Object Oriented Programming

(Code : 210243)

Semester III - Computer Engineering

(Savitribai Phule Pune University)

Strictly as per the New Credit System Syllabus (2019 Course)
Savitribai Phule Pune University w.e.f. academic year 2020-2021

Dr. Rachana Anil Satao
Ph.D. (Computer Engineering)
Assistant Professor,
Department of Computer Engineering,
Smt. Kashibai Navale College of Engineering,
Vadgaon Budruk, Pune-411041

Araddhana Arvind Deshmukh
M.E. Computer Engineering (SPPU), Pune
Ph.D. Research Scholar, Aarhus University, Denmark
Asst. Professor, Department of Computer Engineering,
Smt. Kashibai Navale College of Engineering,
Vadgaon Budruk, Pune-411041

Deepa Ashish Padalkar
M.E. Computer Engineering
Assistant Professor,
Department of Computer Engineering,
Smt. Kashibai Navale College of Engineering,
Vadgaon Budruk, Pune-411041

Ansari M. A.
Ph.D. (Computer Engineering)*
Assistant Professor,
Department of Computer Engineering,
Smt. Kashibai Navale College of Engineering,
Vadgaon Budruk, Pune-411041

Techknowledge Publication

For Lab Manual visit our website https://techknowledgebooks.com/library/#student_downloads

TechKnowledge Publications

PO121A Price ₹ 195/-

(Book Code : PO121A)

6.9 Constructor, Destructor and Exception Handling 6-11

6.10 Exception and Inheritance 6-12

Chapter 7 : Templates 7-1 to 7-10

7.1 Introduction to Generic Programming 7-1

7.2 Power of Templates 7-3

7.2.1 Need of Templates in C++.....7-4

7.3 Function Template / Applying Generic Function 7-4

7.4 Overloading Function Templates.....7-6

7.5 Overloading Class Template 7-7

7.6 Class Template and Nontype Parameters 7-8

7.7 Template and Friends Generic Functions.....7-8

7.8 The Type Name and Export Keywords 7-9

Unit - VI

Chapter 8 : Standard Template Library (STL) 8-1 to 8-20

8.1 Introduction to Standard Template Library (STL).....8-1

8.1.1 Components of STL 8-1

8.2 Containers.....8-2

8.2.1 Sequence Container and Associative Containers.....8-2

8.2.2 Container Adapters.....8-3

8.3 Functions in Container Classes.....8-4

8.3.1 Array 8-4

8.3.2 Deque 8-4

8.3.3 Queue 8-8

8.4 STL Vector 8-8

8.5 STL Stack 8-9

8.6 STL Lists 8-10

8.7 Algorithms 8-11

8.8 Basic Searching and Sorting Algorithms.....8-12

8.9 Min Max Algorithms 8-13

8.10 Set Operations 8-13

8.11 Heap Sort 8-14

8.12 Iterators - Input, Output, Forward, Bidirectional and Random Access 8-15

8.12.1 Types of Iterators 8-16

8.13 Object Oriented Programming - Road Map to Future.....8-17

- **Model Question Paper (End Sem.) .....Q-1 to Q-2**

Technical Knowledge Publication

❑❑❑

Object Oriented Programming (Code : 210243)

Dr. Rachana A. Satao, Araddhana A. Deshmukh, Deepa A. Padalkar, Ansari M. A.

(Semester III – Computer Engineering, (Savitribai Phule Pune University))

Copyright © by Authors. All rights reserved. No part of this publication may be reproduced, copied, or stored in a retrieval system, distributed or transmitted in any form or by any means, including photocopy, recording, or other electronic or mechanical methods, without the prior written permission of the publisher.

This book is sold subject to the condition that it shall not, by the way of trade or otherwise, be lent, resold, hired out, or otherwise circulated without the publisher's prior written consent in any form of binding or cover other than which it is published and without a similar condition including this condition being imposed on the subsequent purchaser and without limiting the rights under copyright reserved above.

First Printed in India : January 2015 (Pune University)

First Edition : August 2020

This edition is for sale in India, Bangladesh, Bhutan, Maldives, Nepal, Pakistan, Sri Lanka and designated countries in South-East Asia. Sale and purchase of this book outside of these countries is unauthorized by the publisher.

TechKnowledge Publications

ISBN : 978-93-89889-53-6

Published by

TechKnowledge Publications

Head Office : B/5, First floor, Maniratna Complex, Taware Colony, Aranyeshwar Corner,
Pune - 411 009. Maharashtra State, India
Ph : 91-20-24221234, 91-20-24225678.
Email : info@techknowledgebooks.com,
Website : www.techknowledgebooks.com

[210243] (FID : PO121) (Book Code : PO121A)

We dedicate this Publication soulfully and wholeheartedly,

in loving memory of our beloved founder director,

Late Shri. Pradeepji Lalchandji Lunawat,

who will always be an inspiration, a positive force and strong support behind us.

"My work is my prayer to God"

- Lt. Shri. Pradeepji L. Lunawat

Soulful Tribute and Gratitude for all Your
Sacrifices, Hardwork and 40 years of Strong Vision…

Preface

My Dear Students,

We are extremely happy to come out with this book on **"Object Oriented Programming"** for you. The topics within the chapters have been arranged in a proper sequence to ensure smooth flow of the subject.

We present this book in the loving memory of **Late Shri. Pradeepji Lunawat,** our source of inspiration and a strong foundation of **"TechKnowledge Publications"**. He will always be remembered in our heart and motivate us to achieve our milestone.

We are thankful to Shri. J. S. Katre, Shri. Shital Bhandari, Shri. Arunoday Kumar and Shri. Chandroday Kumar for the encouragement and support that they have extended. We are also thankful to Seema Pradeep Lunawat for technology enhanced reading, E-books support and the staff members of TechKnowledge Publications for their efforts to make this book as good as it is. We have jointly made every possible efforts to eliminate all the errors in this book. However if you find any, please let us know, because that will help us to improve further.

We are also thankful to our family members and friends for their patience and encouragement.

- Authors

TechKnowledge Publication

Syllabus

210243 : Object Oriented Programming

Teaching Scheme	Credit Scheme	Examination Scheme and Marks
Lectures : 03 Hours/Week	03	**In-Sem (TH) :** 30 Marks **End-Sem (TH) :** 70 Marks

Prerequisite Courses :

Programming and Problem Solving

Course Objectives :

The course is intended to provide the foundations and in-depth understanding of a modern object-oriented language and develop skills in software development, through an algorithmic approach and the application of principles of objected oriented programming.

1. To learn the object-oriented programming paradigm, focusing on the definition and use of classes along with the fundamentals of object-oriented design.
2. To learn the syntax and semantics of the C++ programming language.
3. To understand the concept of data abstraction and encapsulation, how to design C++ classes for code reuse, how to implement copy constructors and class member functions, to overload functions and operators in C++.
4. To learn how inheritance and virtual functions implement dynamic binding with polymorphism.
5. To learn how to design and implement generic classes with C++ templates and how to use exception handling in C++ programs.

Course Outcomes : On completion of the course, learner will be able to :

CO1 : **Apply constructs :** sequence, selection and iteration; classes and objects, inheritance, use of predefined classes from libraries while developing software.

CO2 : **Design** object-oriented solutions for small systems involving multiple objects

CO3 : **Use** virtual and pure virtual function and complex programming situations.

CO4 : **Apply** object-oriented software principles in problem solving.

CO5 : **Analyze** the strengths of object-oriented programming.

CO6 : **Develop** the application using object oriented programming language(C++).

UNIT - I : Fundamentals of Object Oriented Programming **(07 Hours)**

Introduction to Object Oriented Programming, Need of object-oriented programming, Fundamentals of object-oriented programming : Namespaces, Objects, Classes, Data members, Methods, Messages, Data encapsulation, Data abstraction and information hiding, Inheritance, Polymorphism. Benefits of OOP, C++ as object oriented programming language.

C++ Programming : C++ programming basics, Data Types, Structures, Enumerations, Control structures, Arrays and Strings, Class, Object, Class and data abstraction, Access specifiers, Separating interface from implementation. **Functions :** Function, Function prototype, Accessing function and utility function, Constructors and destructor, Types of constructor, Objects and Memory requirements, Static members: variable and functions, Inline function, Friend function. **(Refer Chapter 1)**

Technical Publications

(Book Code : PO121A)

UNIT - II : Inheritance and Pointers **(07 Hours)**

Inheritance : Base class and derived class, Protected members, Relationship between base class and derived class, Constructor and destructor in derived class, Overriding member functions, Class hierarchies, Public and Private inheritance, Types of inheritance, Ambiguity in multiple inheritance, Virtual base class, Abstract class, Friend class, Nested class.

Pointers : Declaring and initializing pointers, Indirection operators, Memory Management: new and delete, Pointers to Objects, this pointer, Pointers Vs Arrays, Accessing arrays using pointers, Arrays of pointers, Function pointers, Pointers to pointers, Pointers to derived classes, Passing pointers to functions, Return pointers from functions, Null pointer, void pointer. **(Refer Chapters 2 and 3)**

UNIT - III : Polymorphism **(07 Hours)**

Polymorphism : Introduction to Polymorphism, Types of Polymorphism, Operator Overloading- concept of overloading, Operator overloading, Overloading unary operators, Overloading binary operators, Data conversion, Type casting (implicit and explicit), Pitfalls of operator overloading and conversion, Keywords : explicit and mutable. Function overloading, Run time polymorphism - Pointers to base class, Virtual function and its significance in C++, Pure virtual function and Virtual table, Virtual destructor, Abstract base class. **(Refer Chapter 4)**

UNIT - IV : Files and Streams **(07 Hours)**

Data hierarchy, Stream and files, Stream classes, Stream errors, Disk file I/O with streams, File pointers and error handling in file I/O, File I/O with member functions, Overloading the extraction and insertion operators, Memory as a stream object, Command-line arguments, Printer output. **(Refer Chapter 5)**

UNIT - V : Exception Handling and Templates **(07 Hours)**

Exception Handling : Fundamentals, Other error handling techniques, Simple exception handling - Divide by zero, Multiple catching, Re-throwing an exception, Exception specifications, User defined exceptions, Processing unexpected exceptions, Constructor, Destructor and exception handling, Exception and inheritance.

Templates : The power of templates, Function template, Overloading function templates, and class template, Class template and nontype parameters, Template and friends generic functions, The type name and export keywords.

(Refer Chapters 6 and 7)

UNIT - VI : Standard Template Library (STL) **(07 Hours)**

Introduction to STL, STL Components, Containers : Sequence container and associative containers, Container adapters, Application of Container classes : vector, list,

Algorithms : Basic searching and sorting algorithms, Min-max algorithm, Set operations, Heap sort,

Iterators : Input, Output, Forward, Bidirectional and random access. Object Oriented Programming -a road map to future. **(Refer Chapter 8)**

❑❑❑

Technknowledge Publication

(Book Code : PO121A)

Unit - I

Chapter 1 : Fundamentals of Object Oriented Programming 1-1 to 1-55

1.1 Introduction to Computer System 1-1
1.2 Storage Structure 1-2
1.3 Operating Systems 1-4
1.3.1 Examples of OS 1-4
1.4 Programming Language 1-5
1.4.1 Challenges in Software Development 1-5
1.4.2 Software Evolution 1-6
1.4.3 Types of Programming Language 1-6
1.4.4 Most Popular Programming Language 1-7
1.4.5 Qualities of Good Programmer 1-7
1.4.6 Terminologies in Programming 1-8
1.5 Introduction to Procedural, Modular Programming 1-9
1.5.1 Characteristics of Procedure Oriented Programming 1-10
1.5.2 Limitations of Procedural Programming 1-11
1.6 Introduction to Object - Oriented Programming 1-11
1.6.1 Need of Object-Oriented Programming (OOP) 1-12
1.6.2 Applications of OOP 1-12
1.6.3 Difference between Procedure Oriented and Object Oriented Programming 1-12
1.7 Object Oriented Languages and Object Based Languages 1-13
1.8 Fundamentals of Object-Oriented Programming 1-13
1.9 Benefits of Object - Oriented Programming 1-15
1.10 C++ as Object Oriented Programming Language 1-16
1.10.1 C++ Programming Basics 1-16
1.10.2 ++ Extension of C 1-16
1.10.3 Structure of C++ Program 1-17
1.10.4 C++ Program Execution 1-18
1.10.5 Different Tools for C++ Program Development 1-19
1.10.6 Main Function in C++ 1-19
1.11 Building Blocks of C++ 1-20
1.11.1 Comments 1-20
1.11.2 Tokens 1-20
1.11.2(A) Keywords 1-20
1.11.2(B) Identifiers 1-20
1.11.2(C) Constant 1-21
1.11.3 Const Keyword 1-21
1.12 Data Types 1-22
1.13 Structures 1-23
1.14 Enumerations 1-23
1.15 Control Structure 1-24
1.15.1 Selection Statement 1-24
1.15.2 Looping 1-26
1.16 Arrays and Strings 1-28
1.17 Class 1-29
1.18 Object 1-30
1.19 Class Scope and Accessing Class Members 1-31
1.20 Access Specifiers 1-32
1.21 Class and Data Abstraction 1-33
1.22 Separating Interface from Implementation 1-34
1.23 Programming in C++ 1-34
1.23.1 Variable Declarations 1-34
1.23.2 Global Scope 1-34
1.23.3 Reference Variables 1-35
1.23.4 Operators in C++ 1-36
1.23.5 Scope Resolution Operator 1-37
1.23.6 'cin' and 'cout' Operator 1-38
1.23.7 Formatting and I/O Manipulators 1-39
1.24 Functions 1-39
1.24.1 Function Prototypes 1-40
1.24.2 Default Parameters 1-41
1.24.3 Default and Constant Arguments 1-42

Techknowledge Publications

1.24.4 Accessing Function and Utility Function 1-43

1.25 Constructors 1-43

1.26 Types of Constructor 1-44

1.26.1 Parameterized Constructor 1-44

1.26.2 Multiple Constructors within Class 1-44

1.26.3 Constructor with Default Parameters 1-45

1.26.4 Copy Constructor 1-45

1.26.5 Default Copy Constructor 1-46

1.27 Destructors 1-46

1.28 Objects and Memory Requirements 1-47

1.29 Static Members (Variable and Functions) 1-48

1.30 Static Data Members 1-49

1.31 Inline Functions 1-50

1.31.1 Call by Value and Call by Reference 1-51

1.32 Friend Functions 1-52

1.32.1 Advantages and Disadvantages of Friend Functions 1-54

Unit - II

Chapter 2 : Inheritance 2-1 to 2-21

2.1 Inheritance 2-1

2.1.1 Containership 2-2

2.1.2 Base Class and Derived Class 2-3

2.1.3 Defining Inheritance 2-3

2.2 Protected Members 2-4

2.3 Relationship between Base Class and Derived Class 2-5

2.4 Constructor and Destructor in Derived Class 2-5

2.5 Overriding Member Functions 2-7

2.6 Aggregation : Classes within Classes 2-8

2.7 Class Hierarchies 2-9

2.8 Public and Private Inheritance 2-9

2.8.1 Public Inheritance 2-9

2.8.2 Private Inheritance 2-10

2.8.3 Member Access Control 2-11

2.9 Types of Inheritance 2-11

2.9.1 Single Inheritance 2-12

2.9.2 Multiple Inheritance 2-13

2.9.3 Hierarchical Inheritance 2-15

2.9.4 Hybrid Inheritance 2-15

2.10 Ambiguity in Multiple Inheritance 2-16

2.11 Virtual Base Class 2-17

2.12 Abstract Class 2-18

2.13 Friend Classes 2-19

2.14 Static Class 2-20

2.15 Nested Class 2-20

Chapter 3 : Pointers 3-1 to 3-14

3.1 Pointer : Indirection Operators 3-1

3.2 Memory Management : New and Delete 3-2

3.3 Pointers to Objects 3-3

3.4 this Pointer 3-4

3.5 Pointers vs Arrays 3-5

3.6 Accessing Arrays using Pointers 3-6

3.6.1 Pointer and String 3-7

3.7 Arrays of Pointers 3-7

3.8 Function Pointers 3-9

3.9 Pointers to Pointer 3-10

3.10 Pointers to Derived Classes 3-10

3.11 Passing Pointers to Functions 3-11

3.12 Return Pointers from Functions 3-11

3.13 Null Pointer 3-12

3.14 Void Pointer 3-13

- **Model Question Paper (In Sem.) Q-1**

Techknowledge Publication

Unit - III

Chapter 4 : Polymorphism 4-1 to 4-19

4.1 Introduction to Polymorphism 4-1
4.2 Types of Polymorphism 4-2
4.2.1 Function Overloading 4-2
4.2.2 Operator Overloading 4-3
4.2.3 Function Overriding 4-4
4.2.4 Difference between Compile Time and Run Time Polymorphism 4-4
4.3 Concept of Overloading 4-5
4.4 Overloading Unary Operators 4-5
4.5 Overloading Binary Operators 4-6
4.6 Data Conversion 4-7
4.6.1 Conversion between Basic and User Defined Data Types 4-7
4.6.2 Conversion between User Defined Data Types 4-8
4.7 Type Casting (Implicit and Explicit) 4-10
4.7.1 Implicit Type Casting 4-10
4.7.2 Explicit Type Casting 4-11
4.8 Pitfalls of Operator Overloading and Conversion 4-11
4.9 Keywords : Explicit and Mutable 4-11
4.10 Pointer to Base Class 4-14
4.11 Virtual Function and its Significance in C++ 4-15
4.12 Pure Virtual Function and Virtual Table 4-16
4.12.1 Pure Virtual Functions 4-16
4.12.2 Virtual Table 4-16
4.13 Virtual Destructor 4-18
4.14 Abstract Base Class 4-18

Unit - IV

Chapter 5 : Files and Streams 5-1 to 5-20

5.1 Data Hierarchy 5-1
5.2 Streams and Files 5-1
5.3 Stream Classes 5-2
5.4 Stream Errors 5-4
5.5 Files 5-6
5.5.1 File Handling in C++ 5-6
5.5.2 C++ Classes for File Handling 5-7
5.6 Disk File I/O with Streams 5-7
5.6.1 File Opening Mode 5-10
5.6.2 Reading and Writing Class Object from File 5-10
5.6.3 Reading and Writing Multiple Class Objects 5-11
5.7 File Pointers 5-12
5.8 Error Handling in File I/O 5-14
5.9 File I/O with Member Functions 5-15
5.10 Overloading the Extraction and Insertion Operators ..5-16
5.11 Memory as a Stream Object 5-17
5.12 Command-Line Arguments 5-18
5.13 Printer Output 5-19

Unit - V

Chapter 6 : Exception Handling 6-1 to 6-13

6.1 Fundamentals of Exception Handling 6-1
6.2 Other Error Handling Techniques 6-2
6.3 Simple Exception Handling - Divide by Zero 6-3
6.4 Multiple Catching 6-5
6.5 Re-throwing an Exception 6-5
6.6 Exception Specifications 6-7
6.7 User Defined Exception 6-9
6.8 Processing Unexpected Exceptions 6-10

Technknowledge Publications

CHAPTER 1

Unit - I

Fundamentals of Object Oriented Programming

Syllabus

Introduction to Object Oriented Programming, Need of object-oriented programming, Fundamentals of object-oriented programming : Namespaces, Objects, Classes, Data members, Methods, Messages, Data encapsulation, Data abstraction and information hiding, Inheritance, Polymorphism. Benefits of OOP, C++ as object oriented programming language.

C++ Programming : C++ programming basics, Data Types, Structures, Enumerations, Control structures, Arrays and Strings, Class, Object, Class and data abstraction, Access specifiers, Separating interface from implementation. **Functions :** Function, Function prototype, Accessing function and utility function, Constructors and destructor, Types of constructor, Objects and Memory requirements, Static members: variable and functions, Inline function, Friend function.

Technknowledge Publication

1.1 Introduction to Computer System

- Before start with very interesting subject of object oriented programming, we start with basic building blocks of computers. This will explain what are the different part of the computer system, what are the function of each and every components, how the components are organized with each other, what are the examples of the different components.
- This point is just for knowledge purpose and not for examination.
- Computer system is available in different format like desktop system, laptop system, server system, handheld computer (Palmtop), etc. All the computer system have similar building block.
- Fig. 1.1.1 depicts the organization of computer system which is applicable for any computer system. The organization is very general and not bound to any computer system.

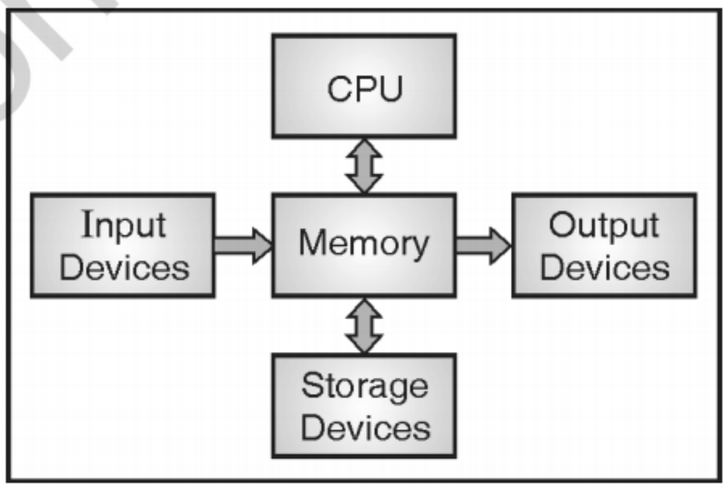

Fig. 1.1.1 : Block diagram of computer system

- Basically every computer system contains the processing units, memory devices, storage system, input/output devices, etc. All the components of the computer system are connected using the wires. Bunch of wires can be considered as bus.

The system can be divided into different parts like input unit, output unit, central processing units, and peripheral devices.

- **Input unit :** Input unit provides the information to computer. Unlike human being who gets the input through ear, nose, and eyes or by sensing. Number of input units are available in computer system like Keyboard, mouse, scanner, joystick, touch screen monitor, etc.

- **Output unit :** It gives the result of computations performed by computer. Famous examples of output devices are monitor, printer, sound system, etc. Unlike human being who gives the output by speaking, writing or by gestures.
- **CPU :** It refers to central processing unit, which is the heart of computer system. It processes the instructions. Major work in computer system is done by CPU only. Unlike human being, brain processes all the instructions. Along with Processor it also consists of different parts; RAM, SMPS, motherboard, storage devices, system buses, etc.
- **Processor :** It processes the instructions. Different processors from different manufacturers are available in market right from Pentium-I to I-7. E.g. Pentium-I,II,III,IV, Intel dual core, Intel core 2 duo, Intel centrino, Intel i-3, intel i-5, intel i-7, AMD Athlon processor, Celeron processors, etc. The evolution of processor started from microprocessor 8086 or even before that. Intel is leading organization manufacturing the processors but other organizations are also there in the market.
- **SMPS :** It is the Switched Mode Power Supply. An SMPS is usually employed to efficiently provide a regulated output voltage, typically at a level different from the input voltage. It deals with wide range of power supply and voltages.
- **RAM :** It is the Random Access Memory which is volatile one. User can read and write the data in this kind of memory. Specified address can be located in fixed amount of time that's why it is called as random access memory. It is the primary memory from where the programs are executed as processor can communicate only with primary memory. It is volatile kind memory which means data can be accessed as long as power is on.. It requires continuous power supply to retain the contents. Different types of RAM are available. Basically RAM can be of two type; Static and dynamic RAM. Today DDR, DDR2, DDR3, etc are famous in market.
- **Storage devices :** These are non-volatile type of memory devices where the data can be stored permanently. Data can be retained even when power supply will goes off. e.g. Hard disk, CD-ROM, DVD-ROM, electronic storage like USB, floppy disk, etc.
- **System bus :** Bus is nothing but the collection of wires or pins. The different hardware units via system bus are connected to each other. Buses can be data bus, address bus, and control bus. Data bus carries information. Address bus carries address which signifies where data should be delivered whereas control bus controls the operation of data and address bus. Front side bus carries information between CPU and memory controllers. Back side bus generally connects CPU and Cache memory.

1.2 Storage Structure

- This point is just for knowledge purpose and not for examination.
- Storage system or memory systems deals with how data is stored in computer. Memory system can be classified into primary and secondary memory.
- Primary memory is also referred as main memory. Programs are executed by processor which should be stored in main memory. Some of the examples of main memories are RAM (Random access memory), cache memory, registers, etc. Cache memory is used to store the frequently used data between processor and main memory.
- Registers are another kind of memory which is part of the processors. Before processing the data by Processor, it is stored in the processor registers. Register can be accumulator, data register, counter register, base registers, etc.
- Primary memories are volatile in nature which means data is lost when power supply is off. Primary memory is relatively small in size and is very costly. Speed of primary memory is highest than that of secondary memory.

Techknowledge Publications

- Secondary memory is used to store the data permanently. It is also referred as backing store. In secondary storage devices, data is stored permanently. We are able to retrieve the contents of this type of devices even after power failure. This is the reason why this type of memory is referred as non-volatile memory. They are slow in speed of operation. They are comparatively less costly than primary memory. File system is used to organize the data in secondary storage devices. Some of the examples of secondary storage devices are Hard Disk (HDD), CD-ROM (Compact Disk Read Only Memory), DVD-ROM (Digital Video Disk Read Only Memory), pen drive (USB drive), Floppy drive, etc.
- As shown in the Fig. 1.2.1, the processors are very fast and expensive memory system. They are very small in size and have small capacity. They work only when the power is on.

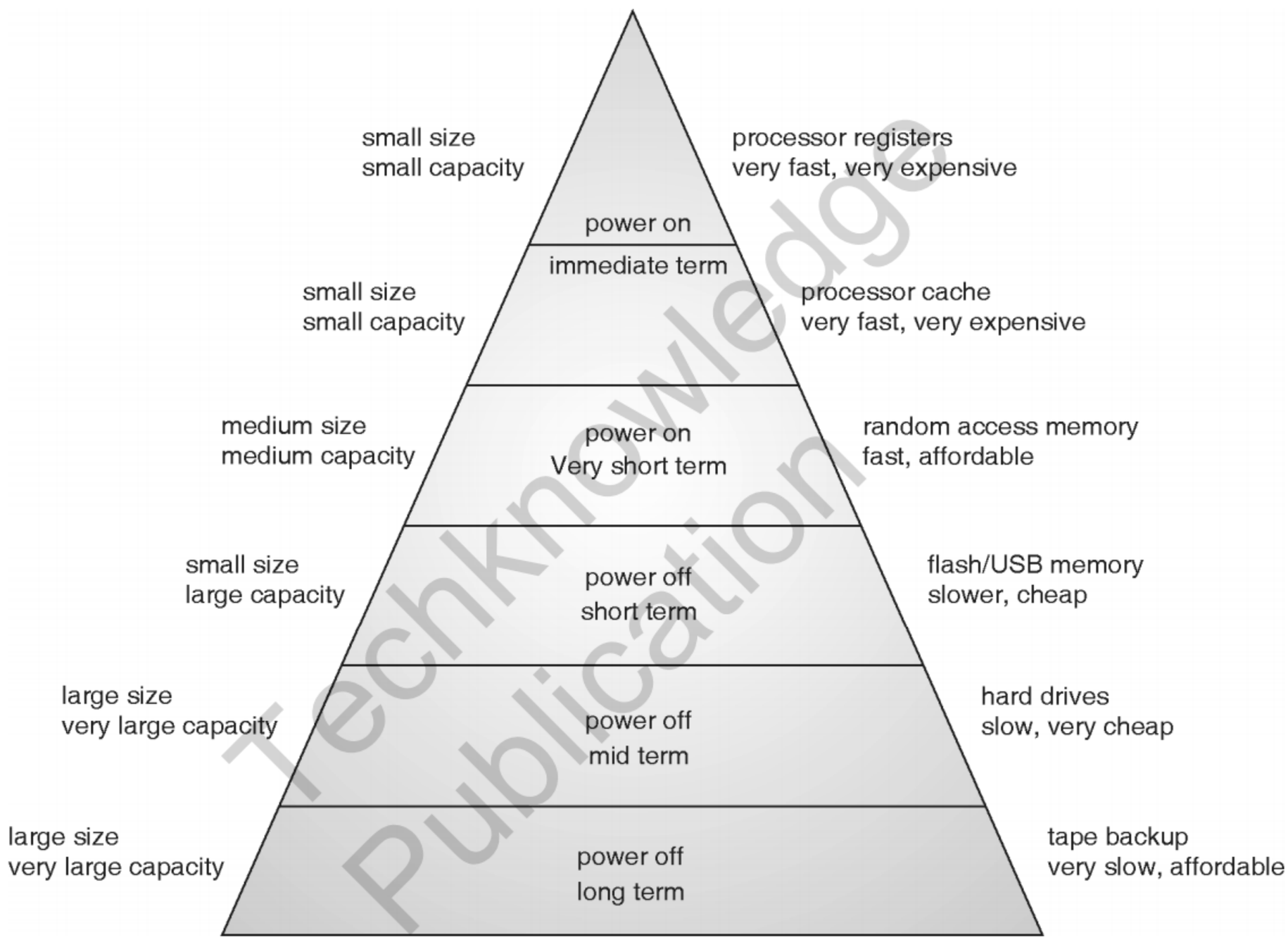

Fig. 1.2.1 : Memory hierarchy of computer system

- Next type of memory in the hierarchy is cache memory which is less expensive and have higher capacity than registers. They have high speed of operation.
- Next is Random Access Memory (RAM). They have moderate speed of operation. Cost is less than cache memory but more than secondary storage devices. That's why the size of RAM is less than size of secondary storage devices. E.g. typical computer system has 2-8 GB of RAM whereas the Hard disk size varies between 320 GB up to 1TB.
- Secondary storage devices are less costly and have higher size. Typically used for storing the data permanently.
- Tape back are kind of memory system which was used very earlier. They used magnetic tape to store the data. They are commonly comes in the format of cartridges and cassettes. E.g. Audio cassettes, video cassettes.

Memory Units

Measuring size of memory system is called as memory units. Memory system consists of number of semiconductors memory cells. One cell is able to store 1 bit of data. Data in memory system is in binary format means in terms of 0's and 1's. That's why size of any memory system is in the power of 2. E.g. suppose number of memory cells are 8 then size of memory system would be 2^8 that is 256. It means maximum decimal number stored would be 256 and not beyond it. Conversely to store decimal number 65536 we need 16 bit that is 2 bytes of memory. Table 1.2.1 shows the different memory units used.

Table 1.2.1 : Memory units

1 bit	0 or 1 (means stores one number)
1 byte	8 bits
1024 bytes	1 KB
1024 KB	1 MB
1024 MB	1 GB
1024 GB	1 TB
1024 TB	1 Petabyte
1024 Petabyte	1 Exabyte
1024 Exabytes	1 Zettabyte
1024 Zettabyte	1 Yottabyte.

1.3 Operating Systems

- This point is just for knowledge purpose and not for examination point of view.
- Operating system is software running on computer hardware. It contains programs, applications and data which runs on computer hardware. It acts as manager of computer resources. It provides the environment wherein user can perform variety of activity with computer.
- For example, word application is run by operating system which provides the environment to create the word document.
- Another example would be user can install the game application and execute it with the help of operating system.
- Operating system is nothing but the interface between user and computer hardware. Application software for its smooth functioning always calls the operating system. Operating system is part of all computer devices right from mobiles to handheld devices and even supercomputer or all the servers.
- All web servers are using the operating system though they are bit different and have more functionalities. It is similar to the government who is not doing anything but whose presence is important. It is similar to Principal of college of whose presence is important for smooth functioning of the system.

1.3.1 Examples of OS

Following are some of the examples of operating systems. These are commonly used operating systems :

- **Windows :** Operating system is invented by Microsoft organization. Very first operating system by Microsoft was DOS. Over the years number of other operating systems was developed by Microsoft. Some of the examples of Windows Operating Systems are Windows-1.0, Windows-95, Windows-98, Windows-2000 NT, Windows-XP, Windows-2003 server, Windows-Vista, Windows-7, Windows-8, etc.
- **UNIX :** It was invented in late 1969 by Ken Thompson, Douglas McIlroy, Michael Lesk, Brian Kernighan, Joe Ossanna and Dennis Ritchie. Originally developed in assembly language but later recoded in C language. Unix is not Open source operating system. Only large organizations purchase the license of Unix. All Unix system are commercial. Unix can be considered as complete system as everything is bundled in it. Popularly used Unix systems are as follows :
  - HP-UX
  - IBM AIX
  - Sun Solaris
  - Mac OS X
  - IRIX

Techknowledge Publications

- **Linux :** Linux is Unix clone written by Linus Torvalds from start. Linux is just kernel whereas the Linux distribution includes GUI utilities, GUI, installation tools, language compilers, application software, etc. Linux is just a kernel but with the help of distribution the system will get completed. Famous Linux distributions are as follows :
  - Fedora Linux
  - Red Hat Enterprise Linux
  - Suse Enterprise Linux
  - Debian Linux
  - Ubuntu Linux
- **Google Chrome OS :** Chrome is an operating system based on the Linux kernel and designed by Google.
- **Mobile OS :** It includes Nokia's Symbian OS, Google's android OS, Apple's iOS, Blackberry OS, etc.

1.4 Programming Language

- We are using the computer system as general purpose computing machine.
- Computer performs the activities which are directed by the human being.
- Computer can understand the language in terms of 0 and 1 that is in binary format.
- Human being cannot deal with binary language easily.
- Human being can direct the computer system using different instructions.
- Instruction is nothing but the order or command to computer system.
- Set of instructions forms a program. It means program is collection of instructions.
- The way of writing and organizing the instructions is taken care by programming languages.
- Programming language specify the rules and regulations for writing the program.
- Programming language specifies how to start the program, how to end the program, how to start the instructions, how to terminate the instruction, etc.
- All the rules and regulations needed to create the program are given by the programming languages.
- Before programming we need to know the programming languages specifications then only we can create the program.
- Varieties of programming languages are available.
- Some of the popular programming languages are C, C++, Java, .net, HTML, etc.
- Software developers can choose the programming language as per their requirement.
- Due to rapid changes in the technology, we need to choose the different approaches and different programming languages for software development.

1.4.1 Challenges in Software Development

- Day by day software requirement is rapidly getting changed.
- Technology also continuously evolving because of which advance and new hardware are getting introduced.
- There are many challenges for software developers in this continuously evolving environment.
- One of the challenge is how to represent the actual situation in the software. End user requirements are very different than we think.
- Another challenge is how to create user interface for intended users. Software developers have to always think by the mindset of end users. He always thinks how user will feel this software.
- How to divide the software into small entities is also big challenge. Reason for dividing the software reusability.
- Designing the software in such way that they will need less maintenance in the future.
- Developing the software which incurs minimum cost is very important in successful software development.
- Improving the productivity of the developed software is very important.

Techknowledge Publications

- The very important factor in software development is quality of the software.
- Developing the software which will fulfil the deadlines of end users.

1.4.2 Software Evolution

- The software is continuously evolving.
- There are many reasons for evolution.
- One of the major reason is changes in the hardware.
- Another major reason is rapidly changing user requirement.
- To cope with these challenges we need to evolve the software continuously.
- Some of the quality issues we need to keep in mind while evolving the software. These quality issues are listed below :
  - Correctness of the developed software.
  - Software should consider all the security concerns.
  - User friendliness is the critical factor in software evolution.
  - New changes in the software should easily integrate in the existing software.
  - Components of the software should be reusable.
  - Software should have openness and interoperability.
  - The developed software should be easily portable across the different operating system.
  - Software should be easily maintainable.

1.4.3 Types of Programming Language

- There are varieties of programming languages. Some of the category of programming languages is as follows :

1. Machine level language
2. Assembly level language
3. High level language
4. Object oriented language

- The hierarchy is shown in Fig. 1.4.1.

C, C++, Java, .net, HTML, etc.
High level language
Assembly language
Machine language
Hardware

Fig. 1.4.1 : Level of programming language

1. Machine Level Language

- It is low level language.
- Program is written in terms of 0 and 1.
- It is not user friendly programming language.
- Complicated program cannot be written using binary language.
- It is not preferred language for general purpose programming.

2. Assembly Level Language

- It is next level of programming language.
- It is written using assembly instructions.
- Assembler is used to convert the assembly program into machine readable format.
- Some of the example of assembly instructions is MOV, SUB, ADD, DIV, MUL, etc.
- It is mostly used for system level programming.
- Example of assembly program is as follows :

```
section .data
msg db 'This is Hello world in assembly language!'
len equ $ - msg
section .text
global _start
_start:
mov eax,4
mov ebx,1
mov ecx,msg
mov edx,len
int 0x80
```

Tech Knowledge Publications

```
mov eax,1
mov ebx,0
int 0x80
```

3. High Level Language

- This language is more like English language.
- It is written using high level instructions.
- Program created in high level language is converted into machine readable language using compilers and interpreters.
- Some of the famous high level language includes C, C++, Java, .net, HTML, LISP, etc.
- Steps in creating typical C program are as follows.
  1. Create the program.
  2. Save the program with .c extension.
  3. Compile the program which creates the .obj file.
  4. Link and execute the program. It creates the .exe file.
- Sample hello word program in C is given below :

```
#include<stdio.h>
#include<conio.h>
void main()
{
        clrscr();
        printf("\n Hello word");
        getch();
}
```

- Sample hello word program in C++ is given below :

```
#include<iostream>
using namespace std;

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

- Sample hello word program in Java is given below:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World");
  }
}
```

1.4.4 Most Popular Programming Language

- Some of the most demanding language in 2016 is as follows. This is just suggestive list. It may vary time to time.
  - SQL
  - Java
  - JavaScript
  - C#
  - C++
  - Python
  - PHP
  - Ruby
  - iOS

1.4.5 Qualities of Good Programmer

- Some of the good qualities of successful software programmer are as follows. These are just suggestive qualities.
  - Programmer should have deep understanding of at-least one programming language.
  - Programmer should be comfortable with algorithmic understanding.
  - He should be comfortable with mathematical thinking.
  - Good in structuring the program code.
  - He should have positive attitude.
  - Good communication skills.
  - Great in time and work management.
  - Should have high end user focus.
  - Impressive technical skills.
  - Always ready to learn
  - Good debugging skills

Technknowledge Publications

- Programmer should have good problem solving skills.
- Work environment should be matching.
- He should have passion of work.
- He should have good ability to plan the work.
- He should have ability to handle the failure.
- He should be good team member.
- He should respect the software deadlines.
- Should have willingness to research.

1.4.6 Terminologies in Programming

Some of the frequently used terminologies in programming are as follows. This may not be part of examination but useful to understand the subject.

– **Program :** It is a sequence of instructions written to perform the specified task by computer / processor. Program is nothing but sequence of order to processor of computer. Program can be developed in any programming language. Program can be developed in C, C++, Java, etc.

– **Programming language :** It is nothing but the language to communicate with computer. It defines rules and regulations to write the program. It is more like communication language like Marathi, Hindi, English. Popular programming languages are C, C++, Java, .net, PHP, Perl, Python, HTML, etc.

– **Software :** It is collection of program for some purpose. E.g. Word application, paint, power point, game application, device drivers, etc.

– **Instructions :** It is command or order to processor.

– **Error :** It is nothing but mistake in the program which causes program to work incorrectly. Missing semicolon at the end of statement or instruction is error in the program. Error can be logical or syntactical.

– **Syntactical error :** This type of error occurs due incorrect syntax the in the program. Famous examples are missing semicolon, missing closing brackets, etc. Following example illustrate syntactical error.

Program for addition of two numbers

```
1. #include<stdio.h>
2. void main()
3. {
4. int a=10, b=20, c;
5. c=a-b
6. printf("%d", c);
7. }
```

Explanation : *Program will not compile as we are missing semicolon on statement number 5.*

– **Logical error :** This type of errors occurs due to incorrect programming. Program is syntactically correct but still it does not produce desired output. It generally occurs due to misinterpretation of the problem statement. Consider the example where we are writing program to do the addition of two numbers. But by mistake we mentioned – (minus) sign instead of + (plus) sign in addition statement. This program will be syntactically correct but does not produce desired outcome due to logical error.

Program for addition of two numbers :

```
1. #include<stdio.h>
2. void main()
3. {
4. int a=10, b=20, c;
5. c=a-b;
6. printf("%d", c);
7. }
```

Explanation : *Program will produce the output -10 but desired output was 30. Logical error present on line number 5. This type of error can be considered logical errors.*

– **Run time error :** These error occurs during program execution. Generally program is syntactically correct but does not produce desired outcome. Array index out of bound, divide by zero are example of run time error. It can be due to software or hardware problem.

Tech Knowledge Publications

- **Editor :** It is used to type the program. Many editors are available to type the program. Popular editors are notepad, WordPad, eclipse, NetBeans, vi editor, visual C++, etc.
- **Compilation :** Compilation is frequently used term in programming. When programmer write any program it should be get checked whether it is as per the language specification or not. The task of checking the correctness of the program is done by the compiler. Complier is system program provided by programming language. It just checks the syntax error and does not do anything else.
- **Linker :** It is the system program which links different object module to generate executable program.
- **Loader :** It loads the program in memory of computer to get it execute. It is considered system program.
- **Program execution :** It means executing the application/program. When we start executing any program, it is first loaded into memory. After loading the program into memory, processor executes the instructions one by one.
- **Debugger :** Step by step program execution is possible with debugger. It is helpful in identifying the errors in the program. It does not correct the program instead just assist to identify the error in the program.
- **IDE :** It is nothing but the integrated development environment. It consists of everything to develop the program like editor, compiler, debugger, linker, loader, etc. Example include Eclipse, Turbo C, NetBeans, etc.
- **Program output :** It is nothing but end result of correct program execution.
- **Process :** Process in operating system is executing instance of the program. Along with the program code, it also contains the additional data related to it. Most of the operating system uses the concept of multithreading which is part of the process only. Threading is useful to improve the performance of the system.
- **Resource :** Resource is any component in the computer system. The resource can be physical or logical. Physical resources include processor, memory, printer, etc. Logical resources includes files or CPU time. All the hardware devices connected to computer can be considered as resources. Virtual system resources include files, network connections and memory areas. E.g. printer, scanner, CPU time, memory, storage devices, files, monitors, keyboard, mouse, etc.
- **Job :** Job and process refers to same thing.
- **CPU :** It is central processing unit. The term processor or CPU is used interchangeably.
- **Source code :** It is nothing but the program code which contains the instructions.
- **Kernel :** It is core part of the operating system. It performs very basic and low level operations. It mostly deals with computer hardware. It is also referred as mini operating system.

1.5 Introduction to Procedural, Modular Programming

Q. When would you choose to use C rather than C++ ? **SPPU - Dec. 17, 4 Marks**

- This type of program consists of collections of procedures or modules.
- Module or procedure is nothing but the part of whole program which is performing some particular task.
- Generally, program is divided into different module or procedure in this kind of programming languages.
- Some example of procedure oriented programming languages is C, COBOL, FORTRAN, assembly programming, etc.

Techknowledge Publication

- In this type of programming, entire software is considered as collection of activities executed in particular order.
- Generally, the execution sequence of the program is not changed.
- Large program is divided into small subroutines. Subroutines are nothing but the module or procedures. It means subroutine, module, procedure, functions, etc. is one as the same thing.
- It is mainly based on the functions and less focus is on data.
- It just gives the list of instructions to be executed by the computer system.
- Security issues are not taken into considerations.
- Main focus is on completing the work.
- Many times we are using the global variables which are less secure in programming.
- Actual problems in the worlds are not considered and represented properly.
- They are mainly used for fundamental programming.
- They are also used in system programming.
- Structure of procedure oriented program is shown in Fig. 1.5.1.

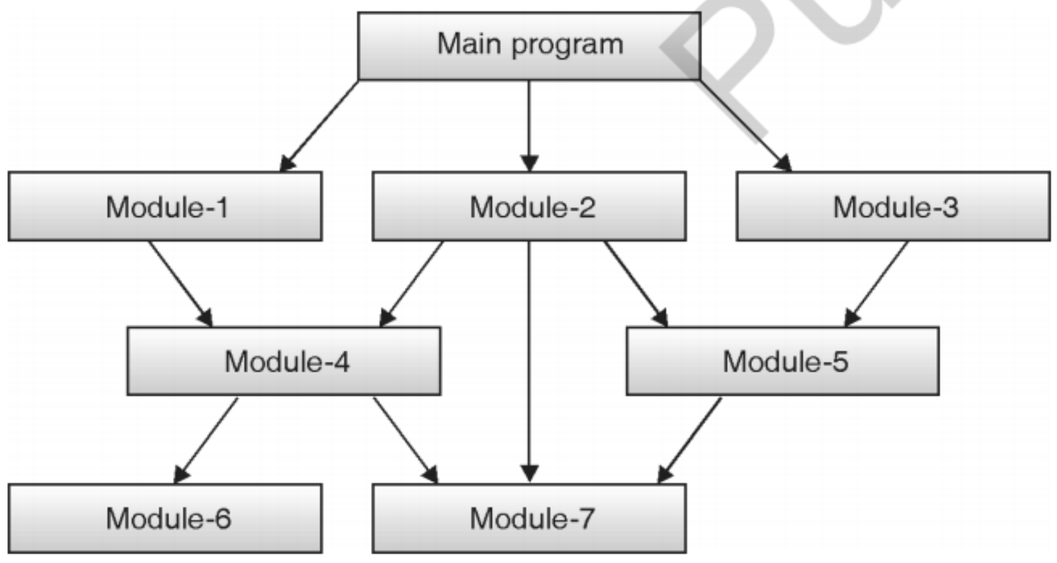

Fig. 1.5.1 : Structure of procedure oriented program

1.5.1 Characteristics of Procedure Oriented Programming

- Main focus is on functions.
- Less focus on data.
- Uses top down approach.
- Large program is divided into small subroutines.
- We can transfer data from one function to another function very easily. Data can be transferred with the help of variables.

➢ **Program 1.5.1 : Example of procedure oriented program**

```
#include<stdio.h>
#include<conio.h>
void add(int, int);
void sub(int, int);
void mul(int, int);
void div(int, int);

void main()
{
   int a,b;
   printf("\n Different arithmetic operations");
   a=20;
   b=10;
   add(a, b);
   sub(a, b);
   mul(a, b);
   div(a, b);
   getch();
}
void add(int a, int b)
{
   int c=a+b;
   printf("\n Addition of two number is:%d", c);
}
void sub(int a, int b)
{
   int c=a-b;
   printf("\n Subtraction of two number is:%d", c);
}
void mul(int a, int b)
{
   int c=a*b;
   printf("\n Multiplication of two number is:%d", c);
}
void div(int a, int b)
```

Technknowledge Publication

```
{
   int c=a/b;
   printf("\n Division of two number is:%d", c);
}
```

Output

```
Different arithmetic operations
Addition of two number is:30
Subtraction of two number is:10
Multiplication of two number is:200
Division of two number is:2
```

1.5.2 Limitations of Procedural Programming

- Security is main problem in procedure oriented programming.
- Scalability is another problem which means we cannot easily scale the software.
- Data hiding is limitation of procedure oriented programming.
- We cannot reuse the software code from procedure programming.
- Less maintainability.
- Difficulty in creating the new data types.
- Not suitable for rapidly changing software requirement.

1.6 Introduction to Object - Oriented Programming

- Object oriented programming is new approach to modern programming.
- It is uses totally different approach to develop the program than that of procedure oriented programming.
- It is invented to overcome the drawbacks of procedure oriented programming.
- Most of the modern software are developed using the object oriented concept.
- It is most widely used and popular programming concepts.
- Object oriented programming are concepts and not the language.
- Using the object oriented concept many programming languages are invented like C++, Java, .net, PHP, etc.
- It uses the different object oriented concepts like object, classes, polymorphism, encapsulation, inheritance, message passing, data binding, etc.
- Main features of object oriented programming :
  1. Main focus of this kind of programming is on data rather than functions.
  2. The whole program is divided into different objects.
  3. Functions and data of object oriented program tied together.
  4. Data used in this type of programming is hidden and cannot be accessed easily.
  5. Object in the program can communicate with the help of functions.
  6. We can add or delete the data and procedures very easily as and when required.
  7. It uses bottom up approach in programming.
  8. Reusability is possible with object oriented programming.
  9. Can easily maintain the software.
- The structure of object programming is shown in Fig. 1.6.1.

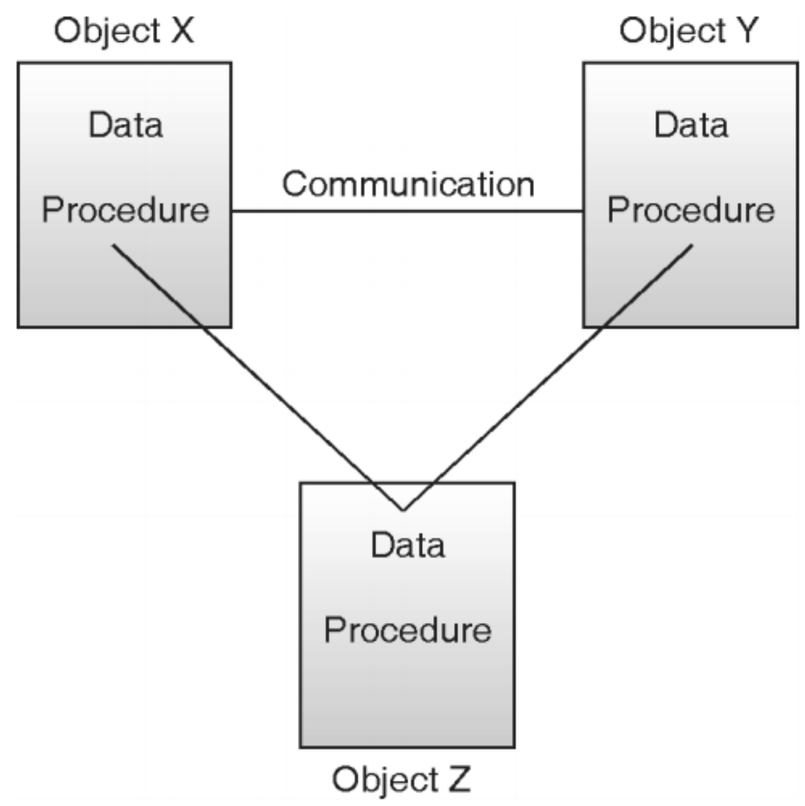

Fig. 1.6.1 : Structure of object oriented programming

Tech Knowledge Publications

1.6.1 Need of Object-Oriented Programming (OOP)

- There are many reasons for using the object oriented programming.
- It provides many benefits for software development.
- Object oriented programming provides object, class, inheritance, encapsulation, data hiding, abstraction, etc.
- It is very easy to develop the software using the above object oriented concept.
- With the help of inheritance concept, we are able to remove the repeated code in the program.
- With the help of classes and objects we can easily divide the code into different components.
- We can develop the code which hides the data from other users or from other program.
- We can easily scale the software for more requirements.
- Software code reusability is possible with object oriented programming.
- Complexity in the program development can be easily managed in object oriented programming.
- Communication between different modules is possible with message passing concept.
- They develop the software which focuses mainly on data which results in more secure software.
- Very easy to map the real world problem into programming.

1.6.2 Applications of OOP

Following are some of the famous application of objects oriented programming:

- They are used to design the real time system.
- They are used in the simulation and modelling.
- They are used to develop object oriented database.
- They are used in artificial intelligence and expert system.
- They are used in the concept of neural networks as well as in the parallel and multithreaded programming.
- Office automation software are developed which are based on object oriented programming.
- The concepts are used in the mechanical software like computer aided design.
- Basically most of the modern software is based on object oriented concept.
- They are used in image processing and pattern recognition software.
- They are useful in computer aided teaching.
- They are used in database management systems.
- They are used in distributed computing, parallel and mobile computing.

1.6.3 Difference between Procedure Oriented and Object Oriented Programming

Q. Compare Procedure Oriented Programming Vs Object Oriented Programming.

SPPU - Dec. 16, Dec. 19, 4 Marks

Table 1.6.1 : Difference between procedure oriented and object oriented programming

Sr. No.	Procedure oriented programming	Object oriented programming
1.	Program is divided into small functions.	Program is divided into objects.
2.	It depends upon functions.	It depends upon object.
3.	More focus on functions.	More focus on data.
4.	Less secure.	More secure.
5.	Very less reusable.	More reusable.
6.	Less maintainability.	More maintainable.
7.	It uses top down approach.	It uses bottom up approach.

Technknowledge Publications

Sr. No.	Procedure oriented programming	Object oriented programming
8.	It does not provides any access specifier.	It provides the different access specifier like public, private, protected.
9.	Directly we can access the data members.	Data movement is possible only with the help of functions.
10.	Does not provides classes, object, inheritance, encapsulation, etc.	Provides classes, object, inheritance, encapsulation, data hiding, message passing, abstraction, etc.
11.	Many times global variables are used which are used by all the functions in the program.	Data access is possible with the help of functions only.
12.	Data hiding is not possible.	Data hiding is possible.
13.	Function overloading is not possible.	Function overloading is possible.
14.	Famous examples are C, VB, FORTRAN, PASCAL.	Famous examples are Java, C++, .net, etc.

1.7 Object Oriented Languages and Object Based Languages

- Object oriented programming are just the concepts and not any programming language.
- Object oriented programming concepts can be implemented by any programming languages which support object orientation.
- Even though it can be implemented by any language, it may create confusion. It is better to use language which has special support for OOP concept.
- The underlying language should support most of the feature so that we can consider it is object oriented programming.
- Depending upon the feature available in the language they can be divided into :
  - Object based programming languages.
  - Object oriented programming languages.
- Object based programming generally have different features like object identity and encapsulation.
- Important features in object based programming languages are data encapsulation, data hiding and access mechanism, automatic initialization, clear up the objects after use, operator overloading.
- Programming language which supports programming with object can be said object based programming languages. Inheritance and dynamic binding cannot be supported.
- Typical example of object based programming language is Ada. Object oriented programming language supports all feature of OOP.
- Typical examples are C++, Java.

1.8 Fundamentals of Object-Oriented Programming

Q. List and explain various features of object oriented programming.

SPPU - May 17, 3 Marks, May 18, 4 Marks, Dec. 19, 6 Marks

- Object oriented programming is based on object.
- Along with concept of object they are providing many fundamental concepts which are as follows :

1. Namespaces

- A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it.
- Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

Technknowledge Publication

– All identifiers at namespace scope are visible to one another without qualification. Identifiers outside the namespace can access the members by using the fully qualified name for each identifier

2. Classes

– The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

– **For Example :** Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

– We can say that a **Class in C++** is a blue-print representing a group of objects which shares some common properties and behaviours.

3. Objects

An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

```
class person
{
   char name[20];
   int id;
public:
   void getdetails(){}
};
int main()
{
person p1;        // p1 is a object
}
```

4. Data members

– "Data Member" and "Member Functions" are the new names/terms for the members of a class, which are introduced in C++ programming language.

– The variables which are declared in any class by using any fundamental data types (like int, char, float etc.) or derived data type (like class, structure, pointer etc.) are known as Data Members, and the functions which are declared either in private section of public section are known as Member functions.

– There are two types of data members/member functions in C++:

  - Private members
  - Public members

5. Methods

– A method is a procedure or function in OOPs Concepts. Whereas, a function is a group of reusable code which can be used anywhere in the program. This helps the need for writing the same code again and again. It helps programmers in writing modular codes.

– A method also works the same as that of function.

– A method is defined inside a class. For example : main() in Java

– A method can be private, public, or protected.

– The method is invoked by its reference/object only. For example : If class has obj as an object name, then the method is called by: obj.method();

– A method is able to operate on data that is contained within the class

– Each object has it's own method which is present in the class.

6. Messages

– Objects communicate with one another by sending and receiving information to each other.

Techknowledge Publication

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

7. Data encapsulation

- In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit.
- In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

8. Data abstraction and information hiding

- Data abstraction is another important feature of object oriented programming.
- It refers to using very essential features without adding the background details of functions.
- It means only less number of details are added and implementation details are not given.
- It is way information hiding from the outside world.
- Data abstractions also referred as abstract data types (ADT).
- Data abstraction and data encapsulation are ways of information hiding.

9. Inheritance

- The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.
- **Sub Class :** The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class :** The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability :** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

10. Polymorphism

- The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person possess different behaviour in different situations. This is called polymorphism.
- An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

1.9 Benefits of Object - Oriented Programming

There are many advantages of object oriented programming. It has been benefited by programmer as well as user of the program.

- Quality of the software is increased due to use of object oriented programming concepts.
- Productivity of the software increases with OOP.
- Code reuse is most important benefit of OOP. It is achieved using inheritance.
- With code reuse we can use part of code in other program also which reduces development time.
- Program can be built from existing modules without writing them right from the beginning. It increases speed of software development.
- With the help of data hiding we can develop secure program.

Tech Knowledge Publications

- Program division is very easy with OOP.
- The program or software which is developed with OOP can be easily upgraded from small to large system.
- Many state of the object can exist without any interference with each other.
- OOP uses data centred which enables us to capture more details.
- Efficient message passing system enable us to communicate easily between objects.
- Complexity in software development can be easily managed.
- It hides many implementation details from the users.

1.10 C++ as Object Oriented Programming Language

- C++ is object oriented program language.
- Object oriented programming's are just concepts.
- C++ is actual language which implements object oriented programming features.
- It was invented by Bjarne Stroustrup at AT & T bell laboratories in USA in 1980.
- Limitation of C results into creation of C++.
- That is why it is called C++ (increment operator in C is ++ so we called it C++). It means C with increment, C++.
- It was also called as C with classes.
- It is superset of C.
- Most of the concepts are applicable to C++ from C.
- But are many additions in C++ than C.
- Many times same compiler is used to execute C and C++.
- Object oriented feature in C++ allows programmer to design and development large applications.
- Variety of applications can be developed using C++.

1.10.1 C++ Programming Basics

- As we rightly said, object oriented programming language is not a programming language.
- They are set of concept and which are implemented using different programming language.
- Different programming languages which uses object oriented concepts are C++, Java, .net, etc.
- All the concepts are explained by taking the example of C++.
- You can use any operating system or any development tools, most of the program code will be same.
- We have tested all programs on Windows and Linux also.

1.10.2 ++ Extension of C

- We are using the C language as starting point for high level languages.
- It is very fundamental language before going for programming.
- Unlike C, C++ is first object oriented programming language.
- Before going for any object oriented programming, it is very important to learn C++.
- Before going for any other object oriented programming like Java, .net.
- C++ implements all the features of object oriented programming.
- C++ supports objects, classes, inheritance, polymorphism, data hiding, etc.
- Along with object oriented feature it also supports the C features.
- All the C features are available in C++ also.
- This is the reason why we are referring C++ as extension of C.
- Sometimes C++ can be considered as C with object and classes.
- C uses the ++ operator for getting the next value of the variable. That's why next version of C is C++.

Tech Knowledge Publications

1.10.3 Structure of C++ Program

- Typical structure of C++ program is shown in Fig. 1.10.1.
- This structure is almost identical for all object oriented programming.

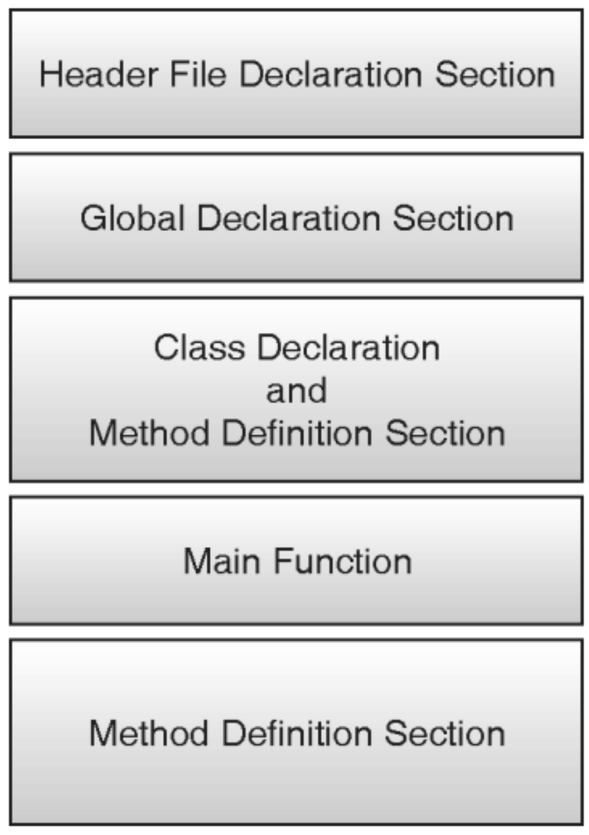

Fig. 1.10.1 : Structure of C++ program

- As shown in the Fig. 1.10.1, it consists of different sections.
- A first section is header file declaration. Here required header files are added.
- Next section is global declaration section. Here global variables are declared. Most of the time global variables are ignored in C++ program.
- Next section is class declaration. Here required classes in the program are declared. If needed functions are defined inside the class only.
- Next is a main function. It is nothing but the starting point of the program.
- If we have not defined the functions inside the classes then after the main functions all the functions are defined.
- Typical C++ looks like following snippet.

➢ **Program 1.10.1**

```
#include<iostream>
//more header file goes here if needed
int a;
//more global variables goes here if needed
class example {
   public:
      void accept();
      void display();
};
//more class declaration goes here if needed
void example:: accept()
{
   //definition
}
void example::display()
{
   //definition
}
//more function definition goes here if needed
int main()
{
   //body of main function
}
//more method definition can also goes here if needed
```

Hello word program with C++

Simple version

```
#include<iostream>
using namespace std;
int main()
{
cout<<"\n Hello World in C++!";
}
```

Line by line explanation

Line 1 : Used to add header file.

Line 2 : It used to define namespace.

Line 3 : Start of the functions.

Line 5 : Used to print the message. In this case we have used the put operator cout to print the message which is more like printf() in C.

Techknowledge Publication

Hello word program with object oriented concept in C++

```
1.  #include<stdio.h>
2.  #include<iostream>
3.  using namespace std;
4.  class Hello
5.  {
6.  public:
7.  Hello() {}
8.  ~Hello() {}
9.  void display()
10. {
11. cout<<"\n This is Hello World in C++";
12. }
13. }
14.
15. int main()
16. {
17. Hello myobject;
18. myobject.dispay();
19. return 0;
20. }
```

Line by Line explanation

Line 1 and 2 : Used to add header file.
Line 3 : Define namespace.
Line 4 : It is used to define new class.
Line 6 : Public specifies whatever it follows will be accessed by everyone.
Line 7 : Hello() is used to define constructor.
Line 8 : ~Hello() is used to define destructor.
Line 9 : It is user defined function named display().
Line 11 : cout is used to display the message on screen.
Line 15 : Start of program.
Line 17 : It is used to create object named myobject.
Line 18 : It is used to call display() function using myobject.

1.10.4 C++ Program Execution

- The process of executing the C++ depends upon which type of tools and operating system we are using.
- C++ program should be typed in any convenient editor.
- After typing we need to save the program with .cpp extension.
- Compile the program using C++ compiler. Well known C++ compilers are Turbo C++, Borland C++, GCC, Intel C++, visual C++, etc.
- After compiling we need to link the program using linker.
- Last step is program execution.
- If we are using Turbo C++ then direct options are given for compiling and executing the program.
- If we are using GCC then we need to compile and execute the program using following two commands. :

```
gcc   -o  myprogram_output myprogram.cpp
./myprogram_output
```

Steps in C++ program development

Step 1 : Type program.
Step 2 : Save the program with .cpp extension.
Step 3 : Compile the program. Correct errors if present.
Step 4 : Execute the program.

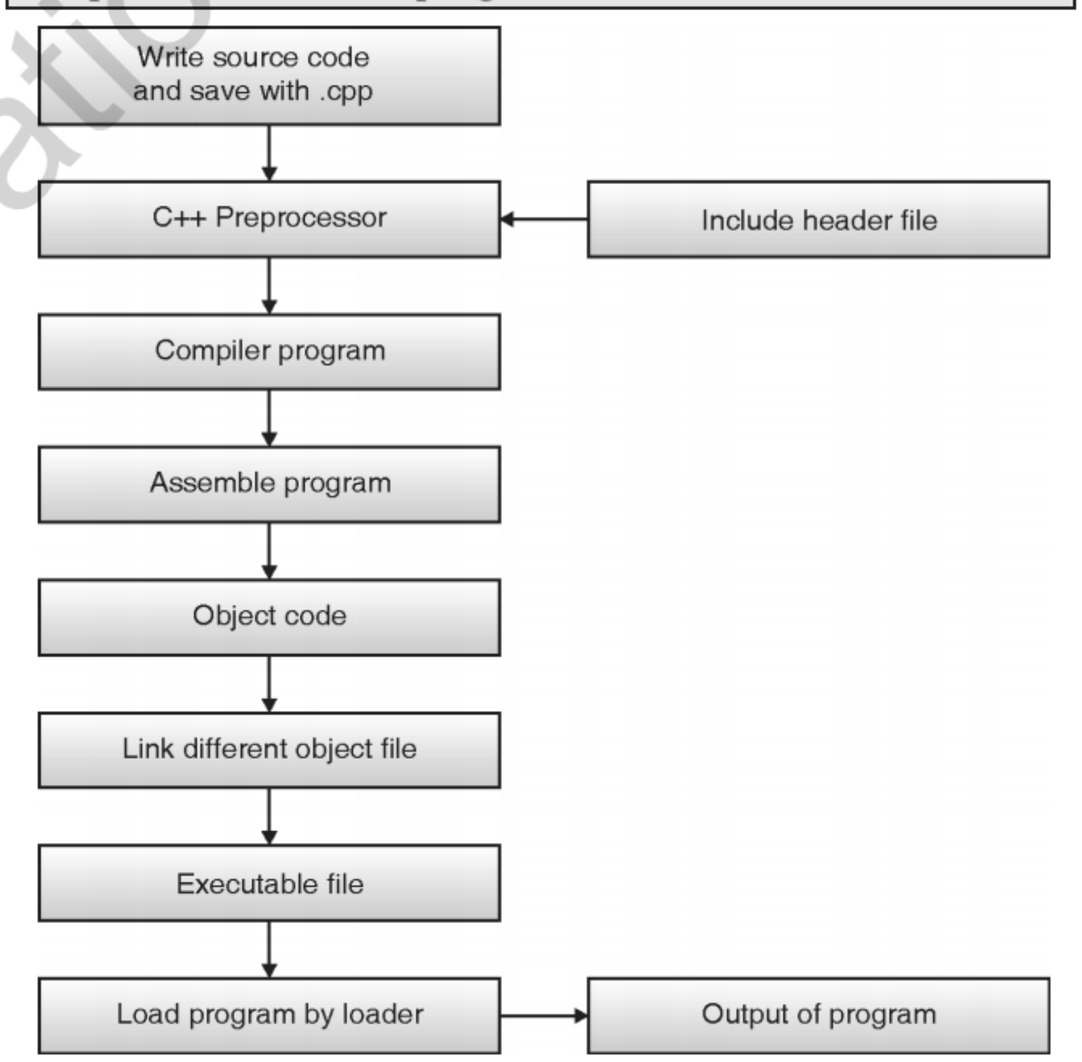

Fig. 1.10.2 : Steps in C++ program development

Consider the case where Example.cpp is being executed. Following are the steps in executing Example.cpp program.

Tech Knowledge Publications

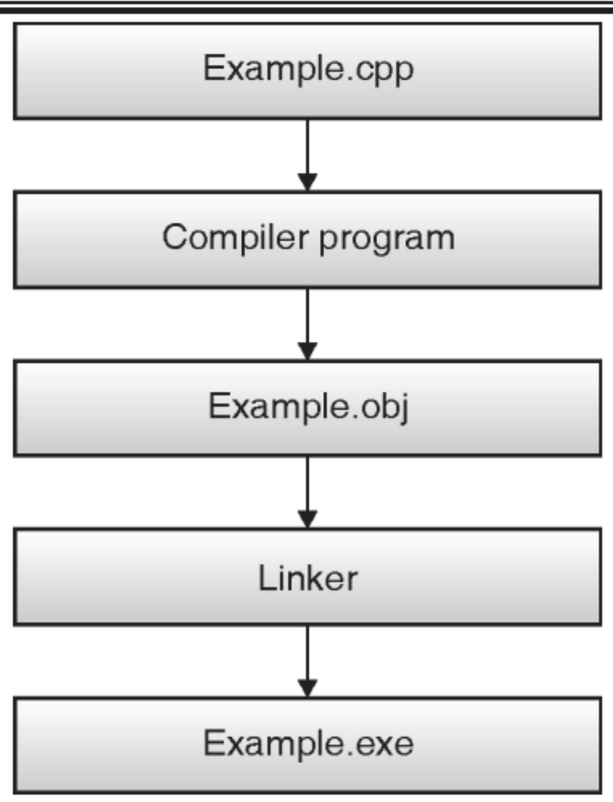

Fig. 1.10.3 : Executing Example.cpp program

1.10.5 Different Tools for C++ Program Development

Following are some of the popular C++ development tools. This is just suggestive list. There are many more tools available. Some tools can work only with Windows or Linux. Some tools works on both.

- Turbo C++
- Microsoft visual C++
- GCC
- Eclipse
- Netbeans
- DevCPP
- Anjuta
- Borland C++
- Embedded visual C++
- Free Intel compiler.
- VIDE
- XCode

Run C++ program online

It is possible to run C++ program online without installing any software. Just internet connection is enough to run the program.

Following are some of the suggestive websites where C++ program can be executed directly.

- https://www.codechef.com/ide
- https://ideone.com/
- http://cpp.sh/
- http://compileonline.com/
- http://codepad.org/
- http://www.onlinecompiler.net/
- http://rextester.com/
- https://www.remoteinterview.io/online-c-compiler

1.10.6 Main Function in C++

- Every C++ program should start with main function.
- main() is the starting point for every C++ program.
- The return data type of main() function should be int in C++.
- Last statement in every C++ program should be return 0.
- We can pass the argument to main function also.

Main function

```
#include<iostream.h>
using namespace std;
int main()
{
.
.
.
return 0;
}
```

Main function with passing parameter

```
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{
   cout<<"\n Number of arguments are :"<<argc;
   cout<<"\n Actual arguments are:";
   for(int i=0;i<argc;i++)
   {
        cout<<argv[i];
   }
   return 0;
}
```

Techknowledge Publication

Above program can be executed using following steps (Ubuntu).

Step 1 : Type the program.

Step 2 : Save the program with .cpp extension.

Step 3 : Compile the program using

```
gcc -o output myprogram.cpp
```

Step 4 : Run the program using following command

```
./output first_arg second_arg
```

We are passing two argument namely first_arg and second_arg

1.11 Building Blocks of C++

1.11.1 Comments

- Comments are the part of program which is not considered during the program execution.
- They are used for documentation purpose.
- At the time of programming, programmer can add comments to get the significance of certain part.
- Comments in C++ can be single line or multiline comments.
- Single line comments start with //, whereas multiline comments start with /* and end with */.
- Following is the snippet of single and multiple comments :

Single line comment

```
//this is c++ program
//this is part of the comment
```

Multiline comments

```
/* this is start of multiline comment
It can have multiple lines
This is end comment */
```

1.11.2 Tokens

- Smallest element in the program is known as tokens.
- C++ has following type of tokens :
  - Keywords
  - Identifiers
  - Constants
  - String
  - Operators

1.11.2(A) Keywords

- These are predefined word in the program.
- They have specific functionality assign by programming language.
- They are reserved by language and cannot be used for any other purpose.
- Many keywords are common in C and C++.
- Some of important keywords are as follows :

auto	break	case	default	catch
char	class	const	do	continue
double	else	enum	float	goto
friend	if	int	long	new
private	public	protected	switch	union
this	return	short	signed	sizeof
static	struct	throw	try	virtual

1.11.2(B) Identifiers

- It is used to refer the name of variable, constant, array, function, classes, etc.
- It is also called as variables.
- Variables can be declared anywhere in the program.
- Variable declaration just reserves the memory space.
- Variable definition actually stores the value on reserved memory location.
  - int a;

    above statement is variable declaration.
  - a=10;

    above statement is variable definition.
- They are generally created and managed by the programmer as per requirement.

Technknowledge Publications

- C++ has different rules and regulations to create the identifiers. Some of the rules are as follows.
  - Only alphabets, number and underscore is allowed in variable declaration.
  - Variable name cannot start with number.
  - Variable name are case sensitive means uppercase and lowercase are different.
  - Keyword cannot be used as variable name.
- Some valid and invalid variable names are as follows.
  - int a; //valid
  - int abc; //valid
  - int abc123; //valid
  - int abc_123; //valid
  - int 123; //invalid
  - int 123abc; //invalid
  - int _abc; //valid

1.11.2(C) Constant

- It is used to refer the fix value.
- Value of constant cannot change during program execution.
- We cannot change the value during program execution. If we are trying to modify the value, run time error occurs.
- Constant can be created using const keyword.

```
const int a=10;
```

above statement create variable a whose value is 10 which cannot be changed during execution.

- Some of the constant are as follows :
  - 12345
  - 123.45
  - "abc"
  - 'a'
- Program 1.11.1 illustrate const concept.

➢ **Program 1.11.1**

```
#include<iostream>
using namespace std;
int main()
{
   const int a=10;
   cout<<"value of a is "<<a;
   a++;
   cout<<"Modified value of a is "<<a;
}
```

Output

```
In function 'int main()'
8:6: error: increment of read-only variable 'a'
```

Explanation : *Program 1.11.1 cannot be executed as constant value cannot be incremented.*

1.11.3 Const Keyword

- It is used to create the constant variable.
- Value of the constant variable cannot be changed or modified.
- Attempt to modify the value of the constant variable results into compiler error.
- Constant can be created with the help of const keyword or with the help of #define.
- With const keyword we can create the constant using following syntax.

```
const data_type variable_name=value;
```

- With #define we can create constant using the following syntax. The #define should be at the start of program only.

```
#define variable_name value
```

- Both the options have explained using Program 1.11.2.

➢ **Program 1.11.2**

```
#include<iostream>
#define max 100
using namespace std;
int main()
{
   const int a=50;
   cout<<"\n This is Constant with const keyword:
         "<<a;
```

Technknowledge Publications

```
  cout<<"\n This is constant using #define:
        "<<max;
  return 0;
}
```

Output

```
This is Constant with const keyword:50
This is constant using #define:100
```

1.12 Data Types

Q. What are primitive data types and user defined data types? **SPPU - Dec. 18, 3 Marks**

- Data type is used to specify the type of value stored in the variable.
- Data types specify how much memory should be reserved for certain variable.
- Data types in C++ are more like C only.
- C++ data types are shown in Fig. 1.12.1.

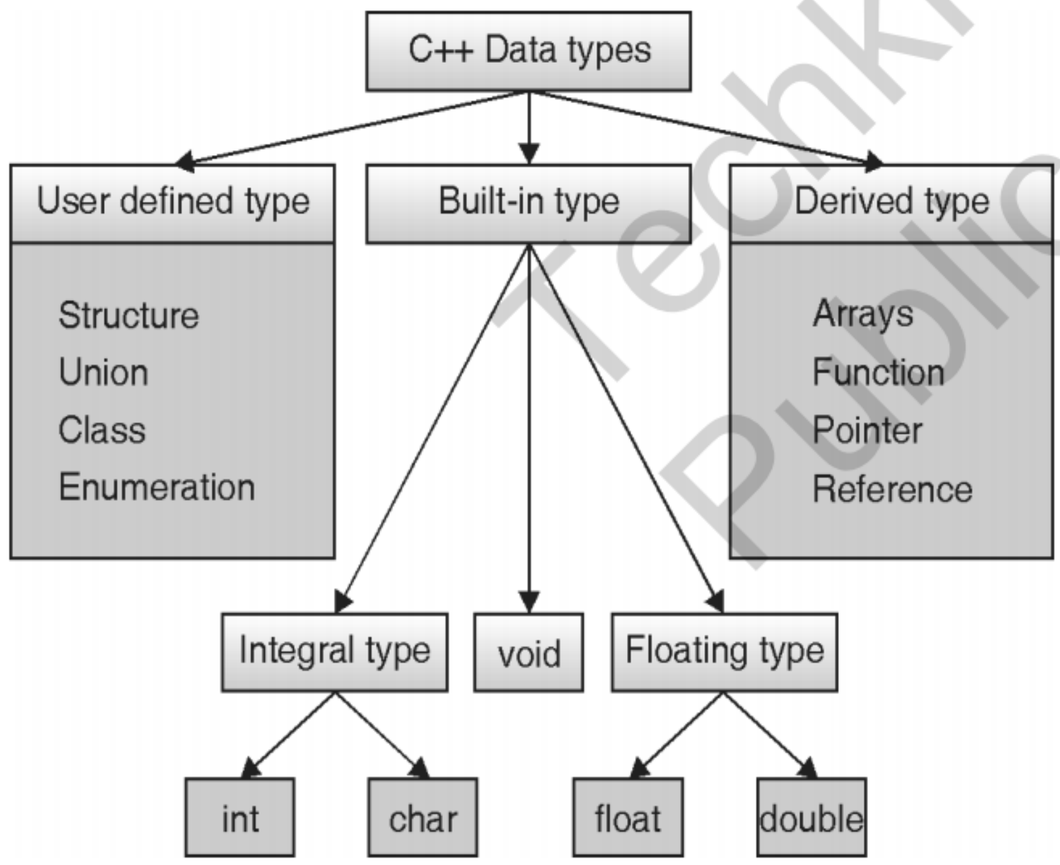

Fig. 1.12.1 : C++ data types

- Data types can be classified into user defined data types, built in data types, derived data types, etc.
- Built in data types consists of int, char, void, float, double, long, etc.
- User defined data type means data types created as per the user requirement. It contains structure, union, class and enumeration.
- Derived data types means data types derived from built in data types. It contains arrays, functions, pointer and reference.
- List of built in data type with memory requirement are listed below :

Data type	Memory requirement
int	4 byte
char	1 byte
singed char	1 byte
unsigned char	1 byte
float	4 byte
double	8 byte
long int	8 byte

➢ **Program 1.12.1 : Program print the size of every type of variable.**

```
#include<iostream>
using namespace std;
int main()
{
 cout<<"\n Size of integer is :"<<sizeof(int);
 cout<<"\n Size of double is :"<<sizeof(double);
 cout<<"\n Size of long is :"<<sizeof(long);
 cout<<"\n Size of long integer is :"<<sizeof(long
          int);
 cout<<"\n Size of char is :"<<sizeof(char);
 cout<<"\n Size of float is :"<<sizeof(float);
 cout<<"\n Size of long double is :"<<sizeof(long
         double );
 return 0;
}
```

Output

```
Size of integer is :4
Size of double is :8
Size of long is :8
Size of long integer is :8
Size of char is :1
Size of float is :4
Size of long double is :16
```

Technknowledge Publication

Note : Output may change. It depends upon which type of operating system and compiler we are using.

1.13 Structures

- Structure is user defined data types.
- It allows combining several data types together.
- Generally in array we can store similar elements. But in structure we can combine dissimilar elements also.
- Structures are used to store record like customer information, student information, item information, etc.
- Structure can be defined as follows :

```
struct stud
{
   int roll;
   char name[20];
   float marks;
};
```

Above block of code create structure having three elements namely roll, name and marks. We can individually access the elements of structure also.

- Array of structure is also possible.
- Element of structures are accessed by creating variable of structure.
- Element of structure are accessed using structure variable and structure element using .(dot) operator.
- Syntax for accessing elements of structures.

```
struct_variable.structure_element;
```

➢ **Program 1.13.1 : Illustrates the concept of structure.**

```
#include <iostream>
using namespace std;
int main()
{
   struct stud{
      int roll;
      float marks;
      char grade;
   };

   struct stud s;
   s.roll=10;
   s.grade='A';
   s.marks=66.66;

   cout<<"\n Student information is as follows: ";
   cout<<"\n Roll Number: "<<s.roll;
   cout<<"\n Marks: "<<s.marks;
   cout<<"\n Grade: "<<s.grade;

   return 0;
}
```

Output

```
Student information is as follows:
Roll Number: 10
Marks: 66.66
Grade: A
```

1.14 Enumerations

- Enumerations are user defined data type.
- It is useful to give name to numbers.
- It is useful in code simplicity.
- It automatically assigns numbers to name.
- It is useful in creating symbolic constant.
- Examples are as follows :

```
enum shape {square, circle, triangle};
enum color {red, green, blue, white, black};
enum switch_pos {on, off};
```

- In the above example number is assigned to value automatically. It means in above shape enum, square will assign 0, circle 1 and triangle 2 and so on.
- For color enum, red will be assigned 0, green 1 and so on.

Technknowledge Publication

- Enumerated values can be used in place of int value. It means we can write square if we wanted to write 0, triangle can be written if we wanted to write 2, etc.
- By default enum data type assign values from 0. We can change the starting value also. Means first value can be assigned 10 then next values will be 11, 12, and so on.
- Following is example with different assign value.

```
enum example {red=10, green=18, yellow=30};
```

In above example red will be assigned 10, green with 18, yellow with 30.

- Practically enum are used to create symbolic constant.

➢ **Program 1.14.1 : Illustrate the concept of enum.**

```
#include<iostream>
using namespace std;
int main()
{
   enum shape {circle, triangle, square};
   enum color {red=10, green, blue, white, black};
   enum example {A=15, B=20, C, D, E};

   cout<<"Shape value is : "<<circle<<"
      "<<triangle<<" "<<square ;
   cout<<"\n Color value is : "<<red<<"
      "<<green<<" "<<blue <<" "<<white<<"
      "<<black ;
   cout<<"\n Example value is : "<<A<<"
      "<<B<<" "<<C<<" "<<D<<" "<<E ;

   return 0;
}
```

Output

```
Shape value is : 0   1   2
Color value is : 10   11   12   13   14
Example value is : 15   20   21   22   23
```

1.15 Control Structure

- It is used to control the flow of program.
- Different control structure is used to control the flow of the program.
- It can be divided into selection statement and looping statement.
- Selection statement include if, if...else, switch, etc.
- Looping statement include while, do...while, for, etc.

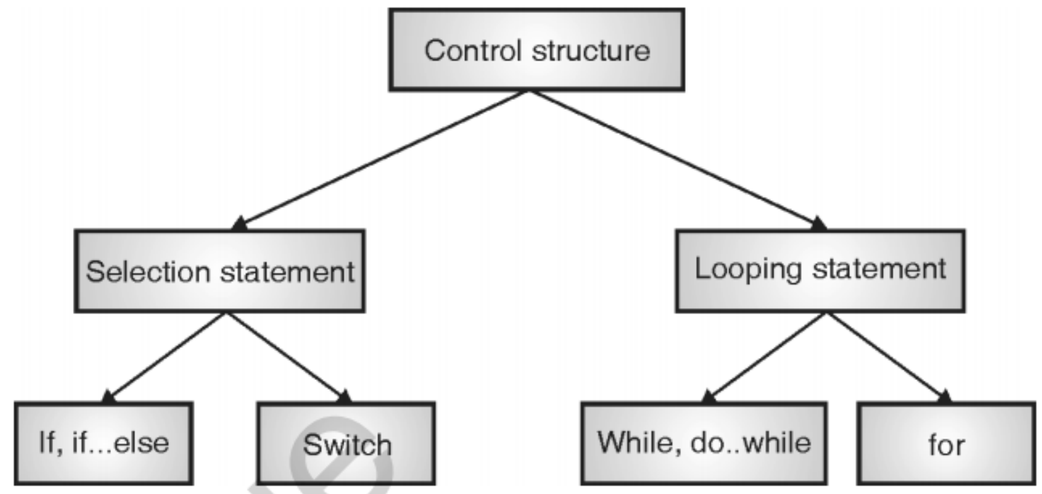

Fig. 1.15.1 : Control Structure

1.15.1 Selection Statement

If statement

- If statement is used as control statement.
- It alters the flow of program execution.
- It can be implemented as simple if statement or if....else statement.
- With simple if statement, if condition satisfy then that block of code will be executed.
- With if...else statement, if condition satisfies then if block will be executed otherwise else block will be executed.
- Condition can be specified using different relational operator like ==, !=, <, >, <=, >=, etc.

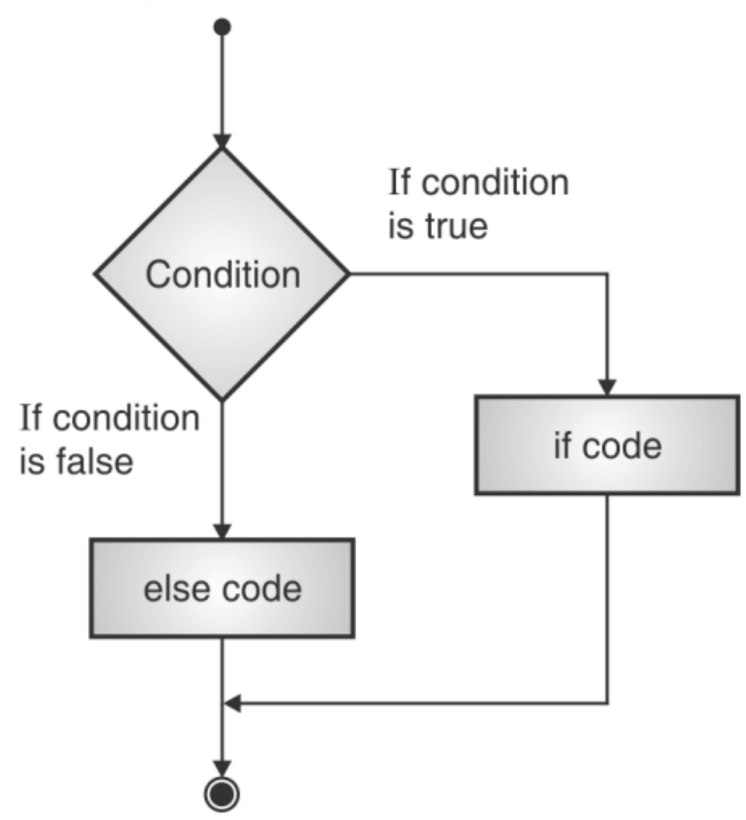

Fig. 1.15.2 : if...else statement

Tech Knowledge Publications

Syntax for if statement

```
if (condition)
{
   Statements1;
Statement 2;
.
.
Statement n;
}
```

Syntax of if...else statement

```
if (condition)
{
   Statements1;
Statement 2;
.
.
Statement n;
}
else
{
   Statements1;
Statement 2;
.
.
Statement n;
}
```

➢ **Program 1.15.1 : Illustrates if...else example.**

```
#include<iostream>
using namespace std;
int main()
{
   int a=10;
   if(a==10)
   {
      cout<<"\n Given number is 10";
   }
   else
   {
      cout<<"\n Given number is not 10";
   }
   return 0;
}
```

Output

```
Given number is 10
```

Switch statement

- It is multiple branch statement.
- Multiple conditions can be checked using switch statement.
- With if statement we can test only one condition where as with switch we can test many condition.
- In switch if first condition is not satisfied then it will check next condition, if not then next condition and so on. If no switch statement matches then it will come out of switch statement without executing any block of code.
- We can insert default case in switch statement. It will execute when no condition matches.
- We need to mention break statement after every case. If break is not mentioned then rest of all cases will be executed.
- Case value should be constant value only. We cannot mention variable as case value.

➢ **Program 1.15.2**

```
#include<iostream>
using namespace std;
int main()
{
   int choice=3;
   switch(choice)
   {
      case 1:
         cout<<"\n One";
         break;
```

Techknowledge Publication

```
    case 2:
        cout<<"\n Two";
        break;
    case 3:
        cout<<"\n Three";
        break;
    case 4:
        cout<<"\n Four";
        break;
    case 5:
        cout<<"\n Five";
        break;
    default:
        cout<<"\n Number is greater than 5";
  }
  return 0;
}
```

Output

```
Three
```

Explanation : *In Program 1.15.2 we have written 5 choices and one default case. It means 5 conditions we have written. During program execution it will check case 1, if not matching then it will check case 2, then case 3 and so on. If no case matches then it will execute default case. If Program 1.15.2 case 3 matches so it will get executed and break statement will be executed as a result of which rest of cases will not be matched. Suppose if we have given choice=7 then it will execute default. Default is optional block.*

1.15.2 Looping

- It is used to repeat block of statement.
- Condition is mentioned which decides how many times block of statements should be executed.
- Looping is possible with
  - For
  - While
  - Do....while

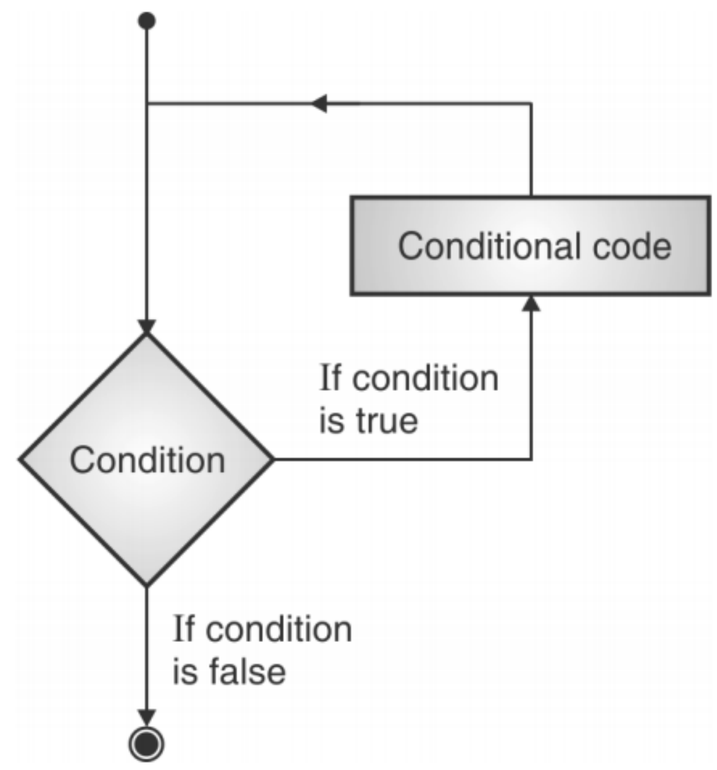

Fig. 1.15.3 : Looping

For loop

- It is used to execute certain block of statement for many times.
- In for loop we can mention loop initialization, condition checking and increment/decrement operation.
- In body of for loop we can write statements which should be executed repeatedly.

Syntax

```
for(initialization; condition; increment/decrement)
{
//Statements;
}
```

➢ **Program 1.15.3 : Illustrate the example of for loop.**

```
#include<iostream>
using namespace std;
int main()
{
    int i;
    for(i=0;i<5;i++)
    {
    cout<<"\n Loop is executing for "<<i<<"   time";
    }
    return 0;
}
```

Technknowledge Publication

Output

```
Loop is executing for 0   time
Loop is executing for 1   time
Loop is executing for 2   time
Loop is executing for 3   time
Loop is executing for 4   time
```

While loop

- It is simple form of for loop.
- for loop and while loop does the same thing.
- Only difference between for and while is in syntax.
- While loop is also used for executing the same block of code for many times.

Syntax

```
Initialization;
while(condition)
{
//Statements;
//increment/decrement;
}
```

➢ **Program 1.15.4**

```
#include<iostream>
using namespace std;
int main()
{
   int i;
   i=0;
while(i<5)
   {
      cout<<"\n Loop is executing for "<<i<<"
                time";
      i++;
   }
   return 0;
}
```

Output

```
Loop is executing for 0   time
Loop is executing for 1   time
Loop is executing for 2   time
Loop is executing for 3   time
Loop is executing for 4   time
```

Do....while

- Do....while is also same like while statement.
- Difference between while and do...while is that in while statement condition is checked before loop execution whereas in do....while condition is checked after loop execution.
- Simply while loop checks condition first whereas do...while checks condition afterwards.

Syntax

```
Initialization;
do
{
//Statements;
//increment/decrement;
}while(condition);
```

➢ **Program 1.15.5**

```
#include<iostream>
using namespace std;

int main()
{
   int i;
   i=0;
   do
   {
      cout<<"\n Loop is executing for "<<i<<"
                time";
      i++;
   }while(i<5);
   return 0;
}
```

Output

```
Loop is executing for 0   time
Loop is executing for 1   time
Loop is executing for 2   time
Loop is executing for 3   time
Loop is executing for 4   time
```

Tech Knowledge Publication

Break and Continue statement

- Break statement is used to stop looping.
- It breaks looping in for, while, do…while looping.
- Continue is opposite of break statement. It continue the execution of some loop.

➢ **Program 1.15.6 : (break.cpp) :**
Illustrate use of break and continue statement.

```
#include<iostream>
using namespace std;
int main()
{
   int i;
   for(i=0;i<5;i++)
   {
      cout<<"\n Number is   "<<i;
      if(i==2)
         break;
   }
   return 0;
}
```

Output

```
Number is   0
Number is   1
Number is   2
```

Explanation : *In the Program 1.15.6, loop get terminated when the value of i is 2. No further iterations will be executed.*

➢ **Program 1.15.7 : (contiune.cpp)**

```
#include<iostream>
using namespace std;
int main()
{
   int i;
   for(i=0;i<5;i++)
   {
      cout<<"\n Number is   "<<i;
      if(i==3)
         continue;
   }
   return 0;
}
```

Output

```
Number is   0
Number is   1
Number is   2
Number is   3
Number is   4
```

Explanation : *In the Program 1.15.7, loop will skipped for third iteration. It skips third iteration and continue the execution of rest of the iteration.*

1.16 Arrays and Strings

- It is nothing but collection of similar element.
- It is more like array in C only.
- Array can be single or multidimensional.
- Collection of character is known as string.
- Array can be created as follows :

```
int a[10];
float f[20];
char c[30];
```

- Above statement creates single dimension array. Array a is be created of size 10 it means we can store 10 different integer number. Array f will be created of size 20 means we can store 20 different floating point numbers. Array c will be created of size 30 means we can store 30 different characters.
- Array can initialized at the time creation also.

```
int a[5]={10,20,30,40,50};
```

0	1	2	3	4
10	20	30	40	50

10 will be stored on 0th position, 20 on 1st position and so on.

- Index position of array always start with 0. It means index will be 0 to 9 for array with 10 size.
- If we store character in single dimensional array with character data type then it is called as string.

Techknowledge Publication

➢ **Program 1.16.1 : Illustrate array and string concept.**

```
#include<iostream>
using namespace std;
int main()
{
    int a[10];
    char c[20];
    a[0]=100;        //store value in array at position 0
    a[1]=200;        //store value in array at position 1

    cout<<"\n element position 0 is "<<a[0];
    cout<<"\n element position 1 is "<<a[1];

    c[0]='a';
    c[1]='b';
    cout<<"\n element position 0 is "<<c[0];
    cout<<"\n element position 1 is "<<c[1];

    return 0;
}
```

Output

```
element position 0 is 100
element position 1 is 200
element position 0 is a
element position 1 is b
```

– String is nothing but sequence of characters. Internally string is stored as single dimensional array of characters.

– Character is written in single quotes whereas string is written in double quotes.

– Example of string and character is as follows :

```
char c='a';
            //create variable c and store single character a
char name[12]="MAAnsari";
                //create character array and store name
```

– Memory layout of single dimensional character array is shown below :

0	1	2	3	4	5	6	7	8	9	10	11
'M'	'A'	'A'	'n'	's'	'a'	'r'	'i'	'\0'			

– Every element in the string will be stored as character.

– '\0' indicates end of the string. It means string will be read upto '\0' character only. This is called as backslash null character. '\0' character will be automatically added by compiler. No need to add by user.

– '\0' character is useful in string manipulation.

– Every character in array will be stored as ASCII value. It means 65 will be stored instead of 'A', 66 for 'B' and so on.

➢ **Program 1.16.2 : Illustrate the concept of string.**

```
#include <iostream>
using namespace std;
int main()
{
    char Name[20]="MAAnsari";
    cout<<"Name is : " <<Name;

    return 0;
}
```

Output

```
Name is : MAAnsari
```

1.17 Class

Q. What is class and object ? Differentiate between class and object. **SPPU - Dec. 17, 6 Marks**

– Class is user defined data type.

– It is used to create new data type.

– Objects are created whose data type is class.

– It is one of the important features of C++.

– Class creation consists of two parts

Techknowledge Publication

- Class declaration
- Class function definition

– Class declaration defines type and scope of class members. Behaviour of class is defined by functions.

– Class declaration is similar to structure declaration.

– It starts with class keyword.

– Body of class is enclosed in { } and terminated with ;.

– Class contains declaration of variables and functions.

– We can specify variables and functions in the class.

– Variables are called as data members whereas functions are called as member functions.

– Only member functions can access private data member of the class.

– Public members can be accessed from anywhere in the program.

– Different access specifiers can be mentioned for class members otherwise will be considered private.

– The syntax of class is shown below :

```
class class_name
{
   public:
            //Public variables and functions
  private:
            //Private variables and functions
  protected:
            //Protected variables and functions
};
```

– Simple class example is as follows.

➢ **Program 1.17.1**

```
class employee
{
   public:
   char name[30];
   int emp_id;
   float salary;
   void add_employee();
   int del_employee();
   void disp_details();
   private:
      void get_salary();
      void get_personal_info();
}
class student
{
   public:
   char name[30];
   int roll_number;
   float marks;
   void add_student();
   int del_student();
      void search_student();
   private:
      void get_marks();
};
```

– Generally class name should be meaningful.

– After class declaration, class name becomes new data type.

1.18 Object

Q. What is class and object ? Differentiate between class and object. **SPPU - Dec. 17, 6 Marks**

– Class declaration just creates the template.

– Until we instantiate class, it is of no use.

– After class declaration we can create variable of its type.

– Class variable is known as objects.

– Memory is not allocated to class instead allocated to object.

– Consider following example.

```
student A;      //memory allocated to A.
```

– Above statement create variable A of type student. Variable is known as object.

Techknowledge Publications

- We can create many objects in one statement also.

  student A, B, C;
- Object creation is similar to variable declaration.
- Memory is allocated to object at this point of time.
- We can create object during class declaration also.

```
class student{
int roll;
float mark;
void accept();
void display();
}A, B, C;
```

Explanation : *Above example create class as well as create three objects namely A, B, C.*

- Class members can be accessed using object only.
- We can access class member using object name and .(dot) operator.

```
Object_name.member_name;
```

- Above statement access the class member.
- Example of class and object is below :

➢ **Program 1.18.1**

```
#include<iostream>
using namespace std;

class account{
   private:
      int account_number;
      float balance;
   public:
      void set_value(){
         account_number=10;
         balance=100;
      }
      void display(){
        cout<<"Account Number: "<<account_number;
       cout<<"\n Balance: "<<balance;
      }
};
int main()
{
   account A;

   A.set_value();
   A.display();
   return 0;
}
```

Output

```
Account Number: 10
Balance: 100
```

1.19 Class Scope and Accessing Class Members

- Everything declared inside class can be considered in class scope.
- Data member and member functions of class belong to class scope.
- In object oriented programming languages we use word access rather than scope.
- Scope of object oriented programming is divided into three parts namely public, private and protected.
- Public member of class are accessible from anywhere in the program.
- Private member of class is accessed only through methods of class in which they have defined.
- Protected member of class is accessible to their own class and all subclasses of that class.
- Within class scope, members are immediately accessible to all members of class by name.
- But public member of class can accessible outside class scope using object of that class. E.g. there are two classes say A and B. Suppose we need to access public member of A from B class then, it is possible by creating object of class A.

Tech Knowledge Publications

1.20 Access Specifiers

Access Specifiers

Q. Define the term Member access control. **SPPU - May 14, 2 Marks**

Q. Explain public, private and protected keywords using program. **SPPU - Dec. 17, 6 Marks**

- There are three access specifier in C++.
- Access specifiers are public, private and protected.
- Data hiding feature can be achieved using these access specifiers.
- Class can have public, private and protected access specifiers.
- Default access specifiers for members and classes is private.
- They are used to specify boundary of variable access.

```
class example{
    public:
//public member
    private:
//private member
    protected:
//protected member
};
```

- Accessibility from different classes is shown in Table 1.20.1.

Table 1.20.1 : Access specifiers in C++

Class member access specifier	Access from own class	Accessible from derived class	Accessible from object
Private member	Yes	No	No
Protected member	Yes	Yes	No
Public member	Yes	Yes	Yes

- Public variables or functions can be used by everyone. It can be accessed by member functions of the class as well as from other classes also.

```
class example
{
    public:
    int a, b, c;
    void disp();
};
```

- In the above example a, b, c and disp() function is public.
- Private members can be accessed only using the member functions of the class. Outside the class they are not accessible.
- By default members of class are private. It means if we are not specifying the access specifier then all the members becomes private.

```
class example
{
    private:
    int a, b, c;
    void disp();
};
```

- In the above example a, b, c and disp() function is private.
- Protected members of the class are accessible by class functions. They are also accessible to subclass of that class. Other than parent and child class they are inaccessible.

```
class example
{
    protected:
    int a, b, c;
    void disp();
};
```

- In the above example a, b, c and disp() function is protected.

Techknowledge Publication

– They will be accessible to class as well as his subclass. The complete example is given below.

➢ **Program 1.20.1**

```
#include<iostream>
using namespace std;
class abc
{
   private:
   int b;
   void disp_private()
   {
      b=200;
      cout<<"\n This is private function";
      cout<<"\n Value of private function is"<<b;
      cout<<"\n Value of public function is"<<a;
   }
   public:
   int a;
   void disp_public()
   {
      a=100;
      cout<<"\n This is public function";
      cout<<"\n Value of public function is"<<a;
   }
};
int main()
{
   abc x;
   x.disp_public();
   cout<<"\n Value of a from public class is:"<<x.a;
   return 0;
}
```

Output

```
This is public function
Value of public function is100
Value of a from public class is:100
```

1.21 Class and Data Abstraction

Data Abstraction

Q. What is abstraction ? Why it is important ?

SPPU - May 15, 4 Marks

– Data abstraction is nothing but hiding implementation details from the end users.

– It only provides essential information.

– It present the information without providing implantation details.

– Data abstraction technique separates interface and implementation details.

– Consider real time example where we access internet, browse sites, download data, etc. But end user is totally unaware about its internal. From where data is getting accessed, which programming languages is used, which operating system is used, which connection method used, etc. It means internet totally separate internal representation from external interface.

– In C++ also there are many functionalities which are used by programmer without knowing their internals.

– Consider simple hello word example.

➢ **Program 1.21.1**

```
#include<iostream>
using namespace std;
int main()
{
  cout<<"Hello C++"
return 0;
}
```

– In the above program internals of the program is hidden from programmer or user.

– How cout function is worked is hidden from user.

– What happens when iostream file included in the program that is also hidden. Data abstraction gives many benefits like Class internals are protected from errors.

Techknowledge Publication

– Any class implemented with public and private members can be considered as example of data abstraction.

1.22 Separating Interface from Implementation

– As seen in data abstraction which is nothing but hiding implementation details from end users.
– It aims at reducing implementation details from the end users.
– All the member functions are stored in .h header file.
– Programmer is not aware of the implementation of these file.
– Mostly we create generic class to provide the functionality.
– Functions are also implemented as generic functions.

1.23 Programming in C++

1.23.1 Variable Declarations

– Variable are used to store value.
– Type of variable is specified by the data types. We have already discussed different data types.
– Generally, all the variables are declared at the start of the function.
– In C++ it is not necessary to declare the variable at the start.
– We can declare variable in the program as and when required by the programmer. Only requirement is that it should be declared before its use.
– When we declare the variable, it signifies that much memory is reserved by the operating system.
– With variable declaration we just reserve the memory for later use.
– With variable definition we are storing the value on the reserved memory locations.
– In short variable refers to reserved memory location.

➢ **Program 1.23.1**

```
#include<iostream>
using namespace std;
int main()
{
   int a;          //declaration of variable
   int b=20;       //declaration and definition
   a=50;           //definition of variable
   cout<<"\n Value of a and b are :"<<a<<"
           "<<b;
   for(int i=0;i<=5;i++)        // i is declared here
   {
     cout<<"\n value of i is :"<<i;
   }
   return 0;
}
```

Output

```
Value of a and b are :50    20
 value of i is :0
 value of i is :1
 value of i is :2
 value of i is :3
 value of i is :4
 value of i is :5
```

1.23.2 Global Scope

– Scope is nothing but the region of program.
– Scope signifies the lifetime of program elements.
– Local scope is bound to certain block where it is declared.
– Global scope is applicable for the entire program. Value in the global scope can be accessed by the entire program.
– In the following program c is global variable which can be accessed by the entire program.
– Variable a and b are local variable which can be accessed only through main() function.

Techknowledge Publication

➢ **Program 1.23.2**

```
#include<iostream>
using namespace std;
void myfunction();
int c=50;          //global variable
int main()
{
   int a;              //local variable
   int b=20;           //declaration and definition
   a=10;
   cout<<"\n Local variable a and b are :"<<a<<"
          "<<b;
   cout<<"\n Value of global variable is "<<c;
   myfunction();
   return 0;
}
void myfunction()
{
   cout<<"\n This is in function";
                          // a and b cannot be accessed.
    cout<<"\n Value of global variable is :"<<c;
}
```

Output

```
Local variable a and b are :10    20
Value of global variable is 50
This is in function
 Value of global variable is :50
```

– If we have same local and global variable then local variable will be accessed over the global variable.

➢ **Program 1.23.3**

```
#include<iostream>
using namespace std;
int c=50;    //global variable
int main()
{
   int c=100;
   cout<<"\n value of c variable is:"<<c;
   return 0;
}
```

Output

```
value of c variable is : 100
```

Note : In Program 1.23.3, C is available as local as well as global variable. But still local value is used.

– If two variables with same name are present and we wanted to access global then it is possible with :: operator.

– If we precede :: with variable name then global value is referred and not local value.

➢ **Program 1.23.4**

```
#include<iostream>
using namespace std;
int c=50;    //global variable
int main()
{
   int c=100;
   cout<<"\n value of c variable is:"<<::c;
   return 0;
}
```

Output

```
value of c variable is:50
```

1.23.3 Reference Variables

Q. Write a short note on Reference variable with example. **SPPU - May 16, 3 Marks**

Q. What is difference between pointer and references ? Explain using program. **SPPU - Dec. 17, 4 Marks**

– This facility is available in C++.

– It is used to create the alias name to already created variable.

– If we make some variable reference to some other variable then both variable behave identically.

– If we change the value of one variable then it reflects in another variable also.

– Reference variable can be created using following syntax :

```
data_type & reference_variable_name = old_variable_name
```

Tech Knowledge Publications

– Example of reference variable is as follows.

```
int abc=500;
int & xyz=abc;
```

– The example with all possibilities are given below.

➢ **Program 1.23.5**

```
#include<iostream>
using namespace std;
int main()
{
   int abc=500;
   int & xyz=abc;
   cout<<"\n The value of abc is :"<<abc;
   cout<<"\n The value of xyz is :"<<xyz;
   abc=1000;
   cout<<"\n The value of abc is :"<<abc;
   cout<<"\n The value of xyz is :"<<xyz;
   xyz=2000;
   cout<<"\n The value of abc is :"<<abc;
   cout<<"\n The value of xyz is :"<<xyz;
   return 0;
}
```

Output

```
The value of abc is :500
The value of xyz is :500
The value of abc is :1000
The value of xyz is :1000
The value of abc is :2000
The value of xyz is :2000
```

– It is mainly used in parameter passing in functions.

– Example is given in Program 1.23.6.

➢ **Program 1.23.6**

```
#include<iostream>
using namespace std;
void function(int &a);
int main()
{
  int abc=500;
  cout<<"\n In main";
  cout<<"\n The value of abc is :"<<abc;
  function(abc);
  cout<<"\n In Main Function ";
  cout<<"\n Modified Value of variable is "<<abc;
  return 0;
}
void function(int &a)
{
   cout<<"\n In function ";
   cout<<"\n Value of variable is "<<a;
   a=1000;
   cout<<"\n In function ";
   cout<<"\n Modified Value of variable is "<<a;
}
```

Output

```
In main
The value of abc is :500
In function
Value of variable is 500
In function
Modified Value of variable is 1000
In Main Function
Modified Value of variable is 1000
```

1.23.4 Operators in C++

– C++ uses different operators for variety of purpose.

– Some of the operators are listed below.

Table 1.23.1 : Operators in C++

Operator	Meaning
+	Addition
-	Subtraction
/	Division
*	Multiplication
%	Modulus
++	Increment
--	Decrement

Techknowledge Publication

Operator	Meaning
==	Equality
!=	Not equal
>	Greater than
<	Less than
&&	Logical AND
||	Logical OR
!	Logical NOT
&	Bitwise AND
|	Bitwise OR
^	XOR
<<	Left shift
>>	Right shift
=	Assignment
::	Scope resolution operator
new	Memory allocation operator
cin	Get operator
cout	Put operator
setw	Field width separator
sizeof	Return size of variable

Operator precedence

It is nothing but the priority of the operator. It means if two operators are present in the same statement then which operator will be evaluated first will be decided by operator precedence.

For example multiplication will have higher precedence than addition operator. Table 1.23.2 shows the operator precedence.

Table 1.23.2 : Operator precedence

High priority	
::	Scope resolution
->, (), [], , postfix ++, postfix --	
Prefix ++, prefix --, unary +, unary -	
*, /, %	Arithmetic operator
+, -	Arithmetic operator
<<, >>	Bitwise left and right shift
<, <=, >, >=	Relational operator
==, !=	Relational operator
&	Bitwise AND operator
^	Bitwise XOR
|	Bitwise OR
&&	Logical AND
||	Logical OR
,	Comma
Low priority	

1.23.5 Scope Resolution Operator

- It is used to resolve the name conflict of variable names in the program.
- Program can have same variable name in different scope.
- It is means program can same name for global variable and function variable.
- If same name is used in different scope then which variable will be referred.
- Again for some purpose if we wanted to use certain variable only then what to do.
- This situation can be resolved using scope resolution operator.

➢ **Program 1.23.7 : Illustrate the use of scope resolution operator.**

```
#include <iostream>
using namespace std;
int a=10;            //global variable
int main()
{
   int a=200;     //local variable
   cout<<"\n Value of variable a is  "<<a;
   return 0;
}
```

Output

```
Value of variable a is 200
```

TechKnowledge Publications

Explanation : *Always local variable will be used. If we wanted to access the global variable then it is not possible. If we wanted to use global variable the scope resolution operator must be used. :: acts as scope resolution operator. It will always refer to global copy of variable and not local copy.*

– Program 1.23.8 illustrates the use of global variable.

➢ **Program 1.23.8 : (with scope resolution operator)**

```
#include<iostream>
using namespace std;
int a=10;    //global variable
int main()
{
   int a=200;     //local variable
   cout<<"\n Value of variable a is  "<<::a;
   return 0;
}
```

Output

```
Value of variable a is 10
```

1.23.6 'cin' and 'cout' Operator

– cin and cout are the two operator in C++ for printing and getting the input from the users.

– They are more like printf() and scanf() in C.

– cout is considered as put operator whereas cin is called as get operator.

– cout prints the specified value on screen.

– cin accepts the value from the user and store in the variable.

– It is part of iostream header file.

– iostream contains the declaration of cin and cout.

– More specifically we can use istream.h for cin and ostream.h for cout. If we wanted to use both cin and cout then we can add iostream.h.

– The different example of cin and cout are given below :

➢ **Program 1.23.9 : Printing message with cout**

```
#include<iostream.h>
using namespace std;
int main()
{
        cout<<"this is message from C++";
        return 0;
}
```

➢ **Program 1.23.10 : Printing variable value with message**

```
#include<iostream.h>
using namespace std;
int main()
{
        int a=10;
        cout<<"Value of a is :"<<a;
        return 0;
}
```

➢ **Program 1.23.11 : Printing message on next line**

```
#include<iostream.h>
using namespace std;
int main()
{
        cout<<"\n This is message from C++ on next
                line";
        return 0;
}
```

➢ **Program 1.23.12 : Printing multiple variable values**

```
#include<iostream.h>
using namespace std;
int main()
{
        int a=10, b=20, c=30;
        cout<<"\n This is value of a, b,
                c"<<a<<b<<c;
        return 0;
}
```

Technknowledge Publication

➢ **Program 1.23.13 : Getting single value from user**

```
#include<iostream.h>
using namespace std;
int main()
{
        int a;
        cout<<"\n Enter the value";
        cin>>a;
        cout<<"\n The value of a is:"<<a;
        return 0;
}
```

➢ **Program 1.23.14 : Getting the multiple values from the users**

```
#include<iostream.h>
using namespace std;
int main()
{
        int a, b, c;
        cout<<"\n Enter three value";
        cin>>a>>b>>c;
        cout<<"\n The value of a, b, c   are:
              "<<a<<b<<c;
        return 0;
}
```

1.23.7 Formatting and I/O Manipulators

Q. What are different formatting activities supported by Stream manipulators ? Support with suitable examples. **SPPU - May 15, 4 Marks**

Q. Explain the term manipulators. **SPPU - Dec. 15, 2 Marks**

Formatting activities supported by Stream manipulators

- Output in C++ can be formatted using many options.
- We can print the output on next line using \n.
- We can print the output with tab on same line using \t.
- endl is also used for next line.
- setw is used specify the width of output. If we specify the 40 width then output will be printed in 40 characters on right side.
- setfill() is used to draw some character.
- Floating point precision can be specified using setprecision(). It is possible to specify maximum number of digits in output.

➢ **Program 1.23.15**

```
#include<iostream>
#include<iomanip>
using namespace std;
 int main()
{
   cout<<"\n This is on next line \n";
   cout<<"\t This is with tab\n";
   cout<<endl<<"This is on next line"<<endl;
   cout<<setw(40)<<"This is after 40 spaces\n";
   cout<<setfill('+')<<setw(40)<<"+"<<endl;
   float f=10.33333333333;
   cout<<"\n Floating point
   value"<<setprecision(4)<<f;
   return 0;
}
```

Output

```
This is on next line
This is with tab
This is on next line
            This is in 40 width
++++++++++++++++++++++++++++++++
+++
Floating point value10.33
```

1.24 Functions

- It is nothing but group of statement to perform certain task.
- Every C++ program have atleast one function but it can have more functions also.
- Every C++ program has main() function which is starting point for the program.

Technknowledge Publications

- Program code can be divided into different functions for better management.
- Function in program involve three activity.
  - Function declaration
  - Function definition
  - Function calling.
- Function declaration means writing the name of program. It is mandatory part in the function.
- Function definition is nothing but writing body of the function. Task performed by the function is written in the definition. It is mandatory part for function.
- Function is called whenever required. It is optional part in the program.

➢ **Program 1.24.1 : Illustrate the concept of functions.**

```
#include <iostream>
using namespace std;
int add(int a, int b);
int sub(int a, int);
int main()
{
   int a=20, b=10, c;
   cout<<"\n Addition is ";
   c=add(a,b);
   cout<<c;
   cout<<"\n Subtraction is ";
   c=sub(a,b);
   cout<<c;
}
int add(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int sub(int a, int b)
{
   int c;
   c=a-b;
   return c;
}
```

Output

```
Addition is 30
Subtraction is 10
```

1.24.1 Function Prototypes

- It is nothing but the signature of function.
- It consists of name of functions, passing parameters to the functions, returns data type of functions.
- Functions creation consists of three steps.
- In first steps function is declared. Mentioning name of variable in the passing parameter is not necessary.

```
Return_type function_name (para_one, para_two, ....);
```

- In second step functions is defined.

```
return_type function_name(para_one, para_two, ....);
{
   ----
   ----
   ----
}
```

- In third step functions is used as per the requirement.

```
function_name (value_para_one, value_para_two, .....);
```

- Whenever functions are called, compiler checks the template of the function with the help of function declaration. If the functions calling matches with function declaration template then only function is called.
- We can have function without any passing parameters also.
- The example of function prototyping is explained with the help of Program 1.24.2.

Techknowledge Publication

➢ **Program 1.24.2**

```
#include<iostream>
using namespace std;
void fun();
void fun1(int );
void fun2(int a, float b);
int fun3();
int fun4(int a, int b);
int main()
{
 fun();
 fun1(10);
 fun2(100, 20.5);
 int a=fun3();
 cout<<"\n Value from fun3 is "<<a;
 int b=fun4(100,200);
 cout<<"\n The value from fun4 is "<<b;
 return 0;
}
void fun()
{
   cout<<"\n This is function without passing
            parameters";
}
void fun1(int a)
{
   cout<<"\n This is function with one passing
            parameters";
   cout<<"\n Value of variable is :"<<a;
}
void fun2(int a, float b)
{
   cout<<"\n Function with two passing parameters";
   cout<<"\n Value of variables are "<<a<<"
            "<<b;
}
int fun3()
{
cout<<"\n Function with no passing parameter but
          return value";
     return 100;
}
int fun4(int a, int b)
{
   cout<<"\n Function with two passing parameter
            and return value";
   return (a+b);
}
```

Output

```
This is function without passing parameters
This is function with one passing parameters
Value of variable is :10
Function with two passing parameters
Value of variables are 10020.5
Function with no passing parameter but return value
Value from fun3 is 100
Function with two passing parameter and return value
The value from fun4 is 300
```

1.24.2 Default Parameters

- It means default value is provided at the time of function declaration.
- While calling the function, if we do not provide the passing parameters then default value of parameters are used which are specified during function declaration.
- Generally end variables should make default. We can make first variables as ordinary and rest of the variables as default. First parameter is default and rest of the variables are normal will not be allowed.

➢ **Program 1.24.3**

```
#include<iostream>
using namespace std;
void fun(int a=10, float b=20.5);
void fun1(int a, float b=10.5);
int main()
{
  cout<<"\n Function with both the parameters are
            default";
  fun();
```

Techknowledge Publication

```
    cout<<"\n Passing the required value to fun()";
    fun(500, 600.5);
    cout<<"\n Function with one of the parameters are
            default";
    fun1(100);
    cout<<"\n Function with one of the parameters are
            default";
    fun1(200,200.5);
    return 0;
}
void fun(int a, float b)
{
    cout<<"\n Value of variables are "<<a<<"
            "<<b;
}
void fun1(int a, float b)
{
    cout<<"\n Value of variables are "<<a<<"
            "<<b;
}
```

Output

```
Function with both the parameters are default
Value of variables are 10     20.5
Passing the required value to fun()
Value of variables are 500    600.5
Function with one of the parameters are default
Value of variables are 100    10.5
Function with one of the parameters are default
Value of variables are 200    200.5
```

1.24.3 Default and Constant Arguments

- Default argument is already explained in the earlier section.
- In this section we will focus on constant argument.
- Whenever we are passing the argument to function, we can make use of that value for different purposes. Sometimes we just use the variable. Sometimes we modify the passed argument.
- If we wanted to keep the passed value remain intact then we can make the passing parameters as constant.
- Constant value of the passing parameters is specified during function declarations only.
- The function cannot modify the constant value, it can just use it.
- The concept of default and constant parameters are explained with the help of Program 1.24.4.

➢ **Program 1.24.4**

```
#include<iostream>
using namespace std;
void fun(const int a=10, const float b=20.5);
void fun1(int a, const float b=10.5);
int main()
{
    cout<<"\n Function with both the parameters
            constant ";
    fun();
    cout<<"\n Function with one of the parameter
            constant";
    fun1(100);
    return 0;
}
void fun(const int a, const float b)
{
    //int c=a+b;      //error
    cout<<"\n Value of variables are "<<a<<"
            "<<b;
}
void fun1(int a, const float b)
{
    a++;            //allowed as not constant variable
    //b++;          //error
    cout<<"\n Value of variables are "<<a<<"
            "<<b;
}
```

Output

```
Function with both the parameters constant
Value of variables are 10     20.5
Function with one of the parameter constant
Value of variables are 101    10.5
```

Tech Knowledge Publications

1.24.4 Accessing Function and Utility Function

- Access functions are used to read and display data.
- Access functions are also used to check true or false condition in the program.
- These functions are also referred as predicate functions.
- Some example of access functions are isEmpty(), isFull(), etc in stack operations using STL.
- Utility functions are used for number conversions, variable length, argument processing, searching, sorting, random number generations.
- To use utility functions stdlib header file is used.
- Some of the important functions in C++ are as follows :

Function	Meaning
abort()	Abnormal program termination
abs()	Get absolute value
atof()	Convert string into double
atoi()	Convert string into integer
bsearch()	Useful to perform binary search
rand()	Useful to generate random number
exit()	Terminate the program
qsort()	It is useful for quicksort.

1.25 Constructors

Q. Define Constructor. **SPPU - May 18, 2 Marks**

Q. What do you mean by constructor ? **SPPU - May 19, 2 Marks**

- Constructor is used to initialize the objects.
- It is special function which is automatically get called during object creation.
- Some of the rules and regulations are given for creating the contractors :
  - Name of constructor should be same as that of class name.
  - It should not have return data types.
  - It can have passing parameters.
  - Constructors can have default argument also.
  - Constructor without passing parameter is called as non-parameterized constructor.
  - Constructor with passing parameters are called as parameterized constructor.
  - We cannot call the constructor; it is automatically get called.
  - Constructor cannot be virtual unlike virtual function.
  - We cannot refer the address of constructor.
  - Every class have one default constructor. If we are not creating the constructor, then this default constructor will get automatically created which is not visible to users.
  - If we are created at least one constructor, then default constructor by the system will not be created.
  - It should be in the public section of the class.
  - It can be defined inside as well as outside the class.
- Syntax for constructor declaration is as follows :

```
class example{
    public:
        example(){

            //declaration and definition of constructor
            //definition
        }
};
```

➢ **Program 1.25.1 : Write a program to explain use of constructor.** **SPPU - May 18, 4 Marks**

```
#include<iostream>
using namespace std;
class construct_example
{
  public:
  construct_example()
```

Tech Knowledge Publications

```
  {
     cout<<"\n This message is by constructor";
  }
};
int main()
{
   cout<<"\n Program for constructor";
   construct_example a;
   return 0;
}
```

Output

```
Program for constructor
This message is by constructor
```

Explanation : *As shown in the output of the Program 1.25.1, we have not called the constructor explicitly but still it is called during object creation in main*

1.26 Types of Constructor

Following are various types of constructors :

1. Parameterized Constructor
2. Multiple Constructors within Class
3. Constructor with Default Parameters
4. Copy Constructor
5. Default Copy Constructor

1.26.1 Parameterized Constructor

- Parameterized constructor is used to pass the argument to constructor during object creation.
- Passing values can be used for initialization purposes.
- Example is given in Program 1.26.1.

➢ **Program 1.26.1**

```
#include<iostream>
 using namespace std;
class construct_example
{
  public:
  construct_example(int a, int b)
  {
     cout<<"\n This message is by constructor";
     cout<<"\n Passing parameter of the
              constructor"<<a<<"      "<<b;
  }
};
int main()
{
   cout<<"\n Program for constructor";
   construct_example a(10,20);
   return 0;
}
```

Output

```
Program for constructor
This message is by constructor
Passing parameter of the constructor10 20
```

1.26.2 Multiple Constructors within Class

- We can have multiple constructors in class also.
- It means class can have combination of default, parameterized, etc. constructor.
- The example is given in Program 1.26.2.

➢ **Program 1.26.2**

```
#include<iostream>
 using namespace std;
 class construct_example
{
  public:
  construct_example()
  {
     cout<<"\n This is non-parameterized
              constructor";
  }
  construct_example(int a, int b)
  {
     cout<<"\n This is parameterized constructor";
```

Technknowledge Publication

```
    cout<<"\n Passing parameter of the constructor
        "<<a<<"     "<<b;
  }
};
int main()
{
   cout<<"\n Program for constructor";
   construct_example a();
   construct_example b(10,20);
   return 0;
}
```

Output

```
Program for constructor
This is parameterized constructor
Passing parameter of the constructor  10     20
```

1.26.3 Constructor with Default Parameters

- We can have constructor with default parameters also.
- If we are not passing the value to during object creation then default values are used.
- All the rules for default constructor will be applicable.

➢ **Program 1.26.3**

```
#include<iostream>
 using namespace std;
 class construct_example
{
  public:
  construct_example(int a, int b=100)
  {
     cout<<"\n This is constructor with default
              parameter";
     cout<<"\n Passing parameter of the constructor
              "<<a<<"     "<<b;
  }
};
int main()
{
   cout<<"\n Program for constructor";
   construct_example a(50);
   construct_example b(30,40);
   return 0;
}
```

Output

```
Program for constructor
This is constructor with default parameter
Passing parameter of the constructor  50     100
This is constructor with default parameter
Passing parameter of the constructor  30     40
```

Explanation : *During object creation of a we are passing only one argument that time, it is default value 100. At the time of creating another object b, we are passing both the values that time passing parameters are used by the constructor.*

1.26.4 Copy Constructor

Q. Write short note on : Copy constructor.

SPPU - Dec. 14, 3 Marks, Dec. 15, 2 Marks

- It is special constructor in C++ to create new object as copy of exiting object. It is standard way of copying object in C++. First argument to such constructor is a reference to the object of the same type.
- It is used to initialize one object from another object of same type.
- It can be used to copy an object to pass it as argument to function.
- Copy an object to return it from function.
- Typical syntax is as follows.

```
Classname (const classname &objectName){
            ///constructor
}
```

- Copy constructor example is given in Program 1.26.4.

Techknowledge Publications

➢ **Program 1.26.4 : Write appropriate C++ program which uses copy constructor.**

SPPU - May 19, 2 Marks

```
#include<iostream>
using namespace std;

class copy1
{
private:
   int a, b;
public:
   copy1(int a1, int b1) { a = a1; b = b1; }

   // Copy constructor
   copy1(const copy1 &p2) {a = p2.a; b = p2.b; }

   int getA()          {  return a; }
   int getB()          {  return b; }
};

int main()
{
   copy1 c1(100, 200);
   copy1 c2 = c1;
   cout<<"c1.a = " << c1.getA() << ", c1.b = "
      << c1.getB();
   cout<<"\nc2.a = " << c2.getA() << ", c2.b = "
      << c2.getB();

   return 0;
}
```

Output

```
c1.a = 100, c1.b = 200
c2.a = 100, c2.b = 200
```

1.26.5 Default Copy Constructor

- C++ create default constructor if we don't define it.
- Constructor created by compiler has empty body. It does not assign default values to data members.
- Compiler also creates a copy constructor if we don't write our own copy constructor.
- Unlike default constructor, body of compiler created copy constructor is not empty, it copies all data members of passed object to the object which is being created.

➢ **Program 1.26.5**

```
#include<iostream>
using namespace std;
class copy1
{
private:
   int a, b;
public:
   copy1(int a1, int b1) { a = a1; b = b1; }

   int getA()          {  return a; }
   int getB()          {  return b; }
};

int main()
{
   copy1 c1(100, 200);
   copy1 c2 = c1;
   cout<<"c1.a = " << c1.getA() << ", c1.b = "
      << c1.getB();
   cout<<"\nc2.a = " << c2.getA() << ", c2.b = "
      << c2.getB();
   return 0;
}
```

Output

```
c1.a = 100, c1.b = 200
c2.a = 100, c2.b = 200
```

Explanation : *If we observe in the Program 1.26.5, we have not defined the copy constructor but still program is working as copy constructor is created by default by the compiler of the system.*

1.27 Destructors

Q. Define Destructor. **SPPU - May 18, 2 Marks**

Q. What do you mean by destructor? **SPPU - May 19, 2 Marks**

- It is used to destroy the objects in C++. They destroy the object which are created by the constructor.
- It is also member of class and have same name with starting character as ~(tilde).

TechKnowledge Publications

– The syntax is given below.

```
~class_name()
{
  ----
  ----
}
```

– It cannot have passing parameter.

– It is automatically called by the compiler.

– It is called at the end of program execution. When we terminate the program, just before the program termination, destructor is get called.

– It is used to deallocate the memory used by the object which is allocated during object creation.

– It is used for clean-up activity and for doing garbage collection.

– It is recommended to add the destructor in the program.

– Example is given in Program 1.27.1.

➢ **Program 1.27.1 : Write a program which uses default constructor, parameterized constructor and destructor.**

SPPU - Dec. 17, 6 Marks May 18, 4 Marks

```
#include<iostream>
using namespace std;
class construct_example
{
  public:
  construct_example()
  {
    cout<<"\n This is non-parameterized
        constructor";
  }
  construct_example(int a, int b)
  {
    cout<<"\n This is parameterized constructor";
    cout<<"\n Passing parameter of the constructor
        "<<a<<"     "<<b;
  }
  ~construct_example()
  {
    cout<<"\n Created object has been deleted";
  }
};
int main()
{
  cout<<"\n Program for constructor";
  construct_example a();
  construct_example b(10,20);
  return 0;
}
```

Output

```
Program for constructor
This is parameterized constructor
Passing parameter of the constructor   10     20
Created object has been deleted
```

1.28 Objects and Memory Requirements

– Memory is allocated to object when they are created.

– Memory is not allocated for class.

– But member functions of class are created and placed in memory only once. When we create many object of same class then no separate memory will be allocated to function.

– All objects belonging to same class use same copy of functions.

– When different objects are created, memory is allocated to data members only.

Technknowledge Publication

– Fig. 1.28.1 illustrate the concept of memory requirement for object.

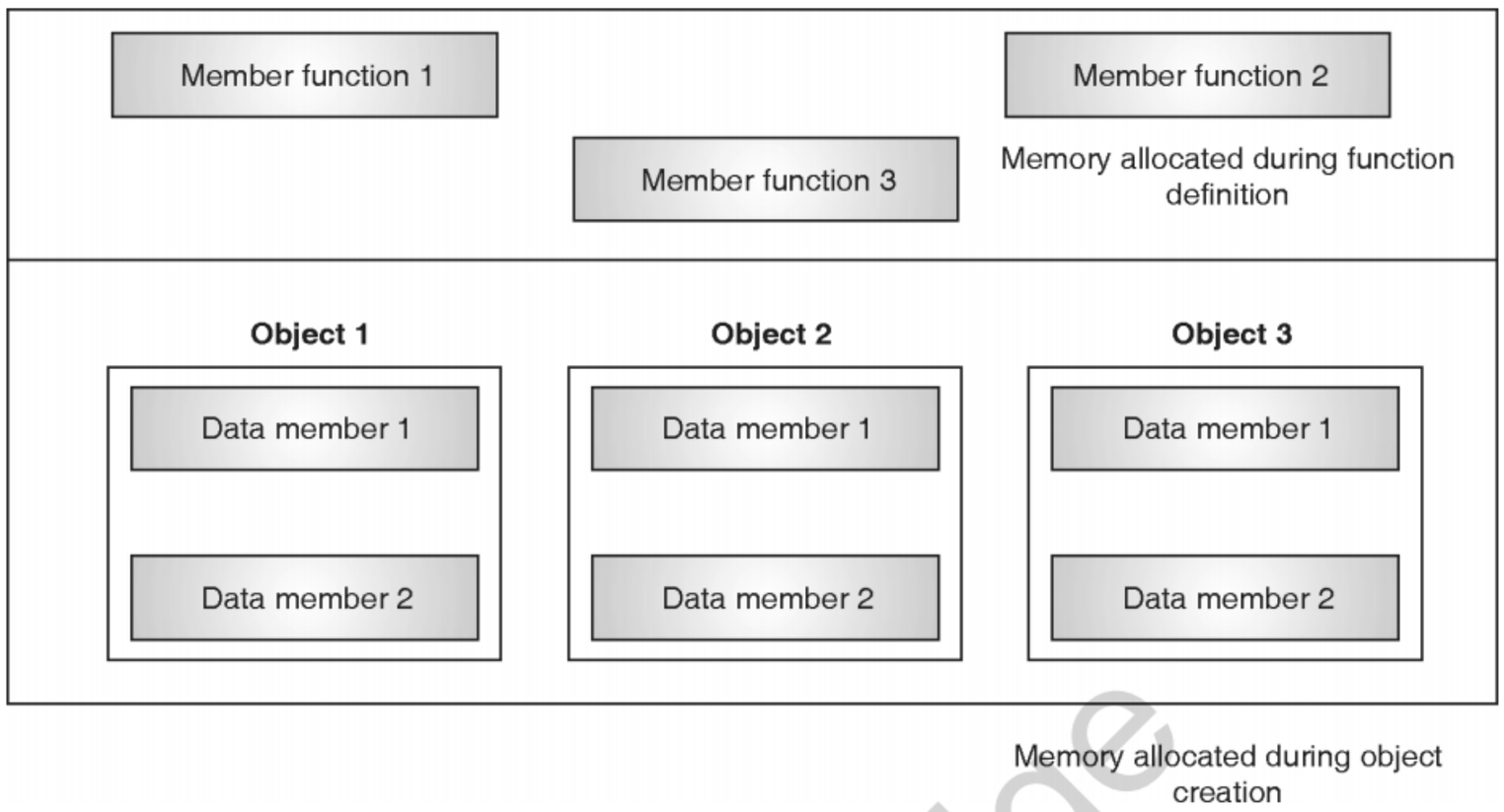

Fig. 1.28.1 : Memory requirement for object

1.29 Static Members (Variable and Functions)

Q. What is static member function? Give an example for the same.

SPPU - May 14, 4 Marks, Dec. 19, 3 Marks

– Unlike static data members we have static member function also.

– Static member functions can access only static members.

– They are called using the class name and not by the object name.

– Function should be declared with static keyword.

– We can call static member functions even if object of the class does not exists.

– They are mostly used to check whether some object of the class have created or not.

➢ **Program 1.29.1**

```
#include<iostream>
 using namespace std;
 class static_function
{
  static int count;
  int a;
  public:
  void fun()
  {
    a=10;
    cout<<"\nThis is normal function "<<a;
  }
  static void fun1()
  {
    cout<<"\n This is static member function";
    cout<<"\n It can access only static member";
    cout<<"\n Initial value of static variable is
            :"<<count;
  }
};

int static_function:: count;

int main()
{
    cout<<"\n Program for static member function";
    static_function a;
    a.fun();
    static_function:: fun1();
                          //function called without object
    return 0;
}
```

Techknowledge Publications

Output

```
Program for static member function
This is normal function  10
This is static member function
It can access only static member
Initial value of static variable is :0
```

1.30 Static Data Members

Q. Define the term Static data members.

SPPU - May 14, Dec. 15, 2 Marks

- Static variables have allocated static memory.
- It means such type of variables can be used in the lifetime of program.
- When we load the program, static members are initialized. They are initialized to 0 when we load the program.
- Only one copy of variable is created in the entire program.
- They are mainly used to maintain the state of program.
- In C++ static data members are part of the class.
- Even though they are part of class, only one copy of static member will be created.
- If we create 10 object of class which is having static member, only one copy of static data member will be created and not 10 copies.
- If we have normal member in the class and if we create 10 object of such class than 10 copies of variables will be created.
- We can declare the variable inside the class using following syntax.

```
static int a;
```

- Once declared we need to define the static member outside the class.

```
int class_name:: variable_name;
```

- Programs 1.30.1 and 1.30.2 explain the concept of static data members.

➢ **Program 1.30.1 : Without static member**

```
#include<iostream>
 using namespace std;
 class static_member
{
  int count=0;
  public:
   void increment()
  {
     count++;
     cout<<"\n  Incremented  value  of  count  is
              "<<count;
  }
};
int main()
{
  cout<<"\n Program for static member";
  static_member a;
  a.increment();
  static_member b;
  b.increment();
  static_member c;
  c.increment();
  static_member d;
  d.increment();
  return 0;
}
```

Output

```
Program for static member
Incremented value of count is   1
Incremented value of count is   1
Incremented value of count is   1
Incremented value of count is   1
```

➢ **Program 1.30.2 : With static member**

```
#include<iostream>
 using namespace std;
 class static_member
{
  static int count;
  public:
```

TechKnowledge Publications

```
    void increment()
  {
     count++;
     cout<<"\n Incremented value of count is
          "<<count;
  }
};
int static_member :: count;
int main()
{
   cout<<"\n Program for static member";
   static_member a;
   a.increment();
   static_member b;
   b.increment();
   static_member c;
   c.increment();
   static_member d;
   d.increment();
   return 0;
}
```

Output

```
Program for static member
Incremented value of count is  1
Incremented value of count is  2
Incremented value of count is  3
Incremented value of count is  4
```

Explanation : *It means only one copy of variable is created. The same copy is used by all the object of that class.*

1.31 Inline Functions

Q. What are inline functions ? What are their advantages ? Give example.

SPPU - May 14, 2 Marks, Dec. 18, 3 Marks, May 19, 6 Marks, Dec. 19, 4 Marks

- Functions are used to store the frequently used instructions.
- It is also to modularize the program.
- Function saves the memory space of the program.
- Whenever function is called, instruction pointer is jumped to definition of the function. After complete execution of function, instruction pointer come back to the instruction from where it made a jump.

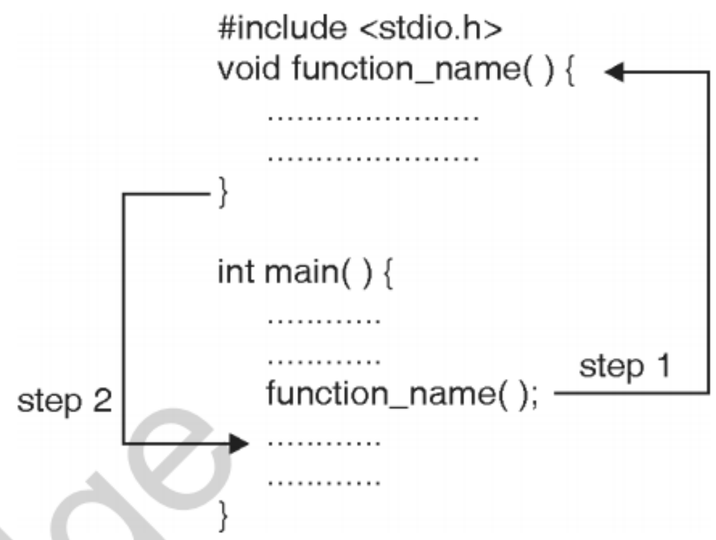

Fig. 1.31.1 : Function Execution

- Program execution with function is shown in Fig. 1.31.1.
- Program execution with function incurs extra overhead to jump and come back to previous position. This problem can be solved using the concept of inline functions.
- In inline function, function call is replaced by actual program code.
- It does not jump to the block of code.
- Generally they are recommended for small computation.
- It is more like normal function with inline keyword at the time of declaration of function.
- Syntax for creating inline functions is as follows :

```
inline  function_prototype()
{
      //Definition of function
}
```

➢ **Program 1.31.1 : Inline program is given below.**

```
#include<iostream>
#include<string>
using namespace std;
inline void display()
{
```

Techknowledge Publications

```
   cout<<"\n This is inline function";
}
inline int add(int a, int b)
{
   return (a+b);
}
int main()
{
   cout<<"\n This program is for inline function";
   display();
   int a=add(10,20);
   cout<<"\n Addition of number is :"<<a;
   return 0;
}
```

Output

```
This program is for inline function
This is inline function
Addition of number is :30
```

➢ **Program 1.31.2 : Define class number which has inline functions mult() and cube() for calculating the multiplication of 2 double numbers given and cube of the number given.**

SPPU - Dec. 16, 4 Marks

```
#include<iostream.h>
#include<conio.h>
class number
{
  public:
     inline float mult(float x,float y)
     {
        return(x*y);
     }
     inline float cube(float x)
     {
     return(x*x*x);
     }
};
void main()
{
     number obj;
     float val1,val2;
     clrscr();
     cout<<"Enter two values: ";
     cin>>val1>>val2;
     cout<<"\nMultiplication value is :
        "<<obj.mult(val1,val2);
     cout<<"\n\nCube value is :
        "<<obj.cube(val1)<<"\t"<<obj.cube(val2);
     getch();
}
```

– Conditions to use inline functions are as follows :

1. They should be small for better efficiency.
2. It increases the efficiency but all the functions should not be inline.
3. Inline functions does not work where switch and goto exists.
4. Inline functions does not use static variables.
5. It cannot be recursive.

Advantages of inline function

1. It does not require function calling overhead.
2. It also save overhead of variables push/pop on the stack, while function calling.
3. It also save overhead of return call from a function.
4. It increases locality of reference by utilizing instruction cache.
5. After in-lining compiler can also apply intra-procedural optimization, if specified. In this way compiler can now focus on dead code elimination and can give more stress on branch prediction, induction variable elimination etc.

1.31.1 Call by Value and Call by Reference

– Usually parameter to functions are passed by value.

– Called function create copy of variables and copy the value passed into it.

– Function does not have access to original value. This is acceptable until we need to modify the value of variable.

– If we want to modify the value of variable then call by value is not sufficient.

Tech Knowledge Publications

– In this situation we can use call by reference. In call by reference, we pass the address of variable instead value. So any changes in the variable will be reflected.

– Programs 1.31.3 and 1.31.4 illustrate use of call by value and call by reference concept.

➢ **Program 1.31.3 (call by value) :**

```
#include<iostream>
using namespace std;
void swap (int a,int b);
int main()
{
   int a=10,b=20;
   cout<<"\n Original value of a and b are :
         "<<a<<" "<<b;
   swap(a,b);
   cout<<"\n Modified value of a and b are :
         "<<a<<" "<<b;
   return 0;
}
void swap(int c, int d)
{
   int temp;
   temp=c;
   c=d;
   d=temp;
}
```

Output

```
Original value of a and b are : 10  20
Modified value of a and b are : 10  20
```

Explanation : *We can observe, even though value of a and b are swapped but not reflected in main program. It is because we are passing value of variable and not address.*

➢ **Program 1.31.4 : (call by reference)**

```
#include<iostream>
using namespace std;
void swap(int &a, int &b);
int main()
{
   int a=10,b=20;
   cout<<"\n Original value of a and b are :
         "<<a<<" "<<b;

   swap(a,b);

   cout<<"\n Modified value of a and b are :
         "<<a<<" "<<b;
   return 0;
}
void swap(int &c, int &d)
{
   int temp;
   temp=c;
   c=d;
   d=temp;
}
```

Output

```
Original value of a and b are : 10  20
Modified value of a and b are : 20  10
```

Explanation : *We can observe, values of a and b has been changed after execution of swap function*

1.32 Friend Functions

Q. What is a friend function ? How do you declare one ? When is friend function compulsory ? Give an example.

SPPU - May 14, 6 Marks, Dec. 16, 2 Marks

Q. Explain friend function with example.

SPPU - May 17, May 19, 4 Marks

Q. What are friend functions.

SPPU - May 18, Dec. 18, 3 Marks

Friend function

– It is functions which are not part of the class but it can access the private and protected members of the class.

– They have special privilege to access the members of the class but not part of the class.

Techknowledge Publication

– The prototype is same as that of normal function with friend keyword at the start of the function.

```
friend  function_prototype
```

– They should be declared inside the class.

– Definition of friend function is outside the class scope.

➢ **Program 1.32.1**

```
#include<iostream>
using namespace std;
class friend_class
{
   private:
int x;
   public:
friend_class()
      {
         x=0;
      }
      friend int myfunction(friend_class);
};
int myfunction(friend_class c)
{
c.x=500;
   return c.x;
}
int main()
{
cout<<"\n Friend function program";
friend_class f;
cout<<"\n Value of variable is : "<<myfunction(f);
   return 0;
}
```

Output

```
Friend function program
Value of variable is : 500
```

– We can deal with different object using friend functions.

– The example is given in Program 1.32.2.

➢ **Program 1.32.2**

```
#include<iostream>
using namespace std;
class xyz;
class abc
{
   private:
int a;
   public:
abc()
      {
         a=100;
      }
      friend int myfunction(abc, xyz);
};
class xyz
{
   private:
int b;
   public:
xyz()
      {
         b=100;
      }
      friend int myfunction(abc, xyz);
};
int myfunction(abc obj1, xyz obj2)
{
int c=obj1.a+obj2.b;
   return c;
}
int main()
{
cout<<"\n Friend function program";
abc obj1;
   xyz obj2;
cout<<"\n  Value  of  variable  is  :  "<<my
        function(obj1, obj2);
   return 0;
}
```

Techknowledge Publication

Output

```
Friend function program
Value of variable is : 200
```

1.32.1 Advantages and Disadvantages of Friend Functions

Q. Explain the term with respect to object oriented programming : Merits and demerits of friend function. **SPPU - May 15, 2 Marks**

Merits and demerits of friend function

1. Provides additional functionality which is kept outside the class.
2. Provides functions that need data which is not normally used by the class.
3. Allows sharing private class information by a non member function.
4. It is used to create many input/output functions.
5. Friend function is used for operator overloading.
6. It is useful when members of a class are interrelated to other sections of the program.
7. It is used to create effective code.
8. It doesn't have a storage class specifier.
9. A derived class can't inherit friend functions.

Important Points

- Procedure oriented programming
- Object oriented programming
- Need of OOP
- Benefits of OOP
- Applications of OOP
- Object oriented paradigm
- Building blocks of C++
- Data types
- Enumeration
- Array and string
- Reference variables
- Scope resolution operator
- Functions
- Access and utility functions
- Inline functions
- Class
- Object
- Access control
- Object and memory requirement
- Static class member
- Constructor
- Destructor
- Copy constructor

Review Questions

Q. 1 What is procedure oriented programming? State its limitations.

Q. 2 What is basic concept of object oriented programming?

Q. 3 Why there is need of OOP?

Q. 4 What are benefits of OOP?

Q. 5 What are different applications of OOP?

Q. 6 What is difference between procedure oriented and object oriented programming?

Q. 7 What is object oriented language and object based languages?

Q. 8 What are different object oriented paradigm?

Q. 9 Explain use of const keyword.

Q. 10 What are different data types in C++?

Q. 11 What are user defined data types in C++?

Q. 12 Explain enumeration with program?

Q. 13 Write program to understand array and strings?

Q. 14 Write program illustrate use of reference variables?

Q. 15 Write program illustrate use of scope resolution operator?

Technknowledge Publication

Q. 16 Explain function prototype?

Q. 17 How to handle default arguments in functions?

Q. 18 Write note on access and utility functions?

Q. 19 Write program to illustrate the use of inline functions.

Q. 20 Explain call by value and call by reference with example.

Q. 21 What is difference between classes and structure?

Q. 22 Explain class and object?

Q. 23 How access control works in C++?

Q. 24 What is private member functions?

Q. 25 How to pass object as parameter to functions?

Q. 26 How to return object from functions?

Q. 27 Write program to create array of objects?

Q. 28 Explain object and memory requirement?

Q. 29 What is static class members?

Q. 30 How to create constructor and destructor in C++?

Q. 31 How to deal with copy constructor?

❑❑❑

Techknowledge Publication

CHAPTER 2

Unit - II

Inheritance

Syllabus

Inheritance : Base class and derived class, Protected members, Relationship between base class and derived class, Constructor and destructor in derived class, Overriding member functions, Class hierarchies, Public and Private inheritance, Types of inheritance, Ambiguity in multiple inheritance, Virtual base class, Abstract class, Friend class, Nested class.

2.1 Inheritance

Q. Discuss the ways in which inheritance promotes software reuse, saves time during program development and helps prevent errors.

SPPU - Dec. 17, 5 Marks

Q. What is the difference between Inheritance and Containment ? Write a program in C++ to defend your answer. **SPPU - May 18, 6 Marks**

- Object oriented programming having different unique and important features like classes, object, inheritance, polymorphism, etc.
- These all are very important features and playing crucial role in success of object oriented programming.
- Inheritance is one of the important concepts in C++.
- Inheritance is nothing but creating the more special class of the existing class.
- Inheritance provides the reusability in program code.
- If programming provides reusability then programming efforts are going to decrease and cost of development is also reduces.
- Speed of development is going to increase with reusability which is provided by inheritance.
- If we consider the example of animal. We can create animal as base class. Animal class can be further specialized like dog class, cat class, etc.

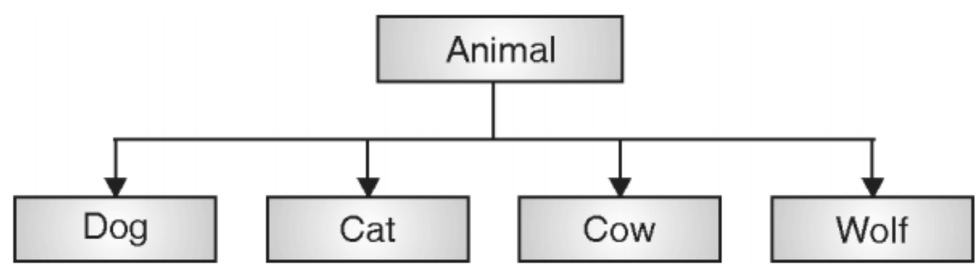

Fig. 2.1.1 : Inheritance example

- In the Fig. 2.1.1, animal class has some functions and data members. Derived classes like dog, cat, cow, wolf, etc inherits all the functionalities of base class. They can add their extra facilities also. Like dog can add type of food he eats, cat also can add type of food needed, cow also add type of food she eats, etc. They can also add the extra member like type of shelter they needed.
- Another example would be bank account class. It can be further inherited to saving account, current account, fixed deposit account, salary account, etc.

Technknowledge Publication

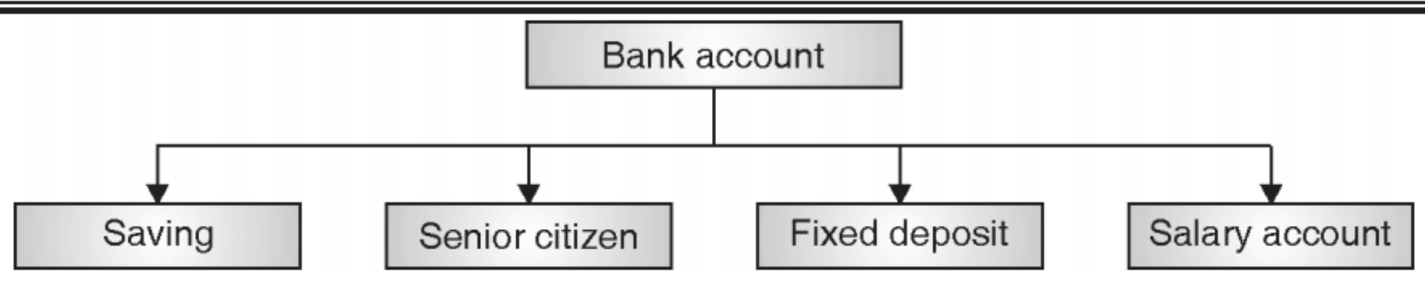

Fig. 2.1.2 : Inheritance example

- In the Fig. 2.1.2, bank account would be the general class. They have some common facilities like account number, name, address, phone, etc. Derived class gets all the facilities of base class. They can also add their own extra facilities. Saving account can add the minimum balance need to be maintained. In senior citizen account, we can add interest rate offered for senior citizens, etc.
- During the inheritance, we can use all the base class facility in derived class as well as we can add more facility in derived class also.
- A derived class inherits all base class methods with the following exceptions :
  - Constructors, destructors and copy constructors of the base class.
  - Overloaded operators of the base class.
  - The friend functions of the base class.

Following are the ways to promote inheritance :

1. Software reuse

A new class extends an existing class and uses its code. Code in the super class is immediately available to the new subclass extending it.

2. Time saving

Software re-use improves the software-development process by shortening development time.

3. Prevention of errors

- Declaring a subclass does not affect its super class's source code. Inheritance preserves the integrity of the super class.
- Implement a subclass of Date called DateTime. Assume it has instance variables hour (type int), minute (type int) and second (type int).

2.1.1 Containership

Q. What is the difference between Inheritance and Containment ? Write a program in C++ to defend your answer.

SPPU - May 18, 6 Marks

In simple terms, the Containership means a class been called in another class.

For example :

```
Class A
{.......} obj1;
Class B
{
A obj1;
}
```

Tech Knowledge Publications

– Here we see that an object of class A is called inside class B so class B contains obj1 of class A, the obj1 act as a (user defined) variable for class B, this is referred to as containership. While inheritance is class deriving properties from another class.

For example :

```
Class A
{.....}
Class B: public A
{....}
```

– Here 'public' is access specifier and allow all the public and protected member of A to act as they are written in B. Inheritance allows to connect two classes so that if there is need to update data, one can do this easily by using inheritance so in all derived classes the data will be updated by updating a few lines.

2.1.2 Base Class and Derived Class

– In inheritance, we use the concept of base and derived class.

– Base class is the class which is parent of all class. All the classes are derived from base class.

– Derived class is inherited from the base class. It is also called as child class.

– Derived class may create another class. That time derived class becomes bases class and newly created class would be the derived class.

– Derived class can be from more than one base class also.

– Fig. 2.1.3 illustrates the base and derived class concept.

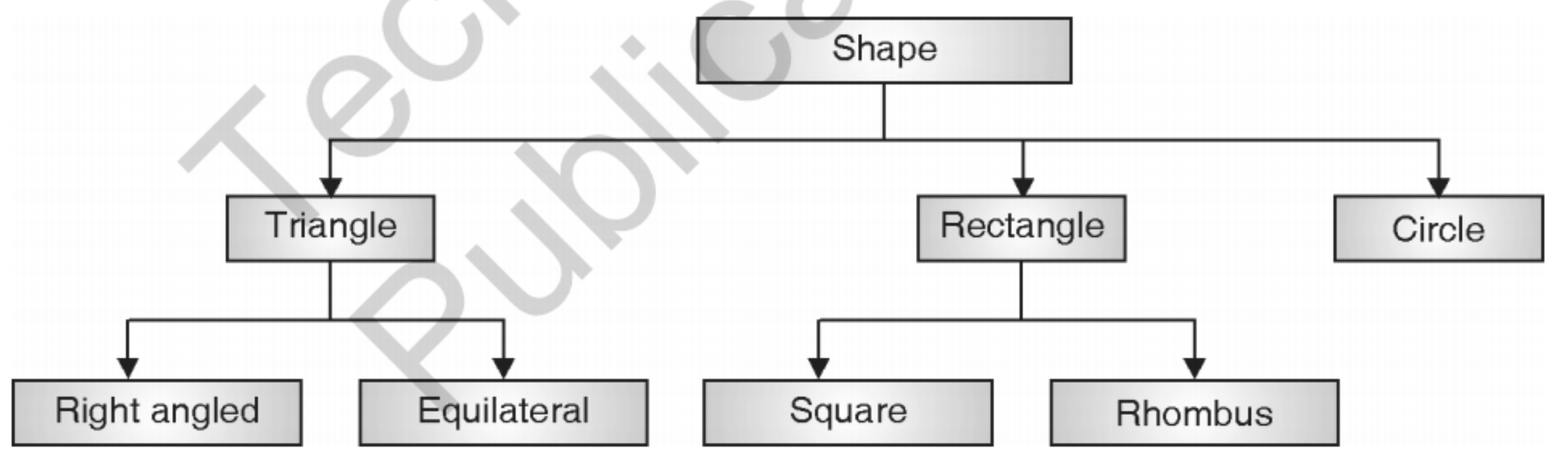

Fig. 2.1.3 : Base and derived class

– In Fig. 2.1.3, shape is base class and triangle, rectangle, circle are derived class.

– Triangle is base class whereas right angled triangle and equilateral are derived classes.

– Rectangle is base class whereas square and rhombus are derived classes.

2.1.3 Defining Inheritance

– Inheritance can be defined by creating base and derived class.

– Base class will be normal function having data member and member functions.

– Derived class will be another class but we need to mention relationship with base class.

– Derived class have their own variables and functions as well as it shares members from base class also.

– Syntax for creating inheritance is as follows.

```
class child_class_name: visibility base_class_Name{
public:
//members
Private:
//members
```

Techknowledge Publications

```
//protected
//members
};
```

– Program 2.1.1 illustrates the concept of inheritance.

➢ **Program 2.1.1**

```
#include<iostream>
using namespace std;
class base{
int a;
   public:
int b;
      void displayB(){
cout<<"\n This is base class";
      }
};
class derived:public base{
int c;
   public:
      void displayD(){
cout<<"\n This is derived class";
      }
};
int main(){
   base b;
   derived d;
d.displayB();   //derived object access base function
d.displayD();   //derived object access own function
b.displayB();   //base object access own function
  //b.displayD();   //error not allowed
   return 0;
}
```

Output

```
This is base class
This is derived class
This is base class
```

2.2 Protected Members

– Protected members from base class can be accessed by own class and its all subclasses.

– Protected members are not accessible other than own class and derived class.

– They are created solely for inheritance.

– They are combination of private and public access control.

– They are private to own class and public to derived class.

➢ **Program 2.2.1 : Illustrate the use of protected members.**

```
#include<iostream>
using namespace std;
class parent
{
protected:
int c;
public:
   void display(){
      c=100;
cout<<"\n This is base class"<<c;
   }

};
class child : public parent
{
public:
   void disp()
   {
      c=200;
cout<<"\n This is child class"<<c;
   }
};
int main( )
{
  child obj;
obj.disp();
        //calls derived class functions can access
        //protected variable c
obj.display();
        //calls base class function, already have access
        //to variable c
```

Techknowledge Publication

```
  //obj.c=100;
        //error cannot access protected member using
        //object
  return 0;
}
```

Output

```
This is child class200
This is base class100
```

2.3 Relationship between Base Class and Derived Class

- Consider inheritance hierarchy containing types of Bank accounts in a bank. Consider that there are two types of bank accounts (classes) :
  - Base class Account
  - Derived class SavingAccount
- SavingAccount class derived from Base class as a public inheritance.
- This relationship between Base class and Derived class can be exhibited from the following Program 2.3.1 :

➢ **Program 2.3.1**

```
#include<iostream>
#include<vector>
class Account
{
public:
   virtual void PrintBalance()
   {
      std::cout<<"Base"<<std::endl;
   }
};
class SavingAccount : public Account
{
public:
  void PrintBalance()
  {
     std::cout<<"SavingAccount"<<std::endl;
  }
};
int main()
{
   1)Account* p = new SavingAccount();
   p->PrintBalance();
   2)Account* p1 = new Account();
   p1->PrintBalance();
   3)SavingAccount* p3 = new Account();
   p3->PrintBalance();
   4)SavingAccount* p4 = new SavingAccount();
   p4->PrintBalance();
   return 0;
};
```

2.4 Constructor and Destructor in Derived Class

- Constructor is important part in object oriented programming.
- Base and derived class can have their own constructor.
- Constructor is used to construct the object whereas destructor is used to destroy the object.
- Order of calling constructor /destructor is in following sequence.
  - First base class constructor is called.
  - Next derived class constructor is called (both constructors are called when derived class object is created).
  - Next destructor of derived class is called.
  - Next destructor of base class is called.
- Simple base and derived class constructor/destructor is shown in Program 2.4.1. For this program constructor with parameters are not considered instead simple constructors without arguments are considered.

➢ **Program 2.4.1**

```
#include<iostream>
using namespace std;
class parent
```

Technknowledge Publications

```
{
 public:
 parent()
 {
cout<<"\n Parent class constructor called";
 }
 ~parent()
 {
cout<<"\n Parent class destructor called";
 }
};

class child : public parent
{
 public:
 child()
 {
cout<<"\n Child class constructor called";
 }
 ~child()
 {
cout<<"\n Child class destructor called";
 }
};

int main()
{
 child a;
 return 0;
}
```

Output

```
Parent class constructor called
Child class constructor called
Child class destructor called
Parent class destructor called
```

- If base class constructor is not taking argument, then there is no need to define constructor in derived class.
- If base class is having constructor with argument, then we need to define constructor in derived class and pass argument to base class constructor.
- Generally we create object using derived class in inheritance. So it is necessary to pass the argument to base class.
- If both base and derived class contains constructor, then base constructor is executed first then derived class constructor.
- In multiple inheritance, base classes are constructed in the order in which they appear in the declaration of derived class.
- In multilevel inheritance also constructors are executed in the order of inheritance.
- Derive class take care of arguments for base class. We need to provide all necessary arguments which are required by all base classes.
- Base constructors are called and executed before executing the statement in the body of derived constructors.

➢ **Program 2.4.2**

```
#include<iostream>
using namespace std;

class A
{
   private:
int a;
   public:
      A(int i)
      {
         a = i;
cout<<"\n Class A created \n";
      }
      void dispA()
      {
cout<<"\n a = "<<a;
      }
};
class B
{
   private:
      float b;
   public:
```

TechKnowledge Publications

```
    B(float j)
    {
      b = j;
cout<<"\n class B created \n";
    }
    void dispB()
    {
cout<<"\n b = "<<b;
    }
};
class C : public B, public A
{
  private:
int n, m;
  public:
    C(int a, float b, int c, int d): A(a), B(b)
    {
      m = c;
      n = d;
cout<<"\n Class C Created \n";
    }
    void dispC()
    {
cout<<"\n m = "<<m;
cout<<"\n n = "<<n;
    }
};
int main()
{
  C c(25, 27.65, 330, 4100);
cout<<"\n";
c.dispA();
c.dispB();
c.dispC();

  return 0;
}
```

Output

```
class B created

Class A created

Class C Created
a = 25
b = 27.65
m = 330
n = 4100
```

2.5 Overriding Member Functions

- Overloading means use of same thing for different purposes.
- It is one of the main features of object oriented programming.
- Overloading can be of different types :
  1. Function overloading
  2. Operator overloading
- In this section we will see the function overloading.
- It refers to using the same function name for different purposes.
- At the time of creating such functions we need to keep in mind that name of function should be same.
- Same functions name should take care of the following points.
  - Must have different data types of passing parameters.
  - Must have different return types.
  - Number of argument should be different.
- Make different combination using above conditions.
- These are some valid prototypes for function overloading :
  - void function()
  - int function(int a)
  - int function(int a, int b)
  - int function(int a, int b, int c)
  - float function(int a, float f)
- Example of function overloading is given in Program 2.5.1.

Tech Knowledge Publications

➢ **Program 2.5.1**

```
#include<iostream>
using namespace std;
void function()
{
cout<<"\n return void, no parameter";
}
int function(int a)
{
cout<<"\n return int, one parameter";
      return a;
}
int function(int a, int b)
{
cout<<"\n return int, two parameter";
      return a+b;
}
int function(int a, int b, int c)
{
cout<<"\n return int, three parameter";
      return a+b+c;
}
float function(int a,  float f)
{
cout<<"\n return float, int and float parameter";
      return a+f;
}
int main()
{
function();
int a=function(100);
cout<<"\n Return value is:"<<a;
   a=function(100,200);
cout<<"\n Return value is:"<<a;
   a=function(100,200,300);
cout<<"\n Return value is:"<<a;
   float f=function(100,200,300.5);
cout<<"\n Return value is:"<<f;
  return 0;
}
```

2.6 Aggregation : Classes within Classes

Aggregation and inheritance and class relationships

- If class B is derived from class A then we say B is kind of A.
- It means B has all characteristics of A and their own characteristics.
- Aggregation is called has a relationship.
- In object oriented programming, aggregation occurs when one object is attribute of another class.

```
Class Parent
{
};
Class Child{
Parent p;          //it defines object of parent class
};
```

- C++ can have nested class also. It means class is declared inside another class.
- Nested class is member of outer class so it has same access right like other members of class.
- Member of outer class has no special access to member of nested class.
- Program 2.6.1 illustrates the concept of nested class.

➢ **Program 2.6.1**

```
#include<iostream>
using namespace std;
class outer
{
   public:
   class inner
   {
      private:
int sum;
      public:
         void addition(int x, int y)
         {
```

Techknowledge Publication

```
            sum=x+y;
        }
        void display( )
        {
cout<<"\nSum of x and y is:: " << sum;
        }
    };
};

int main()
{
outer::inner  a;        //object creation
a.addition(112, 110);
a.display();
return 0;
}
```

Output

```
Sum of x and y is:: 222
```

2.7 Class Hierarchies

- It is nothing but ordering of abstraction classes.
- We have two hierarchical schemes in object oriented programming.
- One is "is part of" which means some object is part of larger object than it. Car engine is part of car.
- Another is "is a" scheme which is nothing but inheritance.
- Inheritance allows to reuse the software code.
- Inheritance allows to create more specialized class from existing class.
- In inheritance one class is said to be kind of other class.
- While traversing from top to bottom in inheritance hierarchy, we go to more specialized class from generalized class.
- More specialization is possible by adding more facility in derived classes.
- Derived class inherit all attributes of base class.
- Object does not inherit anything.

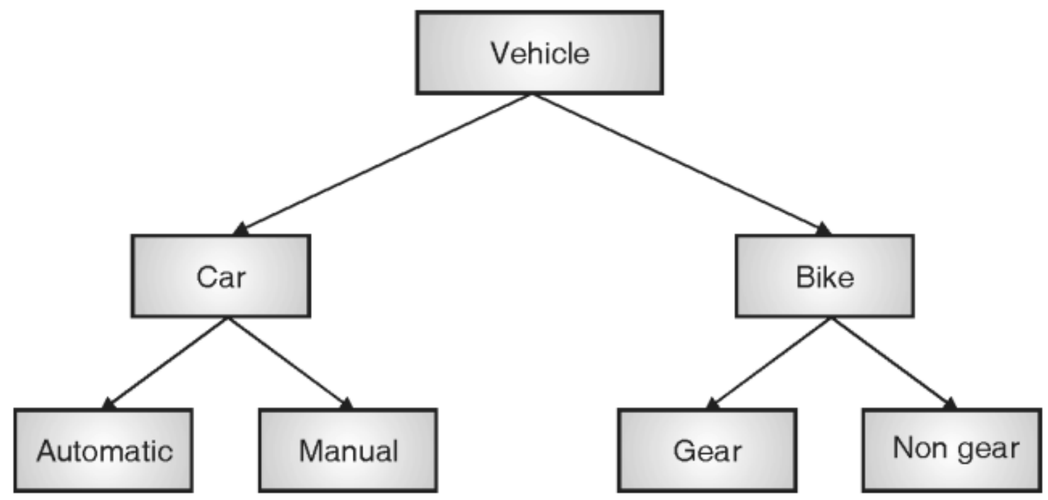

Fig. 2.7.1 : Class hierarchies

2.8 Public and Private Inheritance

- Upto this point we have seen class is inherited publicly. It means derived class can access the public members of base class.
- There is another way to derive a class that is private inheritance.
- Public derivation of class is as follows :

```
Class derived: public base;
```

- Public keyword in the above statement signifies, all public members of base class are accessible in the derived class.
- Private derivation of class is as follows :

```
Class derived: private base;
```

- Private keyword in above statement signifies, no member of base class will be accessible.
- Programs 2.8.1 and 2.8.2 illustrates the public and private derivation.

2.8.1 Public Inheritance

➢ **Program 2.8.1**

```
#include<iostream>
using namespace std;
class A
{
   private:
int b;
   public:
int a;
```

Tech Knowledge Publications

```
   void display()
   {
cout<<"\n This is Base class";
cout<<"\n Value of a is "<<a;
   }
};
class B: public A
{
   public:
int c;
   void disp()
   {
cout<<"\n This is derived class";
cout<<"\n Value of a is:"<<a;
cout<<"\n Value of c is:"<<c;
   }
};
int main()
{
   B obj;
obj.a=100;      //base class member accessible
   //obj.b=300;
                //not accessible as private member
obj.c=200;
obj.display();      //access base class function as public
   //obj.b=300;      //will not work as private member
obj.disp();
   return 0;
}
```

Output

```
This is Base class
Value of a is 100
This is derived class
Value of a is:100
Value of c is:200
```

Explanation : *As we can see all public member of base class are accessible because we have derived class publicly.*

2.8.2 Private Inheritance

➢ **Program 2.8.2**

```
#include<iostream>
using namespace std;
class A
{
   private:
int b;
   public:
int a;
   void display()
   {
cout<<"\n This is Base class";
cout<<"\n Value of a is "<<a;
   }
};
class B: private A
{
   public:
int c;
   void disp()
   {
      a=100;
cout<<"\n This is derived class";
cout<<"\n Value of a is:"<<a;
cout<<"\n Value of c is:"<<c;
   }
};
int main()
{
   B obj;
   //obj.a=100;      //not accessible
   //obj.b=300;      //not accessible as private member
obj.c=200;
   //obj.display();    //display not accessible
obj.disp();
   return 0;
}
```

Output

```
This is derived class
Value of a is:100
Value of c is:200
```

Explanation : *As shown in Program 2.8.2 no member of base class is accessible through derived class object as class is derived privately.*

Techknowledge Publications

2.8.3 Member Access Control

Q. Explain why and when do we use protected instead of private? **SPPU - Dec.15, 4 Marks**

Q. Explain visibility modes in Inheritance. **SPPU - Dec.19, 2 Marks**

- C++ provides three access control public, private and protected.
- Public member can be access by own class, derived class and outside class also.
- Private members of the class can be accessible only through member functions of the class. They cannot be accessed by derived class or from outside the class.
- Protected members are more like private members. But difference is that they can be accessed by derived class member functions also.
- Different access types are given in Table 2.8.1.

Table 2.8.1

Access	Public	Protected	Private
Same class	Yes	Yes	Yes
Derived classes	Yes	Yes	No
Outside classes	Yes	No	No

- Table 2.8.1, signifies, public members can be accessed from same class, derived class and from outside class also.
- Table 2.8.2 signifies how class members are inherited to derived class.

Table 2.8.2

Base Class Visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

2.9 Types of Inheritance

Q. With suitable example, explain different types of inheritance. **SPPU - Dec. 15, 4 Marks**

Q. Write a short note on types of inheritance with respect to : (i) Single (ii) Multiple (iii) Hierarchical inheritance. **SPPU - May 16, Dec. 19, 6 Marks**

- There are different types of inheritance, which type of inheritance to use depends upon the requirement and it is developers choice.
- Different types of inheritance are as follows :

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

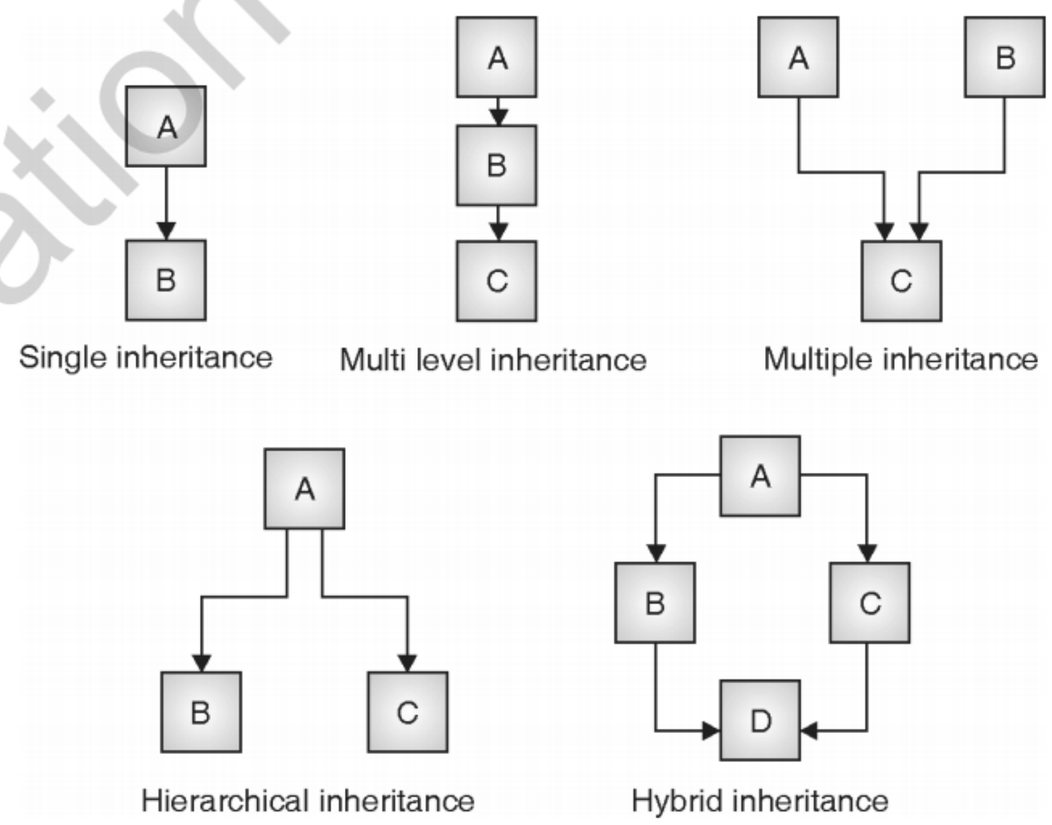

Fig. 2.9.1 : Types of inheritance

1. **Single inheritance :** Single inheritance have only one base class and one derive class.
2. **Multilevel inheritance :** Multilevel inheritance have one base and one derive class. Derive class can create another derived class and so on.
3. **Multiple inheritance :** In this case, class is derived from multiple base classes.
4. **Hierarchical inheritance :** In this case, one base class creates many subclasses.

Tech Knowledge Publications

5. **Hybrid inheritance :** It is nothing but the combination of above four inheritances.

2.9.1 Single Inheritance

- In single inheritance, there is only one base class and one derived class.
- Derived class inherits the base class members also.
- Single inheritance is shown in Fig. 2.9.2.

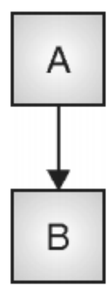

Fig. 2.9.2 : Single inheritance

- In C++ inheritance can be created using following syntax :

```
class A
{
      public:
      class members
};
class B: public/private/protected A
{
     public:
     class members
};
```

- In the above example, A is name of base class. B is name of derived class.
- Derived class can have access to all the members of subclass as well as public and private members of base class.

➢ **Program 2.9.1 : Illustrates the concept of single inheritance.**

```
#include<iostream>
using namespace std;
class A
{
   private:
int b;
   public:
int a;
   void display()
   {
cout<<"\n This is Base class";
cout<<"\n Value of a is "<<a;
   }
};
class B: public A
{
   public:
   void disp()
   {
cout<<"\n This is derived class";
cout<<"\n Value of a is:"<<a;
   }
};
int main()
{
cout<<"\n Single inheritance example";
   A object1;
   object1.a=10;
   object1.display();
   B object2;
   object2.a=20;
   object2.disp();
   object2.display();
    //derive class object can access base and derive
    //class members
    //we cannot access b as it is private in base class
//object2.b=30;   //will not work
   return 0;
}
```

Output

```
Single inheritance example
This is Base class
Value of a is 10
This is derived class
Value of a is:20
This is Base class
Value of a is 20
```

Techknowledge Publication

- In Program 2.9.1, Base class is A having two public members a and display().
- D is derived class having public member disp().
- Using derived class, we can access derived class as well as base class members.
- It means derive class can access disp(), display() and a variable also.
- We can make private member to be accessed by derived which is possible by making this type of member protected.

➢ **Program 2.9.2**

```
#include<iostream>
using namespace std;
class A
{
   protected:
int b;
   public:
int a;
   void setdata(int temp1, int temp2)
   {
      a=temp1;
      b=temp2;
   }
   void display()
   {
cout<<"\n This is Base class";
cout<<"\n Value of a is "<<a;
cout<<"\n Value of b is "<<b;
   }
};
class B: public A
{
   public:
   void disp()
   {
cout<<"\n This is derived class";
cout<<"\n Value of a is:"<<a;
cout<<"\n Value of b is "<<b;
   }
};
int main()
{
cout<<"\n Single inheritance with protected member
        example";
   A object1;
   object1.setdata(10,20);
   object1.display();
   B object2;
   object2.setdata(30,40);
   object2.disp();
   object2.display();
   //derive class object can access base and derive
   //class members
   //we can access b as it is protected in base class
   return 0;
}
```

Output

```
Single inheritance with protected member example
This is Base class
Value of a is 10
Value of b is 20
This is derived class
Value of a is:30
Value of b is 40
This is Base class
Value of a is 30
Value of b is 40
```

2.9.2 Multiple Inheritance

Q. What is Multiple Inheritance? Give suitable example to demonstrate Multiple inheritance.

SPPU - May 19, 4 Marks

- In multiple inheritance, class can be derived from more than one base class.
- In multiple inheritance, derived class can access the members of both the bases classes.
- Sometimes, it becomes complicated as we are getting the member from both the classes. Problem arises when both the bases class contains the data having same name.

Techknowledge Publication

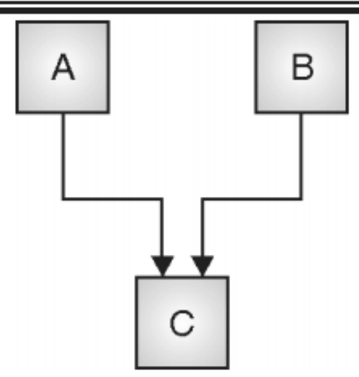

Fig. 2.9.3 : Multiple inheritance

- It creates the ambiguity in multiple inheritance.
- In Fig. 2.9.3, derive class C can access its own members.
- Class C can also access the public and protected members of class A and Class B.
- Program 2.9.3 illustrates the concept of multiple inheritance.

➢ **Program 2.9.3**

```
#include<iostream>
using namespace std;
class A
{
   protected:
int a;
   public:
int b;
   void setdata_A(int temp1, int temp2)
   {
      a=temp1;
      b=temp2;
   }
   void displayA()
   {
cout<<"\n This is first Base class";
cout<<"\n Value of a is "<<a;
cout<<"\n Value of b is "<<b;
   }
};
class B
{
   protected:
int c;
   public:
int d;
   void setdata_B(int temp1, int temp2)
   {
      c=temp1;
      d=temp2;
   }
   void displayB()
   {
cout<<"\n This is second Base class";
cout<<"\n Value of c is "<<c;
cout<<"\n Value of d is "<<d;
   }
};
class C: public A, public B
{
   public:
   void display_derived()
   {
cout<<"\n This is derived class inherited from A and
        B";
cout<<"\n This class can access public and protected
        members of A and B";
cout<<"\n Value of a is:"<<a;
cout<<"\n Value of b is "<<b;
cout<<"\n Value of c is:"<<c;
cout<<"\n Value of d is "<<d;
   }
};
int main()
{
cout<<"\n Multiple inheritance example";
   C object3;
   object3.setdata_A(50, 60);
   object3.setdata_B(70, 80);
   object3.display_derived();
   return 0;
}
```

Output

```
Multiple inheritance example
This is derived class inherited from A and B
This class can access public and protected members of
A and B
```

Technknowledge Publication

```
Value of a is:50
Value of b is 60
Value of c is:70
Value of d is 80
```

2.9.3 Hierarchical Inheritance

- Inheritance is the process of inheriting properties of objects of one class by objects of another class. The class which inherits the properties of another class is called Derived or Child or Sub class and the class whose properties are inherited is called Base or Parent or Super class.
- If more than one class is inherited from the base class, it's known as hierarchical inheritance. In hierarchical inheritance, all features that are common in child classes are included in the base class.
- For example: Physics, Chemistry, Biology are derived from Science class.

Syntax

```
class base_class {
    ... .. ...
}
class first_derived_class: public base_class {
    ... .. ...
}
class second_derived_class: public base_class {
    ... .. ...
}
class third_derived_class: public base_class {
    ... .. ...
}
```

2.9.4 Hybrid Inheritance

- As we said earlier, hybrid inheritance is combination of single, multiple, multilevel, hierarchical inheritance.
- It faces the problem of ambiguity at the time of accessing the members from multiple base classes.
- Hybrid inheritance is shown in Fig. 2.9.4.
- In the Fig. 2.9.4, class B and C is derived from class A. Class D is derived from class B and C.

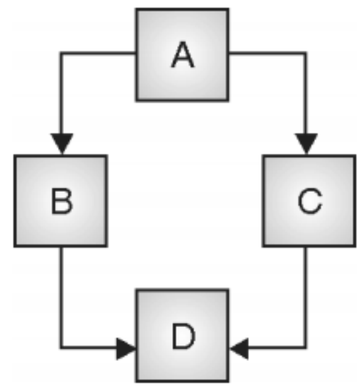

Fig. 2.9.4 : Hybrid inheritance

- Program 2.9.4 illustrates the concept of hybrid inheritance.

➢ **Program 2.9.4**

```
#include <iostream>
using namespace std;
class A
{
   public:
int a;
   void displayA()
   {
cout<<"\n This is Base class";
cout<<"\n Value of a is "<<a;
   }
};
class B: public A
{
   public:
int b;
   void displayB()
   {
cout<<"\n This class is derived from A class";
cout<<"\n Value of b is "<<b;
   }
};
class C: public A
{
   public:
```

TechKnowledge Publications

```
int c;
   void displayC()
   {
cout<<"\n This class is derived from class A";
cout<<"\n Value of c is "<<c;
   }
};
class D: public B, public C
{
   public:
int d;
   void display_derived()
   {
cout<<"\n This class is derived from A and B";
cout<<"\n This class can access public and protected
         members of B and C";
cout<<"\n Value of b is "<<b;
cout<<"\n Value of c is:"<<c;
cout<<"\n Value of d is "<<d;
   }
};
int main()
{
cout<<"\n Hybrid inheritance example";
   D object1;
   object1.b=100;
   object1.c=200;
   object1.d=300;
   object1.display_derived();
   return 0;
}
```

Output

```
Hybrid inheritance example
This class is derived from A and B
This class can access public and protected members of
B and C
Value of b is 100
Value of c is:200
Value of d is 300
```

2.10 Ambiguity in Multiple Inheritance

Q. What is the ambiguity in multiple inheritance?

SPPU - May 19, 2 Marks

- In object oriented programming, multiple inheritance faces the problem of ambiguity.
- This situation arises when multiple base class have members with same name.
- Derived class faces the ambiguity as which base class version of the member they should use.
- It is the state in which compiler confuses which version of base class member to use.
- Ambiguity can be resolved by using scope resolution operator.
- At the time of calling the member we can specify the name of the class from which we wanted to access the members.

```
object_name.BASE ::abc();
```

- The above statement access the abc() function from class BASE.
- Ambiguity is explained in the Program 2.10.1.

➢ **Program 2.10.1 : With error (Ambiguous program)**

```
#include<iostream>
using namespace std;
class A
{
   public:
   void display()
   {
cout<<"\n This is first Base class";
   }
};
class B
{
   public:
   void display()
   {
cout<<"\n This is second Base class";
```

Techknowledge Publication

```
   }
};
class C: public A, public B
{
   public:
   void display_derived()
   {
cout<<"\n This is derived class inherited from A and
         B";
   }
};
int main()
{
cout<<"\n Ambiguity in multiple inheritance";
   C object_name;
object_name.display_derived();
object_name.display();
  return 0;
}
```

Error

```
In function 'int main()': 35:17: error: request for
member 'display' is ambiguous 15:10: error:
candidates are: void B::display() 7:10: error: void
A::display()
```

Corrected program by resolving the ambiguity

➢ **Program 2.10.2**

```
#include<iostream>
using namespace std;
class A
{
   public:
   void display()
   {
cout<<"\n This is first Base class";
   }
};
class B
{
   public:
   void display()
   {
cout<<"\n This is second Base class";
   }
};
class C: public A, public B
{
   public:
   void display_derived()
   {
cout<<"\n This is derived class inherited from A and
         B";
   }
};
int main()
{
cout<<"\n Ambiguity in multiple inheritance";
   C object_name;
object_name.A::display();
object_name.B::display();
object_name.display_derived();
   return 0;
}
```

Output

```
Ambiguity in multiple inheritance
This is first Base class
This is second Base class
This is derived class inherited from A and B
```

2.11 Virtual Base Class

Q. Explain virtual base class with example.

SPPU - Dec. 16, May 17, 2 Marks

- Virtual bases class is used to remove the ambiguity in multiple inheritance.
- This situation arises when there is multiple path to access member of base class.
- Sometimes child class can get duplicate member inherited from single base class. This problem can be solved by virtual base class.

Techknowledge Publication

- If we are creating virtual base class then compiler takes care of removing the duplicate members during inheritance.
- The Program 2.11.1 explains virtual base class.

➢ **Program 2.11.1**

```
#include<iostream>
using namespace std;
class base
{
   public:
int i;
};

class derive1 : virtual public base
{
     public:
int j;
};
class derive2: virtual public base
{
      public:
int k;
};
class derive3: public derive1, public derive2
{
   public:
        int mul;
};
int main()
{
       derive3 a;
a.i = 10;
a.j = 20;
a.k = 30;
a.mul = a.i * a.j * a.k;
cout<<"Value of i is : "<<a.i<<"\n";
cout<<"Value of j is : "<<a.j<<"\n";
cout<<"Value of k is :"<<a.k<<"\n";
cout<<"Multiplication is : "<< a.mul <<"\n";
   return 0;
}
```

Output

```
Value of i is : 10
Value of j is : 20
Value of k is :30
Multiplication is : 6000
```

2.12 Abstract Class

- Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Following are some characteristics of Abstract Class :

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for upcasting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

➢ **Program 2.12.1 : Example of Abstract Class in C++**

```
//Abstract base class
class Base
{
   public:
   virtual void show() = 0;    // Pure Virtual Function
};

class Derived:public Base
{
   public:
   void show()
   {
```

Techknowledge Publication

```
    cout<<"Implementation of Virtual Function in
            Derived class\n";
  }
};

int main()
{
   Base obj;   //Compile Time Error
   Base *b;
   Derived d;
   b = &d;
   b->show();
}
```

Output

```
Implementation of Virtual Function in Derived class
```

2.13 Friend Classes

- Unlike friend function we can have friend class also.
- Friend functions are used to access the private members of class.
- Class can be declared friend of some other class also.
- Whenever, we create friend class then all member functions of friend class also becomes friend of the other class.
- Friend class can access private member of another class where it declared to become friend.
- It requires friend class should be declared first.
- Program 2.13.1 illustrates the friend class concept.

➢ **Program 2.13.1**

```
#include <iostream>
using namespace std;
class classA
{
   //declaration of friend class
   friend class friend_class;
 private:
      int a;
 public:
      classA()
      {
         a=100;
      }
      void print()
      {
            cout<<"\n This is normal class";
            cout<<"\n Value of a is "<<a;
      }
};
class  friend_class
{
   public:
      void alter( classA&obj, int x )
      {
            obj.a = x;
      }
};
int main()
{
   classA object1;
   friend_class object2;
   object1.print();
   object2.alter(object1, 200);
   cout<<"\n Altered value";
   object1.print();
   return 0;
}
```

Output

```
This is normal class
Value of a is 100
Altered value
This is normal class
Value of a is 200
```

Explanation : *If we see carefully, private member of classA can been changed from friend_class which is friend class. If friend_class is normal then it is not possible to access the private member of classA from another class.*

Technknowledge Publication

2.14 Static Class

- Static class is more like non static class only.
- One difference between static and non static class is that static class cannot be instantiated.
- We cannot use new keyword to create object with static class.
- We can access member of class using name of the class itself.
- Consider the case where name of static class is myStaticClass which contains function whose name is myMethod(). Then we can access function using following syntax :

```
myStaticClass.myMethod();
```

- Static is used for convenient containers for collection of methods. These methods don't have set or get property on internal instance of the fields.
- Famous example is Math library in .net framework wherein we can use different function directly to perform some operations. Examples are given below.

```
float var = -1.4564;
Console.WriteLine(Math.Abs(var));
Console.WriteLine(Math.Floor(var));
Console.WriteLine(Math.Round(Math.Abs(var)));
```

- Static class contains only static members.
- Static class cannot be instantiated.
- Static class are sealed.
- Static class does not contain instance constructors.

2.15 Nested Class

- A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member.
- The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.
- Program 2.15.1 demonstrates nested classes in C++ is as follows :

➢ **Program 2.15.1**

```
#include<iostream>
using namespace std;
class A {
  public:
  class B {
    private:
    int num;
    public:
    void getdata(int n) {
      num = n;
    }
    void putdata() {
      cout<<"The number is "<<num;
    }
  };
};
int main() {
  cout<<"Nested classes in C++"<< endl;
  A :: B obj;
  obj.getdata(9);
  obj.putdata();
  return 0;
}
```

Output

```
Nested classes in C++
The number is 9
```

Important Points

- Inheritance
- Levels of inheritance
- Friend class
- Static class
- Overriding member functions
- Abstract class

Technknowledge Publication

Review Questions

Q. 1 What is inheritance?

Q. 2 What do you mean by base and derived class?

Q. 3 How to define inheritance?

Q. 4 What do you mean by protected member?

Q. 5 Explain constructor and destructor in inheritance with suitable program.

Q. 6 What are different levels of inheritance?

Q. 7 Write program to implement multiple inheritance?

Q. 8 How to resolve ambiguity in multiple inheritance?

Q. 9 What is public and private inheritance?

Q. 10 Write program for friend class?

Q. 11 What is virtual base class?

Q. 12 Write program to illustrate overriding member functions?

Q. 13 What is classes within class?

Q. 14 Write program to illustrate use of abstract class?

❑❑❑

Techknowledge Publication

CHAPTER 3

Unit - II

Pointers

Syllabus

Pointers : Declaring and initializing pointers, Indirection operators, Memory Management: new and delete, Pointers to Objects, this pointer, Pointers Vs Arrays, Accessing arrays using pointers, Arrays of pointers, Function pointers, Pointers to pointers, Pointers to derived classes, Passing pointers to functions, Return pointers from functions, Null pointer, void pointer.

Technoknowledge Publication

3.1 Pointer : Indirection Operators

Q. What is Pointer? Explain using diagram and program. **SPPU - May 17, 3 Marks**

Q. What is difference between pointer and references ? Explain using program. **SPPU - Dec. 17, 4 Marks**

Q. Explain pointer to a variable. Use suitable example. **SPPU - May 18, 2 Marks**

Q. Explain various arithmetic pointer operations. **SPPU - Dec. 18, 4 Marks**

- Pointer is important concept in C and C++.
- C++ pointer is easy understand and learn.
- It offers very efficient method to handle the information.
- Data handling is very efficient because we directly deal with address of information.
- Pointer is nothing but the address only.
- Pointer is derived data type. It stores variables memory address.
- It can also refer to another pointer.
- Pointer can be declared using following syntax :

```
Data_type *pointer_variable;
```

- Pointer_variable is name of pointer in which we can store address. Data type can be any data type.
- Some tasks are very easily performed with pointers like dynamic memory allocation.
- Every variable is stored in memory and every memory cell have address associated with it.

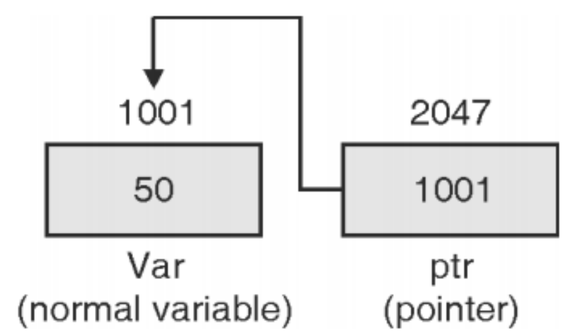

Fig. 3.1.1 : Pointer

- We can print address of variable using & operator.
- Program 3.1.1 illustrates the address printing of variable.

➢ **Program 3.1.1**

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
```

```
{
int a=10;
int b=20;
cout<<"\n Address of a is : "<<&a;
cout<<"\n Address of b is : "<<&b;
  return 0;
}
```

Output

```
Address of a is : 0x28fefc
Address of b is : 0x28fef8
```

- We must declare pointer before its use.
- After declaration, pointer initialization is also important which initialize the pointer to valid memory location. After declaration and initialization, we use pointer to print some value.
- Value can be printed using * operator with pointer variable name.

➢ **Program 3.1.2**

```
#include<iostream>
using namespace std;
int main()
{
int  a = 1000;
int  *p;
  p = &a;
cout<<"\n  Value  of  var  variable  using  variable
        name: ";
cout<< a << "\n";
cout<<"\n address of a, which is stored in p ";
cout<<p << "\n";
cout<<"\n Value of *p variable: ";
cout<<*p ;
  return 0;
}
```

Output

```
Value of var variable using variable name: 1000
address of a, which is stored in p 0x28fef8
Value of *p variable: 1000
```

- Pointer can be incremented or decremented.
- Pointer increment will point to next memory locations. Pointer decrement will point to previous memory location.
- It means integer pointer will be incremented by two memory location. Character pointer will be incremented by one memory location, and so on.
- We can add or subtract any integer from pointer.
- It does not make any sense by dividing or multiplying pointers.
- We can have array of pointers also.
- Pointer to functions
- Pointer to object
- Pointer and string
- Pointer to derived classes

3.2 Memory Management : New and Delete

Q. Compare and contrast memory allocation and deallocation using new, delete.

SPPU - May 19, 4 Marks

- New and delete operators are used in memory allocation in C++.
- New is used to allocate memory whereas delete is used to deallocate the memory.
- It is more like malloc() and free() in C.
- We can allocate the required memory using new operator.
- We can use new operator to allocate memory to object also.

Techknowledge Publication

Comparison between memory allocation and deallocation using new, delete

Table 3.2.1 : Comparison between memory allocation and deallocation using new, delete

Feature	Memory Allocation (new)	Memory Deallocation (delete)
Memory allocated	'Free Store'	'Heap'
Returns	Fully typed pointer	void*
On failure	Throws (never returns NULL)	Returns NULL
Required size	Calculated by compiler	Must be specified in bytes
Handling arrays	Has an explicit version	Requires manual calculations
Reallocating	Not handled intuitively	Simple (no copy constructor)
Call of reverse	Implementation defined	No
Overridable	Yes	No
Use of constructor / destructor	Yes	No

➢ **Program 3.2.1 : Write a program to explain new and delete operator.** **SPPU - May 18, 4 Marks**

```
#include<iostream>
#include<new>
using namespace std;
int main()
{
int count;
int * numbers;
cout<<"How many number you wanted to store";
cin>>count;
  numbers = new (nothrow) int[count];

 for (int i=0; i<count; i++)
 {
  cout<<"\n Enter the number: ";
  cin>>numbers[i];
 }
 cout<<"You have entered the following numbers: ";
  for (int j=0; j<count; j++)
  cout<<numbers[j] << "\t";
  delete[] numbers;
  return 0;
}
```

Output

```
How many number you wanted to store5
Enter the number: 1
Enter the number: 2
Enter the number: 3
Enter the number: 4
Enter the number: 5
You have entered the following numbers:
1   2   3   4   5
```

3.3 Pointers to Objects

- Pointer is used to access class members.
- Pointer can point to object also.
- Pointer can be created in normal way.

```
class example {
};
Example e;        //normal object
Example *p;
//pointer of type class that is example in this case
```

- Object pointers are useful in creating object at run time.
- Pointer is used to access elements of objects.

Techknowledge Publication

– Elements can be access using different manner.

```
(*p).function_name();     //using .(dot) operator
p->function_name();       // using indirection operator
```

– It is recommended to use indirection operator (->) instead of .(dot) operator to access elements of class using pointers. It is because .(dot) operator is having higher priority than -> (indirection) operator.

➢ **Program 3.3.1**

```
#include<iostream>
#include<cstring>
using namespace std;

class example{
private:
int a, b;
public:

   void disp(int c){
      a=c;
cout<<"\n This function is called using function
        pointer: "<<a;
   }
   void print(int d){
      b=d;
cout<<"\n This is another function called using
        function pointer:"<<b;
   }
};

int main()
{
   example e;     //object
   example *p;    //pointer

   p=&e;

cout<<"\n using indirection operator";
   p->disp(100);
   p->print(200);
cout<<"\n using dot operator";
   (*p).disp(300);
   (*p).print(400);
   return 0;
}
```

Output

```
using indirection operator
This function is called using function pointer: 100
This is another function called using function
pointer:200
using dot operator
This function is called using function pointer: 300
This is another function called using function
pointer:400
```

3.4 this Pointer

Q. Write short note on : 'this' pointer.

SPPU - May 14, Dec. 15, 3 Marks, Dec. 19, 2 Marks

Q. What is use of 'this' pointer ?

SPPU - Dec. 16, Dec. 17, 2 Marks

'this' pointer

– 'this' pointer is important concept in C++.

– It is used to get the address of own object.

– It has implicit parameter to all member functions.

– We can use this pointer inside any member function for invoking the object.

– If two variable names are same that time we can use this pointer to differentiate between the variables.

– The Program 3.4.1 illustrates the concept of this pointer.

➢ **Program 3.4.1**

```
#include<iostream>
 using namespace std;
 class this_pointer
{
int count;
```

Techknowledge Publication

```
  public:
this_pointer(int count)
  {
     this->count=10;
cout<<"\n The value of count passed by object is
        "<<count;
cout<<"\n The value of count in this class is
        "<<this->count;
  }
};
int main()
{
cout<<"\n Program for this pointer";
this_pointer a(300);
   return 0;
}
```

Output

```
Program for this pointer
The value of count passed by object is 300
The value of count in this class is 10
```

Explanation : *We have two count variable in the class. One count variable is data member whereas another is part of constructor declaration. We can differentiate between these two variables by using this pointer.*

– this pointer can be useful for returning current object also.

➢ **Program 3.4.2**

```
#include<iostream>
using namespace std;

class test
{
private:
int z;
public:
test() {
   }
test(int d) {
   z = d;
   }
void print() {
cout<<z;
   }
test& operator = (test& a) {
z = a.z;
cout<<"\n Assignment operator invoked";
return *this;
}
};

int main()
{
test a1(500);
test x,y;
y = x = a1;     //invoke overloaded =, twice
cout<<"\nx="; x.print();       //display a2
cout<<"\ny="; y.print();       //display a3
cout<<endl;
return 0;
}
```

Output

```
Assignment operator invoked
Assignment operator invoked
x=500
y=500
```

3.5 Pointers vs Arrays

– The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type.

– For example, consider following two declarations :

```
int myarray [20];
int * mypointer;
```

– The following assignment operation would be valid:

```
mypointer = myarray;
```

TechKnowledge Publications

– After that, mypointer and myarray would be equivalent and would have very similar properties. The main difference being that mypointer can be assigned a different address, whereas myarray can never be assigned anything, and will always represent the same block of 20 elements of type int. Therefore, the following assignment would not be valid:

```
myarray = mypointer;
```

– Let's see Program 3.5.1 that mixes arrays and pointers:

➢ **Program 3.5.1**

```
#include<iostream>
using namespace std;
int main()
{
 int numbers[5];
 int *p;
 p = numbers;  *p = 10;
 p++;  *p = 20;
 p = &numbers[2];  *p = 30;
 p = numbers + 3;  *p = 40;
 p = numbers;  *(p+4) = 50;
 for (int n=0; n<5; n++)
   cout<<numbers[n] << ", ";
 return 0;
}
```

Output

```
10, 20, 30, 40, 50,
```

– Pointers and arrays support the same set of operations, with the same meaning for both. The main difference being that pointers can be assigned new addresses, while arrays cannot.

3.6 Accessing Arrays using Pointers

– Traditionally elements of array can be accessed using index.

– It is also possible to access the elements of array using pointer.

– Program 3.6.1 illustrates array element access using pointer.

➢ **Program 3.6.1**

```
#include<iostream>
using namespace std;
int main()
{
int a[5]={10,20,30,40,50};
int *p;
   p=a;

cout<<"\n Printing array elements using pointer";
   for(int i=0;i<5;i++)
   {
cout<<"\n "<<*p;
      p++;      //point to next element
   }

 return 0;
}
```

Output

```
Printing array elements using pointer
 10
 20
 30
 40
 50
```

– Increment / decrement pointer is illustrated in Program 3.6.2. But pointer increment / decrement should be careful.

➢ **Program 3.6.2**

```
#include<iostream>
using namespace std;
int main()
{
int a[5]={10,20,30,40,50};
int *p;
```

Technknowledge Publications

```
   p=a;

cout<<"\n Printing array elements using pointer";
   for(int i=0;i<5;i++)
   {
cout<<"\n "<<*p;
      p++;
   }
        //pointer is already reach to end of array
        //decrement pointer
   p--;
cout<<"\n Previous element: "<<*p;
                        //decrement pointer
   p--;
cout<<"\n Previous element: "<<*p;
                        //decrement pointer
   p--;
cout<<"\n Previous element: "<<*p;
                        //increment pointer
   p++;
cout<<"\n Next element: "<<*p;
  return 0;
}
```

Output

```
Printing array elements using pointer
10
20
30
40
50
Previous element: 50
Previous element: 40
Previous element: 30
Next element: 40
```

3.6.1 Pointer and String

- String manipulation is possible with pointers.
- As we already know string is nothing but collection of characters stored in character array.
- The way elements of array are accessed, similarly elements of character array (string) can be accessed.
- String manipulation is easier with pointers.

➢ **Program 3.6.3**

```
#include<iostream>
#include<cstring>
using namespace std;
int main()
{
   char str[ ]="This is sample string";
   char *p;
   p=str;
cout<<"\n String manipulation with pointer";
cout<<"\n First character: "<<*p;
   p++;
cout<<"\n next character: "<<*p;
cout<<"\n Print all character with pointer:";
   p=str;     //reset pointer to starting location
   for(int i=0;i<strlen(str);i++)
   {
cout<<"\t"<<*p;
      p++;
   }
  return 0;
}
```

Output

```
String manipulation with pointer
First character: T
next character: h
Print all character with pointer:
T  h  i  si  s  s  a  mp  l  es  t  r  i  n  g
```

3.7 Arrays of Pointers

- Array and pointers are closely related to each other. In C++, the name of an array is considered às a pointer, i.e., the name of an array contains the address of an element. C++ considers the array name as the address of the first element.

Techknowledge Publications

– For example, if we create an array, i.e., marks which hold the 20 values of integer type, then marks will contain the address of first element, i.e., marks[0]. Therefore, we can say that array name (marks) is a pointer which is holding the address of the first element of an array.

– For better understanding, consider the Program 3.7.1.

➢ **Program 3.7.1**

```
#include<iostream>
using namespace std;
int main()
{
   int *ptr;            //integer pointer declaration
   int marks[10];       //marks array declaration
   std::cout<<"Enter the elements of an array :" <<
            std::endl;
   for(int i=0;i<10;i++)
   {
      cin>>marks[i];
   }
   ptr=marks;
    // both marks and ptr pointing to the same element.
   std::cout<<"The value of *ptr is :" <<*ptr<< std
            ::endl;
   std::cout<<"The value of *marks is :" <<*marks
            <<std::endl;
}
```

Output

```
Enter the elements of an array :
1
2
3
4
5
6
7
8
9
10
The value of *ptr is : 1
The value of *marks is : 1
```

– An array of pointers is an array that consists of variables of pointer type, which means that the variable is a pointer addressing to some other element. Suppose we create an array of pointer holding 5 integer pointers; then its declaration would look like:

```
int *ptr[5];        //array of 5 integer pointer.
```

– In the above declaration, we declare an array of pointer named as ptr, and it allocates 5 integer pointers in memory.

– The element of an array of a pointer can also be initialized by assigning the address of some other element. Let's observe this case through an example.

```
int a;       //variable declaration.
ptr[2] = &a;
```

– In the above code, we are assigning the address of 'a' variable to the third element of an array 'ptr'.

– We can also retrieve the value of 'a' be dereferencing the pointer.

```
*ptr[2];
```

– Let's understand through Program 3.7.2.

➢ **Program 3.7.2**

```
#include<iostream>
using namespace std;
int main()
{
   int ptr1[5];   //integer array declaration
   int *ptr2[5]; //integer array of pointer declaration
   std::cout<<"Enter five numbers :" << std::endl;

   for(int i=0;i<5;i++)
   {
      std::cin>>ptr1[i];
   }
   for(int i=0;i<5;i++)
   {
      ptr2[i]=&ptr1[i];
   }
```

Techknowledge Publication

```
    // printing the values of ptr1 array
    std::cout<<"The values are" << std::endl;
    for(int i=0;i<5;i++)
    {
        std::cout << *ptr2[i] << std::endl;
    }
}
```

Output

```
Enter five numbers :
4
5
1
2
3
The values are
4
5
1
2
3
```

- In the above code, we declare an array of integer type and an array of integer pointers. We have defined the 'for' loop, which iterates through the elements of an array 'ptr1', and on each iteration, the address of element of ptr1 at index 'i' gets stored in the ptr2 at index 'i'.

3.8 Function Pointers

Q. Explain pointer to a function. Use suitable example. **SPPU - May 18, 4 Marks**

Q. What is concept of function pointer? Give suitable example in C++. **SPPU - May 19, 5 Marks**

- Function pointers are used in dynamic binding.
- It acts as base for pointer to members.
- It is also known as call back functions.
- Function pointer can be used to refer to functions.
- With function pointer it is possible to select function dynamically.
- Function can be passed as argument to another function. When function is passed, it is passed as pointer during.
- It is possible to compare two function pointers.
- Two types of function pointers are available in C++.
  - Function pointer pointing to static member functions
  - Function pointer pointing to non static member functions.
- Function pointer can be created using following syntax :

```
Return_data_type (*function_name)();
```

- Program 3.8.1 shows how function pointers are used.

➢ **Program 3.8.1**

```
#include<iostream>
#include<cstring>
using namespace std;
typedef void (*ptr)();      //pointer declaration
void disp(){
cout<<"\n This function is called using function
        pointer";
}
void print(){
cout<<"\n This is another function called using
        function pointer";
}
int main()
{
ptr p;                  //create pointer
   p=&disp;             //initialize pointer
   p();
   p=&print;
   p();
   return 0;
}
```

Output

```
This function is called using function pointer
This is another function called using function pointer
```

Tech Knowledge Publications

➢ **Program 3.8.2 : Write a program to find the sum of an array Arr by passing an array to a function using pointer.**

SPPU - Dec. 17, 4 Marks

```
#include<iostream.h>
#include<conio.h>
void max(int *p, int l)
{
int i,sum=0;
for(i=0;i<l;i++)
{
sum+=*p;
p++;
}
cout<<"sum="<<sum;
}
void main()
{
int Arr[5]={1,4,20,3,6};
clrscr();
max(Arr,5);
getch();
}
```

Output

```
Sum=34
```

3.9 Pointers to Pointer

- It is possible for pointer to store another pointer.
- It is called as multiple indirection.
- Pointer contains address of variable.
- In pointer to pointer, first pointer contains address of second pointer, second pointer contains address of actual value.
- Variable with pointer to pointer can declared in following manner.

```
data_type **p;
```

➢ **Program 3.9.1**

```
#include<iostream>
using namespace std;
int main()
{
int a;
int *p;
int **pp;
  a = 100;
 //first store address of actual value
  p = &a;
  //now store address of pointer in another pointer
  pp = &p;
//now print values using different options
cout<<"\n  Value of a using variable:" << a;
cout<<"\n Value of a using pointer :" << *p;
cout<<"\n Value of a using pointer to pointer :" <<
        **pp;
//printing address
cout<<"\n Address of variable :"<<&a;
cout<<"\n Value of pointer: "<<p;
cout<<"\n Address of p:"<<&p;
cout<<"\n Value of pointer to pointer: "<<pp;
cout<<"\n Address of pointer to pointer: "<<&pp;
  return 0;
}
```

Output

```
Value of a using variable:100
Value of a using pointer :100
Value of a using pointer to pointer :100
Address of variable :0x28fefc
Value of pointer: 0x28fefc
Address of p:0x28fef8
Value of pointer to pointer: 0x28fef8
Address of pointer to pointer: 0x28fef4
```

3.10 Pointers to Derived Classes

- In C++, we can declare a pointer points to the base class as well as derive class.
- Consider Program 3.10.1 to understand pointer to derived class.

➢ **Program 3.10.1**

```
#include<iostream.h>
class base
{
   public:
```

Tech Knowledge Publication

```
    int n1;
    void show()
    {
        cout<<"\nn1 = "<<n1;
    }
};
class derive : public base
{
    public:
    int n2;
    void show()
    {
        cout<<"\nn1 = "<<n1;
        cout<<"\nn2 = "<<n2;
    }
};
int main()
{
    base b;
    base *bptr;          //base pointer
    cout<<"Pointer of base class points to it";
    bptr=&b;             //address of base class
    bptr->n1=44;  //access base class via base pointer
    bptr->show();
    derive d;
    cout<<"\n";
    bptr=&d;             //address of derive class
    bptr->n1=66;  //access derive class via base
                  //pointer
    bptr->show();
    return 0;
}
```

Output

```
Pointer of base class points to it
n1 = 44
Pointer of base class points to derive class
n1=66
```

3.11 Passing Pointers to Functions

- If a pointer is passed to a function as a parameter and tried to be modified then the changes made to the pointer does not reflects back outside that function.
- This is because only a copy of the pointer is passed to the function. It can be said that "pass by pointer" is passing a pointer by value.
- In most cases, this does not present a problem. But the problem comes when you modify the pointer inside the function. Instead of modifying the variable, you are only modifying a copy of the pointer and the original pointer remains unmodified.

➢ **Program 3.11.1**

```
#include<iostream>
using namespace std;
int global_var = 42;

//function to change pointer value
void changePointerValue(int* pp)
{
    pp = &global_var;
}
int main()
{
    int var = 23;
    int* ptr_to_var = &var;
    cout<<"Passing Pointer to function:" << endl;
    cout<<"Before :" << *ptr_to_var << endl;
                                        // display 23
    changePointerValue(ptr_to_var);
    cout<<"After :" << *ptr_to_var << endl;
                                    // display 23
    return 0;
}
```

Output

```
Passing Pointer to function:
Before :23
After :23
```

3.12 Return Pointers from Functions

- C++ allows you to return a pointer from a function.

Techknowledge Publications

– To do so, we would have to declare a function returning a pointer as in the following example :

```
int * myFunction() {
.
.
.
}
```

– Second point to remember is that, it is not good idea to return the address of a local variable to outside of the function, so you would have to define the local variable as static variable.

– Now, consider the following function, which will generate 10 random numbers and return them using an array name which represents a pointer i.e., address of first array element.

➢ **Program 3.12.1**

```
#include<iostream>
#include<ctime>
using namespace std;
//function to generate and return random numbers
int* getRandom( ) {
  static int  r[10];
  //set the seed
  srand( (unsigned ) time( NULL ) );
  for (int i = 0; i < 10; ++i) {
    r[i] = rand();
    cout<<r[i] << endl;
  }
  return r;
}
//main function to call above defined function
int main() {
  //a pointer to an int
  int *p;
   p = getRandom();
  for(int i = 0; i < 10; i++) {
    cout<<"*(p + " << i << ") : ";
    cout<<*(p + i) << endl;
  }
   return 0;
}
```

– When the above code is compiled together and executed, it produces result something as follows :

Output

```
624723190
1468735695
807113585
976495677
613357504
1377296355
1530315259
1778906708
1820354158
667126415
*(p + 0) : 624723190
*(p + 1) : 1468735695
*(p + 2) : 807113585
*(p + 3) : 976495677
*(p + 4) : 613357504
*(p + 5) : 1377296355
*(p + 6) : 1530315259
*(p + 7) : 1778906708
*(p + 8) : 1820354158
*(p + 9) : 667126415
```

3.13 Null Pointer

– It is always a good practice to assign the pointer NULL to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

– The NULL pointer is a constant with a value of zero defined in several standard libraries, including iostream. Consider the Program 3.13.1.

```
#include<iostream>
using namespace std;
int main() {
  int  *ptr = NULL;
  cout<<"The value of ptr is " << ptr;
   return 0;
}
```

Techknowledge Publications

– When the above code is compiled and executed, it produces the following result :

```
The value of ptr is 0
```

– On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

– To check for a null pointer you can use an if statement as follows :

```
if(ptr)    // succeeds if p is not null
if(!ptr)   // succeeds if p is null
```

– Thus, if all unused pointers are given the null value and you avoid the use of a null pointer, you can avoid the accidental misuse of an uninitialized pointer.

– Many times, uninitialized variables hold some junk values and it becomes difficult to debug the program.

3.14 Void Pointer

– A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be typcasted to any type.

```
int a = 10;
char b = 'x';
void *p = &a; // void pointer holds address of int 'a'
p = &b;       // void pointer holds address of char 'b'
```

Advantages of void pointers

– malloc() and calloc() return void * type and this allows these functions to be used to allocate memory of any data type (just because of void *)

```
int main(void)
{
    // Note that malloc() returns void * which can be
    // typecasted to any type like int *, char *, ..
    int *x = malloc(sizeof(int) * n);
}
```

– Note that the above program compiles in C, but doesn't compile in C++. In C++, we must explicitly typecast return value of malloc to (int *).

– void pointers in C are used to implement generic functions in C. For example compare function which is used in qsort().

➢ **Program 3.14.1**

```
#include<iostream>
using namespace std;

int main() {
    void* ptr;
    float f = 2.3f;

    //assign float address to void
    ptr = &f;

    cout<<&f << endl;
    cout<<ptr << endl;

    return 0;
}
```

Output

```
0xffd117ac
0xffd117ac
```

– Here, the pointer ptr is given the value of &f.

– The output shows that the void pointer ptr stores the address of a float variable f.

– To print the content of a void pointer, we use the static_cast operator.

Techknowledge Publication

– It converts the pointer from void* type to the respective data type of the address the pointer is storing.

➢ **Program 3.14.2 : Printing the Content of Void Pointer**

```
#include<iostream>
using namespace std;
int main() {
  void* ptr;
  float f = 2.3f;
  //assign float address to void pointer
  ptr = &f;
  cout<<"The content of pointer is ";
  //use type casting to print pointer content
  cout<<*(static_cast<float*>(ptr));
  return 0;
}
```

Output

```
The content of pointer is 2.3
```

– This program prints the value of the address pointed to by the void pointer ptr.

– Since we cannot dereference a void pointer, we cannot use *ptr.

– However, if we convert the void* pointer type to the float* type, we can use the value pointed to by the void pointer.

Important Points

– Pointer

– Function pointer

– Pointer to object

– New and delete

– Pointer to pointer

– this pointer

Review Questions

Q. 1 What is pointers?

Q. 2 How to access elements of array using pointers?

Q. 3 How to access elements of string using pointers?

Q. 4 What is function pointers?

Q. 5 What is pointer to object?

Q. 6 Explain new and delete operator with suitable program?

Q. 7 Write program to explain pointer to pointer?

Technical Knowledge Publication

❑❑❑

CHAPTER 4

Unit - III

Polymorphism

Syllabus

Polymorphism : Introduction to Polymorphism, Types of Polymorphism, Operator Overloading- concept of overloading, Operator overloading, Overloading unary operators, Overloading binary operators, Data conversion, Type casting (implicit and explicit), Pitfalls of operator overloading and conversion, Keywords : explicit and mutable. Function overloading, Run time polymorphism - Pointers to base class, Virtual function and its significance in C++, Pure virtual function and Virtual table, Virtual destructor, Abstract base class.

Technknowledge Publication

4.1 Introduction to Polymorphism

Polymorphism

- It refers to ability to take more than one form in the programming.
- Which behaviour to use totally depends upon type of data used in the operation.
- If we consider the simple case of addition of two number. If we call the functions with two number then function should perform the addition.
- But if we pass two different strings then instead of addition, concatenation of string operation should be performed automatically.
- Polymorphism can be compile time polymorphism and run time polymorphism.
- Compile time polymorphism is possible with function and operator overloading whereas run time polymorphism is possible with virtual functions.
- Polymorphism example is shown in Fig. 4.1.1.

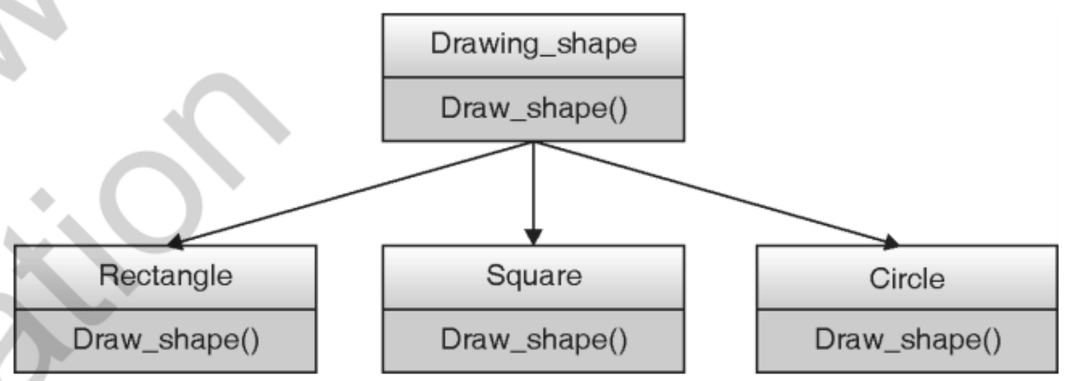

Fig. 4.1.1 : Polymorphism

- As shown in the Fig. 4.1.1, which Draw_shape() to call is totally depend upon the number of parameters to functions.
- If we are passing radius and centre coordinates then circle type of Draw_shape() should get called. If we passing just one side then Draw_shape() for square should get called and so on. Another example is shown in Fig. 4.1.2.

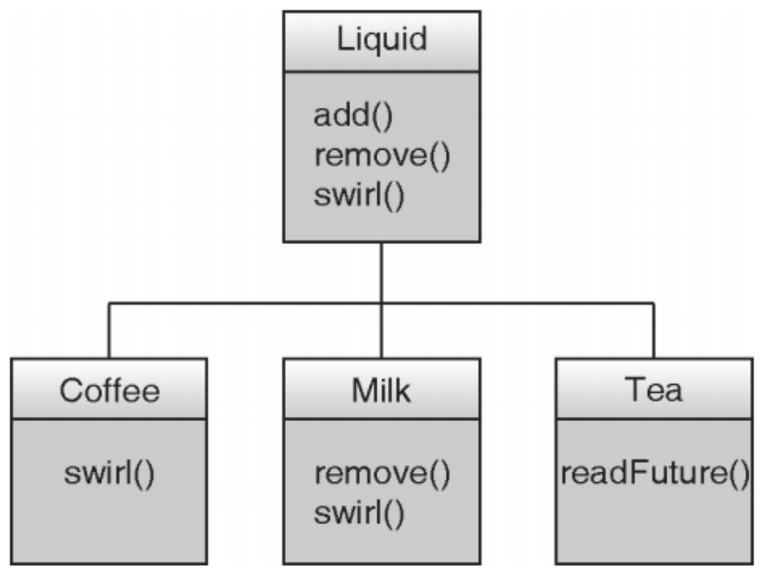

Fig. 4.1.2 : Polymorphism

4.2 Types of Polymorphism

– Polymorphism is a feature of OOPs that allows the object to behave differently in different conditions. In C++ we have two types of polymorphism:

1. Compile time Polymorphism - This is also known as static (or early) binding.
2. Runtime Polymorphism - This is also known as dynamic (or late) binding.

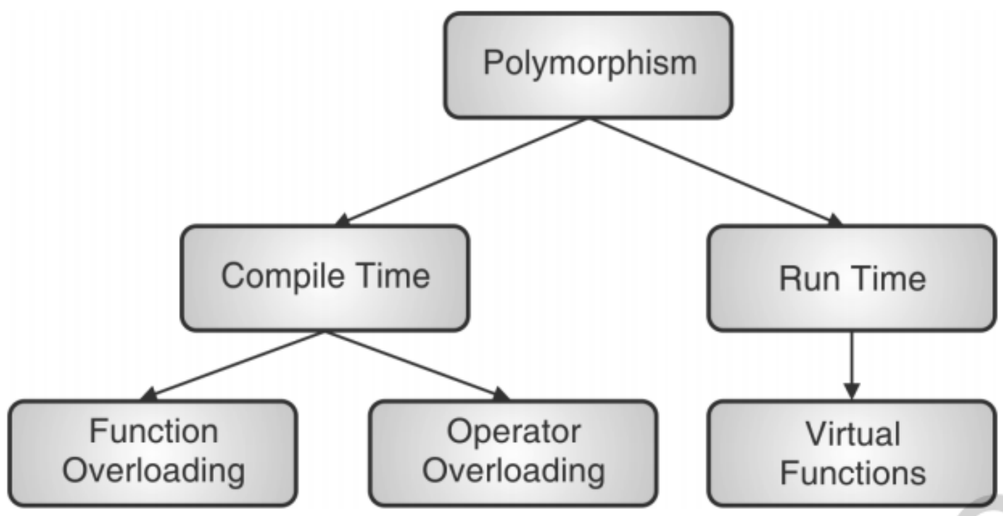

Fig. 4.2.1 :Types of polymorphism

[A] Compile time polymorphism

Function overloading and Operator overloading are perfect example of Compile time polymorphism.

4.2.1 Function Overloading

Q. What is function overloading ? Explain with a suitable example. **SPPU - Dec. 19, 4 Marks**

When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

➢ **Program 4.2.1**

```
//C++ program for function overloading
#include<bits/stdc++.h>
using namespace std;
class FuncOver
{
   public:
   //function with 1 int parameter
   void func(int x)
   {
      cout<<"value of x is " << x <<endl;
   }

   //function with same name but 1 double parameter
   void func(double x)
   {
      cout<<"value of x is " << x <<endl;
   }

   //function with same name and 2 int parameters
   void func(int x, int y)
   {
      cout<<"value of x and y is "<< x << ", " <<
            y <<endl;
   }
};
int main() {

   FuncOver obj1;

   // Which function is called will depend on the
   // parameters passed
   // The first 'func' is called
   obj1.func(7);

   // The second 'func' is called
   obj1.func(9.132);

   // The third 'func' is called
   obj1.func(85,64);
   return 0;
}
```

Output

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

Technknowledge Publication

– In Program 4.2.1, a single function named **func** acts differently in three different situations which is the property of polymorphism.

4.2.2 Operator Overloading

Q. What is operator overloading ? Name the operators that cannot be overloaded in C++ ?

SPPU - May 14, 2 Marks, Dec. 14, 3 Marks

Q. What is Operator overloading?

SPPU - May 18, 2 Marks

– C++ also provides option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands, adds them and when placed between string operands, concatenates them.

– We can overload many operators except some of the operators

– The operator which can be overloaded is listed in Table 4.2.1.

Table 4.2.1

+	-	*	/	%	^	&	|
=	<	>	+=	-=	*=	/=	%=
|=	<<	>>	<<=	>>=	==	!=	<=
||	++	--	,	->*	->	( )	[ ]
new	delete	~	!	^=	&=	>=	&&

Operators that cannot be overloaded in C++

– Operators which cannot be overloaded are as follows :

- Scope resolution operator (::)
- Size of operator
- Conditional operator (?:)
- Class access operator (.)

– Even though we are changing the semantics of the operator, we cannot change syntax for using the operators. + is binary operator which means it requires two values to perform the operation, after overloading also we need to provide the two values to perform the operation.

– With operator overloading we can extend the facility of C++.

➢ **Program 4.2.2**

```
//C++ program to illustrate Operator Overloading
#include<iostream>
using namespace std;
class Complex {
private:
   int real, imag;
public:
   Complex(int r = 0, int i = 0) {real = r; imag = i;}
   //This is automatically called when '+' is used with
   //between two Complex objects
   Complex operator + (Complex const &obj) {
      Complex res;
      res.real = real + obj.real;
      res.imag = imag + obj.imag;
      return res;
   }
   void print() { cout<< real << " + i"
                <<imag<<endl; }
};
int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2;
                        // An example call to "operator+"
   c3.print();
}
```

Output

```
12 + i9
```

Tech Knowledge Publications

– In the Program 4.2.2 the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers (integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers.

[B] Runtime Polymorphism

– This type of polymorphism is achieved by Function Overriding.

4.2.3 Function Overriding

Q. How virtual function different from function overriding. **SPPU - Dec. 19, 3 Marks**

– Function overriding on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

➢ **Program 4.2.3**

```
//C++ program for function overriding
#include<bits/stdc++.h>
using namespace std;
class base
{
public:
   virtual void print()
   { cout<<"print base class" <<endl; }

   void show()
   { cout<<"show base class" <<endl; }
};
class derived : public base
{
public:
   void print()
//print() is already virtual function in derived class,
//we could also declared as virtual void print()
//explicitly
   { cout<<"print derived class" <<endl; }
   void show()
   { cout<<"show derived class" <<endl; }
};
//main function
int main()
{
   base *bptr;
   derived d;
   bptr = &d;
   //virtual function, binded at runtime (Runtime
   //polymorphism)
   bptr->print();
   // Non-virtual function, binded at compile time
   bptr->show();
   return 0;
}
```

Output

```
print derived class
show base class
```

4.2.4 Difference between Compile Time and Run Time Polymorphism

Q. Differentiate compile time and run time polymorphism. **SPPU - Dec. 17, Dec. 18, 4 Marks**

Table 4.2.2 : Difference between compile time and run time polymorphism

Sr. No.	Compile time polymorphism	Run time polymorphism
1.	If object to function binding is at compile time then it is called as compile time polymorphism.	If object to function binding is at run time then it is called as run time polymorphism.
2.	Compile time polymorphism can be of function and operator overloading.	Run time polymorphism can be of virtual function and pointer.
3.	It is also known as Static binding, Early binding and overloading as well.	It is also known as Dynamic binding, Late binding and overriding as well.

Techknowledge Publication

Sr. No.	Compile time polymorphism	Run time polymorphism
4.	In Compile time Polymorphism, call is resolved by the compiler.	In Run time Polymorphism, call is not resolved by the compiler.
5.	Overloading is compile time polymorphism.	Overriding is run time polymorphism.
6.	It provides fast execution because known early at compile time.	It provides slow execution as compare to early binding because it is known at runtime.
7.	Compile time polymorphism is less flexible as all things execute at compile time.	Run time polymorphism is more flexible as all things execute at run time.

4.3 Concept of Overloading

- C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called **function overloading** and **operator overloading** respectively.
- An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).
- When you call an overloaded function or operator, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions.
- The process of selecting the most appropriate overloaded function or operator is called overload resolution.
- The examples of both **function overloading** and **operator overloading** are illustrated in Program 4.2.2.

4.4 Overloading Unary Operators

- The unary operators operate on a single operand and the examples of Unary operators are :
  - The increment (++) and decrement (--) operators.
  - The unary minus (-) operator.
  - The logical not (!) operator.
- The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.
- Program 4.4.1 explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

➢ **Program 4.4.1**

```
#include<iostream>
using namespace std;
class Distance{
private:
   int feet;                    // 0 to infinite
   int inches;                  // 0 to 12

public:
//required constructors
Distance(){
feet=0;
inches=0;
}
Distance(int f,int i){
feet=f;
inches=i;
}
//method to display distance
void display Distance(){
cout<<"F: "<< feet <<" I:"<< inches <<endl;
}
```

TechKnowledge Publications

```
//overloaded minus (-) operator
Distanceoperator-(){
feet=-feet;
inches=-inches;
return Distance(feet, inches);
}
};
int main(){
   Distance D1(11,10), D2(-5,11);
   -D1;                        // apply negation
   D1.displayDistance();       // display D1
   -D2;                        // apply negation
   D2.displayDistance();       // display D2
   return 0;
}
```

Output

```
F: -11 I:-10
F: 5 I:-11
```

4.5 Overloading Binary Operators

- The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.
- Program 4.5.1 explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

➢ **Program 4.5.1**

```
#include<iostream>
using namespace std;
class Box{
double length;          // Length of a box
double breadth;         // Breadth of a box
double height;          // Height of a box
public:
double getVolume(void){
return length * breadth * height;
}
void setLength(double len){
length=len;
}
void setBreadth(double bre){
breadth=bre;
}
void setHeight(double hei){
height=hei;
}
//Overload + operator to add two Box objects
Box operator+(const Box& b){
Box box;
box.length=this->length +b.length;
box.breadth=this->breadth +b.breadth;
box.height=this->height +b.height;
return box;
}
};
//Main function for the program
int main(){
BoxBox1;        //Declare Box1 of type Box
BoxBox2;        //Declare Box2 of type Box
BoxBox3;        //Declare Box3 of type Box
double volume =0.0;
                //Store the volume of a box here
//box 1 specification
Box1.setLength(6.0);
Box1.setBreadth(7.0);
Box1.setHeight(5.0);
//box 2 specification
Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);
//volume of box 1
volume=Box1.getVolume();
cout<<"Volume of Box1 : "<< volume <<endl;
//volume of box 2
volume=Box2.getVolume();
cout<<"Volume of Box2 : "<< volume <<endl;
```

TechKnowledge Publications

```
//Add two objects as follows:
Box3=Box1+Box2;
//volume of box 3
volume=Box3.getVolume();
cout<<"Volume of Box3 : "<< volume <<endl;
return 0;
}
```

Output

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

4.6 Data Conversion

Q. What do you mean by type conversion ? Explain the same with example. **SPPU - May 15, 4 Marks**

– Data conversion in C++ includes conversions between basic types and user-defined types, and conversions between different user-defined types.

– The assignments between types, whether they are basic types or user-defined types, are handled by the compiler with no effort on our part, provided that the same data type is used on both sides of the equal sign.

– The different possible data conversion situations are:

1. **Conversion between basic and user defined data types**
   (a) Conversion from basic to user defined data types
   (b) Conversion from user-defined data type to basic data types
2. **Conversion between user defined data types**
   (a) Conversion in destination object
   (b) Conversion in source object

4.6.1 Conversion between Basic and User Defined Data Types

(a) Conversion from basic to user defined data types

– Conversion from basic to user defined type is done by using the constructor function with one argument of basic type as follows.

Syntax

```
class class_name{
private:
//....
public:
class_name(data_type){
// conversion code
}
};
```

➢ **Program 4.6.1**

```
//conversion from basic type to object
#include<iostream>
using namespace std;
class celsius
{
private:
   float temper;
public:
   celsius()
   {
      temper=0;
   }celsius(float ftmp)
   {
      temper=(ftmp-32)*5/9;
   }
   void showtemper()
   {
      cout<<"Temperature in Celsius: "<<temper;
   }
};
int main()
{
   celsius cel;            //cel is user defined
```

Technknowledge Publication

```
    float fer;                  //fer is basic type
        cout<<"\nEnter  temperature  in  Fahrenheit
                measurement: ";
    cin>>fer;
    cel=fer;          //convert from basic to user-defined;
                      //eqvt to cel = celsius(fer);
    cel.showtemper();
    return 0;
}
```

Output

```
Enter temperature in Fahrenheit measurement:-40
Temperature in Celsius:-40
```

(b) Conversion from user-defined type to basic data types

- Conversion from user-defined type of basic data type is done by overloading the cast operator of basic type as a member function.
- Operator function is defined as an overloaded basic data type which takes no arguments.
- Return type of operator is not specified because the cast operator function itself specifies the return type.

Syntax

```
class class_name{
...
     public:
operator data_type(){
//Conversion code
}
};
```

➢ **Program 4.6.2**

```
//conversion from user to basic type
#include<iostream>
using namespace std;
class celsius
{
private:
   float temper;
public:
   celsius()
   {
      temper=0;
   }
   operator float()
   {
      float fer;
      fer=temper *9/5+32;
      return (fer);
   }
   void get temper()
   {
      cout<<"\n Enter Temperature in Celsius:";
      cin>>temper;
   }
};
int main()
{
   celsius cel;                //cel is user defined
   float fer;                  //fer is basic type
   cel.gettemper();
   fer=cel;          //convert from user-defined to basic;
                     //eqvt to  fer = float(cel);
   cout<<"\nTemperature in Fahrenheit measurement:
      "<<fer;
}
```

Output

```
Enter Temperature in Celsius:-40
Temperature in Fahrenheit measurement:-40
```

4.6.2 Conversion between User Defined Data Types

(a) Conversion in destination object

- This conversion is exactly like conversion of basic to user defined data type i.e. one argument constructor is used.
- Conversion routine is defined as a one argument constructor in destination class and takes the argument of the source class type.

Tech Knowledge Publications

```
classA objA;
classB objB;
objA=objB;
```

– Here, objB is the source of the class classB and objA is the destination object of class classA.

Syntax

```
//source object class
class classB
{
   //body of classB
}

//destination object class
class classA
{
private:
   //....
public:
   classA(classB objB)        //object of source class
   {
      //code for conversion from classB to classA
   }
};
```

(b) Conversion in source object

– This conversion is exactly like conversion of user-defined type to basic type i.e. overloading the cast operator is used.

```
classA objA;
classB objB;
objA=objB;
```

– Here, objB is the source of the class classB and objA is the destination object of class classA.

Syntax

```
//destination object class
class classA
{
   //body of classA
}
//source object class
class classB
{
private:
   //....
public:
   operator classA()    //cast operator destination types
   {
      //code for conversion from classB to classA
   }
};
```

➢ **Program 4.6.3**

```
//conversion from user defined to user defined
//conversion routine in source class
#include<iostream>
#include<cmath>
using namespace std;
class Cartesian
{
private:
   float xco, yco;
public:
   Cartesian()
   {
      xco=0;
      yco=0;
   }
   Cartesian(float x, float y)
   {
      xco=x;
      yco=y;
   }
   void display()
   {
      cout<<"("<<xco<<","<<yco<<")";
   }
};
class Polar
{
```

Techknowledge Publications

```
private:
   float radius, angle;
public:
   Polar()
   {
      radius=0;
      angle=0;
   }
   Polar(float rad, float ang)
   {
      radius=rad;
      angle=ang;
   }
   operator Cartesian()
   {
      float x=static_cast<int>(radius * cos(angle));
      float y=static_cast<int>(radius * sin(angle));
      return Cartesian(x,y);
   }
   void display()
   {
      cout<<"("<<radius<<","<<angle<<")";
   }
};
int main()
{
   Polar pol(10.0,0.78);
   Cartesian cart;
   cart=pol;
   cout<<"\nGiven Polar: ";
   pol.display();
   cout<<"\nEquivalent cartesian: ";
   cart.display();
   return 0;
}
```

Output

```
Give Polar : (10.0, 0.78)
Equivalent cartesian: (7,7)
```

4.7 Type Casting (Implicit and Explicit)

Q. Explain what is type casting, implicit type casting and explicit type casting. **SPPU - Dec. 17, 6 Marks**

- Type casting is converting one data type into another one. It is also called as data conversion or type conversion. It is one of the important concepts introduced.
- There are two types of type casting operations:
  - Implicit type casting
  - Explicit type casting

4.7.1 Implicit Type Casting

- Implicit type casting means conversion of data types without losing its original meaning. This type of type casting is essential when you want to change data types without changing the significance of the values stored inside the variable.
- Implicit type conversion happens automatically when a value is copied to its compatible data type. During conversion, strict rules for type conversion are applied.
- If the operands are of two different data types, then an operand having lower data type is automatically converted into a higher data type. This type of type conversion can be seen in the Program 4.7.1.

➢ **Program 4.7.1**

```
#include<stdio.h>
int main(){
   short a=10; //initializing variable of short data type
   int b;          //declaring int variable
   b=a;            //implicit type casting
   printf("%d\n",a);
   printf("%d\n",b);
}
```

Output

```
10
10
```

Techknowledge Publication

4.7.2 Explicit Type Casting

– In implicit type conversion, the data type is converted automatically. There are some scenarios in which we may have to force type conversion. Suppose we have a variable div that stores the division of two operands which are declared as an int data type.

```
int result, var1=10, var2=3;
result=var1/var2;
```

– In this case, after the division performed on variables var1 and var2 the result stored in the variable "result" will be in an integer format. Whenever this happens, the value stored in the variable "result" loses its meaning because it does not consider the fraction part which is normally obtained in the division of two numbers.

➢ **Program 4.7.2**

```
#include<stdio.h>
int main()
{
   float a = 1.2;
   //intb  = a; //Compiler will throw an error for this
   int b = (int)a + 1;
   printf("Value of a is %f\n", a);
   printf("Value of b is %d\n",b);
   return 0;
}
```

Output

```
Value of a is 1.200000
Value of b is 2
```

4.8 Pitfalls of Operator Overloading and Conversion

Consider the following example :

```
   class Complex
          {
public:
Complex(double = 0, double = 0);
            Complex operator+(Complex b) const;
            Complex operator-(Complex b) const;
            Complex operator*(Complex b) const;
            Complex operator/(Complex b) const;
            Complex operator^(Complex b) const;
            // ...
private:
double _re, _im;
          };

int main()
{  Complexi(0, 1);
cout<< i^2 + 1;          // i*i  is -1
return 0;
 }
```

– In the above example, if we look, we see that it will not print (0,0)

– See the line :

```
cout<< i^2 + 1;
```

– Using the C/C++ operator precedence rules, we can add parentheses as follows :

```
cout<< (i ^ (2 + 1));
```

– The ^ operator is weaker than + (but stronger than <<)

– You cannot change the operator precedence when overloading operators. Do not overload an operator if its precedence is not intuitive for the problem domain.

4.9 Keywords : Explicit and Mutable

Q. Explain the use of explicit and mutable keywords with suitable examples. (Hint : explicit is used with constructor and mutable is used with data member declaration).

SPPU - Dec. 18, 6 Marks, May 19, 4 Marks

– The two unusual keywords: **explicit** and **mutable**. They have quite different effects, but are grouped together here because they both modify class members.

Techknowledge Publication

– The explicit keyword relates to data conversion, but mutable has a more subtle purpose.

Use of explicit keyword

– Consider the Program 4.9.1 and predict the output :

➢ **Program 4.9.1**

```
#include<iostream>
using namespace std;

class Complex
{
private:
   double real;
   double imag;

public:
   //default constructor
   Complex(double r = 0.0, double i = 0.0) : real(r),
              imag(i) {}

   //a method to compare two Complex numbers
   bool operator == (Complex rhs) {
   return (real == rhs.real&&imag == rhs.imag)?
          true : false;
   }
};

int main()
{
   //a Complex object
   Complex com1(3.0, 0.0);

   if (com1 == 3.0)
   cout<<"Same";
   else
   cout<<"Not Same";
   return 0;
}
```

Output

```
Same
```

– As discussed in C++, if a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor because such a constructor allows conversion of the single argument to the class being constructed.

– We can avoid such implicit conversions as these may lead to unexpected results. We can make the constructor explicit with the help of explicit keyword. For example, if we try the Program 4.9.2 that uses explicit keyword with constructor, we get compilation error.

➢ **Program 4.9.2**

```
#include<iostream>
using namespace std;

class Complex
{
private:
   double real;
   double imag;

public:
//default constructor
explicit Complex(double r = 0.0, double i = 0.0) :
real(r), imag(i) {}

   //a method to compare two Complex numbers
   bool operator == (Complex rhs) {
   return (real == rhs.real&&imag == rhs.imag)?
          true : false;
   }
};
int main()
{
   //a Complex object
   Complex com1(3.0, 0.0);

   if (com1 == (Complex)3.0)
   cout<<"Same";
   else
```

Technknowledge Publications

```
    cout<<"Not Same";
    return 0;
}
```

Output

```
Same
```

Use of mutable keyword

- Sometimes there is requirement to modify one or more data members of class / struct through const function even though you don't want the function to update other members of class / struct.
- This task can be easily performed by using mutable keyword. Consider Program 4.9.3 where use of mutable can be useful.
- Suppose you go to hotel and you give the order to waiter to bring some food dish. After giving order, you suddenly decide to change the order of food. Assume that hotel provides facility to change the ordered food and again take the order of new food within 10 minutes after giving the 1st order. After 10 minutes order can't be cancelled and old order can't be replaced by new order. See the Program 4.9.3 for details.

➢ **Program 4.9.3**

```
#include<iostream>
#include<string.h>
using std::cout;
using std::endl;
class Customer
{
    char name[25];
    mutable char placedorder[50];
    int tableno;
    mutable int bill;
public:
    Customer(char* s, char* m, int a, int p)
    {
        strcpy(name, s);
        strcpy(placedorder, m);
        tableno = a;
        bill = p;
    }
    void changePlacedOrder(char* p) const
    {
        strcpy(placedorder, p);
    }
    void changeBill(int s) const
    {
        bill = s;
    }
    void display() const
    {
        cout<<"Customer name is: " << name
            <<endl;
        cout<<"Food ordered by customer is: "
            <<placedorder<<endl;
        cout<<"table no is: " <<tableno<<endl;
        cout<<"Total payable amount: " << bill
            <<endl;
    }
};

int main()
{
    const Customer c1("Pravasi Meet", "Ice Cream", 3,
                      100);
    c1.display();
    c1.changePlacedOrder("GulabJammuns");
    c1.changeBill(150);
    c1.display();
    return 0;
}
```

Output

```
Customer name is: Pravasi Meet
Food ordered by customer is: Ice Cream
table no is: 3
Total payable amount: 100
Customer name is: Pravasi Meet
Food ordered by customer is: GulabJammuns
table no is: 3
Total payable amount: 150
```

Tech Knowledge Publications

4.10 Pointer to Base Class

- One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature.
- The Program 4.10.1 about the rectangle and triangle classes can be rewritten using pointers taking this feature into account:

➢ **Program 4.10.1**

```
//pointers to base class
#include<iostream>
using namespace std;

class Polygon {
protected:
int width, height;
public:
void set_values (int a, int b)
{ width=a; height=b; }
};

class Rectangle: public Polygon {
public:
int area()
{ return width*height; }
};

class Triangle: public Polygon {
public:
int area()
{ return width*height/2; }
};

int main() {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
cout<<rect.area() <<'\n';
cout<<trgl.area() <<'\n';
return 0;
}
```

Output

```
20
10
```

- Function main declares two pointers to Polygon (named ppoly1 and ppoly2).
- These are assigned the addresses of **rect** and **trgl**, respectively, which are objects of type Rectangle and Triangle.
- Such assignments are valid, since both Rectangle and Triangle are classes derived from Polygon.
- Dereferencing **ppoly1** and **ppoly2** (with *ppoly1 and *ppoly2) is valid and allows us to access the members of their pointed objects.
- For example, the following two statements would be equivalent in the Program 4.10.1:

1.	ppoly1->set_values(4,5);
2.	rect.set_values (4,5);

- But because the type of **ppoly1** and **ppoly2** is pointer to Polygon (and not pointer to Rectangle nor pointer to Triangle), only the members inherited from Polygon can be accessed, and not those of the derived classes Rectangle and Triangle.
- That is why the program above accesses the area members of both objects using **rect** and **trgl** directly, instead of the pointers; the pointers to the base class cannot access the area members.
- Member area could have been accessed with the pointers to Polygon if area were a member of Polygon instead of a member of its derived classes, but the problem is that Rectangle and Triangle implement different versions of area, therefore there is not a single common version that could be implemented in the base class.

Techknowledge Publication

4.11 Virtual Function and its Significance in C++

Q. What is virtual function ?

SPPU - Dec. 19, 2 Marks

A virtual function is a member function which is declared within a base class and is re-defined (Overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

- Virtual functions cannot be static and also cannot be a friend function of another class.
- Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
- The prototype of virtual functions should be same in base as well as derived class.
- They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
- A class may have virtual destructor but it cannot have a virtual constructor.
- Consider the Program 4.11.1 showing run-time behaviour of virtual functions.

➢ **Program 4.11.1**

```
//C++ program to illustrate
//concept of Virtual Functions

#include<iostream>
using namespace std;

class base {
public:
   virtual void print()
   {
      cout<<"print base class" <<endl;
   }

   void show()
   {
      cout<<"show base class" <<endl;
   }
};
class derived : public base {
public:
   void print()
   {
      cout<<"print derived class" <<endl;
   }
   void show()
   {
      cout<<"show derived class" <<endl;
   }
};
int main()
{
   base* bptr;
   derived d;
   bptr = &d;
   // virtual function, binded at runtime
   bptr->print();
   // Non-virtual function, binded at compile time
   bptr->show();
}
```

Techknowledge Publication

Output

```
print derived class
show base class
```

4.12 Pure Virtual Function and Virtual Table

4.12.1 Pure Virtual Functions

Q. When do we make a virtual function "pure" ? What are the implications of making a function a pure virtual function **SPPU - May 14, 3 Marks**

Implications of making a function a pure virtual function

- It is type of virtual function which is not defined in the base class.
- Only derived class can define the function.
- It has notation =0 during the declaration and no definition for function.
- If class contains atleast one pure virtual function then that class is called as abstract class.
- It is solely created for extending the class. No functions are defined in the base class.
- All derive classes should define the functions until it is abstract.
- Declaration of pure virtual function is as follows :

```
virtual return_typefun_name()=0;
```

- Example is given in Program 4.12.1.

➢ **Program 4.12.1**

```
#include<iostream>
using namespace std;
class base
{
public:
virtual void display()=0;
};
class derive1 : public base
{
public:
void display()
    {
cout<<"\nThis is first derived class";
    }
};
class derive2 : public base
{
public:
void display()
    {
cout<<"\nThis is second derived class";
    }
};
int main()
{
base *b;
  derive1 d1;
  derive2 d2;
  b = &d1;
b->display();
  b = &d2;
b->display();
return 0;
}
```

Output

```
This is first derived class
This is second derived class
```

4.12.2 Virtual Table

- A Virtual Method Table (VMT), virtual function table, virtual call table, dispatch table, vtable, or vftable is a mechanism used in a programming language to support dynamic dispatch (or run-time method binding).
- To implement virtual functions, C++ uses a special form of late binding known as the virtual table. The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner. The virtual table sometimes goes by other names, such as “vtable”, “virtual function table”, “virtual method table”, or “dispatch table”.

Technknowledge Publication

- Because knowing how the virtual table works is not necessary to use virtual functions, this section can be considered optional reading.
- The virtual table is actually quite simple, though it's a little complex to describe in words. First, every class that uses virtual functions (or is derived from a class that uses virtual functions) is given its own virtual table. This table is simply a static array that the compiler sets up at compile time.
- A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.
- Second, the compiler also adds a hidden pointer to the base class, which we will call *__vptr. *__vptr is set (automatically) when a class instance is created so that it points to the virtual table for that class. Unlike the *this pointer, which is actually a function parameter used by the compiler to resolve self-references, *__vptr is a real pointer. Consequently, it makes each class object allocated bigger by the size of one pointer. It also means that *__vptr is inherited by derived classes, which is important.
- Consider following simple example :

```
class Base
{
public:
   virtual void function1() {};
   virtual void function2() {};
};

class D1: public Base
{
public:
   virtual void function1() {};
};
class D2: public Base
{
public:
   virtual void function2() {};
};
```

- Because there are 3 classes here, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2.
- The compiler also adds a hidden pointer to the most base class that uses virtual functions. Although the compiler does this automatically, we'll put it in the next example just to show where it's added:

```
class Base
{
public:
   FunctionPointer *__vptr;
   virtual void function1() {};
   virtual void function2() {};
};
 class D1: public Base
{
public:
   virtual void function1() {};
};
class D2: public Base
{
public:
   virtual void function2() {};
};
```

- When a class object is created, *__vptr is set to point to the virtual table for that class. For example, when a object of type Base is created, *__vptr is set to point to the virtual table for Base. When objects of type D1 or D2 are constructed, *__vptr is set to point to the virtual table for D1 or D2 respectively.

Tech Knowledge Publication

4.13 Virtual Destructor

- Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behaviour.
- To correct this situation, the base class should be defined with a virtual destructor.
- For example, Program 4.13.1 results in undefined behaviour.

➢ **Program 4.13.1**

```
//C++ program without virtual destructor causing
//undefined behaviour
#include<iostream>
using namespace std;
class base {
public:
   base()
   { cout<<"Constructing base \n"; }
   ~base()
   { cout<<"Destructing base \n"; }
};
class derived: public base {
public:
   derived()
   { cout<<"Constructing derived \n"; }
   ~derived()
   { cout<<"Destructing derived \n"; }
};
int main(void)
{
derived *d = new derived();
base *b = d;
delete b;
getchar();
return 0;
}
```

Output

```
Constructing base
Constructing derived
Destructing base
```

4.14 Abstract Base Class

Q. Describe abstract base class. Illustrate an example to explain it.

SPPU - May 15, Dec. 16, 4 Marks, May 17, May 19, 2 Marks

- They are classes that can only be used as base classes, and thus are allowed to have virtual member functions without definition (known as pure virtual functions).
- The syntax is to replace their definition by =0 (an equal sign and a zero):
- An abstract base Polygon class could look like this:

```
//abstract class Polygon
class Polygon {
protected:
int width, height;
public:
voidset_values (int a, int b)
{ width=a; height=b; }
virtual int area() =0;
};
```

- Notice that area has no definition; this has been replaced by =0, which makes it a pure virtual function. Classes that contain at least one pure virtual function are known as abstract base classes.
- Abstract base classes cannot be used to instantiate objects. Therefore, this last abstract base class version of Polygon could not be used to declare objects like :

```
Polygon mypolygon;
   // not working if Polygon is abstract base class
```

- But an abstract base class is not totally useless. It can be used to create pointers to it, and take advantage of all its polymorphic abilities. For example, the following pointer declarations would be valid:

```
Polygon * ppoly1;
Polygon * ppoly2;
```

Techknowledge Publication

– And can actually be dereferenced when pointing to objects of derived (non-abstract) classes. Here is the entire example:

➢ **Program 4.14.1**

```
//abstract base class
#include<iostream>
using namespace std;
class Polygon {
protected:
int width, height;
public:
void set_values (int a, int b)
{ width=a; height=b; }
virtual int area (void) =0;
};
class Rectangle: public Polygon {
public:
int area (void)
{ return (width * height); }
};
class Triangle: public Polygon {
public:
int area (void)
{ return (width * height / 2); }
};
int main() {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
cout<< ppoly1->area() <<'\n';
cout<< ppoly2->area() <<'\n';
return 0;
}
```

Output

```
20
10
```

Important Points

– Operator overloading

– Overloading binary operator

– Overloading unary operator

– Type casting

– Polymorphism

– Abstract base class

Review Questions

Q. 1 Explain operator overloading?

Q. 2 How to overload unary operator? Write suitable program for the same.

Q. 3 How to overload binary operator? Write suitable program for the same.

Q. 4 What is type casting? Explain implicit and explicit type casting?

Q. 5 What are different pitfalls of operator overloading?

Q. 6 Explain explicit and mutable keyword with program.

Q. 7 What is polymorphism ?

Q. 8 What is Pure virtual function ?

Q. 9 Write program to illustrate use of abstract base class?

Techknowledge Publication

❑❑❑

CHAPTER 5

Unit - IV

Files and Streams

Syllabus

Data hierarchy, Stream and files, Stream classes, Stream errors, Disk file I/O with streams, File pointers and error handling in file I/O, File I/O with member functions, Overloading the extraction and insertion operators, Memory as a stream object, Command-line arguments, Printer output.

Technknowledge Publication

5.1 Data Hierarchy

- Ultimately, all data items that computers process are reduced to combinations of 0s and 1s. This occurs because it is simple and economical to build electronic devices that can assume two stable states, one represents 0 and the other represents 1. It is remarkable that the impressive functions performed by computers involve only the most fundamental manipulations of 0s and 1s.
- The smallest data item that computers support is called a bit (short for "binary digit" a digit that can assume one of two values). Each data item, or bit, can assume either the value 0 or the value 1. Computer circuitry performs various simple bit manipulations, such as examining the value of a bit, setting the value of a bit and reversing a bit (from 1 to 0 or from 0 to 1).
- Data items processed by computers form a data hierarchy as shown in Fig. 5.1.1, in which data items become larger and more complex in structure as we progress from bits to characters to fields to larger data aggregates :

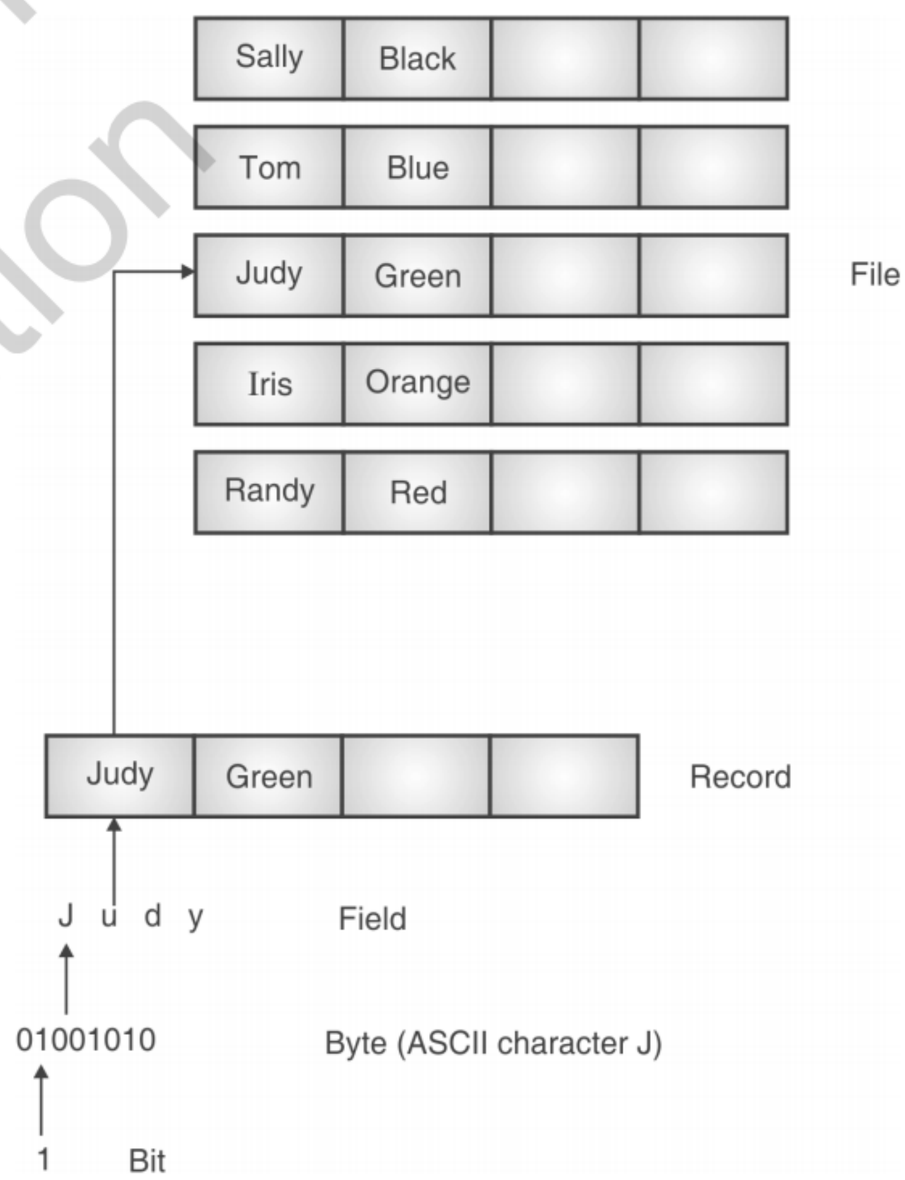

Fig. 5.1.1 : Data Hierarchy

5.2 Streams and Files

Q. What is a stream?

SPPU - Dec. 16, Dec. 17, May 18, Dec. 18, Dec. 19, 2 Marks

- Stream is nothing but sequence of characters.

- Characters may be normal or wild characters.
- Stream work with all built in data types.
- Stream is serial interface to storage, buffer files, or any other storage medium.
- We can overload stream by overloading << (insertion) operator.
- iostream is nothing but standard input output stream library.
- It provides input output functionality using streams.
- Stream is abstraction which represents device on which input and output operations are performed.
- To operate with stream C++ provides iostream library.
- It is mainly declared in header file <iostream>.
- It is derived from istream and ostream classes which is also subclass of ios class.
- istream is for input purpose whereas ostream is for output purpose.
- For file handling we have ifstream for input purpose and ofstream for output purpose.
- It acts as both source and destination.
- In case of input, stream acts as source where as in case of output it acts as destination.
- Source which provides input to the program is known as input stream. Keyboard, file, devices are example of input stream.
- Destination stream is one where program can send the output. Monitor, printer, file are example of output stream.
- Program extracts sequence of byte from the input stream.
- Program adds sequence of bytes to output stream.
- The working of stream is shown Fig. 5.2.1.

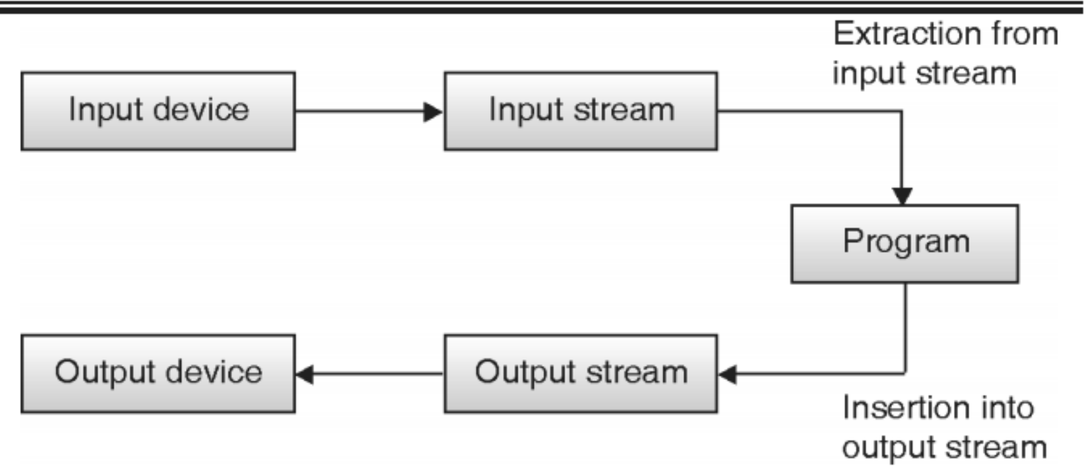

Fig. 5.2.1 : C++ streams

5.3 Stream Classes

Q. Explain types of streams available in C++.
SPPU - Dec. 16, Dec. 17, May 18, Dec. 18, 2 Marks

Q. What are cin and cout? Explain iostream.
SPPU - Dec. 16, May 17, May 18, 3 Marks

- In C++, input/output consists of hierarchy of classes which are useful to manage input output of the system.
- These input output classes is called as stream classes.
- All the classes are declared in header files.
- This is reason why we are adding corresponding header file to work with input output system.
- Classification of stream classes is shown in Fig. 5.3.1.

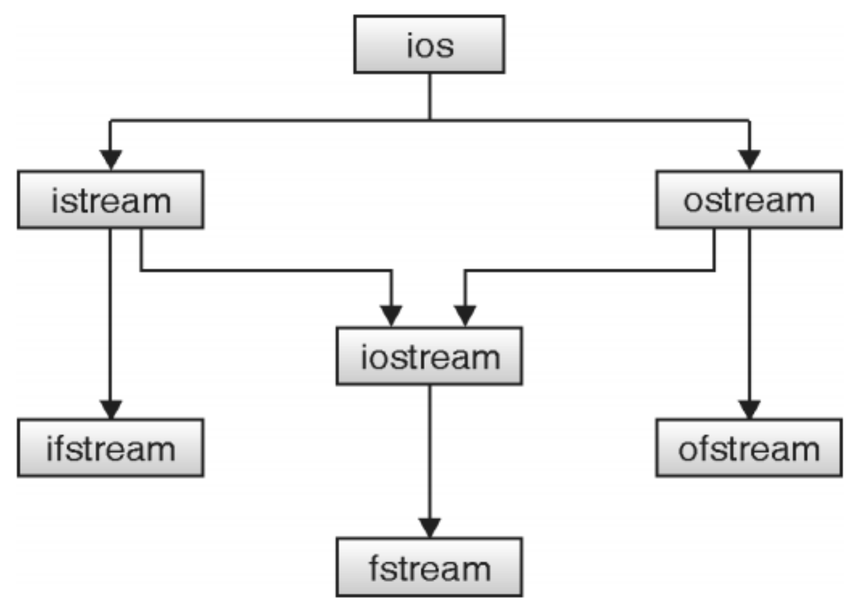

Fig. 5.3.1 : Stream classes in C++

- ios is base of all classes.
- istream is used for input purpose.
- ostream is used for output purpose.

Technknowledge Publications

- iostream is used for both input and output.
- ifstream is used for input from file.
- ofstream is used for output to file.
- fstream is used for both input and output with file.

– Elements of iostream are as follows :

Element	Meaning
ios	Base class for stream
istream	Input stream
ostream	Output stream
iostream	Input output stream
ifstream	Input file stream
ofstream	Output file stream
fstream	Input output file stream

– For ease of use different objects are created.

– Different object in iostream classes is as follows.

- **cin :** Standard input stream object. It is object of istream class. cin is actually object and not function or anything else.
- **cout :** Standard output stream object. It is object of ostream class. cout is object and not function.
- **cerr :** Standard output stream for error.

– iostream class also contains different manipulators. Some of the manipulators are as follows.

Manipulator	Meaning
Boolalpha	Alphanumerical bool values
Dec	Use decimal base
Endl	Newline
Ends	Insert null
Fixed	Use fixed point notation
Flush	Flush stream buffer
Left	Print output to left
Right	Print output to right
Setfill	Set fill character
Setw	Set field width
Ws	Extract whitespaces

Standard output stream (cout)

– cout is object of ostream class.

– cout object is connected to standard output device.

– Standard output device is screen.

– It is used with insertion operator (<<).

– With cout we can display any build in data type values. It means we can display string, char, double, int, float using cout.

– We can overload << to display object.

– Program 5.3.1 shows the working of standard output stream.

➢ **Program 5.3.1**

```
#include<iostream>
 using namespace std;
 int main()
{
   char str[] = "Hello word in C++";
    cout<<str;
}
```

Output

```
Hello word in C++
```

– We can overload >> also.

– Program 5.3.2 shows overloaded << operator to display object. In Program 5.3.2 we have overloaded the operator to display the object.

➢ **Program 5.3.2**

```
#include<iostream>
#include<string>
#include<cstring>
```

Tech Knowledge Publications

```
using namespace std;
class str_op
{
   private:
      char str[40];
   public:

      void get_string()
      {
         cout<<"\n Enter String";
         cin>>str;
      }
                              //display string
      friend void operator<<(ostream &a, str_op &b)
      {
         a<<b.str;
      }
};

int main()
{
   str_op s1;
   s1.get_string();
   cout<<s1;
   return 0;
}
```

Output

```
Enter String maansari
maansari
```

Explanation : *Observe carefully we have displayed object s1 with the help of cout. It is possible because we have overloaded << operator. Otherwise we would not have displayed object.*

Standard input stream (cin)

- It is object of istream class.
- cin object is attached to standard input stream.
- We consider standard input as keyboard.
- It is used with get operator (>>).
- We can overload the >> operator also.

➢ **Program 5.3.3**

```
#include<iostream>
using namespace std;
int main()
{
  char name[50];
  cout<<"Please enter your name: ";
  cin>>name;
  cout<<"Your name is: " << name << endl;
}
```

Output

```
Please enter your name: maansari
Your name is: maansari
```

Standard Errors Stream

- cerr is object of ostream class.
- It is attached to standard error device.
- Standard error device is screen only.
- Object cerr is unbuffered means each stream insertion to cerr causes its output to appear immediately.
- cerr is also used to with << put operator.

➢ **Program 5.3.4**

```
#include<iostream>
using namespace std;
int main()
{
char str[] = "Unable to read....";
cerr << "Error message : " << str << endl;
}
```

Output

```
Error message : Unable to read....
```

5.4 Stream Errors

- Input output system in C++ is based on stream.
- Stream is nothing but sequence of characters.

Techknowledge Publication

- Generally stream works well with cin and cout for input and output purpose.
- But there are many problems during dealing with stream.
- Famous problems are entering character instead of number, entering string instead of character, and so on.
- There may be some hardware failure also.
- Whenever there is error in input or output, it will be reported by stream state.
- Every stream have state associated with it.
- Whenever there are errors with stream which can be handled by observing state of the stream.
- There are many functions to check the state of the streams.
- Some of the functions are listed below :
  - bool ios bool ios::good() : true when everything is fine.
  - bool ios::eof(): return true when if the input operation reached end of input sequence.
  - bool ios::bad(): returns true if the stream is corrupted
  - bool ios::fail(): returns true if the input operation failed to read characters.
  - ios::goodbit : indicates there is no error with streams.
  - ios::eofbit: indicates that the input operation reached end of input sequence.
  - ios::badbit: indicates that the stream is corrupted
  - ios::failbit: indicates that input/output operation failed.
- Program 5.4.1 illustrates the stream error concept.

➢ **Program 5.4.1 : Write a program to illustrate the stream error concept.**

SPPU - Dec. 19, 4 Marks

```
#include<iostream>
using std::cout;
using std::endl;
using std::cin;
using std::ios;
int main()
{
  int a;
   cout<<"block of code before input operations ";
   cout<<"\n cin.rdstate(): "<<cin.rdstate();
   cout<<"\n cin.eof():"<<cin.eof();
   cout<<"\n cin.fail(): "<<cin.fail();
   cout<<"\n cin.bad(): "<<cin.bad();
   cout<<"\n cin.good(): "<<cin.good();
   cout<<"\n\n Desired input is int but please enter
          character : ";
   cin>>a;
   cout<<"\n";
   cout<<"After wrong input from user: ";
   cout<<"\n cin.rdstate(): "<<cin.rdstate();
   cout<<"\n cin.eof():"<<cin.eof();
   cout<<"\n cin.fail(): "<<cin.fail();
   cout<<"\n cin.bad(): "<<cin.bad();
   cout<<"\n cin.good(): "<<cin.good();
   cin.clear(ios::goodbit);
   cout<<"After cin.clear()";
   cout<<"\n cin.fail(): "<<cin.fail();
   cout<<"\n cin.good(): "<<cin.good()<<"\n";

 return 0;
}
```

Output

```
block of code before input operations
cin.rdstate(): 0
 cin.eof():0
cin.fail(): 0
cin.bad(): 0
cin.good(): 1
Desired input is int but please enter character : asdf
After wrong input from user :
cin.rdstate(): 4
 cin.eof():0
cin.fail(): 1
cin.bad(): 0
```

Techknowledge Publication

```
cin.good(): 0
cin.fail(): 0
cin.good(): 1
```

Explanation : *We can observe if we enter wrong input then different flags are affected.*

5.5 Files

- Upto this point we have deal with temporary data only.
- It means we read and write the data from variables.
- When program terminates, we are not able to access the data from variables.
- Large volume of data cannot be handled with the help of variables.
- Consider a cases where we need to store the information of student in class. There are 100 students in class. No suitable data structure is available to store this much amount of information. Again we need to access the stored information again and again. We cannot enter the information again and again. The above problem can be solved using the concept of files.
- Files are part of secondary storage devices. It means once we store the data on this devices, it will be permanently stored in it.
- Data storage with secondary storage devices is possible using file concept. Example of secondary storage devices are hard disk, pen drive, floppy, etc.
- We can read or write data in the file whenever required.
- C++ provides very user friendly list of functions to manipulate the files.
- Different file operations are as follows :
  - Create file
  - Open file
  - Read from file
  - Write to file
  - Close file
  - Append contents in file
  - Modify file
  - Truncate file
- There are many more functions available to manipulate the information in the file.
- Program can store the information in the file or read the data from file. Files can be character or binary files.

5.5.1 File Handling in C++

- As we already know input output is handled in C++ with the help of stream classes.
- File handling is also possible using stream classes but different stream classes which are made for file handling only.
- I/O system handles file more like input output with console.
- They use file stream interface between program and file for data manipulation.
- Stream which feeds data to program is known as input stream and stream which receives the data is known as output stream.
- Steam which feeds data to program through file is known as file input stream and stream which stores program data to file is known as file output stream.
- The working of input output file streams are shown in Fig. 5.5.1.

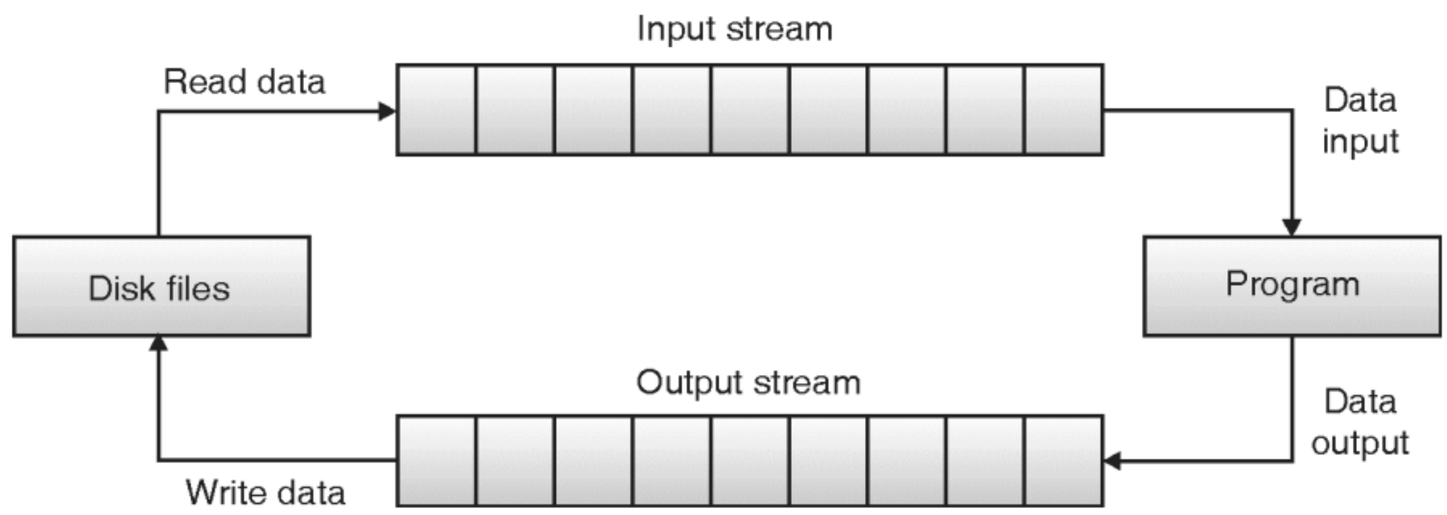

Fig. 5.5.1 : Input output file stream

TechKnowledge Publications

5.5.2 C++ Classes for File Handling

– We have already seen hierarchy of stream classes.

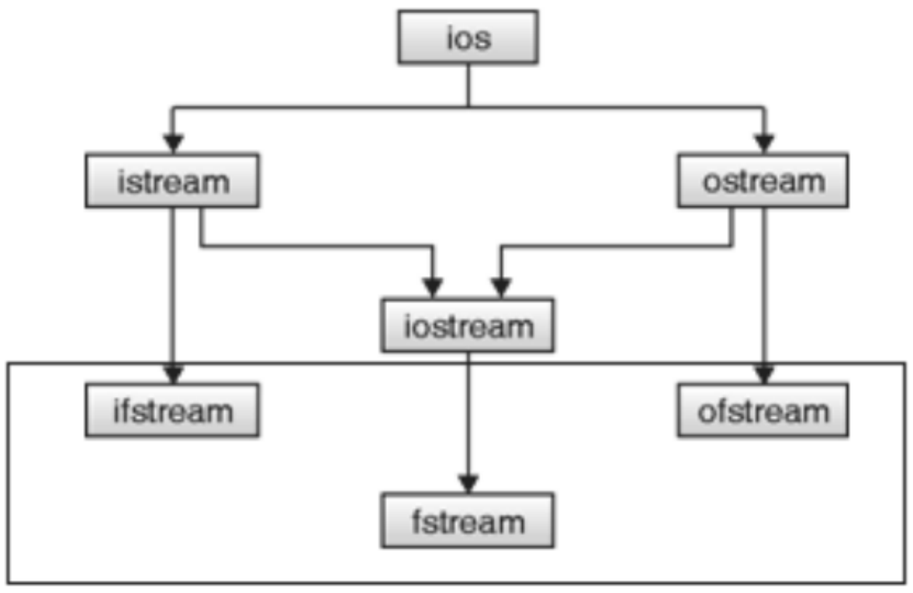

Fig. 5.5.2 : Stream classes

– In the Fig. 5.5.2, ifstream, ofstream and fstream classes are file classes.

– If we wanted to read the information from file then ifstream class can be used. Different functions like open(), get(), getline(), read(), seekg(), tellg(), etc. functions are available.

– If we want to write the information in file then ofstream class can be used. With ofstream class, different functions are available like open(), put(), seekp(), tellp(), write(), etc.

– If we wanted to perform both the operations then fstream class is used.

– As shown in Fig. 5.5.2, we can see ifstream is subclass of istream, ofstream is subclass of ostream and fstream is subclass of iostream. It means these classes enjoy many functionalities from their parent classes.

5.6 Disk File I/O with Streams

Q. Explain following file handling functions :

I. open() II. get() III. getline()

IV. seekg() V. tellg()

SPPU - May 18, Dec. 18, 6 Marks

1. Opening file

– Whenever we need to read or write data in file, we need to open the file.

– For file creation and file opening same function is used.

– After read or write operation we close the file.

– For opening file we need to decide where to store the file and what name to give to identify the file.

– ofstream class is used to open the file.

– We need to create object of ofstream class to open the file.

– After creating object, it will open the file. If file is not existing in the system, then new file will be created. If file is existing in the file system, then file will be truncated. Truncated means all the contents of file will be erased.

– ofstream header file need to included to run the program.

– Program 5.6.1 illustrate file opening.

➢ **Program 5.6.1**

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
   cout<<"\n File opening operation";
   ofstream fout("c:\\example.txt");
   return 0;
}
```

Output

```
File opening operation
```

– It is also possible to open the file with open() function.

– Program 5.6.2 shows how to open file with open() function.

Technknowledge Publications

➢ **Program 5.6.2**

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
   cout<<"\n File opening with operation";
   ofstream fout;
   fout.open("c:\\example.txt");
   cout<<"\n File opened successfully";
   fout.close();
   return 0;
}
```

Output

```
File opening with operation
File opened successfully
```

2. File closing operation

– Whenever we open the file we need to close it.

– Generally C++ automatically closes the file. No need to close it explicitly.

– But it is better to close file explicitly.

– It is possible by close() function.

➢ **Program 5.6.3**

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
   cout<<"\n File closing operation";
   ofstream fout("c:\\example.txt");
   fout.close();
   return 0;
}
```

Output

```
File closing operation
```

3. Writing data to file

– For writing data to file we need to create or open file first.

– If file is existing then existing file will be truncated.

– We can write data to file using << put operator more like put operator with cout statement.

➢ **Program 5.6.4**

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
   int roll;
   char name[10];
   float marks;
   cout<<"\n File closing operation";
   ofstream fout("c:\\example.txt");
   cout<<"\n Enter roll number: ";
   cin>>roll;        //get input from user
   fout<<roll;       //write roll number in file
   cout<<"\n Enter Name:  ";
   cin>>name;        //get input from user
   fout<<name;       //write name in file
   cout<<"\n Enter marks: ";
   cin>>marks;       //get input from user
   fout<<marks ;     //write marks in file
   fout.close();
   return 0;
}
```

Output

```
File closing operation
Enter roll number: 100
Enter Name:  maansari
Enter marks: 66.66
```

Explanation : *After program execution above information will be written to file named example.txt. if we open the file from C:\ then contents of file will look like*

100

maansari

66.66

Tech Knowledge Publications

4. Reading data from file

- Now we will see how to read data from file.
- It is possible by using object of ifstream and >> operator.
- It is more like cin and >>.

➢ **Program 5.6.5**

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
   int roll;
   char name[10];
   float marks;
   cout<<"\n File reading operation";
                        //first write data
   ofstream fout("c:\\example.txt");
   cout<<"\n Enter roll number: ";
   cin>>roll;        //get input from user
   fout<<roll;      //write roll number in file
   cout<<"\n Enter Name:  ";
   cin>>name;       //get input from user
   fout<<name;      //write name in file
   cout<<"\n Enter marks: ";
   cin>>marks;      //get input from user
   fout<<marks ;     //write marks in file
   fout.close();

    //now read data in temporary variable
   ifstream fin("c:\\example.txt");
   fin>>roll;
   fin>>name;
   fin>>marks;

    //now print data
   cout<<"\n Roll: "<<roll;
   cout<<"\n name: "<<name;
   cout<<"\n marks: "<<marks;
   return 0;
}
```

Output

```
File reading operation
Enter roll number: 10
Enter Name:  maansari
Enter marks: 66.666
Roll: 10
name: maansari
marks: 66.666
```

Explanation : *Observe carefully we have first stored data in file, then read data from file in temporary variable and then print using cout.*

5. Detecting end of file

- It is necessary to detect the end of file at the time of reading the data from file.
- It is possible using different options.
- ifstream object return 0 after reaching the end of file during the reading.
- We can detect end of file using eof() function which is part of ifstream class.
- Program 5.6.6 illustrate the use of eof() function.

➢ **Program 5.6.6**

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
   ifstream iFile("input.txt");
   while (!iFile.eof())
   {
    int x;
    iFile >> x;
   }
   return 0;
}
Input.txt
10
20
30
```

Techknowledge Publication

Output

```
10
20
30
```

5.6.1 File Opening Mode

- We can open file in different modes also.
- This mode should be used in open() function.
- They are applicable to open existing file, or create new file.
- Generally open() function takes single argument that is name of file. But it also takes two arguments that is name of file and file opening mode.
- Syntax for opening file in different mode is as follows :

```
Object_name.open("file_name", "file_opening_mode");
```

file_name is nothing but name of the file where as file_opening_mode is name of mode in which we want to open file.

- Different file opening modes are as follows :

File opening mode	Meaning
ios::app	Useful to append content in existing file.
ios::ate	Open file and go to end of the file.
ios::binary	Open file in binary mode.
ios: :in	Open file for only reading purpose.
ios::nocreate	Opening fails if file is not existing.
ios::noreplace	Opening fails if file existing.
ios::out	Opening file for writing purpose.
ios::trunc	Delete all the contents of file.

5.6.2 Reading and Writing Class Object from File

- Unlike built in data types, we can write object in file also.
- Generally objects are written in binary mode.
- Binary writing helps in maintaining information in same format.
- Object is important part in object oriented programming. So we focus on writing or reading object in file.
- read() and write() functions from ifstream and ofstream classes are used to read or write object in file.
- Read and write function syntax is given below :

```
File_object.read(address_of_object, sizeof_object);
File_object.write(address_of_object, sizeof_object);
```

Example

Suppose file object is file and object to read/write is myobject

```
file.read((char *)&myobject, sizeof(myobject);
file.write((char *)&myobject, sizeof(myobject);
```

- Program 5.6.7 illustrates reading /writing class object in file.

➢ **Program 5.6.7 : Write the C++ file input and output program using seekg(), tellg(), read() etc. member functions.** **SPPU - May 17, 7 Marks**

```
#include<iostream>
#include<fstream>
using namespace std;
class file_object{
    char name[20];
    int emp_id;
    float salary;
    public:
        void accept()
        {
            cout<<"\n Enter name, emp_id, salary";
            cin>>name;
```

Techknowledge Publication

```
        cin>>emp_id;
        cin>>salary;
    }
    void display()
    {
        cout<<"\n employee information is ";
        cout<<"\n"<<name<<"\t"<<emp_id<<
                "\t"<<salary;
    }
};
int main()
{
    file_object o;
    fstream file;

    file.open("input", ios::in | ios::out);
                //file opened in read and write mode
    o.accept();                 //accept input from user
    file.write((char* ) &o, sizeof(o));


    file.seekg(0);  //reset file pointer to start of the file
    file.read((char* ) &o, sizeof(o));
                                //read data from file
    o.display();                //display data
    return 0;
}
```

Output

```
Enter name, emp_id, salary  maansari
101
70000
employee information is
maansari    101 70000
```

5.6.3 Reading and Writing Multiple Class Objects

- It is also possible to read or write multiple objects in file.
- Program 5.6.8 illustrate reading and writing multiple objects in file.

➢ **Program 5.6.8 : Write a program to create a file, read and write the record into it. Every record contains Employee Name, Id and Salary. Store and retrieve at least 3 employee data.**

SPPU - May 18, Dec. 18, 7 Marks

```
#include <iostream>
#include<fstream>

using namespace std;
class file_object{
    char name[20];
    int emp_id;
    float salary;
    public:
        void accept()
        {
            cin>>name;
            cin>>emp_id;
            cin>>salary;
        }
        void display()
        {
        cout<<"\n"<<name<<"\t"<<emp_id<
            <"\t"<<salary;
        }
};
int main()
{
    file_object o[5];
    fstream file;
    int i;

    file.open("input", ios::in | ios::out);
                //file opened in read and write mode.
    cout<<"\n Enter information of 3 employees (Enter
            name, emp_id, salary)";
    for(i=0;i<3;i++)
    {
     cout<<"\n  Enter  information  of  "<<i<<"
            employee  ";
      o[i].accept();
                        //accept input from user
```

TechKnowledge Publications

```
    file.write((char* ) &o[i], sizeof(o[i]));
                        //write object in file
  }
  file.seekg(0);
                //reset file pointer to start of the file.
   cout<<"\n Information of employee is as follows";
  for(i=0;i<3;i++)
  {
    file.read((char* ) &o[i], sizeof(o[i]));
                        //read data from file
    o[i].display();       //display data
  }
  return 0;
}
```

Output

```
Enter information of 3 employees (Enter name, emp_id, salary)
Enter information of 0 employee  rasatao
101
70000

Enter information of 1 employee  maansari
102
75000

Enter information of 2 employee  abc
103
8000

Information of employee is as follows
rasatao      101 70000
maansari     102 75000
abc           103 8000
```

➢ **Program 5.6.9 : Write a program using open(), eof() and getline() member functions to open and read a file content line by line.**

Dec. 16, May 19, 7 Marks

```
#include<iostream.h>
#include<fstream.h>
using namespace std;
void main()
{
   char filenm[70];
  ifstream ipfile;
  char fl[70];
  char sl[70];
  char tl[70];
//prompt user for file name to be opened
    cout<<"Enter the filename to be opened: ";
//read and store the file name
  cin>>filenm;
//test open file for reading
ipfile.open(filenm);
//if not the end of file
  if(!ipfile.eof())
{
   cout<<"\nThe first line of text is: \n";
  ipfile.getline(fl, 70);
  cout<<fl<<'\n';
  cout<<"The second line of text is: \n";
   ipfile.getline(sl, 70);
  cout<<sl<<endl;
  cout<<"The third line of text is: \n";
  ipfile.getline(tl, 70);
  cout<<tl<<endl;
//we leave the other line of text
}
}// main ends
```

5.7 File Pointers

Q. What are various functions which are used to manipulate file pointers ? Explain using examples.

SPPU - May 17, Dec. 17, 7 Marks

Q. Explain the concept of file pointers.

SPPU - Dec. 19, 6 Marks

– Each file has two file pointers associated with it.

– One is input pointer and another is output pointer which are useful for reading and writing purpose.

– We can move this pointer to and from for reading and writing file.

Technknowledge Publication

- We can read or write data on specified location with file pointers.
- Whenever we read or write data, pointer automatically advances to next location.
- When we open file in read mode, input pointer is automatically located to start of the file. From start of the file we can start reading the file.
- Whenever we open file in write mode, all contents of files are deleted and file pointer is set to beginning of the file. In this case we write the contents in the file right from the beginning of the file.
- When we wanted to open existing file for new data addition then we need to open that file in append mode and not write mode.
- When file is opened in append mode, file pointer is set to end of the file. New contents can be added after the current file pointer.
- Fig. 5.7.1 shows the position of file pointer in different modes.

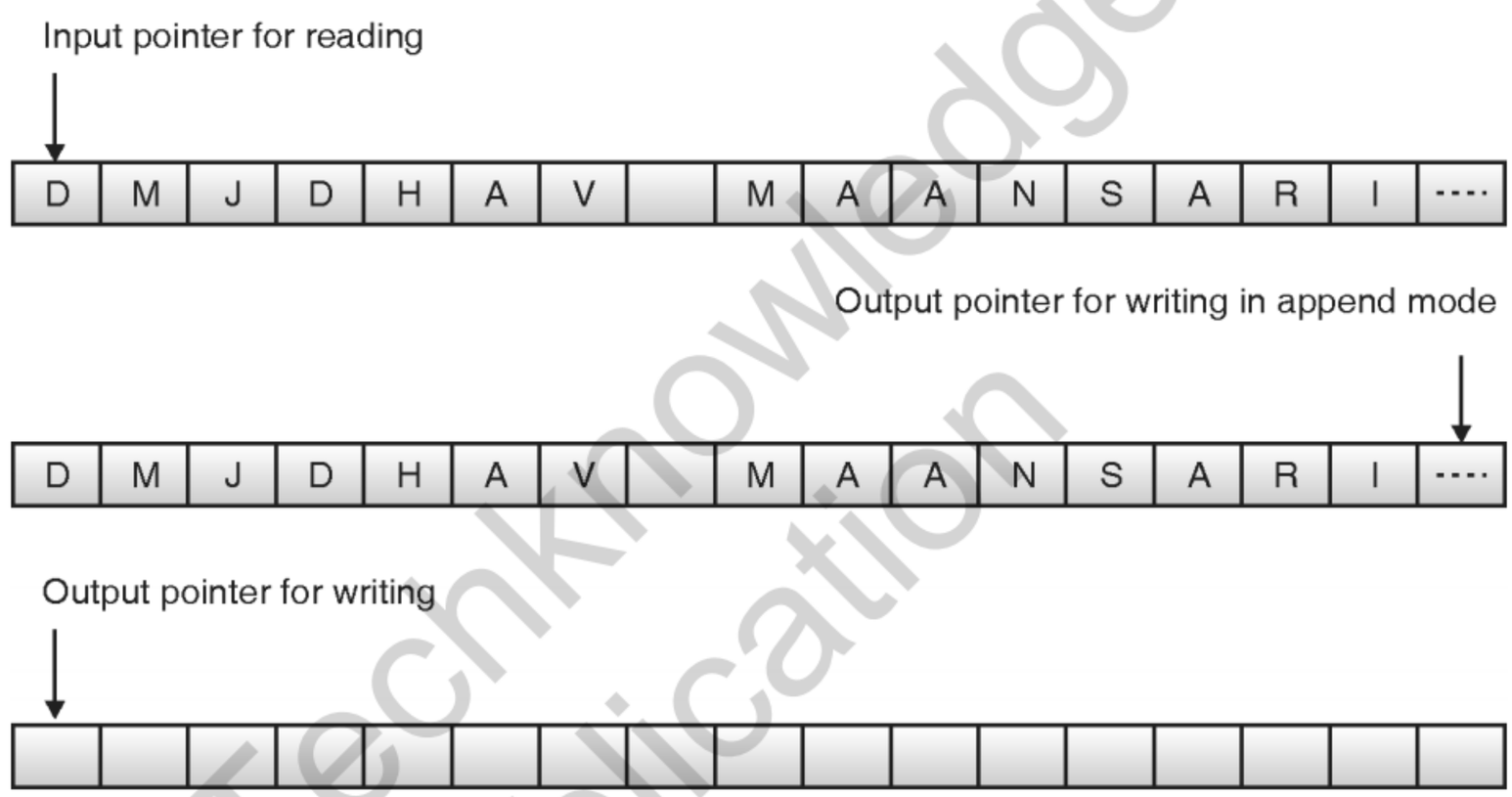

Fig. 5.7.1 : File pointer position

- Different functions are available for manipulating the file pointers.
- These functions can be used to move the file pointer in the file for different purpose.

Function	Meaning	Example
seekg()	Move input pointer to given position	fin.seekg(20); move pointer to byte 20
seekp()	Move output pointer to given position	fout.seekp(20); move pointer to byte 20
tellg()	Get the current position of get pointer	int g=fin.tellg();
tellp()	Get the current position of put pointer	int p=fout.tellg();

Technknowledge Publications

- Seek functions can also be used to offset in the file.
- It means we move in the file from beginning, end, or current position.
- Syntax with offset and reference position is given below :

```
seekg(offset, ref_position);
```

offset is nothing but number of byte to be traversed in the file, ref_position is nothing but reference point in the file like start, end, current.

- Reference position can be specified in the following manner.

```
ios::beg        //start of the file
ios:: cur       //current position in the file
ios:: end       //from the end of file
```

- Some of the actual example are as follows.

Example	Meaning
fout.seekg(10, ios::start)	Move to 10 bytes from start of the file.
fout.seekg(20,ios::end)	Move 20 bytes from the end of file. It means, it goes backward 20 bytes from end of the file.
fout.seekg(30, ios::cur)	Moves 30 bytes from current file pointer.

5.8 Error Handling in File I/O

Q. Explain error handling in file I/O with suitable program. **SPPU - Dec. 19, 5 Marks**

- Error handling is very important in file handling.
- So far we have considered ideal situation for file handling.
- There may be error present during file handling.
- Some of the errors are file not found, could not open file, could not read file, could not write in the file, etc.
- But for efficient programming we need to check these error conditions to get the desired outcome from the program.
- Other error may occurs like file we are trying to open does not exists and so on.
- Sometimes error could occurs due to invalid file name.
- We may attempts to read or write file without opening it.
- Error handling is possible by checking the return value from functions of the object.
- Some of the error handling functions are useful in dealing with unwanted situations.

Function	Meaning
eof()	It returns true if end of file is reached otherwise false.
fail()	It return true when input or output operation failed.
bad()	Return true when invalid operation is attempted to perform.
good()	Return true if no error present.

- Program 5.8.1 illustrate error handling during file operations.

➢ **Program 5.8.1**

```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
  ifstream file;
  file.open("xyz");
  if (file.is_open())
  {
    cout<<"File successfully open";
    file.close();
  }
  else
  {
    cout<<"Error opening file";
  }
  return 0;
}
```

Technknowledge Publications

Output

```
Error opening file
```

Explanation : *As we can see, there is error in opening the file because "xyz" file doesn't exist in the system.*

5.9 File I/O with Member Functions

- In Standard C++, you can do I/O to and from disk files very much like the ordinary console I/O streams cin and cout.
- The object cin is a global object in the class istream (input stream), and the global object cout is a member of the class ostream (output stream).
- File streams come in two flavours :
  - **the class ifstream** (input file stream) inherits from istream, and
  - **the class ofstream (output file stream)** inherits from ostream.
- Thus all of the member functions and operators that you can apply to an istream or ostream object can also be applied to ifstream and ofstream objects.
- However, file streams have some additional member functions and internal information reflecting how they are connected to files on the disk.

Data type	Description
ofstream	It is used to create files and write on files.
ifstream	It is used to read from files.
fstream	It can perform the function of both ofstream and ifstream which means it can create files, write on files, and read from files.

- The syntax for **Opening a file** is as follows :

```
#include<iostream>
#include<fstream>
using namespace std;
int main(){
  ofstream file;
  file.open ("example.txt");
  return 0;
}
```

- We have opened the file 'example.txt' to write on it. 'example.txt' file must be created in your working directory.
- The syntax for **Closing a file** is as follows :

```
#include<iostream>
#include<fstream>
using namespace std;
int main(){
  ofstream file;
  file.open ("example.txt");
  file.close();
  return 0;
}
```

- C++ automatically close and release all the allocated memory. But a programmer should always close all the opened files.
- For **Reading and Writing** on a file, we use **<<** and **>>** to write and read from a file respectively.

```
#include<iostream>
#include<fstream>
using namespace std;
int main(){
  char text[200];
  fstream file;
  file.open("example.txt", ios::out | ios::in );
  cout<<"Write text to be written on file." << endl;
  cin.getline(text, sizeof(text));
 //writing on file
  file << text << endl;
  //reding from file
  file >> text;
```

Technknowledge Publication

```
  cout<<text << endl;

  //closing the file
  file.close();
  return 0;
}
```

5.10 Overloading the Extraction and Insertion Operators

- With cin and cout we can handle all built in data types for input and output.
- >> and << is used with cin and cout to perform the input output operation.
- It is not possible to deal with user defined data types using cin and cout.
- Still C++ provides operator overloading concept wherein we can overload << and >> to deal with user defined data types likes object.
- Keep in mind, make operator overloading function friend functions so that we can call it without creating the object of the class.
- Without operator overloading we need to deal with all members of class individually.

➢ **Program 5.10.1**

```
#include<iostream>
using namespace std;
class distance_example
{
  private:
    int f;
    int i;
  public:
    distance_example(){
      f = 0;
      i = 0;
    }
    distance_example(int feet, int in){
      f = feet;
      i = in;
    }
    friend ostream &operator<<(ostream &out, const
        distance_example &A)
    {
   out<<"Feet : " << A.f << " Inches : " << A.i;
    return out;
    }
    }
 friend istream &operator>>(istream &in,
distance_example &A)
    {
      in>> A.f >> A.i;
      return in;
    }
};
int main()
{
  distance_example A(15, 20), B(25, 30), C;

  cout<<"Enter the value of object : "<<"\n";
  cin>>C;
      //using overloaded >> to get object as input

   //uses << operator to display object
  cout<<"This is 1st distance : " << A <<"\n";
  cout<<"This is 2nd Distance :" << B <<"\n";
  cout<<"This is 3rd Distance :" << C <<"\n";

  return 0;
}
```

Output

```
Enter the value of object :
40 45
This is 1st distance : Feet : 15 Inches : 20
This is 2nd Distance :Feet : 25 Inches : 30
This is 3rd Distance :Feet : 40 Inches : 45
```

- Now we overload the insertion and extraction operator to read or write data in file.
- This program will read or write data from file but it will happen using overloading of << and >> operator.

Techknowledge Publications

➢ **Program 5.10.2**

```
#include<iostream>
#include<fstream>
using namespace std;
class distance_example
{
  private:
    int f;
    int i;
  public:
  distance_example(){
    }
    distance_example(int feet, int in){
      f = feet;
      i = in;
    }
 friend ostream &operator<<(ostream &out, const distance_example &A)
    {
    out<<"Feet : " << A.f << " Inches : " << A.i;
    return out;
     }
friend istream &operator>>(istream &in, distance_example &A)
    {
      in>>A.f >> A.i;
      return in;
    }
};
int main()
{
   ofstream fout;
   fout.open("input");
   distance_example A(10,20);
   fout<<A;       //write first object in file
   distance_example B(30,40);
   fout<<B;       //write second object in file
   distance_example C(50,60);
   fout<<C;       //write third object in file
   fout.close();
                  //now read contents from file
   ifstream fin;
   fin.open("input");
   cout<<"\n Contents of file are";
   fin>>A;
   cout<<"\n Distance = " << A <<endl;
   fin>>B;
   cout<<"Distance = " << B <<endl;
   fin>>C;
   cout<<"Distance = " << C <<endl;
  return 0;
}
```

Output

```
Contents of file are
Distance = Feet : 10 Inches : 20
Distance = Feet : 30 Inches : 40
Distance = Feet : 50 Inches : 60
```

Explanation : *We can observe that instead of cin and cout we have used file object for reading and writing data from file. It means we are writing the information in file and reading information from file only.*

5.11 Memory as a Stream Object

- We can use section of memory to store stream object.
- After creating object we can read or write data in stream object more like file.
- It is useful when we need to format the output.
- It is common in calling functions in GUI environment.
- Family of stream classes implements in-memory formatting.
- For writing to memory we use ostrstream class. It is subclass of ostream class only.
- For reading data from memory we use istrstream which is subclass of istream class.
- For reading and writing purpose we can use strstream class.

Technknowledge Publications

– Program 5.11.1 shows how stream objects can be stored in memory.

➢ **Program 5.11.1**

```
#include<strstream>
#include<iostream>
#include<iomanip>
using namespace std;
const int MAX = 80;
int main()
{
char ch = 'a';
int i = 100;
double num = 1235.567;
char str1[] = "abcd";
char str2[] = "efgh";
char buff[MAX];
ostrstream my_object(buff, MAX);
my_object <<"\n ch="<< ch;
my_object<<"\n i=" << i;
my_object<<setiosflags(ios::fixed)<<setprecision(2)
            << "\n double=" << num;
my_object<<"\n first String =" << str1;
my_object<<"\n Second string=" << str2;
cout<<buff;
return 0;
}
```

Output

```
ch=a
i=100
double=1235.57
first String =abcd
Second string=efgh
```

5.12 Command-Line Arguments

Q. Explain command line argument in arguments in C++. Write program to explain the same.

SPPU - May 17, Dec. 17, May 19, Dec. 19, 6 Marks

– As we know we pass argument to functions.

– Main is also the function which every program have.

– We can pass command line arguments to main function also.

– Arguments can be passed at the time of running the program.

– The syntax for writing main() function with command line arguments is as follows :

```
int main(int argc, char *argv[]);
```

in the above statement, argc indicates number of arguments passed to main function whereas argv[] array actually collected the pass arguments. Number of argument count includes program name also.

– Steps to execute command program are different than normal program.

– Steps in executing command line program in Ubuntu are as follows :

Step 1 : Type program

Step 2 : Save program

Step 3 : Compile program

Example : gcc -o output myprogram.cpp

Step 4 : Run program

./output first_arg second_arg third_arg, ...

– Steps in executing command line program in windows are as follows :

Step 1 : Type program

Step 2 : Compile program

Step 3 : Go command prompt and locate your program

Example : c:\tc\bin

Techknowledge Publication

Step 4 : Now execute the program using following command :

c:\tc\bin\example one two three

Note : In above example c:\tc\bin is path of program, example is name of program, one is first argument, two is second argument, three is third argument

– Program 5.12.1 illustrate command line arguments is given below :

➢ **Program 5.12.1**

```
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{
    cout<<"\n Number of arguments are :"<<argc;
    cout<<"\n Actual arguments are:";
    for(int i=0;i<argc;i++)
    {
            cout<<argv[i];
    }
    return 0;
}
```

– Program 5.12.1 can be executed using following steps (Ubuntu) :

Step 1 : Type the program

Step 2 : Save the program with .cpp extension

Step 3 : Compile the program using

gcc -o output myprogram.cpp

Step 4 : Run the program using following command :

./output first_arg second_arg

– We are passing two argument namely first_arg and second_arg

– In windows it can be executed with following steps :

Step 1 : Type program

Step 2 : Compile program

Step 3 : Go command prompt and locate your program

Example : c:\tc\bin

Step 4 : Now execute the program using following command :

c:\tc\bin\example one two three

Note : In above example c:\tc\bin is path of program, example is name of program, one is first argument, two is second argument, three is third argument.

5.13 Printer Output

– It is easy to use console based program to send data to printer.

– Many special filenames for hardware devices are defined by OS.

– Which makes it possible to treat device like files.

– Hardware device names are as follows :

- con: it is for console
- aur: first serial port
- com2: second serial port
- lpt1: first parallel printer
- lpt2: second parallel printer
- lpt3: third parallel printer
- nul: used as dummy device

– Generally, printer is connected to first parallel port of the system.

– So it is recommended to use lpt1 as filename to print through the printer.

– We have written Program 5.13.1 which will send the output to the printer.

➢ **Program 5.13.1**

```
#include<iostream>
#include<fstream>
using namespace std;
```

Techknowledge Publication

```
int main()
{
   cout<<"\n Printing output using printer";
   ofstream print ("LPT1");
//provided printer is connected to first parallel port
   if(!print)
    //it will exit with 1 if printer is not connect
   {
      cout<<"\n No printer connected";
      return 1;
   }

   print<<"\n This is first line printed";
   print<<"\n Number: "<<12345;
   print<<"\n Float: "<<345.567;
   print<<"\n char: "<<'c';

   print.close();  //close printer
   return 0;
}
```

Output

```
Printing output using printer
No printer connected
```

Important Points

- Streams
- Stream classes
- Standard stream
- Stream errors
- Files
- File handling in C++
- File handling classes
- File operations
- Reading and writing data in file
- File opening modes
- Writing class object in file
- File pointers
- Error handling in file I/O
- Overloading insertion and extraction operator
- Memory as a stream object
- Command line arguments
- Printer output

Review Questions

Q. 1 How to write multiple object in file ?

Q. 2 Can we write object in file? Write suitable program for the same.

Q. 3 Write program to read data from file ?

Q. 4 Write program to write data in file ?

Q. 5 How to detect end of file ?

Q. 6 What are different file opening modes ?

Q. 7 What are different methods to open files ?

Q. 8 What is file ?

Q. 9 What are different file operations.

Q. 10 How to handle file in C++ ?

Q. 11 What are different file handling classes in C++ ?

Q. 12 How to handle stream errors

Q. 13 What do you mean by stream ? Explain in detail.

Q. 14 What are different string manipulators ?

Techknowledge Publication

Q. 15 Write program to illustrate the use of string manipulators.

Q. 16 What is standard input, standard output, standard error stream ?

Q. 17 What is file pointer? Explain in details.

Q. 18 How to handle errors in file i/o ?

Q. 19 Write program to overload insertion and extraction operator ?

Q. 20 How to overload insertion and extraction operator to handle file ?

Q. 21 What is concept of memory as stream object ?

Q. 22 Write program for illustrating command line arguments.

Q. 23 Write program to print output of program through printer ?

❑❑❑

Techknowledge Publication

Exception Handling

Syllabus

Exception Handling : Fundamentals, Other error handling techniques, Simple exception handling - Divide by zero, Multiple catching, Re-throwing an exception, Exception specifications, User defined exceptions, Processing unexpected exceptions, Constructor, Destructor and exception handling, Exception and inheritance.

6.1 Fundamentals of Exception Handling

Q. What is the need of Exception handling ? **SPPU - May 18, 3 Marks**

Q. What is Exception ? How is an exception handled in C++ ? **SPPU - Dec. 19, 5 Marks**

- An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.
- Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**.
- **throw :** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch :** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try :** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
- Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords.
- A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows :

```
try {
  // protected code
} catch( ExceptionName e1 ) {
  // catch block
} catch( ExceptionName e2 ) {
  // catch block
} catch( ExceptionName eN ) {
  // catch block
}
```

- You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Techknowledge Publication

Throwing Exception

– Exceptions can be thrown anywhere within a code block using throw statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

– Following is an example of throwing an exception when dividing by zero condition occurs :

```
double division(int a, int b) {
  if( b == 0 ) {
    throw "Division by zero condition!";
  }
  return (a/b);
}
```

Catching Exceptions

– The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch :

```
try {
  //protected code
} catch( ExceptionName e ) {
  //code to handle ExceptionName exception
}
```

– Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows :

```
try {
  // protected code
} catch(...) {
  // code to handle any exception
}
```

6.2 Other Error Handling Techniques

– C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown in Fig. 6.2.1.

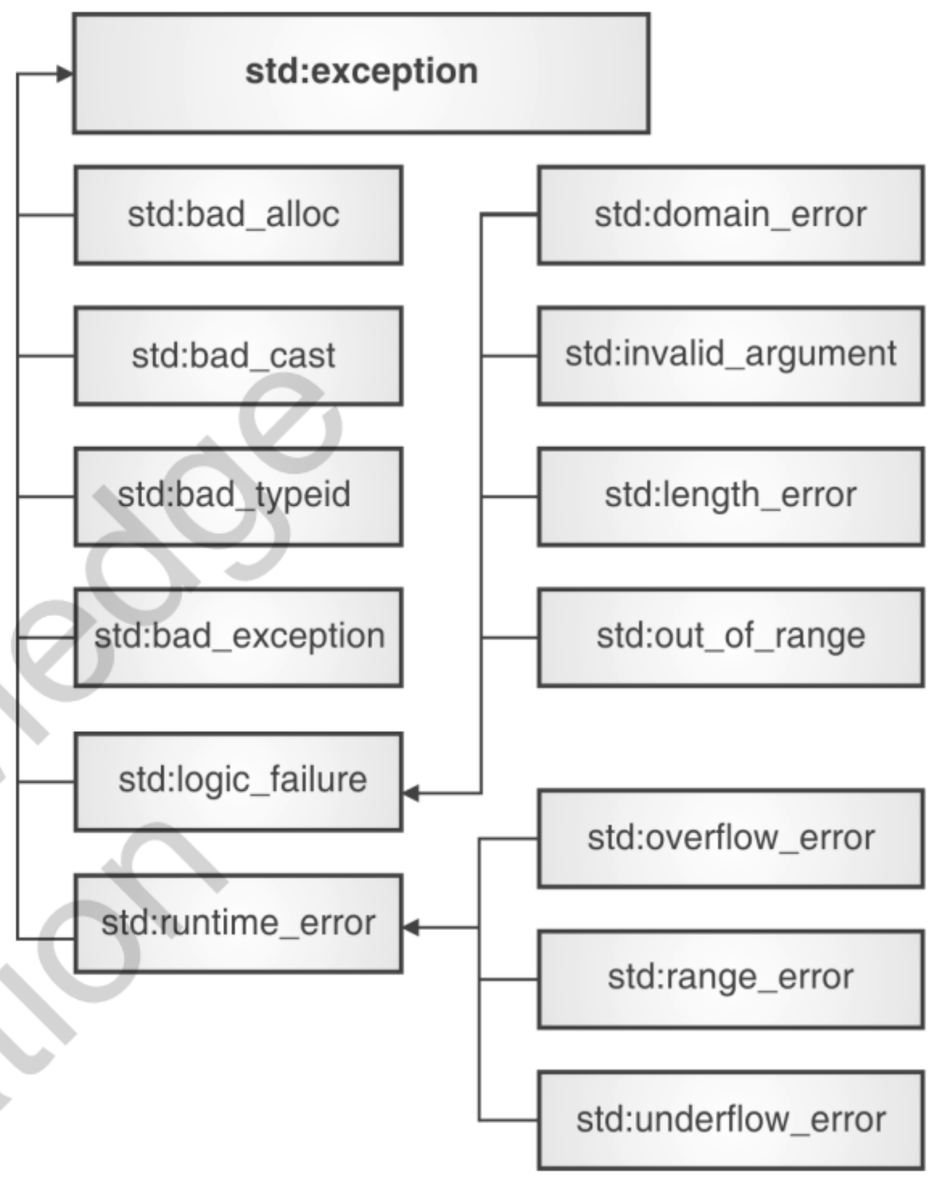

Fig. 6.2.1 : Standard exceptions

– Here is the small description of each exception mentioned in the above hierarchy :

Sr. No	Exception and Description
1.	std::exception An exception and parent class of all the standard C++ exceptions.
2.	std::bad_alloc This can be thrown by new.
3.	std::bad_cast This can be thrown by dynamic_cast.
4.	std::bad_exception This is useful device to handle unexpected exceptions in a C++ program.
5.	std::bad_typeid This can be thrown by typeid.
6.	std::logic_error An exception that theoretically can be detected by reading the code.
7.	std::domain_error This is an exception thrown when a mathematically invalid domain is used.

Tech Knowledge Publication

Sr. No	Exception and Description
8.	std::invalid_argument This is thrown due to invalid arguments.
9.	std::length_error This is thrown when a too big std::string is created.
10.	std::out_of_range This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]()..
11.	std::runtime_error An exception that theoretically cannot be detected by reading the code.
12.	std::overflow_error This is thrown if a mathematical overflow occurs.
13.	std::range_error This is occurred when you try to store a value which is out of range.
14.	std::underflow_error This is thrown if a mathematical underflow occurs.

6.3 Simple Exception Handling - Divide by Zero

Program for exception handling is given in Program 6.3.1.

➢ **Program 6.3.1**

```
#include<iostream>
using namespace std;
int main()
{
int a = 50;
int b = 0;
  double c = 0;
  try {
     if(b==0)
     {
throw(1);
     }
     c=a/b;
cout<<"\n Division is  "<<c;
}catch (inti) {
cout<<"\n Divide by zero error! error number is
   "<<i;
  }
  return 0;
}
```

Output

```
Divide by zero error! error number is 1
```

– Fig. 6.3.1 shows the working of try and catch concept if exception occurs in the program.

```
    1.  # include<iostream>
    2.  using namespace std;
→   3.  int main()
    4.  {
→   5.      int a = 50;
→   6.      int b = 0;
→   7.      double c = 0;
→   8.      try {
→   9.          if(b==0)
    10.         {
→   11.             throw(1);
    12.         }
    13.         c=a/b;
    14. cout<<"\n Division is "<<c;
→   15.     }catch(int i) {
→   16. cout <<\"n Divide by zero
        error!error number is "<<i;
    17.     }
→   18.     return 0;
    19. }
```

Fig. 6.3.1 : Working of try...catch

– As shown in the Fig. 6.3.1 we can observe that after statement 11, it directly executes statement 15 as exception occurs in the program. Arrow indicates flow of program execution.

Tech Knowledge Publications

– Program 6.3.2 illustrates the working of try.... catch if exception does not occur in the program.

➢ **Program 6.3.2**

```
#include<iostream>
using namespace std;
int main()
{
int a = 50;
int b = 22;
  float c = 0;
  try {
     if(b==0)
     {
throw(1);
     }
     c =a/b;
cout<<"\n Division is  "<<c;
}catch (int i) {
cout<<"\n  Divide  by  zero  error!  error  number  is
    "<<i;
  }
cout<<"\n Rest of the program";
  return 0;
}
```

Output

```
Division is 2
Rest of the program
```

Fig. 6.3.2 shows the execution of the Program 6.3.2.

```
      1.  #include<iostream>
      2.  using namespace std;
  →   3.  int main()
      4.  {
  →   5.      int a = 50;
  →   6.      int b = 22;
  →   7.      float c = 0;
  →   8.      try {
  →   9.          if(b==0)
     10.          {
     11.              throw(1);
     12.          }
  →  13.          c=a/b;
  →  14.  cout<<"\n Division is "<<c;
     15.      }catch(int i) {
     16.  cout <<"\n Divide by zero error!error number is "<<i;
     17.      }
  →  18.  cout<<"\n Rest of the program";
  →  19.      return 0;
     20.  }
```

Fig. 6.3.2 : Program execution

– As we can see in the Fig. 6.3.2 after statement 14, it directly executes statement 18 as there is no exception in the program. If exception does not occurs, then catch block will not get executed. Arrow in the Fig. 6.3.2 indicates the flow of program execution.

– Another exception handling program, in this program exception occurs when we try to access array element after the limit.

➢ **Program 6.3.3**

```
#include<iostream>
using namespace std;
int main()
{
int a[5];
int i;
cout<<"\n Enter elements";
   for(i=0;i<5;i++)
cin>>a[i];
try{
a[10]=100;
cout<<"\n Element at 10 position is :"<<a[10];
}catch(int i){
cout<<"\n  Exception  occurs!  array  index  out  of
        bound";
   }
   return 0;
}
```

Output

```
Enter elements 10 20 30 40 50
Exception occurs! array index out of bound
```

Techknowledge Publication

Explanation : *As we can see in the Program 6.3.3 size of array is 5 so only 0-4 index positions are available. 10 index position is not available so it throws the exception.*

6.4 Multiple Catching

Q. How to handle multiple exceptions occurred in a program? **SPPU - May 14, 5 Marks**

Q. Write a short note on multiple exception handling with the help of examples. **SPPU - May 16, 5 Marks**

Handle multiple exceptions

- Whenever exception happens exception handler searches for appropriate handler and catches the exception.
- One try can have multiple catch blocks but many try block cannot have one catch block.
- Example of Multiple exception is given in Program 6.4.1.

➢ **Program 6.4.1 : With multiple exception**

```
#include<iostream>
using namespace std;
int main()
{
int a;
cout<<"\n Example of multiple exception";
cout<<"\n Enter the no  ";
cin>>a;
   try {
     if(a==1)
     {
        throw 10;
     }
     if(a==2)
     {
        throw float (20.4);
     }
     if(a==3)
     {
        throw 'x';
     }
cout<<"\n Exception does not occurs";
   }
catch(int a) {
cout<<"\n This is first exception and value is "<<a;
   }
catch(float a) {
cout<<"\n This is first exception and value is "<<a;
   }
catch(char a) {
cout<<"\n This is first exception and value is "<<a;
   }
   return 0;
}
```

Output

```
Example of multiple exception
Enter the no  2
This is first exception and value is 20.4
```

6.5 Re-throwing an Exception

- Exception handler may decide to rethrow exception without handling it.
- In this situation we can call following statement.

```
throw;
```

- No argument should be passed to throw keyword.
- throw statement causes current exception to be thrown to next try catch clause. Exception is caught by next enclosing try ... catch blocks.
- Program 6.5.1 illustrate the rethrowing exception.

➢ **Program 6.5.1 (without rethrow)**

```
#include<iostream>
using namespace std;
int main()
{
cout<<"\n Rethrowing exception example";
  try
  {
```

Tech Knowledge Publications

```
        try {
                throw 10;
        }
catch(int a){
cout<<"\n Inner try...catch";
        }
  }
catch(int a)
  {
cout<<"\n Exception is caught by outer try...catch
        block"<<a;
  }

cout<<"\n rest of the program";
      return 0;
}
```

Output

```
Rethrowing exception example
Inner try...catch
rest of the program
```

Explanation : *Only inner try…catch block handles the exception and not the outer try….catch block.*

➢ **Program 6.5.2 (with rethrow)**

```
#include<iostream>
using namespace std;
int main()
{
cout<<"\n Rethrowing exception example";

  try
  {
      try {
              throw 10;
      }
catch(int a){
cout<<"\n Inner try...catch";
          throw;
      }
  }
catch(int a)
  {
cout<<"\n Exception is caught by outer try...catch
block"<<a;
  }

cout<<"\n rest of the program";
      return 0;
}
```

Output

```
Rethrowing exception example
Inner try...catch
Exception is caught by outer try...catch block10
rest of the program
```

Explanation : *As we can see inner try….catch block just passes exception and it is again caught in the outer try….catch block.*

➢ **Program 6.5.3**

```
#include<iostream>
using namespace std;
void test()
{
  try
  {
     throw "\n Try catch example";
  }
  catch (const char*)
  {
cout<<"\n Inner try...catch block";
  throw;
//we are rethrowing the exception to outer try...catch
//block.
  }
}
int main()
{
cout<<"\n Rethrowing exception example";
  try
  {
test();
  }
```

Techknowledge Publication

```
catch(const char*)
  {
cout<<"\n Exception is caught by outer try...catch
        block";
  }
cout<<"\n rest of the program";
     return 0;
}
```

Output

```
Rethrowing exception example
Inner try...catch block
Exception is caught by outer try...catch block
rest of the program
```

Explanation : *We can see inner try catch block just passes the exception to outer try catch. After inner try...catch block, outer try...catch block handles the exception.*

6.6 Exception Specifications

- If we wanted to throw only certain exception then also it is possible.
- In this case exception will be thrown only for specified conditions and not for all the conditions.
- It is possible by adding throw list clause during function definition.
- The syntax for specifying the exception is given below.
- Throw list clause can be added during function definition.
- The syntax for creating such exception is as follows :

```
Return_typefunction_name(arguments) throw (type list)
    {
    //body of the function
    }
```

- In the type list we can mention exception which we wanted to throw by given function.
- If we throw any other exception then abnormal program termination occurs.
- We can prevent program from throwing exception by mentioning empty list in throw.
- Example of empty throw list is as follows which guarantees not to throw any type of exception.

```
throw();            //empty list
```

- Program 6.6.1 illustrates the concept of exception.

➢ **Program 6.6.1**

```
#include<iostream>
using namespace std;
void example (int a) throw (int, double)
{
if (a==0) throw 'a';        //char
else
if (a==1) throw a;          //int
else
if (a==-1) throw 231.30; //double

}

int main()
{
try
{
cout<<"testing throw restrictions\n";
cout<<"x== 10\n ";
example(10);
cout<<"x==1 \n";
example(1);
cout<<"x== -1 \n";
example(-1);
cout<<"x==2 \n";
example(2);
}
catch( char c)
{
cout<<"caught a character \n";
}
catch(int m)
```

Technical Knowledge Publication

```
{
cout<<"caught an integer \n";
}
catch (double d)
{
cout<<"caught a double \n";
}
cout<<" end of try catch system \n \n";

return 0;
}
```

Output

```
testing throw restrictions
x== 10
 x==1
caught an integer
 end of try catch system
```

Explanation : *As we can see only int and double exception can be handled by Program 6.6.1. If we try to throw char exception then abnormal program termination occurs. Program 6.6.2 shows the case of abnormal program termination.*

– Program 6.6.2 illustrates abnormal program termination.

➢ **Program 6.6.2**

```
#include<iostream>
using namespace std;
void example (int a) throw (int,double)
{
if (a==0) throw 'a';          //char
else
if (a==1) throw a;            //int
else
if (a==-1) throw 231.30; //double

}
int main()
{
Try
{
cout<<"testing throw restrictions\n";
cout<<"x== 10\n ";
example(0);
cout<<"x==1 \n";
example(1);
cout<<"x== -1 \n";
example(-1);
cout<<"x== 2 \n";
example(2);
}
catch( char c)
{
cout<<"caught a character \n";
}
catch(int m)
{
cout<<"caught an integer \n";
}
catch (double d)
{
cout<<"caught a double \n";
}
cout<<" end of try catch system \n \n";
return 0;
}
```

Output

```
testing throw restrictions
x== 10
```

Explanation : *As we can see from main program we have passed 0 to example function. It should have throw character exception. But we have not mentioned character exception in throw list so abnormal program termination occurs.*

Program 6.6.3 illustrate with empty throw statement.

Tech Knowledge Publication

➢ **Program 6.6.3**

```
#include<iostream>
using namespace std;
void example (int a) throw ()
{
if (a==0) throw 'a';        //char
else
if (a==1) throw a;          //int
else
if (a==-1) throw 231.30; //double
}
int main()
{
try
{
cout<<"Specifying exception\n";
cout<<"x==0\n ";
   example(0);
cout<<"x==1 \n";
example(1);
cout<<"x== -1 \n";
example(-1);
cout<<"x== 2 \n";
example(2);
}
catch( char c)
{
cout<<"caught a character \n";
}
catch(int m)
{
cout<<"caught an integer \n";
}
catch (double d)
{
cout<<"caught a double \n";
}
cout<<" end of try catch system \n \n";
return 0;
}
```

Output

```
Specifying exception
x==0
```

Explanation : *As we can see, no exception is handled and program terminated abnormally. It is because we have not mentioned any type in throw.*

6.7 User Defined Exception

- User Defined Exception or custom exception is creating your own exception class and throws that exception using 'throw' keyword. This can be done by extending the class Exception.
- The following C++ program demonstrates user-defined exceptions. The program derives an exception class from the standard exception class named Divide_By_Zero_Exception. The try block throws the Divide_By_Zero_Exception object and catch block handles it by executing the what() member function.

➢ **Program 6.7.1**

```
/* C++ Program to demonstrate user-defined exceptions */
#include<iostream>
#include<exception>
using namespace std;
 class Divide_By_Zero_Exception : public exception{
   public:
      const char * what() const throw()
      {
         return "Divide By Zero Exception\n";
      }
};
int main()
{
   try
   {
      int a, b;
      cout<<"Enter two numbers : ";
      cin>>a >> b;
```

Techknowledge Publications

```
        // compute a / b
        if (b == 0)
        {
            Divide_By_Zero_Exception d;
            throw d;
        }
        else
        {
            cout<<"a / b = " << a/b << endl;
        }
    }
    catch(exception& e)
    {
        cout<<e.what();
    }
}
```

Output

```
Enter two numbers : 10 2
a / b = 5
Enter two numbers : 1 0
Divide By Zero  Exception
```

➢ **Program 6.7.2 : Create a user defined exception to check whether your employee exist in your data structure (use any data structure to store the employees) and throw exception if name is not in the employees list. Use the catch block to print an appropriate message on the screen.**

SPPU - May 18, Dec. 18, 5 Marks

In java we can create our own exception class and throw that exception using throw keyword.

These exceptions are known as **user-defined** or **custom** exceptions.

```
class MyException extends Exception{
  String str1;
MyException(String str2) {
    str1=str2;
  }
  public String toString(){
    return ("MyException Occurred: "+str1) ;
  }
}
class Example1{
  public static void main(String args[]){
    try{
     throw new MyException("Employee not found");
    }
    catch(MyException exp){
        System.out.println(exp);
    }
  }
}
```

Output

```
MyException Occurred: Employee not found
```

6.8 Processing Unexpected Exceptions

- Even after handling exceptions, there may be unexpected exceptions.
- Unexpected exception terminates the program.
- In this case stack will unwound.
- If no exception handler found then program terminate abnormally.
- It is hard to write to write exception safe code.
- Always use exception when there is really need.
- If some error we could not handle then don't use exception.
- Consider a case where function is for addition of two numbers. Then it is unnecessary to handle exception in this situation.
- The process of removing function from function stack during runtime is called as stack unwinding.
- In C++, when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack. So exception handling involves Stack Unwinding if exception is not handled in same function
- Stack unwinding program is given Program 6.8.1.

Techknowledge Publication

➢ **Program 6.8.1**

```
#include<iostream>
using namespace std;
void fun1() throw (int) {
cout<<"\n This is start of fun1()";
 throw 235;
cout<<"\n This is end of fun1()";

}
void fun2() throw (int) {
cout<<"\n This is start of fun2()";
 fun1();
cout<<"\n This is end of fun2() ";

}
void fun3() {
cout<<"\n This is start fun3()";
 try {
  fun2();

 }
catch(int a) {
cout<<"\n Exception has been Caught: "<<a;
 }
cout<<"\n This is end of fun3()";
}
int main() {
 fun3();
 return 0;
}
```

Output

```
This is start fun3()
This is start of fun2()
This is start of fun1()
Exception has been Caught: 235
This is end of fun3()
```

6.9 Constructor, Destructor and Exception Handling

Q. Explain how exception is handled in case of constructor and destructor with example.

SPPU - May 19, 4 Marks

- Whenever exception is thrown control passes to catch block, destructors are called automatically for all objects constructed since the beginning of try block which are associated with catch block.
- If exception is thrown at the time of creating object which consists of subobjects then destructor are only called for those sub objects which are successfully created before throwing the exception.
- A destructor for a local static object will only be called if the object was successfully constructed.
- If a destructor detects an exception and issues a throw, the exception can be caught if the caller of the destructor was contained within a try block and an appropriate catch is coded.
- If an exception is thrown by a function called from an inner try block, but caught by an outer try block, all objects constructed within both the outer and all inner try blocks are destroyed.
- If you throw an object with a constructor, you can construct an object that contains information relevant to the catch expression.
- The catch block can then access information provided by the thrown object.
- Exception handling can be used in conjunction with constructors and destructors to provide resource management that ensures that all locked resources are unlocked when an exception is thrown.

Tech Knowledge Publication

6.10 Exception and Inheritance

- Exception should be handled carefully in inheritance.
- If both base and derived class caught exception then catch block of derived class must appear before the base class.
- If we put base class catch block first then derived class catch block will never reach.
- It means catch block of derived class should be before catch block of base block.

Incorrect code

➢ **Program 6.10.1**

```
#include<iostream>
using namespace std;

class parent {};
class child: public parent {};
int main()
{
  child c;
  try {
cout<<"\n This is example of try....catch in
        inheritance";
     throw c;
  }
catch(parent p) {
cout<<"\n This is catch block for parent exception";
  }
catch(child c) {
cout<<"\n This is catch block for child exception";
  }
  return 0;
}
```

Explanation : *Program 6.10.1 code will give the warning of unreachable code.*

Correct code

➢ **Program 6.10.2**

```
#include<iostream>
using namespace std;

class parent {};
class child: public parent {};
int main()
{
  child c;

  try {
cout<<"\n This is example of try....catch in
        inheritance";
     throw c;
  }
catch(child c) {
cout<<"\n This is catch block for child exception";
  }
catch(parent p) {
cout<<"\n This is catch block for parent exception";
  }
  return 0;
}
```

Output

```
This is example of try....catch in inheritance
This is catch block for child exception
```

Important Points

- Exception handling
- Try
- Catch
- Throw
- Multiple catch statement
- Rethrowing exception
- Unexpected exception
- Own exception
- Exception handling and inheritance

Techknowledge Publications

Review Questions

Q. 1 What is exception handling?

Q. 2 Explain try…catch and throw keyword?

Q. 3 Write program for handling divide by zero exception?

Q. 4 Write program with multiple catch statement?

Q. 5 What is catch all exceptions?

Q. 6 What do you mean by rethrowing exception?

Q. 7 What is exception specification?

Q. 8 How to process unexpected exception?

Q. 9 Explain exception handling in program with constructor and destructor?

Q. 10 How to define own exception? Write suitable program for the same?

Q. 11 Explain exception handling in inheritance?

❑❑❑

Techknowledge Publication

CHAPTER 7

Unit V

Templates

Syllabus

Templates : The power of templates, Function template, Overloading function templates, and class template, Class template and nontype parameters, Template and friends generic functions, The type name and export keywords.

7.1 Introduction to Generic Programming

Q. Explain Generic Programming. **SPPU - May 14, 3 Marks**

Q. What is generic programming? How is it implemented in c++? **SPPU - May 17, 5 Marks Dec. 17, 7 Marks, Dec. 18, 2 Marks**

Generic Programming

- It is type of computer programming. In this type of programming we are designing the algorithm; where in type of data is specified at run time.
- We are writing the generalized code and compiler decides for what type of data it should work.
- Template is best example of generic programming in C++. In generic programming we are writing the algorithms which are applicable to variety of data types.
- In traditional programming language we need to use different algorithms for different data types. For example, program for performing the arithmetic operations on integer number will not be applicable for performing arithmetic operations on floating point numbers.
- We cannot use same module for performing the operations. We need to create separate copy of functions with some changes.
- Consider the Program 7.1.1.

➢ **Program 7.1.1 (integer version) : Write a program to handle addition of two numbers using template** **SPPU - Dec. 19, 4 Marks**

```
#include<iostream>
using namespace std;
int main()
{
cout<<"\n Addition of two number";
int a=10, b=20;
int c;
   c=a+b;
cout<<"\n Addition of two number is :"<<c;
   c=a-b;
cout<<"\n Sub of two number is :"<<c;
}
```

Output

```
Addition of two number
Addition of two number is :30
Sub of two number is :10
```

Technical Knowledge Publication

***Explanation** : We can do the addition of only integers. Addition of floating point number is not possible with this code. We need to create separate program performing the floating point operations with some changes.*

➢ **Program 7.1.2 : (Floating point version)**

```
#include<iostream>
using namespace std;
int main()
{
cout<<"\n Addition of two number";
   float a=10.5, b=20.3;
   float c;
   c=a+b;
cout<<"\n Addition of two number is :" <<c;
   c=a-b;
cout<<"\n Sub of two number is :"<<c;
}
```

Output

```
Addition of two number
Addition of two number is :30.8
Sub of two number is :9.8
```

***Explanation** : Program 7.1.2 will perform only floating point operations.*

– In Program 7.1.2, we need to create two different programs for integer and floating point numbers. If again data type changes we need to create another program.

– We can solve the above problem using concept of generic programming.

– Generic programming is approach of creating generalize algorithms which are applicable for all data types.

– For example, we design general algorithm for performing arithmetic operations. Same algorithms will be applicable to all data types. Which data type to use will be decided at the time calling the functions.

– The Program 7.1.3 is concept of generic programming. It performs all type of arithmetic operations. This example is with templates.

➢ **Program 7.1.3**

```
#include<iostream>
using namespace std;
template<typename T>
void add(T a, T b)
{
cout<<"\n Addition of two number is :"<<a+b;
}
template<typename T>
void sub(T a, T b)
{
cout<<"\n Sub of two number is :"<<a-b;
}
int main()
{
cout<<"\n Example of generic program";
   //integer operations
   add<int>(10,20);
   sub<int>(50, 20);

   //floating point operations
   add<float>(10.4, 30.6);
   sub<float>(30.4, 60.6);
   return 0;
}
```

Output

```
Example of generic program
Addition of two number is :30
Sub of two number is :30
Addition of two number is :41
Sub of two number is :-30.2
```

***Explanation** : Program 7.1.3 is capable of performing all type of addition and subtraction.*

– Standard template library is another example of generic programming.

Tech Knowledge Publication

– Generic programming in C++ is implemented in templates, container classes, standard template library, etc.

7.2 Power of Templates

Q. What is template? **SPPU - Dec. 19, 2 Marks**

– Templates are newly added feature in C++.
– It allows us to define generic classes and functions.
– It provides the support for generic programming.
– Generic programming is concept where in generic parameters are used in the algorithms.
– With this concept same algorithms can be used for different data types.
– Templates are useful in achieving dynamic polymorphism.
– For example we can create generic search algorithm which is applicable for all data types like char, int, float, etc. In traditional case we need to use different algorithm for int, char and floating point search operations.
– Template is way of creating generalize functions and classes which are applicable for all data types without changing the definition of the program.
– They are used for creating family of functions and classes.
– Template can be function template or class templates.

➢ **Program 7.2.1 : Write a function template for finding the minimum value contained in an array.**

SPPU - Dec. 14, Dec. 19, 3 Marks

```
#include<iostream>
using namespace std;
template<typename T>
void MAX(T a[ ], int n)
{
   T m;
   m=a[0];
   for(int i=0;i<n;i++)
      if(m<a[i])
         m=a[i];
   cout<<"\n Largest number is : "<<m;
}
int main()
{
   int n, i;
   int a[10];
   cout<<"\n Finding largest number from int";
   cout<<"\n How many elements want to enter";
   cin>>n;
   cout<<"\n Enter int elements: ";
   for(i=0;i<n;i++)
      cin>>a[i];
   MAX<int>(a, n);

   float b[10];
   cout<<"\n Finding largest number from float";
   cout<<"\n How many elements want to enter";
   cin>>n;
   cout<<"\n Enter floating point numbers: ";
   for(i=0;i<n;i++)
      cin>>b[i];
   MAX<float>(b, n);
   return 0;
}
```

Output

```
Finding largest number from int
How many elements want to enter5

Enter int elements: 5 4 3 2 1
Largest number is : 5
Finding largest number from float
How many elements want to enter5

Enter floating point numbers: 5.5    599.44    33.22
222.3  4.4

Largest number is : 599.44
```

Tech Knowledge Publications

➢ **Program : 7.2.2 : Write a template function that returns the average of all the elements of an array. The arguments to the function should be the array name and the size of the array (type int). In main(), Exercise the function with arrays of type int, long, double and char.**

SPPU - Dec. 18, 5 Marks

– C++ template function that returns the average of all the elements in an array. The arguments to the function should be the array name and the size of the array.

```
template<typename T>
T average (T const* t, int n)
 {
   T s = t[n - 1];
   for (int i = 0; i < (n-1); i++)
      s += t[i];
   return s / n;
}
```

7.2.1 Need of Templates in C++

Q. Why we need Templates in C++ ?

SPPU - Dec. 16, May 19, 2 Marks

Templates are needed in C++ programming language because it t allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one. You create only one generic version of your class or function instead of manually creating specializations.

Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.

7.3 Function Template / Applying Generic Function

Q. Explain function template with an example.

SPPU - Dec. 16, May 19, 2 Marks

Q. How function overloading and function templates are related to each other ? State valid example to justify your answer.

SPPU - May 18, Dec. 18, 4 Marks

– General syntax for creating function template is as follow :

```
template<typename T>
return_typefunction_name(T type argument)
{
   ----
   ----
}
```

– In the above syntax T is generic variable. We set any data type for T. Datatype of T is decided at the time of calling the function. T is user-defined name and can be some other name also.

– We can pass more than one argument to function template also.

– You generally use templates when you want to do the same set of operations on many different data types.

– You generally would use function overloading when you want to do different operations on certain sets of data.

– Templates cannot take varying numbers of arguments. But Function Overloading can take varying number of arguments.

– In addition, a template indicates that you can operate on any data type, but it's pointless to represent this when in fact, the vast majority of templates would be specializations only.

– Also, overloads can be virtual and template specializations cannot.

– Syntax of function calling with function template is as follows :

```
function_name<data_type>( arg1, arg2, ....);
```

Example

```
add<int> (10, 20);
add<float> (20.4, 30.5);
```

– Program 7.3.1 illustrates the concept of function template and class template.

Technknowledge Publication

➢ **Program 7.3.1 : Function Template**

```
#include<iostream>
using namespace std;

template<typename T>
void MAX(T a, T b)
{
   if(a>b)
cout<<"\n First number is greater";
   else
cout<<"\n Second number is greater";
}
int main()
{
cout<<"\n Example of function template";
cout<<"\n integer comparison";
   MAX<int>(110,20);
cout<<"\n Floating point comparison";
   MAX<float>(-51.5, -50.3);
   return 0;
}
```

Output

```
Example of function template
integer comparison
First number is greater
Floating point comparison
Second number is greater
```

➢ **Program 7.3.2 : Another example of function template**

```
#include<iostream>
using namespace std;
template<typename T>
void exchange(T &a, T &b)
{
   T temp;
   temp=a;
   a=b;
   b=temp;
}
int main()
{
int a=10, b=20;
   float c=30.4, d=50.3;
cout<<"\n Example of function template";
cout<<"\n integer exchange of value";
cout<<"\n Before exchange value of a and b is
      "<<a<<"  "<<b;
   exchange<int>(a, b);
cout<<"\n After exchange value of a and b is
      "<<a<<"  "<<b;
cout<<"\n floating point number exchange of value";
cout<<"\n Before exchange value of c and d is
      "<<c<<"  "<<d;
   exchange<float>(c, d);
cout<<"\n After exchange value of c and d is
      "<<c<<"  "<<d;
   return 0;
}
```

Output

```
Example of function template
integer exchange of value
Before exchange value of a and b is 10   20
After exchange value of a and b is 20   10
floating point number exchange of value
Before exchange value of c and d is 30.4   50.3
After exchange value of c and d is 50.3   30.4
```

– Program 7.3.3 illustrates the function template for finding the minimum value contained in an array .

➢ **Program 7.3.3 : Write a function template for finding the minimum value contained in an array.** **SPPU - Dec. 19, 3 Marks**

```
#include<iostream.h>
#include<conio.h>
template <class T>
T findMin(T arr[],int n)
{
   int i;
   T min;
   min=arr[0];
   for(i=0;i<n;i++)
   {
      if(min > arr[i])
```

Techknowledge Publications

```
        min=arr[i];
    }
    return(min);
}
void main()
{
    clrscr();
    int iarr[5];
    char carr[5];
    double darr[5];
    cout<<"Integer Values \n";
    for(int i=0; i < 5; i++)
    {
      cout<<"Enter integer value "  << i+1 << " : ";
       cin>>iarr[i];
    }
    cout<<"Character values \n";
    for(int j=0; j < 5; j++)
    {
     cout<<"Enter character value " << j+1 << " : ";
       cin>>carr[j];
    }
    cout<<"Decimal values \n";
    for(int k=0; k < 5; k++)
    {
      cout<<"Enter decimal value " << k+1 << " : ";
       cin >> darr[k];
    }
    //calling Generic function...to find minimum value.
    cout<<"Generic Function to find Minimum from
            Array\n\n";
    cout<<"Integer Minimum is :
    "<<findMin(iarr,5)<<"\n";
    cout<<"Character Minimum is :
    "<<findMin(carr,5)<<"\n";
    cout<<"Double Minimum is :
    "<<findMin(darr,5)<<"\n";
    getch();
}
```

7.4 Overloading Function Templates

Q. What is overloading template ? Explain it with an example. **SPPU - May 15, 4 Marks**

Q. How function overloading and function templates are related to each other ? State valid example to justify your answer. **SPPU - May 18, Dec. 18, 4 Marks**

Overloading template

- Template functions can be overloaded.
- It is possible using two options that is template functions or ordinary functions.
- In this situation call to the functions to be very careful as there may be confusion between function template and ordinary functions.
- During function calling if there is no match for ordinary and template function then error will be generated.
- Function overloading is used similarly as function templates, but allows you to use different syntax to carry out the operations for different types. That is to say that, you can represent the values in different ways.
- One obvious example would be what are called **atan** and **atan2** in the C library. With **atan**, we pass the ratio of the "rise to the "run", and we get back the angle that ratio represents. With **atan2** we pass the values for the rise and run separately (which computes roughly the same result, but since it's given slightly more input data, can produce a more complete result as well).
- Program 7.4.1 illustrates overloading function templates.

➢ **Program 7.4.1**

```
#include<iostream>
using namespace std;
 template <class T>
 void print(T a){
```

Techknowledge Publication

```
cout<<"\n This function is with function template:
   "<<a;
 }
 void print(int a){
cout<<"\n This function is ordinary function with int
        parameter" <<a;
 }
int main(){
print(100);
print(200);
print(123.34);
    print ("Message");
   // print (10, 20);
//won't executed because no matching function found
    return 0;
 }
```

Output

```
This function is ordinary function with int
parameter100
This function is ordinary function with int
parameter200
This function is with function template: 123.34
This function is with function template: Message
```

7.5 Overloading Class Template

– A class template defines a family of classes. The syntax for the class template is given as follows :

Syntax

```
template < parameter-list > class-declaration
```

– **class-declaration :** The class name declared becomes a template name.

– **parameter-list :** a non-empty comma-separated list of the template parameters, each of which is either a non-type parameter, a type parameter, a template parameter, or a parameter pack of any of those.

– Member function in a class templates may be overloaded.

– A non-template function is always distinct from a template specialization with the same type. Specializations of different class templates are always distinct from each other even if they have the same type.

– Two class templates with the same return type and the same parameter list are distinct and can be distinguished with explicit template argument list.

– When an expression that uses type or non-type template parameters appears in the function parameter list or in the return type, that expression remains a part of the function template signature for the purpose of overloading.

➢ **Program 7.5.1**

```
#include<iostream>
using namespace std;

int sum (int a, int b)
{
 return a+b;
}

double sum (double a, double b)
{
 return a+b;
}

int main ()
{
 cout<<sum (10,20) << '\n';
 cout<<sum (1.0,1.5) << '\n';
 return 0;
}
```

– Here, sum is overloaded with different parameter types, but with the exact same body.

– The function sum could be overloaded for a lot of types, and it could make sense for all of them to have the same body.

Technknowledge Publication

– For cases such as this, C++ has the ability to define functions with generic types, known as function templates.

7.6 Class Template and Nontype Parameters

Q. Describe type template parameters and non-type template parameters? **SPPU - Dec. 15, 4 Marks**

Type template parameters and non-type template parameters

– Template can have many arguments.

– It is possible to use nontype arguments also.

– It means combination of template and normal arguments can be there.

– Normal arguments can be string, char, int, etc.

➢ **Program 7.6.1**

```
#include<iostream>
using namespace std;
 template <class T, int MAX>
 class example
 {
      T arr[MAX];
   public:
      void accept()
      {
cout<<"\n Enter "<<MAX<<"  elements :";
for(int i=0;i<MAX;i++)
cin>>arr[i];
      }
      void display()
      {
cout<<"\n Content of array are :";
for(int i=0;i<MAX;i++)
cout<<"\t"<<arr[i];
      }
 };
int main(){
                                //deal with float array
    example<float, 5> f;
f.accept();
f.display();
   return 0;
 }
```

Output

```
Enter 5 elements :
10.5
20.6
30.7
40.8
50.9
Content of array are :  10.5    20.6    30.7    40.8
50.9
```

Explanation : *In the Program 7.6.1 non type argument was size of array.*

7.7 Template and Friends Generic Functions

– When writing C++, there are some common situations where you may want to create a friend function, such as implementing operator <<. These are straightforward to write for non-template classes, but templates throw a real wrench in the works here. In C++, there are a few different ways to set up template classes with friend functions.

– The friend declaration appears in a class body and grants a function or another class access to private and protected members of the class where the friend declaration appears.

Techknowledge Publication

– In the following way, we can declare a template class as friend :

Example

```
template<typename T>
struct try
{
   template<typename S>
   friend struct try<S>;
private:
};
```

– Both function template and class template declarations may appear with the friend specifier in any non-local class or class template (although only function templates may be defined within the class or class template that is granting friendship). In this case, every specialization of the template becomes a friend, whether it is implicitly instantiated, partially specialized, or explicitly specialized.

```
class A {
   template<typename T>
   friend class B;      // every B<T> is a friend of A
   template<typename T>
   friend void f(T) {} // every f<T> is a friend of A
};
```

7.8 The Type Name and Export Keywords

– In a template declaration, **typename** can be used as an alternative to class to declare type template parameters and template parameters.

– Inside a declaration or a definition of a template, **typename** can be used to declare that a dependent qualified name is a type.

– Inside a declaration or a definition of a template, **typename** can be used before a non-dependent qualified type name. It has no effect in this case.

– Inside a requirements for type requirements.

– Following is a method for indicating that a dependent name is a type :

Example

```
//Define a generic function that returns the greater of
//its two arguments
template<typename T>
const T& max(const T& x, const T& y)
{
 if (y < x)
   return x;
 return y;
}
```

– The **export** keyword was originally meant to permit the separation of the definition of a template from its usages. It proved to be incredibly difficult to implement.

– An exported declaration must actually declare something.

– The first declaration of an exported entity must be an exported declaration. Subsequent declarations and definitions need not have the **export** keyword:

```
export class Thing;  // Good
export class Thing;  // Okay, but redundant
class Thing;         // Implicit 'export' keyword
class Thing {        // Implicit 'export' keyword
   int a;
   int b;
};
class SomethingElse; // Good. Not exported.
export class SomethingElse;
                 // Illegal! First declaration  is not exported!
```

Important Points

– Templates

– Power of template

– Generic programming

– Function template

– Class template

Techknowledge Publications

- Function overloading
- Overloading function template
- Class template and Nontype parameters

Review Questions

Q. 1 What is generic programming?

Q. 2 What is template? Write program to handle addition of two numbers using template?

Q. 3 What is function template?

Q. 4 What is class template?

Q. 5 Write program for handling stack using templates?

Q. 6 What is template with multiple arguments?

Q. 7 What is power of template

Q. 8 What is function overloading?

Q. 9 Compare function overloading and templates?

Q. 10 How to overload the function template?

Q. 11 What is Class template and Nontype parameters?

❑❑❑

Techknowledge Publication

CHAPTER 8

Unit - VI

Standard Template Library (STL)

Syllabus

Introduction to STL, STL Components, Containers - Sequence container and associative containers, Container adapters, Application of Container classes : vector, list.

Algorithms : Basic searching and sorting algorithms, Min-max algorithm, Set operations, Heap sort, **Iterators :** Input, Output, Forward, Bidirectional and random access. Object Oriented Programming -a road map to future.

Technknowledge Publication

8.1 Introduction to Standard Template Library (STL)

Q. What is STL? Why should a C++ programmer be interested in the STL? What is the design philosophy of the STL? What are the major components of the STL?

SPPU - Dec. 16, May 17, Dec. 17, May 18, Dec. 18, 7 Marks, Dec. 19, 3 Marks

- Standard template library is newly added concept in C++.
- We already know how to create generic programming using templates.
- With STL it is again very simple to do the programming.
- Many predefined functions are provided by STL to help manipulate many data structures.
- It is nothing but collection of functions and classes.
- It contains many functionalities.
- It saves considerable time and space in program development.
- It helps in creating high qualities programs and software.
- It is part of generic programming.
- Three main components of STL are algorithms, containers, iterators.
- It provides build in common classes in C++.
- Algorithms used in STL are independent from data types. They are not designed for specific data types.
- They provide library for many frequently used classes like stack, queue, array, etc.
- They use template concept for its implementation.
- STL ease the programming by providing readymade facilities for frequently used data types.
- We are able to achieve concept of reusability with the help of STL.

8.1.1 Components of STL

- STL is build on different components.
- These are building blocks of STL.
  - Containers

- Algorithms
- Iterators

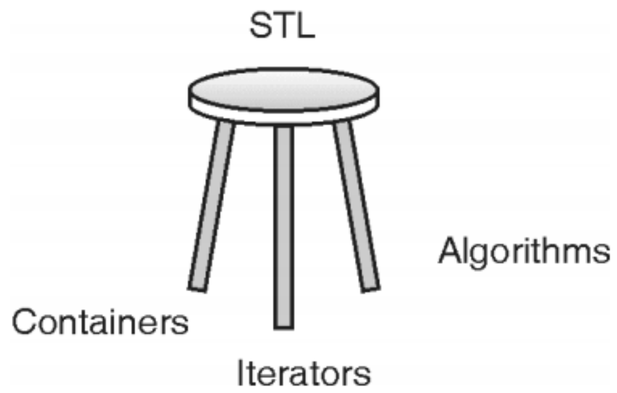

Fig. 8.1.1 : Components of STL

8.2 Containers

Q. List different types of STL containers. **SPPU - May 17, Dec. 19, 2 Marks**

Q. What is container? List the container classes in C++. **SPPU - May 17, May 19, Dec. 19, 4 Marks**

Q. What is the purpose of Container. **SPPU - Dec. 17, 2 Marks**

– Container classes are holders which is capable to store the collection of object.

– They are used to manage collection of objects.

– They are implemented as class templates.

– They do all the storage management of object they are holding in the containers.

– They provides list of functions to access and manipulate the contents of container.

– They provide iterator to access the elements of container.

– They provide many predefined containers classes for stack, queue, priority queue, list, set, map, etc. It is part of generic programming and templates.

– Some of the most common containers are as follows :
  - Array
  - Vector
  - List
  - Deque (double ended queue)
  - Stack
  - Queue
  - Priority queue
  - Set
  - Map

8.2.1 Sequence Container and Associative Containers

Q. Elaborate advantages and disadvantages of the following Basic Sequence Containers : (i) Vector (ii) List **SPPU - Dec. 16, May 18, Dec. 18, 3 Marks**

– The way information is stored and manipulated decides the type of containers.

– It can be sequence container, associative containers and derived containers.

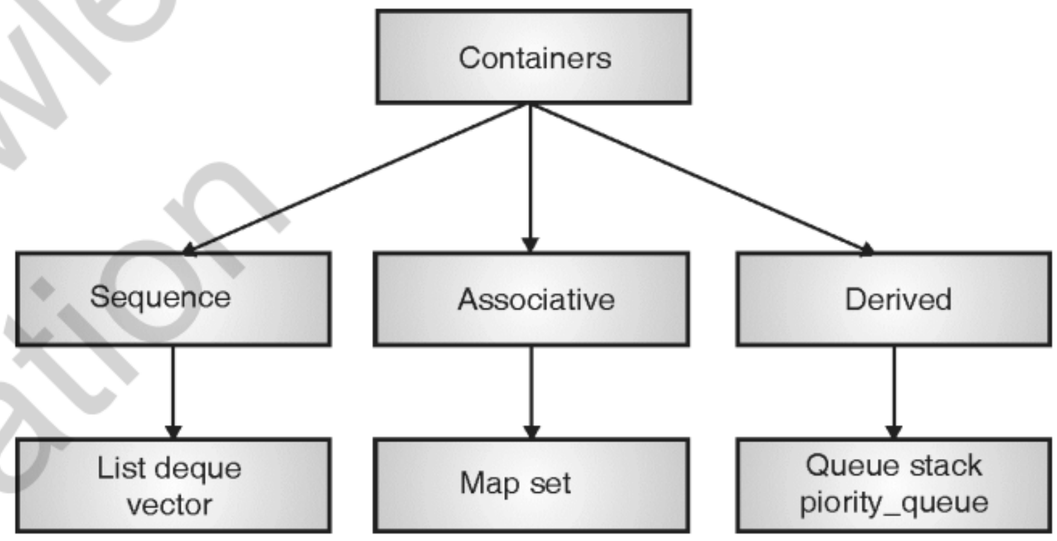

Fig. 8.2.1 : Container

– Sequence containers store elements in sequential manner.

– Element of these containers can be accessed using iterators.

– Example of sequence containers include list, deque, vector, etc.

– With Associative containers it is possible to access elements randomly. It is possible using key.

– Example of associative containers are map, set, etc.

– Elements of associative containers are stored using tree for efficient searching and other operations.

Tech Knowledge Publications

Advantages of Vector

1. Vector files are useful for storing images composed of line-based elements such as lines and polygons, or those that can be decomposed into simple geometrical objects, such as text. More sophisticated formats can also store 3D objects such as polyhedrons and wire-frame models.
2. Vector data can be easily scaled and otherwise manipulated to accommodate the resolution of a spectrum of output devices.
3. Many vector files containing only ASCII-format data can be modified with simple text editing tools. Individual elements may be added, removed, or changed without affecting other objects in the image.
4. It is usually easy to render vector data and save it to a bitmap format file, or, alternately, to convert the data to another vector format, with good results.

Disadvantages of Vector

1. Vector files cannot easily be used to store extremely complex images, such as some photographs, where color information is paramount and may vary on a pixel-by-pixel basis.
2. The appearance of vector images can vary considerably depending upon the application interpreting the image. Factors include the rendering application's compatibility with the creator application and the sophistication of its toolkit of geometric primitives and drawing operations.
3. Vector data also displays best on vectored output devices such as plotters and random scan displays. High-resolution raster displays are needed to display vector graphics as effectively.
4. Reconstruction of vector data may take considerably longer than that contained in a bitmap file of equivalent complexity, because each image element must be drawn individually and in sequence.

Advantages of List

1. Support of Generic programming.
2. The list delivers fast, and efficient.
3. The list is easy to use.
4. It is also robust code.

Disadvantages of List

1. Template by its nature exposes its implementation.
2. In this there is no information hiding.

8.2.2 Container Adapters

– Container contains data which is processed by STL algorithms.

– Adaptors are data types from STL which a container to provide specific interface.

– Example would be stack providing stack interface on top of chosen container.

– When you declare the container adapters, you have an option of specifying which sequence containers form the underlying container.

– A **stack** is a container providing Last-In, First-Out (LIFO) access. Basically, you remove elements in the reverse order you insert them. It's difficult to get to any elements in the middle. Usually this goes on top of a *deque*.

– A **queue** is a container providing First-In, First-Out (FIFO) access. You remove elements in the same order you insert them. It's difficult to get to any elements in the middle. Usually this goes on top of a *deque*.

– A **priority_queue** is a container providing sorted-order access to elements. You can insert elements in any order, and then retrieve the "lowest" of these values at any time.

– Priority queues in C++ STL use a heap structure internally, which in turn is basically array-backed; thus, usually this goes on top of a vector.

Technknowledge Publications

8.3 Functions in Container Classes

8.3.1 Array

Function	Use
begin()	Go to beginning of the array
end()	To end of array
size()	Use to get size of array
max_size()	Used to get maximum size of array
rbegin()	Used to get reverse iterator to reverse beginning
rend()	Used to get reverse iterator to reverse end
empty()	Used to check the whether array is empty or not
at()	Used to get element at particular position
front()	Used to get first element from array
back()	Used to get last element
data()	Used to get pointer to data

8.3.2 Deque

Q. Use minimum 8 functions of Deque STL. Write a program to explain the same.

SPPU - Dec. 17, 6 Marks

It is nothing but double ended queue. We can add or delete element from both side of the queue. STL container class provides following functions to manipulate deque elements.

Function	Use
begin()	Go to beginning
end()	Go to end
size()	Get size
max_size()	Get max size
resize()	Change size
empty()	Check for empty queue
at()	Get particular element
front()	Get first element
back()	Get last element
push_back()	Add element at end
push_front()	Add element at start
pop_back()	Remove last element
pop_front()	Remove first element
insert()	Add element
swap()	Swap the contents
clear()	Clear the contents

Example

– The source code of the C++ Program to demonstrate Deque in STL. The C++ program is successfully compiled and the program output is shown below :

➢ **Program 8.3.1**

```
//C++ Program to Implement Deque in STL
#include<iostream>
#include<deque>
#include<string>
#include<cstdlib>
using namespace std;
int main()
{
   deque<int> dq;
   deque<int>::iterator it;
   int choice, item;
   while (1)
   {
```

Technknowledge Publications

```
    cout<<"\n--------------------"<<endl;
    cout<<"Deque Implementation in Stl"<<endl;
    cout<<"\n--------------------"<<endl;
    cout<<"1.Insert Element at the End"<<endl;
    cout<<"2.Insert Element at the Front"<<endl;
    cout<<"3.Delete Element at the End"<<endl;
    cout<<"4.Delete Element at the Front"<<endl;
    cout<<"5.Front Element at Deque"<<endl;
    cout<<"6.Last Element at Deque"<<endl;
    cout<<"7.Size of the Deque"<<endl;
    cout<<"8.Display Deque"<<endl;
    cout<<"9.Exit"<<endl;
    cout<<"Enter your Choice: ";
    cin>>choice;
    switch(choice)
    {
   case 1:
      cout<<"Enter value to be inserted at the end: ";
      cin>>item;
      dq.push_back(item);
      break;
   case 2:
     cout<<"Enter value to be inserted at the front: ";
     cin>>item;
     dq.push_front(item);
     break;
  case 3:
   item = dq.back();
   dq.pop_back();
   cout<<"Element "<<item<<" deleted"<<endl;
   break;
  case 4:
    item = dq.front();
    dq.pop_front();
    cout<<"Element "<<item<<" deleted"<<endl;
    break;
   case 5:
     cout<<"Front Element of the Deque: ";
     cout<<dq.front()<<endl;
     break;
   case 6:
     cout<<"Back Element of the Deque: ";
     cout<<dq.back()<<endl;
     break;
   case 7:
   cout<<"Size of the Deque: "<<dq.size()<<endl;
    break;
  case 8:
    cout<<"Elements of Deque:  ";
      for (it = dq.begin(); it != dq.end(); it++)
          cout<<*it<<"  ";
      cout<<endl;
      break;
   case 9:
      exit(1);
    break;
    default:
      cout<<"Wrong Choice"<<endl;
    }
  }
  return 0;
}
```

Output

```
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 1
Enter value to be inserted at the end: 9
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
```

Techknowledge Publications

```
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 2
Enter value to be inserted at the front: 8
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 1
Enter value to be inserted at the end: 7
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 1
Enter value to be inserted at the end: 6
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
```

```
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 1
Enter value to be inserted at the end: 5
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 2
Enter value to be inserted at the front: 10
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 2
Enter value to be inserted at the front: 7
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
```

Techknowledge Publications

```
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 7
Size of the Deque: 7
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 8
Elements of Deque:  7  10  8  9  7  6  5
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 5
Front Element of the Deque: 7
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
```

```
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 6
Back Element of the Deque: 5
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 3
Element 5 deleted
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 8
Elements of Deque:  7  10  8  9  7  6
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
```

Tech Knowledge Publications

```
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 4
Element 7 deleted
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 7
Size of the Deque: 5
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 8
Elements of Deque:  10  8  9  7  6
--------------------
Deque Implementation in Stl
--------------------
1.Insert Element at the End
2.Insert Element at the Front
3.Delete Element at the End
4.Delete Element at the Front
5.Front Element at Deque
6.Last Element at Deque
7.Size of the Deque
8.Display Deque
9.Exit
Enter your Choice: 9
```

8.3.3 Queue

Queue is collection of element in which we can add or remove element in certain order only. The function available in queue container class is as follows :

Function	Use
empty()	Check for empty stack
front()	Access next element
push()	Add element
pop()	Remove next element
size()	Get size
back()	Access last element

8.4 STL Vector

Q. Use minimum 8 functions of vector STL. Write a program to explain the same.

SPPU - May 17, Dec. 18, 7 Marks

– It is sequence container used to represent array which can change size. They are more like array.

– Only difference is that their size can be changed. Functions available in vector are as follows :

Function	Use
begin()	Go to beginning
end()	Go to end
size()	Get size
max_size()	Get max size
resize ()	Change the size
capacity()	Return storage capacity
empty()	Check empty vector
reserve()	Request for change in capacity
front()	Get first element

Tech Knowledge Publications

Function	Use
back()	Get last element
at()	Access element at certain position
data()	Access data
push_back()	Adds the element at the end
pop_back()	Erase last element
insert()	Used to insert element
swap()	Swap the contents
clear()	Clear the contents

– Container Vector class is explained using Program 8.4.1.

➢ **Program 8.4.1 : Vector using STL**

```
#include<iostream>
#include<vector>
using namespace std;

int main()
{
  vector<int> myvector;
  int j;

 cout<<"\n Size of vector is = " << myvector.size();

  cout<<"\n Add five element in stack";
  for(j = 0; j < 5; j++){
    myvector.push_back(j);
  }

  if (!myvector.empty())
  {
    cout<<"\n Vector is not empty";
  }
  cout<<"\n Maximum size of vector is
      "<<myvector.max_size();
  cout<<"\n Current size of vector is = " <<
      myvector.size();
  cout<<"\n Element of vector are: ";
  for(j = 0; j < 5; j++){
    cout<<" \t " << myvector[j];
  }
  cout<<"\n Display element using iterator";
  vector<int>::iterator v;
for(v=myvector.begin(); v != myvector.end(); v++)
  {
    cout<<"\t  " << *v ;
  }
  return 0;
}
```

Output

```
Size of vector is = 0
Add five element in stack
Vector is not empty
Maximum size of vector is 4611686018427387903
Current size of vector is  = 5
Element of vector are:     0    1    2    3    4
Display element using iterator 0    1    2    3    4
```

8.5 STL Stack

Q. What is a stack? How is it implemented using STL?

SPPU - Dec. 16, Dec. 17, May 18, May 19, 6 Marks

Q. Explain any one of container class using program.

SPPU - May 17, May 19, Dec. 19, 4 Marks

Stack is collection of element. In stack we can add or delete element in certain order only. The functions available in the stack container are as follows :

Function	Use
empty()	Check for empty stack
top()	Get top element
push()	Add element
pop()	Remove top element
size()	Get size

Techknowledge Publication

➢ **Program 8.5.1 : Stack manipulation using STL**

```
#include<string>
#include<iterator>
#include<iostream>
#include<algorithm>
#include<stack>
using namespace std;
int main()
{
   stack <int> mystack;
   if (mystack.empty())
   {
      cout<<"\n Stack is empty";
   }
   mystack.push(10);
   cout<<"\n Added element in the stack
        "<<mystack.top();
   mystack.push(20);
   cout<<"\n Added element in the stack
        "<<mystack.top();
   mystack.push(30);
   cout<<"\n Added element in the stack
        "<<mystack.top();
   mystack.push(40);
   cout<<"\n Added element in the stack
        "<<mystack.top();
   cout<<"\n Size of stack is "<<mystack.size();
    if (!mystack.empty())
   {
      cout<<"\n Stack is not empty";
   }
   cout<<"\n Popped element is : "<<mystack.top();
   mystack.pop();
   cout<<"\n Element after pop operation";
   while (!mystack.empty())
   {
    cout<<' ' << mystack.top();
    mystack.pop();
   }
   return 0;
}
```

Output

```
Stack is empty
Added element in the stack   10
Added element in the stack   20
Added element in the stack   30
Added element in the stack   40
Size of stack is 4
Stack is not empty
Popped element is : 40
Element after pop operation 30 20 10
```

8.6 STL Lists

- It is another container providing different functionality like bidirectional, linear list.
- It provides efficient insertion and deletion.
- List should be accessed sequentially only.

➢ **Program 8.6.1**

```
#include<iostream>
#include<algorithm>
#include<list>
using namespace std;

int main() {
   list <int> first;
   int temp,i;

   cout<<"\n Enter elements in list ";
   for(i=0;i<5;i++)
   {
      cin>>temp;
      first.push_back(temp);
   }
   list<int>::iterator ip;
   cout<<"\n Original list " ;
   for(ip=first.begin();ip!=first.end();ip++)
      cout<<"\t "<<*ip;
   first.push_front(1000);
   first.push_back(2000);
   cout<<"\n Modified list ";
```

Techknowledge Publications

```
    for(ip=first.begin();ip!=first.end();ip++)
        cout<<"\t "<<*ip;

    return 0;
}
```

Output

```
Enter elements in list 10 20 30 40 50

Original list   10    20    30    40    50
Modified list  1000  10  20  30    40    50    2000
```

8.7 Algorithms

Q. What is the purpose of Algorithms.

SPPU - Dec. 17, 6 Marks

- Algorithms are collection of steps.
- Algorithms acts on containers.
- Algorithms are used to perform the operations on container objects.
- Famous algorithms are searching, sorting, initialization, etc.
- STL algorithms can be classified into :
  - Retrieving algorithms
  - Mutating algorithms
  - Sorting algorithm
  - Searching algorithms
  - Relational algorithms
- List of mutating and non mutating algorithms are given below :

Non mutating algoritms	
adjacent_find()	Search equal, adjacent object
search()	Search subsequence in sequence
count()	Count the occurrences
find()	Search given value
find_end()	Search last occurrence
for_each()	Apply operation to every element

Mutating algorithms	
fill()	Inserted specified value
swap()	Interchange two elements
unique()	Removes equal adjacent elements
remove()	Delete certain element
generate()	Replace element with result of operation
replace()	Replace element with given value
copy()	Copy the sequence

- Sorting algorithms

Function	Meaning
binary_search()	Perform binary search
merge()	Merge sorted sequences
sort()	Sort
equal_range()	Search subrange
lower_bound()	Search first occurrence
make_heap()	Create heap
partial_sort()	Sort part of elements in sequence
partition()	Divides list
pop_heap()	Delete top element
push_heap()	Push element to heap
sort_heap()	Sort given heap
stable_sort()	Perform stable sorting

- Set algorithms are as follows :

Function	Meaning
Includes()	Search weather sequence is sub sequence of another
Set_difference()	Constructs a sequence which is difference of two ordered sets
Set_intersection()	Perform intersection on two sets
Set_symmetric_ difference()	Perform symmetric difference on set
Set_union()	Perform union on two sets

Techknowledge Publications

8.8 Basic Searching and Sorting Algorithms

- We perform basic searching using STL.
- It stores elements in vector and perform searching and sorting on it.

➢ **Program 8.8.1**

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
int main()
{
  vector<int> myvector;
  int j;
  int key;
  cout<<"\n Size of vector is = "<< myvector.size();
  cout<<"\n Add five element in stack";
  for(j = 105; j > 100; j--){
    myvector.push_back(j);
  }
  if (!myvector.empty())
  {
     cout<<"\n Vector is not empty";
  }
  cout<<"\n Display element using iterator";
  vector<int>::iterator v;
  for(v=myvector.begin(); v != myvector.end(); v++)
{
    cout<<"\t  " << *v ;
  }
  cout<<"\n Enter element to be searched";
  cin>>key;
  //first sort vector
  sort(myvector.begin(), myvector.begin()+5);
 if(binary_search(myvector.begin(),      myvector.end(),
    key)){
      cout<<"\n Element found";
   }
  else
      cout<<"\n Element not found";
  return 0;
}
```

Output

```
Size of vector is = 0
Add five element in stack
Vector is not empty
Display element using iterator
 105    104    103    102    101
Enter element to be searched
106

Element not found
```

- Sorting is possible with sort functions.

➢ **Program 8.8.2**

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
int main()
{
  vector<int> myvector;
  int j;
  int key;
   //Add five element in stack
  for(j = 105; j > 100; j--){
    myvector.push_back(j);
  }
  cout<<"\n Display element using iterator\n";
  vector<int>::iterator v;
  for(v=myvector.begin(); v != myvector.end(); v++)
{
    cout<<*v << "\t  ";
  }
  sort(myvector.begin(), myvector.begin()+5);

  cout<<"\n Sorted elements are as follows\n";
  for(v=myvector.begin(); v != myvector.end(); v++)
{
    cout<<*v << "\t  " ;
  }
  return 0;
}
```

Techknowledge Publication

Output

```
Display element using iterator
105     104     103     102     101
Sorted elements are as follows
101     102     103     104     105
```

8.9 Min Max Algorithms

– It is useful to get smallest and largest element in given list.

➢ **Program 8.9.1**

```
#include<iostream>
#include<algorithm>
using namespace std;
int main() {
  auto output = std::minmax({10,20,30,40,50});

  cout<<"minmax({10,20,30,40,50}): ";
  cout<<output.first << ' ' << output.second <<
            '\n';
  return 0;
}
```

Output

```
minmax({10,20,30,40,50}): 10 50
```

8.10 Set Operations

– Different set operations can be performed using STL set class.

– Operations like union, intersection, set difference, etc. are possible.

➢ **Program 8.10.1**

```
#include<iostream>
#include<set>
#include<algorithm>
#include<vector>
using namespace std;
int main()
{
  int a[]={ 50, 40, 30, 20, 10 };  //list of elements
  int b[]={ 10, 30, 50 };  //list of elements
  set<int> first (a,a+5);   //create first set
  set<int> second (b, b+3); //create second set

  vector<int> v(10);
  vector<int>::iterator i;

  set<int>::iterator it;
  cout<<"\n Size of first: " << int (first.size()) <<
            '\n';
  cout<<"\n First set:  ";
  for(it=first.begin();it!=first.end();it++)
     cout<<"\t"<<*it;

  cout<<"\n  Size  of  second  :  "  <<  int
            (second.size()) << '\n';
  cout<<"\n Second set:  ";
  for(it=second.begin();it!=second.end();it++)
     cout<<"\t"<<*it;
  //sort elements
  sort(a, a+5);
  sort(a, b+3);

  //set union
  i=set_union(a, a+5, b, b+3, v.begin());
  v.resize(i-v.begin());
  cout<<"\n Union :  ";
  for(i=v.begin();i!=v.end();i++)
     cout<<"\t"<<*i;

  //set difference
  i=set_difference(a, a+5, b, b+3, v.begin());
  v.resize(i-v.begin());
  cout<<"\n Set difference  :  ";
  for(i=v.begin();i!=v.end();i++)
     cout<<"\t"<<*i;

  return 0;
}
```

Tech Knowledge Publications

Output

```
Size of first: 5

First set:     10    20    30    40    50
Size of second : 3

Second set :    10    30    50
Union :         10    20    30    40    50
Set difference  :     20    40
```

8.11 Heap Sort

- Heap sorting is possible with STL.
- First we need to make heap with make_heap() functions.
- In second step heap need to sorted using sort_heap() functions.

➢ **Program 8.11.1**

```
#include<iostream>
#include<algorithm>
using namespace std;

int main() {

    //Sorting integer
    int a[] = {10, 9, 8,7,6,5,4,3,2,1};
    make_heap(a, a+10);     //create heap
    sort_heap(a,a+10);      //sort heap elements

    cout<<"\n Sorted elements:\n";
    for(int i = 0; i < 10; i++)
        cout<<"\t"<<a[i];
    //Sorting float
  float b[] = {10.9, 9.8, 8.6,7.5,6.4,5.3,4.2,3.1,2.6,1.3};
  make_heap(b, b+10);     //create heap
  sort_heap(b,b+10);      //sort heap elements
  cout<<"\n Sorted elements:\n";
  for(int i = 0; i < 10; i++)
  cout<<"\t"<<b[i];
   return 0;
}
```

Output

```
Sorted elements:
1    2    3    4    5    6    7    8    9    10
Sorted elements:
1.3  2.6  3.1  4.2  5.3  6.4  7.5  8.6  9.8  10.9
```

➢ **Program 8.11.2 : Write a program to illustrate STL heap sort.** **Dec. 19, 7 Marks**

- In C++ STL, there is priority_queue that can directly be used to implement Max Heap. In order to fully understand the code, make sure you are familiar with following concepts in C++ :

```
//C++ program to show that priority_queue is by
//default a Max Heap
#include<bits/stdc++.h>
using namespace std;

//Driver code
int main()
{
   //Creates a max heap
   priority_queue <int> pq;
   pq.push(5);
   pq.push(1);
   pq.push(10);
   pq.push(30);
   pq.push(20);

   // One by one extract items from max heap
   while (pq.empty() == false)
   {
       cout<<pq.top() << " ";
       pq.pop();
   }

   return 0;
}
```

Output

```
30 20 10 5 1
```

- Since elements are printed in descending order, we have a max heap by default.

Techknowledge Publications

8.12 Iterators - Input, Output, Forward, Bidirectional and Random Access

Q. What is the purpose of iterator ?

SPPU - Dec. 16, Dec. 17, May 18, 2 Marks

- Iterators are used to access the element of the containers.
- For example iterator is used to access the element of array one by one.
- Iterator can be of different types.
  - Input iterators
  - Output iterators
  - Forward iterators
  - Bidirectional iterators
  - Random access iterators
- Input iterators are used to read the sequence of values from containers.
- Output iterators are used to write the contents in the iterators.
- Bidirectional iterators are used to move forward and in backward directions.
- Random access iterators can move freely in the containers.
- Iterators is important concept which adds generality in the STL programming.
- Functions is another important concept in the STL.
- Functions perform the actual processing in the containers.
- The stack container example is given

➢ **Program 8.12.1 : Queue manipulation using STL**

```
#include<iostream>
#include<queue>
using namespace std;
int main()
{
   queue<int> Q;
   if (Q.empty())
   cout<<"\n Queue is empty";
   Q.push(10);
   Q.push(20);
   Q.push(30);
   Q.push(40);
   cout<<"\n Size of queue is "<<Q.size();
   cout<<"\n First element is "<<Q.front();
   cout<<"\n Last element is "<<Q.back();
   cout<<"\n Delete element from queue";
   Q.pop();

   cout<<"\n First element is "<<Q.front();
   cout<<"\n Last element is "<<Q.back();
   cout<<"\n Add element in queue";
   Q.push(50);

   cout<<"\n First element is "<<Q.front();
   cout<<"\n Last element is "<<Q.back();
   cout<<"\n Display element of Queue";
   while(!Q.empty())
   {
      cout<<"\t"<<Q.front();
      Q.pop();
   }
  return 0;
}
```

Output

```
Queue is empty
Size of queue is 4
First element is 10
Last element is 40
Delete element from queue
First element is 20
Last element is 40
Add element in queue
First element is 20
Last element is 50
Display element of Queue      20  30  40  50
```

Techknowledge Publications

➢ **Program 8.12.2 : Deque using STL**

```
#include<iostream>
#include<deque>
using namespace std;
int main()
{
   deque<int> Q;
   if (Q.empty())
   cout<<"\n Queue is empty";
   Q.push_front(10);
   Q.push_front(20);
   Q.push_back(30);
   Q.push_back(40);
   cout<<"\n Size of queue is "<<Q.size();
   cout<<"\n First element is "<<Q.front();
   cout<<"\n Last element is "<<Q.back();
   cout<<"\n Delete last element from queue";
   Q.pop_back();

   cout<<"\n First element is "<<Q.front();
   cout<<"\n Last element is "<<Q.back();

   cout<<"\n Add element in queue";
   Q.push_front(50);

   cout<<"\n First element is "<<Q.front();
   cout<<"\n Last element is "<<Q.back();

   cout<<"\n Display element of Queue";
   while(!Q.empty())
   {
      cout<<"\t"<<Q.front();
      Q.pop_front();
   }

  return 0;
}
```

Output

```
Queue is empty
Size of queue is 4
First element is 20
Last element is 40
Delete last element from queue
First element is 20
Last element is 30
Add element in queue
First element is 50
Last element is 30
Display element of Queue      50  20  10  30
```

8.12.1 Types of Iterators

Q. Elaborate forward, bidirectional and random-access iterators with suitable example.

SPPU - Dec. 16, May 18, May 19, Dec. 19, 5 Marks

Forward Iterator

– It is an Iterator that can read data from the pointed-to element.

– Unlike Input Iterator and Output Iterator, it can be used in multipass algorithms.

Example

```
#define _DEFINE_DEPRECATED_HASH_CLASSES 0
#include<iostream>
#include<hash_set>
 using namespace std;
using namespace stdext;
 int main()
{
  hash_set <int> hs1;
  hash_set <int>::iterator hs1_Iter;
  hash_set <int>::const_iterator hs1_cIter;
  hs1.insert( 1 );
  hs1.insert( 2 );
  hs1.insert( 3 );
  hs1_Iter = hs1.begin( );
  cout<<"The first element of hs1 is
       "<<*hs1_Iter<<endl;
  hs1_Iter = hs1.begin( );
  hs1.erase( hs1_Iter );
   //The following 2 lines would err because the iterator
   //is const
```

Techknowledge Publication

```
 // hs1_cIter = hs1.begin( );
 // hs1.erase( hs1_cIter );
  hs1_cIter = hs1.begin( );
 cout<<"The first element of hs1 is now
        "<<*hs1_cIter<<endl;
  return 0;
}
```

– **Bidirectional Iterator :** It is a Forward Iterator that can be moved in both directions that is incremented and decremented.

Example

```
#include<iostream>
#include<string>
#include<list>
 using namespace std;
 int main()
{
  list<int> lst1(8);
  cout <<"\nFirst we enter some integer squares into "
        "a list of size 8, and then display: \n";
   list<int>::iterator p = lst1.begin();
int i = 0;
 while(p != lst1.end())
{
  *p = i*i;
  p++;
  i++;
}
p = lst1.begin();
while(p != lst1.end())
  cout<<*p++<<" ";
   cout<<"\nNext we double the values in the list
        and " "re-display (backwards this time): \n";
p = lst1.begin();
while(p != lst1.end())
{
  *p = *p * 2;
  p++;
}
    //This time, initialize the iterator to lst1.end().
    //And decrement the iterator **before** using it!
  p = lst1.end();
  while(p != lst1.begin())
   cout<<*--p<<" ";    //Note *--p "idiom".
  return 0;
}
```

– **Random-access iterators :** It implement all the functionality of bidirectional iterators, and also have the ability to access ranges non-sequentially: distant elements can be accessed directly by applying an offset value to an iterator without iterating through all the elements in between. These iterators have a similar functionality to standard pointers.

Example

```
#include<iostream>
#include<hash_map>
#define_DEFINE_DEPRECATED_HASH_CLASSES 0
typedef pair <int, int> Int_Pair;
 using namespace std;
using namespace stdext;
 int main()
{
  hash_map <int, int> hm1;
  hash_map <int, int> :: iterator hm1_Iter;
  hash_map <int, int> :: const_iterator hm1_cIter;
  hm1.insert (Int_Pair(0, 0));
  hm1.insert (Int_Pair(1, 1));
  hm1.insert (Int_Pair(2, 4));
  hm1_cIter = hm1.begin ( );
  cout<<"The first element of hm1 is "<<hm1_cIter
        -> first<<"."<<endl;
  hm1_Iter = hm1.begin ( );
  hm1.erase ( hm1_Iter );
            // hm1_cIter = hm1.begin ( );
            // hm1.erase ( hm1_cIter );
   hm1_cIter = hm1.begin( );
   cout<<"The first element of hm1 is now
            "<<hm1_cIter -> first<<"."<<endl;
   return 0;
}
```

8.13 Object Oriented Programming - A Road Map to Future

– Object oriented program is becoming very popular now-a-days.

– After its inception, it is gaining popularity very rapidly.

Techknowledge Publications

- It is become one of the very important programming concepts.
- Many programming languages are object oriented programming languages.
- Due to limitations of procedure oriented programming, it is gaining importance.
- Following programming languages uses the concept of object oriented programming.

C++	Java	C#
Perl	PHP	COBOL
Objective-C	Visual basic	Python
Ruby	Javascript	Smalltalk

- This is just suggestive list.
- Most striking features of object oriented programming languages are as follows.
  - Class
  - Object
  - Inheritance
  - Polymorphism
  - Data hiding
  - Abstraction
  - Message passing
  - Encapsulation
- Object oriented concepts are used in Network protocol also. It useful in getting services in client server environment.
- Object oriented programming languages has been used in dynamic languages also. Dynamic languages include Python, Powershell, Ruby, Perl, PHP, HTML, XML, XHTML, etc.
- Object oriented concept has been used in database also. Such databases are called object oriented Databases. In object oriented databases information is stored in the form of object.
- Object oriented databases are different from relational databases.
- It means object oriented programming concepts are used everywhere.
- There are tremendous job and career opportunities in object oriented programming.

➢ **Program 8.13.1 : Write a program to implement Map using STL.**

SPPU - May 17, Dec. 18, May 19, 6 Marks

```
#include<iostream>
#include<map>
#include<string>
#include<cstdlib>
using namespace std;
int main()
{
   map<char,int> mp;
   map<char, int>::iterator it;
   int ch, value;
   char s1;
   while (1)
   {
      cout<<"\n--------------------"<<endl;
      cout<<"Map Implementation in STL"<<endl;
      cout<<"\n--------------------"<<endl;
      cout<<"1.Insert Element into the Map"<<endl;
      cout<<"2.Delete Element of the Map"<<endl;
      cout<<"3.Size of the Map"<<endl;
      cout<<"4.Find Element at a key in Map"<<endl;
      cout<<"5.Dislplay by Iterator"<<endl;
      cout<<"6.Count Elements at a specific
            key"<<endl;
      cout<<"7.Exit"<<endl;
      cout<<"Enter your Choice: ";
      cin>>ch;
      switch(ch)
      {
      case 1:
         cout<<"Enter value to be inserted: ";
         cin>>value;
         cout<<"Enter the key: ";
         cin>>s1;
         mp.insert(pair<char,int>(s1  ,value));
         break;
```

Techknowledge Publication

```
    case 2:
      cout<<"Enter the mapped string to be
            deleted: ";
      cin>>s1;
      mp.erase(s1);
      break;
    case 3:
    cout<<"Size of Map: ";
    cout<<mp.size()<<endl;
      break;
    case 4:
    cout<<"Enter the key at which value to be
          found: ";
      cin>>s1;
      if (mp.count(s1) != 0)
        cout<<mp.find(s1)->second<<endl;
      else
        cout<<"No Element Found"<<endl;
      break;
    case 5:
    cout<<"Displaying Map by Iterator: ";
        for (it = mp.begin(); it != mp.end(); it++)
        {
    cout << (*it).first << ": " << (*it).second <<
          endl;
        }
        break;
      case 6:
    cout<<"Enter the key at which number of values to
          be counted: ";
        cin>>s1;
        cout<<mp.count(s1)<<endl;
        break;
    case 7:
          exit(1);
           break;
           default:
        cout<<"Wrong Choice"<<endl;
      }
    }
    return 0;
}
```

Output

```
$ g++ map.cpp
$ a.out
---------------------
Map Implementation in STL
---------------------
1.Insert Element into the Map
2.Delete Element of the Map
3.Size of the Map
4.Find Element at a key in Map
5.Dislplay by Iterator
6.Count Elements at a specific key
7.Exit
Enter your Choice: 1
Enter value to be inserted: 1
Enter the key: a

---------------------
Map Implementation in STL
--------------------
1.Insert Element into the Map
2.Delete Element of the Map
3.Size of the Map
4.Find Element at a key in Map
5.Dislplay by Iterator
6.Count Elements at a specific key
7.Exit
Enter your Choice: 1
Enter value to be inserted:2
Enter the key: b

--------------------
Map Implementation in STL
--------------------
1.Insert Element into the Map
2.Delete Element of the Map
3.Size of the Map
4.Find Element at a key in Map
5.Dislplay by Iterator
6.Count Elements at a specific key
7.Exit
```

Tech Knowledge Publications

```
Enter your Choice: 1
Enter value to be inserted: 3
Enter the key: c
--------------------
Map Implementation in STL

--------------------
1.Insert Element into the Map
2.Delete Element of the Map
3.Size of the Map
4.Find Element at a key in Map
5.Dislplay by Iterator
6.Count Elements at a specific key
7.Exit
Enter your Choice:3
Size of Map:3
Map Implementation in STL

--------------------
1.Insert Element into the Map
2.Delete Element of the Map
3.Size of the Map
4.Find Element at a key in Map
5.Dislplay by Iterator
6.Count Elements at a specific key
7.Exit
Enter your Choice: 5
Displaying Map by Iterator: a: 1
b: 2
c: 3
--------------------
Map Implementation in STL
 --------------------
1.Insert Element into the Map
2.Delete Element of the Map
3.Size of the Map
4.Find Element at a key in Map
5.Dislplay by Iterator
6.Count Elements at a specific key
7.Exit
Enter your Choice: 7
```

Important Points

- STL
- Generic programming
- Components of STL
- Container
- Algorithm
- Container adaptors
- Iterator
- Min max algorithm
- STL stack
- STL vector
- STL list
- STL searching and sorting
- STL set

Review Questions

Q. 1 What is STL?

Q. 2 What are different components of STL?

Q. 3 What is containers? What are different container classes?

Q. 4 What are different types of containers?

Q. 5 What is container adaptors?

Q. 6 What is STL algorithms?

Q. 7 What are iterators? Explain different iterator?

Q. 8 Write program for STL Queue?

Q. 9 Write program for STL stack?

Q. 10 Write program to illustrate STL vector?

Q. 11 Write program to illustrate STL deque?

Q. 12 Write program to illustrate STL stack?

Q. 13 Write program to illustrate STL set operations ?

Q. 14 Write program to illustrate STL heap sort?

Q. 15 Write program to illustrate STL list?

Q. 16 Write program to illustrate basic searching and sorting in STL?

Q. 17 Write program to illustrate STL min-max algorithms?

Q. 18 What is future of object oriented programming?

Technknowledge Publication

❑❑❑

Model Question Paper (In Sem.)

Object Oriented Programming

Semester - III (Computer Engineering)

Time : 1 Hour **Maximum Marks : 30**

Instructions to the candidates :

1) **Answer Q. 1 or Q. 2, Q. 3, or Q. 4.**
2) **Neat diagrams must be drawn wherever necessary.**
3) **Figure to the right indicates full marks.**
4) **Make suitable assumptions if necessary.**

Q. 1 **(a)** Compare procedure oriented programming *Vs.* object oriented programming. **(Section 1.6.3)** **(4 Marks)**

(b) Explain the features of Object-oriented Programming. **(Refer Section 1.8)** **(6 Marks)**

(c) What is Inline function ? Explain with suitable program. **(Refer Section 1.31)** **(5 Marks)**

OR

Q. 2 **(a)** Explain Procedure Oriented Programming with an example program. Also Characteristics of Procedure Oriented Programming. **(Refer Section 1.5)** **(6 Marks)**

(b) What are primitive data types and user defined data types ? **(Refer Section 1.12)** **(4 Marks)**

(c) What is class and object ? Differentiate between class and object. **(Refer Sections 1.17 and 1.18)** **(5 Marks)**

Q. 3 **(a)** With suitable example, explain different types of inheritance. **(Refer Section 2.9)** **(6 Marks)**

(b) Explain virtual base class with an example program. **(Refer Section 2.11)** **(2 Marks)**

(c) Define Abstract Class along with its characteristics. Illustrate with an example program in C++.

(Refer Section 2.12) **(5 Marks)**

(d) Define Nested Class with an example. **(Refer Section 2.15)** **(2 Marks)**

OR

Q. 4 **(a)** What is Pointer? Explain using diagram and program. **(Refer Section 3.1)** **(3 Marks)**

(b) Explain pointer to a function. Use suitable example. **(Refer Section 3.8)** **(3 Marks)**

(c) Write a short notes on following : **(4 Marks)**

(i) Null Pointer **(Refer Section 3.13)**

(ii) Void Pointer **(Refer Section 3.14)**

(d) Write a program to find the sum of an array Arr by passing an array to a function using pointer.

(Refer Program 3.8.2) **(5 Marks)**

❑❑❑

Technical Knowledge Publication

Model Question Paper (End Sem.)

Object Oriented Programming

Semester - III (Computer Engineering)

Time : 2 ½ Hour **Maximum Marks : 70**

Instructions to the candidates :

1) **Answer Q. 1 or Q. 2, Q. 3 or Q. 4, Q. 5 or Q. 6, Q. 7 or Q. 8**
2) **Neat diagrams must be drawn wherever necessary.**
3) **Figure to the right indicates full marks.**
4) **Make suitable assumptions, if necessary.**

Q. 1 **(a)** Define Polymorphism in detail. Also explain Compile time polymorphism with an example.

(Refer Sections 4.1 and of 4.2(A)) **(6 Marks)**

(b) Explain the concept of Overloading. How Unary Operators are overloaded? Explain with an example program.

(Refer Sections 4.3 and 4.4) **(6 Marks)**

(c) Explain what is type casting ? Explain Implicit type casting and explicit type casting.

(Refer Section 4.7) **(6 Marks)**

OR

Q. 2 **(a)** Explain the concept of Overloading. How Binary Operators are overloaded? Explain with an example program.

(Refer Sections 4.3 and 4.5) **(6 Marks)**

(b) What do you mean by type conversion ? Explain the same with example. **(Refer Section 4.6)** **(4 Marks)**

(c) Explain the use of explicit and mutable keywords with suitable examples. **(Refer Section 4.9)** **(4 Marks)**

(d) When do we make a virtual function "pure" ? What are the implications of making a function a pure virtual function ? **(Refer Section 4.12.1)** **(4 Marks)**

Q. 3 **(a)** Write the C++ file input and output program using seekg(), tellg(), read() etc. member functions.

(Refer Program 5.6.7) **(7 Marks)**

(b) What is a stream? What are cin and cout ? Explain iostream. **(Refer Sections 5.2 and 5.3)** **(5 Marks)**

(c) What are various functions which are used to manipulate file pointers ? Explain using examples.

(Refer Section 5.7) **(5 Marks)**

OR

Q. 4 **(a)** Write a program to create a file, read and write the record into it. Every record contains Employee Name, Id and Salary. Store and retrieve at least 3 employee data. **(Refer Program 5.6.8)** **(7 Marks)**

(b) Explain command line argument in arguments in C++. Write program to explain the same.

(Refer Section 5.12) **(5 Marks)**

(c) How the Extraction and Insertion Operators are Overloaded ? **(Refer Section 5.10)** **(5 Marks)**

Technical Knowledge Publication

Q. 5 **(a)** How to handle multiple exceptions occurred in a program ? **(Refer Section 6.4)** **(4 Marks)**

(b) Define User Defined Exception. Write a program to demonstrate it. **(Refer Section 6.7)** **(5 Marks)**

(c) Explain how exception is handled in case of constructor and destructor with example.

(Refer Section 6.9) **(4 Marks)**

(d) How Exceptions are handled in Inheritance? Write a program to illustrate. **(Refer Section 6.10)** **(5 Marks)**

OR

Q. 6 **(a)** What is generic programming? How is it implemented in C++ ? **(Refer Section 7.1)** **(7 Marks)**

(b) Write a function template for finding the minimum value contained in an array. **(Refer Program 7.2.1)** **(4 Marks)**

(c) Explain function template with an example. **(Refer Section 7.3)** **(3 Marks)**

(d) Describe type template parameters and non-type template parameters ? **(Refer Section 7.6)** **(4 Marks)**

Q. 7 **(a)** What is STL? Why should a C++ programmer be interested in the STL? What is the design philosophy of the STL? What are the major components of the STL ? **(Refer Section 8.1)** **(6 Marks)**

(b) What is container? List the container classes in C++. **(Refer Section 8.2)** **(5 Marks)**

(c) Use minimum 8 functions of Deque STL. Write a program to explain the same.

(Refer Section 8.3.2) **(6 Marks)**

OR

Q. 8 **(a)** Use minimum 8 functions of vector STL. Write a program to explain the same. **(Refer Section 8.4)** **(7 Marks)**

(b) What is a stack? How is it implemented using STL ? **(Refer Section 8.5)** **(4 Marks)**

(c) Write a program to implement Map using STL. **(Refer Program 8.13.1)** **(6 Marks)**

❑❑❑

Technical Knowledge Publication