill ( candies, toys, cup);
    cout<<"The total bill = Rs. "<<bill<< ;
    getch();
    return 0;
}
int total_bill ( int item1, int item2, int item3)
{
    int total;
    total = item1 + item2 + item3;
    return total;
}

```

Actual Parameters

Formal Parameters

### Output of the Program

The total bill = Rs. 525

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 total\_bill:

*// Some valid actual parameters in the function call*

```

bill = total_bill (35, 45, 67);
bill = total_bill (150+640, 205*2, 10-5);

```

### 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**.



### Teacher Point

Use examples to explain the concept of local, global and static variables.



### 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.

Consider the following example:

```
// local functions
#include<iostream.h>
#include<conio.h>
#include<math.h>
int main()
{
    clrscr();
    int n;
    cin>>n;
    cout<< "absolute value of " << n<< " = "<< abs(n);
    getch();
    return 0;
}
```

### Output of the program

-15

absolute value of -15 =15

In the above program, functions *clrscr()*, *abs()*, and *getch()* are the local functions because they lie inside the *main()* function.

### 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* because, usually, they are defined before the *main()* function and are thus accessible to every part of the program.

Consider the following simple example:

```
// global function
#include<iostream.h>
#include<conio.h>
void print()
{
    cout<< "This is global function";
}

int main()
```

```
{
print();
cout<<endl;
print();
getch();
return 0;
}
```

### Output of the program

This is global function

This is global function

In the above program, the function *print ()* is defined at the start of the program and is thus global to every part of the program (*main () function*). Inside the *main ()* program, it can be accessed easily.

### 6.1.8 Inline Function

Using function calls in program, a lot of CPU time is wasted in passing control from the calling program (**main()** function) to the called function and returning control back to the calling program. This limitation can be overcome by the use of **inline** function.

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 calling function (*main()*) and back to it. 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 contents 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 function declaration*

```
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.



### Teacher Point

Demonstrate various aspects of functions with examples.



*// 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 13 and 32 is: "<<min(13, 32);
```

```
cout <<"\nMinimum out of 34 and 65 is:"<<min(34,65);
```

```
getch();
```

```
return 0;
```

```
}
```

### Output of the program

Minimum out of 13 and 32 is: 13

Minimum out of 34 and 65 is: 34

## 6.2 Passing Arguments and returning Values

When we want to execute functions, we need to pass arguments (values) to them. 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 variables
- Passing arguments by reference

#### a. Passing arguments by constants

While calling a function, arguments are passed to the calling function. In passing arguments by constants, the actual constant values (numeric and character) are passed instead of passing the variables holding these constants.

Consider the following program:

*// passing integer constant*

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

```
#include<conio.h>
void show(int x)
{
    cout << " x= " << x << endl;
}
int main()
{
    show(60); // function call
    getch();
    return 0;
}
```

### Output of the program

x = 60

In the above program, the function call, `show(60)`, passes the argument(60) to the function.

Consider another program defining a function `showGender()` that takes a character constant, 'F' or 'M', as argument from the calling function:

*// passing character constants*

```
#include<iostream.h>
#include<conio.h>
void showGender(char ch)
{
    cout << "The gender marker you passed is : " << ch << endl;
}
int main()
{
    showGender('F'); // function call
    getch();
    return 0;
}
```

### Output of the program

The gender marker you passed is : F

#### b. Passing arguments by variables

By default, arguments in C++ are passed by value. When arguments are **passed by value**, a copy of the argument is passed to the function.

Consider the following program to demonstrate the use of passing arguments by value.

*// passing parameters by value example 2*

```
#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 pass 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 function `foo()` although it changes the value within its body.

Consider another example that uses a function **addition** that gets arguments by values.

*// passing parameters by value example 3*

```
#include <iostream.h>
#include <conio.h>
int addition (int a, int b)
{
    int result;
    result=a+b;
    return (result);
}
int main ()
{
```



### Teacher Point

Teacher should also explain few more related programs according to the chapter topics.

```
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 variables. Using pass by reference, the value of arguments can be changed within the function.

In passing arguments by reference, the original variables should precede by the ampersand sign (&). Consider the following example.

#### *// passing parameters by reference example 1*

```
#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 in the call from **main ()** function:

*// passing parameters by reference example 2*

```
#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 have one return 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)**

Consider the following program to demonstrate the concept of default arguments.

*// default arguments program*

```
#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 fewer  
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: **divide (12);**

We have only specified one argument, but the function **divide()** gets the second value from the default argument, *int b=2*, in the function prototype and thus results in 6.

In the second call:

**divide (20,4);**

There are two parameters, so the default value for *b* (*int b=2*) is ignored and '*b*' takes the value passed as argument, that is 4, making the result returned equal to 5 ( $20/4$ ).

### 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 (*main()* function).

The general syntax of the use of *return* statement is given hereunder.

**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;
```

```
}
```

If a function returns a value by the use of *return* statement then the call to the function should be called from a statement or an expression.

**For example:**

*// function call in the case of using return statement in a function*

```
cout<<cube(x); //first method
```

```
int y=cube(x); //second method
```



## 6.3 Function overloading

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 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 'multiply' that operate on integers and floating points numbers.

*// use of overloaded function*

```
#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 (4, 30) << endl;
    cout << "Output of floats=" << multiply (3.5, 4.5);
    getch();
    return 0;
}
```

### Output of the program

Output of integers=120

Output of floats=15.75

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 other of type **float**. The compiler looks at the arguments and calls its respective function. Here, **multiply** (*int a, int b*) is called for **multiply** (4, 30) because the arguments match with the data types of the formal parameters. Similarly, **multiply** (*float a, float b*) is called for the call **multiply** (3.5, 4.5) because the arguments and formal parameters match each other in types.

### 6.3.1 Advantages of Function overloading

- Using function overloading, we can declare multiple functions with the same name that have slightly different purposes.



- Function overloading can significantly lower complexity of programs.
- 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.

### 6.3.2 Use of Function overloading

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 hereunder:

- 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 A)
{
    cout<< "A="<<A<<endl;
}
void print (float B)
{
    cout<< "B="<<B<<endl;
}
int main ()
{
    print(15);
    print(21.45);
    getch();
    return 0;
}
```

### Output of the program

A=15

B=21.45



### Teacher Point

Teacher should give some home assignments to the students at the end of the chapter.



Here, two functions, *print (int A)* and *print (float B)*, have been used with different types of parameters, *int* and *float*. In *main()* function, two calls are used *print(15)* and *print(21.45)*. 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 this, these functions will need to be given different names.



### Key Points

- 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. The *main ()* function consists of function calls.
- 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 its 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.



## Exercise

### Q1. Select the best answer for the following MCQs.

- 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:
  - Inline Function
  - Nested Function
  - Function overloading
  - Recursive Function
- We declare a function with \_\_\_\_\_ if it does not have any return type
  - long
  - double
  - void
  - int
- Arguments of a functions are separated with \_\_\_\_\_.
  - Comma (,)
  - Semicolon (;)
  - Colon (:)
  - None of these
- Variables inside parenthesis of functions declarations have \_\_\_\_\_ level access.
  - Local
  - Global
  - Module
  - Universal
- Observe the following function declaration and choose the best answer:  
`int divide ( int a, int b = 2 )`
  - Variable b is of integer type and will always have value 2
  - Variable a and b are of int type and the initial value of both variables is 2
  - Variable b is international scope and will have value 2
  - Variable b will have value 2 if not specified when calling function

**Q2. Write answers of the following questions.**

- i. What is a function? Explain different types of functions used in C++ with examples.
- ii. Explain function components with examples.
- iii. Define default arguments. Give the advantages and disadvantages of default argument.
- iv. What is meant by the term function overloading? How a function can be overloaded in C++? Explain it with the help of an example program.
- v. What is function signature? Explain its different parts.
- vi. Explain the scope of different types of variables used in functions.
- vii. What are parameters? Explain their types with examples.

**Lab Activities**

- Write a program with a function that takes two *int* parameters, adds them together, and then returns the sum.
- Write a program with a function name "mean" to read in three integers from the keyboard to find the arithmetic mean.
- 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.
- Write a C++ program having two function names **area** and **perimeter** to find the area and perimeter of a square.
- 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.
- 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.



# 7

## POINTERS



After completing this lesson, you 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

### 7.1 INTRODUCTION TO POINTERS

Pointers are powerful features of C++ that differentiates it from other programming languages. With the help of pointers C++ gives users the power to manipulate the data in the computer's memory directly. Pointers are used in C++ program to access the memory and manipulate the address. The variable name refers to that memory space that is occupied by it. Pointers are used to store the address of a variable. The width of the memory address/location depends on the computer architecture. If the computer architecture is 16-bit then it means that it can have  $2^{16}$  memory locations. Therefore, for a pointer to be able to store any memory location in this computer it should be 16 bits or 2 bytes wide. Similarly for 32-bit and 64-bit architecture we need to have pointers with size 4 bytes (32-bit width) and 8 bytes (64-bit width) respectively.

The main advantage of pointer is it can save memory and run faster because it does not have to duplicate the data.

#### 7.1.1 Pointer variable

Pointer variable is a variable that points to a specific address in the memory pointed by another variable. It holds the address of variable. In memory, each and every variable has an



#### Teacher Point

Explain the concept of pointers with some simple examples.

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.

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 are needed 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. 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 (&) or Address 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 **reference** or **address operator** that is denoted by ampersand (&) symbol. This provides address of a memory location.

To understand it, consider the following segment of code.

```
float x = 6.5;
```

```
float *fPointer;
```

```
fPointer = &x; // assign address of x to fPointer;
```

Conceptually, the above lines of code can be pictorially represented as:

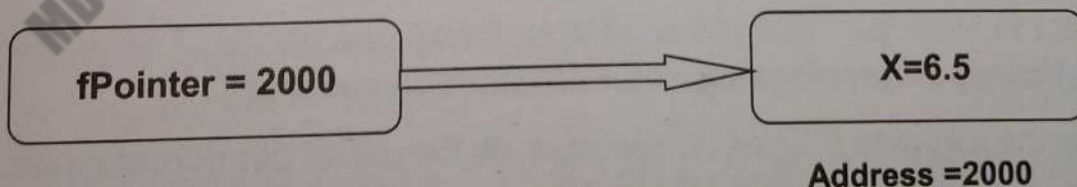


Figure 7.1: Use of the Pointer Variable

Here, the variable *x* has a *float* value 6.5 that is stored at location 2000. The pointer variable *fPointer* points to the variable *x* by holding its address 2000 as its value. In this case, if we want to print the value of the variable *x* it will display 6.5 but if we print the value of *fPointer*



it will display 2000. Note that the value of *fPointer* may differ from 2000 depending upon memory availability.

Consider the following program:

*// use of pointer in a program*

```
#include <iostream.h>
#include <conio.h>
int main()
{
    float x = 6.5;
    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 = 2000

The value of x = 6.5

The value of fPointer = 2000

The output of the program is shown along with the program. It is notable that this program may results some other output (address) on another computer depending upon the availability of the memory.

### 7.1.4 Dereference operator (\*)

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 (\*)**.

Consider the following program to demonstrate the use of **dereference operator (\*)**.

*/\* use of dereference operator (\*) in a program \*/*

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



### Teacher Point

Demonstrate various aspects of pointers with examples.

```

int main()
{
    int n = 200;
    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= 200
The value of Pn = 0x8fc5fff4
The value of (*Pn) = 200
The value of valueN= 200

```

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 *data 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 point 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 statement's.

```
int X;
float Y;
void *pointerVoid;
pointerVoid = &X; // valid
pointerVoid = &Y; // valid
```

In this segment of code, the **void pointer**, *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;
```



#### Teacher Point

Teacher should also explain few more related programs according to the chapter topics.

```
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 only 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 initiated 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:



### Teacher Point

Teacher should give some home assignments to the students at the end of the chapter.

```
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.



## Key Points

- A pointer is a variable that points to or refers to another 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 (\*operator).
- 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 store 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

**Q1. Select the best answer for the following MCQs.**

- If we have the statement `int *Ptr;` then to what Ptr point?
  - Points to an integer type variable
  - Points to a character type variable
  - Points to a floating point type variable
  - None of above
- A pointer is:
  - A keyword used to create variables
  - A variable that stores address of an instruction
  - A variable that stores address of another variable
  - All of the above
- The operator used to get value at address stored in a pointer variable is
  - \*
  - &
  - &&
  - ||
- 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

## Q2. Write answers of the following questions.

- i. What is pointer? Describe the advantages of using pointer variables.
- ii. What is the difference between the dereference operator '\*' and reference operator '&'? Explain with the help of some lines of code.
- iii. How pointer is initialized? Write a simple program to illustrate this concept.
- iv. How the declaration of a pointer variable is different from the declaration of a simple variable.



## Lab Activities

Practice all the programs given in the chapter.



# 8

## CLASSES AND OBJECTS



After completing this lesson, you will be able to:

- Define class and object
- Know the members of class:
  - Data
  - Functions
- Understand and 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

### 8.1 Introduction

The main purpose of C++ programming is to add object orientation to the C++ programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types. A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.



### 8.1.1 Class and object

#### a. Class

The classes are the most important feature of C++ that leads to Object Oriented programming. **Class** is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating instance of that class, which is called object.

The variables inside class definition are called as data members and the functions are called member functions.

A **class** consists of both *attributes* of real world objects and *functions*. Functions perform operations on these attributes. The attributes are called **data members** and the operations are called **member functions**. The general format is:

```
class class_name
{
    access_specifier_1:    // Body of the class
    member1;
    access_specifier_2:
    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 within the *access specifiers*. An access specifier is one of the three types of:

- private
- public
- Protected



#### Teacher Point

Explain the concept of objects and classes with some simple examples.



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 of type **int** (member **x** and member **y**) with **private** access 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

Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects. In other words a variable of type class is called **object**. In C++, when we declare a variable of type class, we call it *instantiating* (from instance) 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; // R1 is the object of class CRectangle**

We can also create more than one object 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.

Consider the following program to implement the class **CRectangle** discussed above.



### Teacher Point

Explain the concept of Inheritance and Polymorphism with daily life examples.

*// Object creation in Class*

```

#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;
}

```

In this program the member function set-values and area are accessed by R1.set-values(44,22) and R1.area().

**Output of the program**

Area of the rectangle=968

**8.1.2 Member of a class**

Class has two members; **data members** and **member functions**.

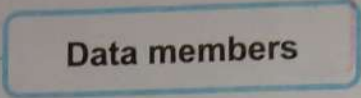
**a. Data member**

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. The following lines of code are taken from the program discussed above to explain data members.

```
class CRectangle
```

```
{
private:
int X;
int Y;
public:
};
```



For each 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*

```
#include<iostream.h>
#include<conio.h>
class Date
{
public:
int Month;
int Day;
int Year;
void SetDate (int nDay, int nMonth, int nYear)
{
Day = nDay;
Month = nMonth;
Year = nYear;
}
void ShowDate()
{
cout<<" Day of birth:"<< Day <<endl;
cout<<" Month of birth:"<< Month <<endl;
cout<<"Year of birth:"<<Year;
}
}
```

```
};
int main ()
{
    Date d1; // Object creation
    d1.SetDate(21,07,2010); // call to SetDate function
    d1.ShowDate(); // call to ShowDate function
    getch();
    return 0;
}
```

### Output of the program

Day of birth: 21  
 Month of birth: 07  
 Year of birth: 2010

### 8.1.3 Access specifiers

**Access specifiers** in C++ define how the members of the class can be accessed. *Access specifiers* determine which member of a class is accessible in which part of the class/program. The most commonly used access specifiers in C++ are given below.

- 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.

Consider the following sample program to demonstrate **private** access specifier.

*// private access specifier example 1*

```
#include<iostream.h>
#include<conio.h>
class TestPrivate
{
private:
    int X;
public:
    void Test()
    {
        X=10;
        cout<< "X is private member and accessed inside the class"<<endl;
    }
}
```

```

cout<< "Value of X="<<X<<endl;
}
};

int main()
{
    TestPrivate P1; //P1 is an object
    P1.Test(); // valid
    //P1.X=10 // invalid
    getch();
    return 0;
}

```

Invalid: because X is private and cannot be accessed from outside the class

A detailed program explaining the use of *private* members within and outside the class, i.e. in *main()* function, is given hereunder.

*// private access specifier example 2*

```

#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 = 12;
    //cDate.Month = 05;
}

```

Okay: because private members are accessible within the class

Invalid: because private members cannot be accessed from outside the class



```
//cDate.Year = 2010;  
cDate.SetDate(12, 05, 2010); //Valid  
getch();  
return 0;  
}
```

### 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 to class PublicTest  
PT1.X; //access public data members  
PT1.showPublic(); // access public member function  
getch();  
return 0;  
}
```



### Teacher Point

Teacher should also explain few more related programs according to the chapter topics.

In the above program, the class *PublicTest* is defined that 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()*.

Here is an example of a class that uses all *access specifiers*.

*// private and public access specifier program*

```
#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 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/Encapsulation**

**Data Hiding** is one of C++ features in which the members of a class are protected against illegal access from outside the class. In C++, we have the facility of hiding data using different access levels: private, protected, public in classes. It is also called **Encapsulation**.

**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.

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

**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.

**a. Constructor**

A **Constructor** is a special type of member function that initializes an object automatically when it is created. Compiler identifies a given member function is a constructor by its name and the return type.

Unlike functions, constructors have specific rules/features for how they must be named:

- Constructors have the same name as that of the class
- Constructors have no return type (not even void)
- Constructor is always public

Consider the following simple program to explain the concept of constructor in classes.

## //Constructor Example 1

```

#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 having 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.

Consider another program to implement the class **CRectangle** including a constructor:

## //Constructor Example 2

```

#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

#### i. Default constructor / implicit constructor

A constructor without any arguments or with default values for every argument, is treated as default / implicit constructor. If we do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like:

*// default/implicit 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.

The following program defines user-defined *constructor*.

*// user-defined 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

**Explanation:**

Here, a constructor "CExample" with two parameters of type *int* has been declared that initializes *private* variables 'a' and 'b'. The object declaration is done as follows:

**CExample CObj (5,10);**

**iii. Constructor overloading**

Using more than one constructor 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 parameter (s) match the arguments used in the function call. In the case of constructor overloading, that constructor is executed whose parameter(s) match(es) the arguments passed on the object declaration. Consider the following program.

*// constructors overloading*

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

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

```
class CRectangle
```

```
{
```

```
int width, length;
```

```
public:
```

```
CRectangle ()      // constructor
```

```
{
```

```
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

### Explanation:

In the above example, the first object r1 passes the arguments (5,14) to the second constructor and thus the output generated is 70. The second object r2 calls the non-parameterized constructor of the class and initializes the variables width with 5 and length with 15 and thus the result calculated 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 already allocated to an object during its creation. They are called automatically when objects are destroyed. They are denoted by ~ destructor.

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.

**Note:** 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 destructor.

*// constructors and destructors example*

```
#include <iostream.h>
#include <conio.h>
class A
{
public:
A () //constructor function
{
cout<<"I am constructor"<<endl;
}
~A () // destructor function
```



```
{  
    cout<<"I am destructor";  
}  
};  
int main ()  
{  
    A a1;  
    getch();  
    return 0;  
}
```

### Output of the program

I am constructor

I am destructor

### Explanation:

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 by `~A ()`, a message, "I am destructor", is displayed to verify that the destructor is executed.

### 8.1.6 Declaration of objects for accessing members of a class

Normally, the execution of each program written in C++ programming language starts from the `main()` function. As we know that class is one of the main building blocks of OOP languages. These classes are written outside the `main ()` body, therefore, they must be brought into the scope of `main` program. For this purpose, objects are declared by the help of which class members are accessed. The following program shows how objects are declared.

```
// Class objects declaration  
#include <iostream.h>  
#include <conio.h>  
class B  
{  
    private:  
    public:  
};  
int main ()  
{  
    B b1,b2,b3; // objects declaration  
    getch();  
    return 0;  
}
```



### a. Accessing data members

To access members of an object, whether data members or member function, dot operator (.) is used with the member name as shown hereunder.

#### Object.member\_name;

Consider the following program:

*// Accessing class data members*

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

class B
{
private:
int x; // defines private data member
public:
int y; // defines public data member
};

int main ()
{
B b1;
//b1.x; (invalid: private members are not accessible outside the class)
b1.y; // access public data member
getch();
return 0;
}
```

#### Explanation:

The above program access the public data member 'y' using the statement "b1.y;" where 'b1' is the object of the class 'B'.

### b. Accessing member function

Member functions of a class can be accessed by the same way as data members are accessed i.e. by the use of dot operator (.) in concatenation with the member function name as given below:

#### Object.member function\_name;

Consider the following program:

*// Accessing class member functions*



```
#include <iostream.h>
#include <conio.h>
class B
{
private:
int x; // defines private data member
public:
int y; // defines public data member
void memberfunction() // defines member function
{
int x=10;
int y= 20;
cout<< "Sum of x and y in member function is: "<<(x+y);
}
};
int main ()
{
B b1;
b1.memberfunction(); // access member function
getch();
return 0;
}
```

### Output of the program

Sum of x and y in member function is: 30

### 8.1.7 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 same thing (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 **Polygon** as base class which has a series of child classes that describe polygons i.e. Square and Pentagon. They have certain common properties, such as both can be described by means of only two sides and angles.

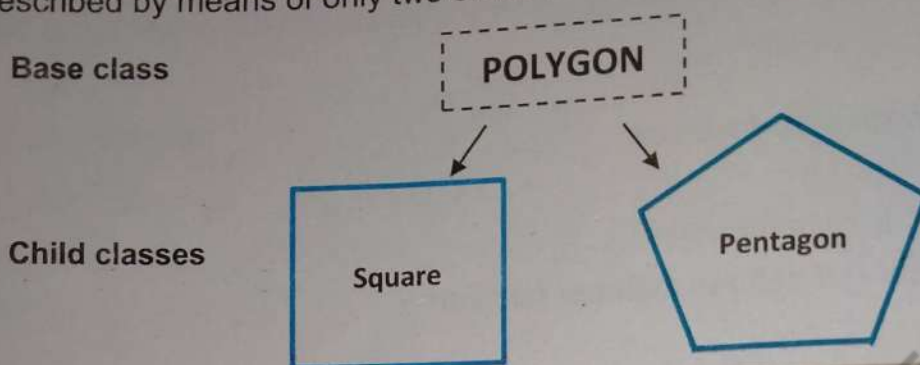


Figure 8.1: Inheritance of Polygon Class

- We can also represent **Patient** class 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:

Name, Father name, Age  
Address, Disease

Ward No, Bed No.  
Next date of visit

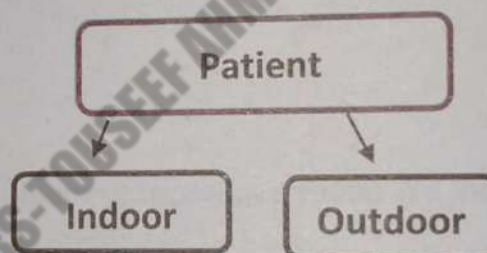


Figure 8.2: Inheritance of Patient Class

**Syntax of using inheritance in classes:**

*// syntax of using inheritance in classes*

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 hereunder.**

*// inheritance of classes*

#include <iostream.h>

#include <conio.h>

**Teacher.Point**

Teacher should give some home assignments to the students at the end of the chapter.



```
class A // base class
{
public:
void showa()
{
cout<<"I am in base class A"<<endl;
}
};

class B: public A //class B is derived from class 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 inheriting function of base class
b1.showb();
getch();
return 0;
}
```

### Output of the program

```
I am in base class A
I am in derived class B
```

### Explanation:

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 class B that have produced the output as shown above.

**Note:** In inheritance, sometimes a class is derived from more than one parent class and the phenomenon is called **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. It is the use of a single function or operator in many ways.

In C++, polymorphism can be achieved by one of the following concepts.

- Function name 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. (*Discussed in detail in unit 6, as topic 6.3*).

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 hereunder.

**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).

### Daily life examples of Inheritance and Polymorphism:

#### Inheritance

Acquiring some of the common qualities from parents (for instance eyes like mother, hair like father, etc) is called inheritance.

A new model car having some inherited features from the old model, like break system and navigation system, etc.

#### Polymorphism

It means multiple possible states for a single property. For example, a person can be a student as well as a friend to another person. Also a person can be an engineer as well as a teacher.



## Key Points

- 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 tilde 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

**Q1. Select the best answer for the following MCQs.**

- 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

**Q2. Write answers of the following questions.**

- i. Explain the terms object and class with examples.
- ii. How objects are created to access members of a class?
- iii. What are access specifiers? Explain private and public specifier with examples.
- iv. What is data hiding? Explain.
- v. Explain constructor and destructor class member functions with examples.
- vi. Explain inheritance and polymorphism with examples.



### Lab Activities

- 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.
- Write a C++ program implementing inheritance between Employee (base class) and Manager (derived class).
- Write a C++ program implementing a class with the name ConstDest having a constructor and destructor functions in its body.
- 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, initially to zero. The class should have another function name ToSecond to convert and display the time pass by the object into seconds.

MDCAT BY FUTURE DOCTORS-TOUSEEF AHMAD-03499815886



# 9

## FILE HANDLING



After completing this lesson, you 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

### 9.1 INTRODUCTION

A **file** is a collection of bytes stored on a secondary storage device, like hard disk. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image. The meaning attached to a particular file is determined entirely by the data structures and operations used by a program to process the file.

**File Handling** concept in C++ language is used for storing data permanently in computer. It provides a mechanism to write output of a program into a file and read from a file. The basic operations involved in file handling are:

- Opening file
- Reading and writing 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

**Text files** can be a stream of characters that a computer can process sequentially. It is not only processed sequentially but only in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time.

Similarly, since text files only process characters, they can only read or write data one character at a time. (In C++ Programming Language, Functions are provided that deal with lines of text, but these still essentially process data one character at a time.) A text stream in C++ is a special kind of file. Depending on the requirements of the operating system, newline characters may be converted to or from carriage-return/linefeed combinations depending on whether data is being written to, or read from, the file. Other character conversions may also occur to satisfy the storage requirements of the operating system. These translations occur transparently and they occur because the programmer has signaled the intention to process a text file.

#### Binary files

Binary file is a collection of bytes. In C++ Programming Language a byte and a character are equivalent. Hence a binary file is also referred to as a character stream, but there are two essential differences.

1. No special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.
2. C++ Programming Language places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

Binary files can be either processed sequentially or, depending on the needs of the application, they can be processed using random access techniques. In C++ Programming Language, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data. This indicates a second characteristic of binary files.

They are generally processed using read and write operations simultaneously.

For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file.



#### Teacher Point

Explain the concept of File handling with some simple examples.



These kinds of operations are common to many binary files, but are rarely found in applications that process text files.

### 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 be fully qualified path name like "C:\myprogs\sale.txt".

#### 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, filename representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

Mode	Description
ios::in	Open for input operations (Reading a file)
ios::out	Open for output operations (Writing a file)
ios::binary	Open in binary mode.
ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::app	Append mode. All output to that file to be appended to the end.
ios::trunc	If the file opened for output operations, its existing 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 *ofstream*, *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.

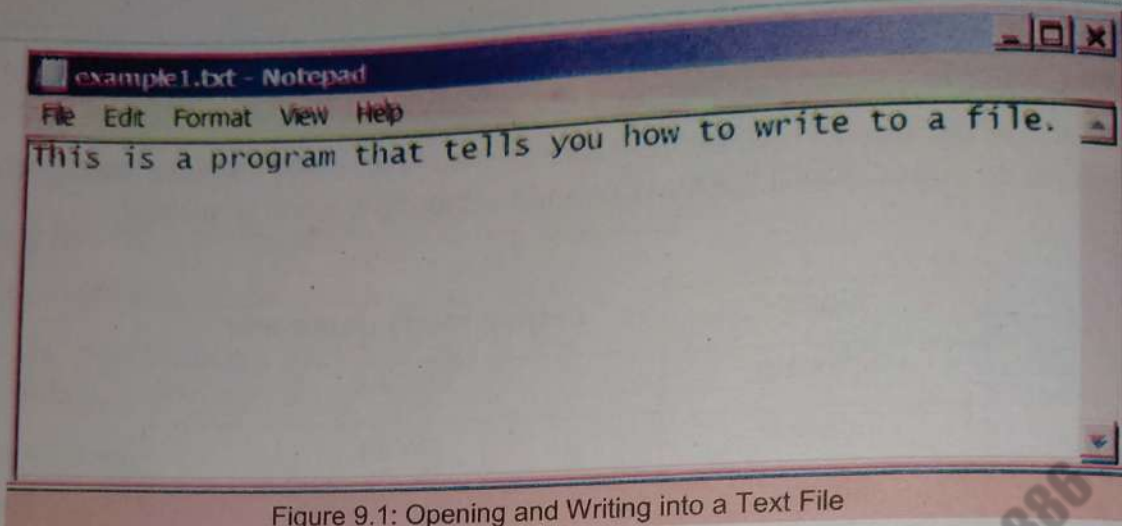


Figure 9.1: Opening and Writing into a Text File

### Reading input file

To read a word from the file we can write as:

```
>> c;
```

So, the first word of the file will be read in `c`, where `c` is a character array. It is similar as with `cin`. There are certain limitations to this. It can read just one word at one time. It on encountering a space, it will stop reading further. Therefore, we have to use it to read the complete file. We can also read multiple words at a time as:

```
> c1 >> c2 >> c3;
```

The first word will be read in `c1`, second in `c2` and third in `c3`. Before reading the file, we know some information regarding the structure of the file. If we have a file of an employee, we 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.

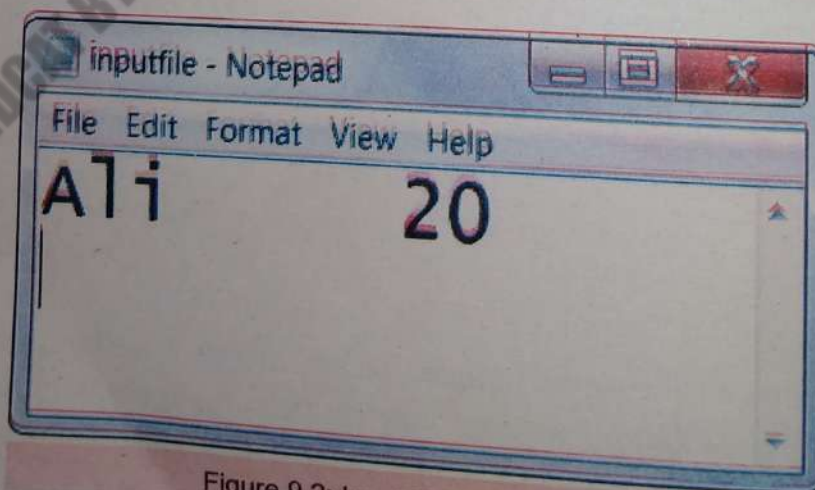


Figure 9.2: Input File to be Read

*//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:

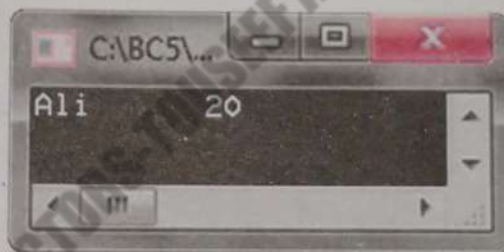


Figure 9.3: Output of Reading Program

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 employees. Figure 9.4 shows **myfile.txt** which is followed by the complete program to describe how it is opened, read and closed.

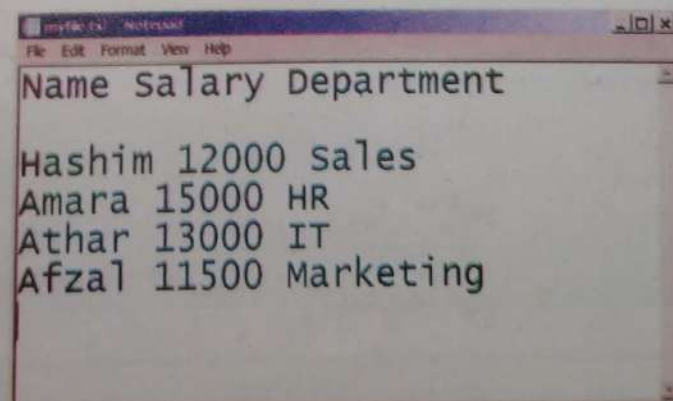


Figure 9.4: Input File to be Read



*// this program reads from the text file "myfile.txt" which contains the employee information*

```
#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
    char inputFileName[] = "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;
}
```

Name	Salary	Department
Hashim	12000	Sales
Amara	15000	HR
Athar	13000	IT
Afzal	11500	Marketing

Figure 9.5: Output of the Program that is Read from Input File "myFile.txt"

**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**.

**c. 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 end of a file respectively. The following sections explain them with the help of proper examples.

**d. 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();
```

**e. eof() – end-of-file**

The **eof()** is a pointer which returns true when there are no more data to be read from input file stream, and false otherwise. It means that this function checks whether control 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..

**Teacher Point**

Teacher should also explain few more related programs according to the chapter

```
// 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
Zonash 21
```

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

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.

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**.

## 9.1.5 Types of 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 "charactersfile.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;
    }
    reads.close();
    getch();
    return 0;
}
```

#### Output of the program

r  
e  
a  
d  
t  
h  
e  
  
f  
i  
l  
e  
  
o

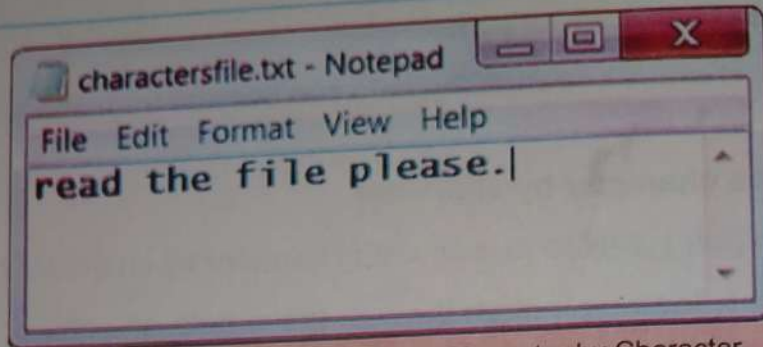


Figure 9.7: Input File to be Read Character by Character

## 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.

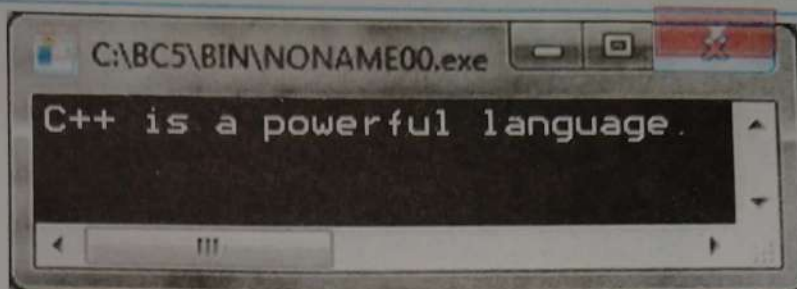


Figure 9.8: File Written by Character Stream

### 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.

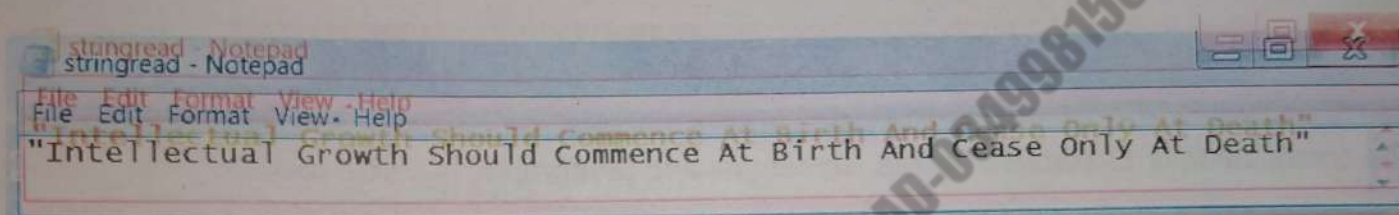


Figure 9.9: Input File that will be Read using String Stream

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;
    }
}
```



### Teacher Point

Teacher should give some home assignments to the students at the end of the chapter.

```

read.close();
getch();
return 0;
}

```

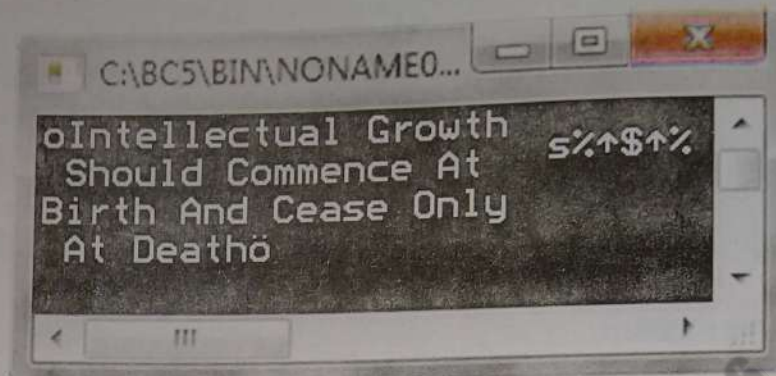


Figure 9.12: File Read by String Stream



### Key Points

- The combination of characters, words, sentences and paragraphs 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 being but readable by the computers. Binary file 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 etc.
- The `bof( )` is a pointer which true if the current position of the pointer is at the beginning 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 as a sequence of bytes of varying length that is used as a buffer to hold data that is waiting to be processed

**Exercise****Q1. Select the best answer for the following MCQs.**

- 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.

**Q2. Write answers of the following questions.**

- i. What is file handling? Explain different types of file.
- ii. Describe different types of operations performed on the files.
- iii. Explain `bof()` and `eof()` functions.
- iv. Define streams. Describe Input and Output streams in detail.
- v. What is meant by the term mode of file opening? Describe different modes of opening file.

**Lab Activities**

Practice all the programs given in the chapter.

# Content Authors

## Mohammad Sajjad Heder

Mohammad Sajjad Heder has done Master (MS) in Computer Science from George Mason University, Virginia, USA and B.E. Mechanical Engineering from Kim Check Engineering College, Korea. He has profound knowledge of computers and is highly experienced in teaching the subject of Computer Science. He has been teaching Computer Science to students of SSC and HSSC for the last twenty years. He is the author of previous HSSC-I & II textbooks of Computer Science. These books were approved by Ministry of Education, Islamabad and prescribed by the Federal Board of Intermediate and Secondary Education for about twenty years.

## Mohammad Khalid

Mohammad Khalid is Head of Computer Science Department at OPF Boys College, Islamabad. He did his MS in Computer Science from the University of Peshawar and earned his B.Ed. degree in Science from Institute of Education and Research, University of Peshawar. He has been instructor and teacher in Computer Science for the last fifteen 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 since 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 many textbooks of Computer Science for different Textbook boards.

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.



## قومی ترانہ

پاک سرزمین شاد باد! کشورِ حسین شاد باد!  
تو نشانِ عزمِ عالی شان ارضِ پاکستان  
سرگزِ یقین شاد باد!

پاک سرزمین کا نظام قوتِ اخوتِ عوام  
قوم، ملک، سلطنت پائندہ تابندہ باد!  
شاد باد منزلِ مسرور!

پرچم ستارہ و ہلال رہبرِ ترقی و کمال  
ترجمانِ ماضی، شانِ حال جانِ استقبال  
سایہ خدائے ذوالجلال!



**National Book Foundation**  
as  
**Federal Textbook Board**  
**Islamabad**