

BCBM201T

OBJECT ORIENTED PROGRAMMING IN C++



DR. C.V. RAMAN UNIVERSITY

Kargi Road, Kota, Bilaspur, (C. G.),

Ph. 07753-253801, 07753-253872

E-mail: info@cvru.ac.in, Website: www.cvru.ac.in

BCBM201T
OBJECT ORIENTED PROGRAMMING IN C++

Credit- 2

Subject Expert Team

Course Edited By:

Unit Written By:

Warning: All rights reserved, No part of this publication may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the publisher.

Edition: 2026

Publish by: DR. C.V. RAMAN UNIVERSITY

Kargi Road, Kota, Bilaspur, (C. G.),

Ph. 07753-253801, 07753-253872

STRUCTURE OF PROGRAM

Title of the Program	:	Bachelor of Computer Application
Program Type	:	Undergraduate
Program Credits	:	120
Program Duration	:	36 Months (3 Years)
Eligibility Criteria	:	12th Pass

PROGRAMME OBJECTIVE

1. To produce employable IT workforce, that will have sound knowledge of IT and business fundamentals that can be applied to develop and customize solutions for Small and Medium Enterprises (SME)
2. To develop skilled manpower in the various areas of information technology like: Data base management, Software Development, Computer-Languages, Software engineering, Web based applications etc.
3. To Work professionally with positive attitude as an individual or in multidisciplinary teams and communicate effectively.”

PROGRAMME OUTCOME

1. Develop competence in basic technical subjects in computer applications like Programming Languages, Data Structures, Databases, Operating Systems, Software Engineering.
2. Identify, analyze, formulate and develop computer applications.
3. Map real life scenarios to various theoretical optimal solutions.
4. Provide simplest automated solutions to various legacy systems.
5. Use modern computing tools and techniques with confidence.

PROGRAMME SPECIFIC OUTCOME

Upon completion of program in Bachelor of Computer application, students will be able:

1. Able to implement computer skill in the areas related to multimedia and website design, Data base, Hardware and networking.
2. Able to demonstrate basic knowledge in the areas such as, Software Engineering, Data communication and Networking, Data base management, Web Technology and Operating Systems for building IT applications.
3. Students will able to understand, analyze and develop computer programs in the areas related to algorithm, system software, web design and networking for efficient design of computer-based system.

4. Apply standard software engineering practices and strategies in software project development using open-source programming environment to deliver a quality of product for business success.
5. Student will able to know various issues, latest trends in technology development and thereby innovate new ideas and solutions to existing problems

STRUCTURE OF COURSE

Course Title	:	Object Oriented Programming in C++
Course Type	:	Core Course (Major)
Course Code	:	BCBM201T
Course Credits	:	4
Semester	:	II Sem

COURSE LEARNING OBJECTIVES -

1. Define and explain core Object-Oriented Programming (OOP) concepts including tokens, expressions, control structures, classes, objects, constructors, destructors, pointers, and files.
2. Be able to write and run C++ programs using functions, constructors, and destructors.
3. Integrate functions, operator overloading, and inheritance to create more efficient and modular C++ programs.
4. Analyze and apply best practices in inheritance and operator overloading to enhance code reusability and maintainability.
5. Summarize and use advanced C++ concepts, including pointers, virtual functions, templates, and exception handling, to address complex programming problems.

COURSE LEARNING OUTCOMES -

By the end of this course, the learner will be able to:

After completion of the course, the student would be able to

1. Define the OOP concepts, tokens, expressions, control structures, classes, objects, constructors, destructors, pointers and files.
2. Executing the programs using functions, constructor and destructor.
3. Integrating the functions, operator overloading and inheritance.
4. Moderating the use of inheritance and operator overloading.
5. Summarize the concept of pointers, virtual functions, templates and exception handling

Course Content

Block / Unit No.	Blocks / Units Name	Page No.
Block 1	Fundamentals of C++ Programming	01-52
Unit 1	Introduction to C++ and OOP Concepts	03
Unit 2	C++ Programming Environment	13
Unit 3	Basic C++ Syntax and Constructs	23
Block 2	Arrays, Functions, and Objects	53-130
Unit 4	Arrays and Multi-Dimensional Data Handling	55
Unit 5	User-Defined Functions and Modularity	63
Unit 6	Introduction to Classes and Objects	77
Block 3	Polymorphism in C++	117-160
Unit 7	Introduction to Polymorphism	119
Unit 8	Function Overloading	127
Unit 9	Operator Overloading	137
Block 4	Inheritance in C++	161-200
Unit 10	Basics of Inheritance	163
Unit 11	Advanced Inheritance Concepts	175
Unit 12	Virtual Functions and Binding	187
Block 5	C++ I/O and Stream Manipulation	201-236
Unit 13	C++ I/O System Basics	203
Unit 14	Formatting Output	213
Unit 15	Custom Manipulators	227

BLOCK 1	FUNDAMENTALS OF C++ PROGRAMMING	01-52
Unit 1	Introduction to C++ and OOP Concepts	03
Unit 2	C++ Programming Environment	13
Unit 3	Basic C++ Syntax and Constructs	23



C++ is a versatile, high-level programming language that extends the capabilities of the C language by introducing object-oriented programming (OOP) concepts. It is widely used in software development, gaming, embedded systems, and high-performance applications due to its flexibility, speed, and efficiency. Understanding C++ fundamentals lays the foundation for developing structured, maintainable, and scalable programs. This block covers the essential concepts needed to start programming in C++. It introduces the history of C++, its evolution from C, and the need for object-oriented programming to address the software crisis—where complex software systems became difficult to manage.

Key concepts of OOP, including classes, objects, encapsulation, inheritance, and polymorphism, are highlighted to demonstrate how C++ simplifies real-world problem modeling. Additionally, learners are introduced to the C++ development environment, including compilers, IDEs, and the standard library. The block emphasizes hands-on knowledge of creating, compiling, and running simple C++ programs, understanding program structure, data types, operators, input/output operations, and basic control statements. By the end of this block, students will have a solid grasp of C++ syntax, program flow, and foundational programming techniques, preparing them for more advanced topics in object-oriented programming.

UNIT 1

INTRODUCTION TO C++ AND OOP CONCEPTS

NOTE

UNIT STRUCTURE

- 1.1 Introduction
- 1.2 Objectives
- 1.3 Overview of C++
- 1.4 Software Crisis
- 1.5 Object Oriented Programming Paradigm
- 1.6 Basic Concepts of OOP
- 1.7 Advantages/Benefits of OOP
- 1.8 Usage/Applications of OOP
- 1.9 Summary
- 1.10 Glossary
- 1.11 Answer to check your progress
- 1.12 Suggested Readings – OER
- 1.13 Reference
- 1.14 Exercise

1.1 INTRODUCTION

Before starting this unit, learners are expected to have a basic understanding of computers, problem-solving methods, and elementary programming concepts such as algorithms, flowcharts, and procedural programming ideas. Knowledge of how instructions are executed sequentially and how programs solve logical problems forms an important prerequisite. This unit serves as the foundation of the entire course by introducing C++ as a programming language and explaining why it was developed. The historical background of C++ and its evolution from the C language are discussed to help learners understand the limitations of procedural programming that led to the software crisis—where large and complex programs became difficult to develop, debug, and maintain.

The unit introduces the object-oriented programming (OOP) paradigm as a solution to these challenges. Core OOP concepts such as objects, classes, encapsulation, inheritance, polymorphism, abstraction, and message passing are explained using simple examples that relate programming concepts to real-world systems. Learners gain insight into how OOP promotes modularity, reusability, and maintainability in software development. The advantages, benefits, and real-world applications of OOP are also discussed, helping learners appreciate the relevance of C++ in modern software engineering. By the end of this unit, learners develop a conceptual understanding of C++ and object-oriented thinking. This conceptual grounding prepares them to move forward into the practical domain, where they will learn about the tools,

NOTE

compilers, standards, and environments required to develop C++ programs, which is the focus of the next unit.

1.2 OBJECTIVES

After completing this unit, learners will be able to:

- Explain the limitations of procedural programming leading to the software crisis.
- Differentiate between procedural and object-oriented programming paradigms.
- Illustrate core object-oriented concepts using real-world examples.
- Analyze the advantages and applications of object-oriented programming.
- Relate the evolution of C++ to modern software development needs.

1.3 OVERVIEW OF C++

C++ is a high-level programming language that was created as an enhancement to the C programming language. It provides procedural concept of C and additional high-level object-oriented features. Bjarne Stroustrup introduced it in the early 1980s. C++ is commonly used in software development for applications requiring speed and efficiency, such as operating systems, game engines, and real-time simulations. C++ contains “object-oriented” programming “features such as encapsulation, classes, inheritance, and polymorphism”, allowing software developers to construct code that is modular, reusable, and organized.

1.4 SOFTWARE CRISIS

Software crisis is about issues in developing large software systems. The demand for better programming paradigms was emanated by issues like an increase in complexity, appalling & maintenance and quality — most of them due to absence or poor project management techniques. C++ was designed to address these issues by bringing object-oriented programming (OOP) that does supports the reusability of code, modularity in designing and ease in testing.

Key Aspects of the Software Crisis:

1. **Complexity:** As software projects grew larger, they were more complicated to conceive, build, and maintain. Managing the complexity of code becomes a major concern in software development.
2. **Cost Overruns:** Many software projects over budget because original estimates failed to account for the unexpected problem of maintaining sophisticated systems.
3. **Schedule Slippage:** Projects frequently miss deadlines due to unexpected issues during development, debugging, and integration.

NOTE

- 4. Reliability:** Many software systems were unreliable, with frequent glitches and failures, causing a lack of trust in software solutions.
- 5. Maintenance Issues:** It was difficult to maintain software over time, especially when the code was not well-structured or documented. This made it difficult to adjust software to new requirements or correct faults without introducing new issues.

C++ and the Software Crisis:

C++ was developed to address some of the challenges associated with the software crisis:

1. C++ introduced “Object-Oriented Programming” (OOP) features such as classes, “inheritance, and polymorphism” to manage complexity through modularity, code reuse, and encapsulation. This makes it easier to disassemble complex systems into manageable components.
2. C++ supports higher degrees of abstraction without losing performance, enabling developers to manage complexity while developing fast code.
3. C++ supports code reuse through features like templates and the Standard Template Library (STL), saving time and effort in program development of programmer.
4. C++'s high-level abstractions and low-level control make it ideal for large-scale software systems that require efficiency and maintainability.

In summary, C++ emerged as a solution to the software crisis by providing tools and paradigms that helped developers manage the growing complexity of software projects, leading to more reliable, maintainable, and scalable software.

1.5 OBJECT ORIENTED PROGRAMMING PARADIGM

“Object-oriented programming” (OOP) is a “programming language” that use concepts of “objects” to create and implement software. Objects are instances of classes, which can be viewed as blueprints for constructing objects. OOP focuses on the use of these objects to represent data and the methods that can manipulate it.

The Need for OOPs software systems grew in complexity, traditional programming methodologies struggled to keep up. OOP was created to overcome the drawbacks of procedural programming, which frequently produced programs that were difficult to maintain and extend. OOP allows you to model “real-world entities in a more natural way”, making your code easier to understand and manage.

Features of OOP

NOTE

1. **Encapsulation:** This feature groups data and methods into a single object. Encapsulation avoids outside influence and misuse of data, enhancing data security.
2. **Abstraction:** This technique conceals complex realities while revealing only essential details. In OOP, you can construct a simple interface while concealing the underlying complexity.
3. **Inheritance:** This feature lets a new class inherit properties and behaviours from an existing class. It enables code reusability, as you can create new functionality based on existing code without rewriting it.
4. **Polymorphism:** Polymorphism enables you to use a single interface to represent different underlying forms (data types). This means that methods can take many forms, allowing for increased flexibility in designs and implementation.

1.6 BASIC CONCEPTS OF OOP

Classes and Objects

- **Class:** A class acts as a blueprint to create objects. It specifies the attributes, and behaviours, or methods that the objects derived from the class will have. Class is used to create user defined data types.
- **Objects:** Objects are class instances or variables of a class. Each object's attributes can have varying values, but they all follow the same structure and behaviour as described by their class. For example, if your class is Car, you may have objects like car1 with properties like colour and model.

Attributes and Methods

- **“Attributes:”** These are the characteristics/properties of an object. For instance, a Car might have attributes like colour, make, and model.
- **Methods/functions:** These are the functions that define the behaviours of an object. For example, a Car class might have methods like start (), stop (), and drive ().

1.7 ADVANTAGES/BENEFITS OF OOP

- **Reusability:** Code can be reused with the help of inheritance and interfaces.
- **Modularity:** Break down large programs into smaller and more manageable sections.
- **Encapsulation:** It allows for minimum influence on other parts of the software, making changes easier to manage.

- **Improved program Development Process:** OOP helps to manage complex software development and improves communication between team members.

1.8 USAGE/APPLICATIONS OF OOP

- 1 **Software Development:** Most modern programming languages (like Java, Python, C++) support OOP principles which are used in software development for building applications that are easier to maintain and scale.
- 2 **Game Development:** OOP simplifies game design by representing game entities (e.g., players, enemies) as objects, each with their properties and behaviors.
- 3 **GUI Applications:** Graphical User Interfaces (GUIs) are often modeled using OOP concepts, where windows, buttons, and other elements are treated as objects.
- 4 **Banking Systems:** Objects can represent accounts, transactions, and users, with methods for deposit, withdrawal, and balance inquiries.
- 5 **Retail Management:** OOP can be used to model products, customers, and sales, allowing for easy management of inventory and sales processes.
- 6 **Artificial Intelligence (AI):** AI and Machine Learning Libraries: C++ is utilized in AI and machine learning for developing algorithms, neural networks, and data processing libraries, taking advantage of its performance and control over system resources.
- 7 **Real time Systems:** Simulation Software: OOP in C++ is used to create simulations for various real-time systems, such as flight simulators, virtual training environments, and industrial automation systems.



Activity 1.1

Procedural vs Object-Oriented Thinking

Objective: To differentiate procedural and object-oriented programming approaches using real-life examples.

Required Materials: Notebook, pen, reference textbook

Prerequisites: Basic knowledge of programming and real-world problem-solving

Steps:

1. List a real-world system such as a library or bank.
2. Describe it using procedural steps.

NOTE

3. Redesign the same system using objects and classes.

4. Identify advantages of the object-oriented approach.

Post-Activity Reflection: How does object-oriented thinking simplify complex systems?

Expected Outcome: Learners can articulate the need for OOP and relate it to C++ development.

☒ **Check Your Progress 1.1**

A. Multiple Choice Question (MCQ)

1. Which of the following is NOT a basic concept of Object-Oriented Programming (OOP)?

a) Encapsulation

b) Inheritance

c) Polymorphism

d) Compilation

B. True/False

1. Object-Oriented Programming helps in solving the problems of software crisis by promoting modularity and reusability.

C. Short Answer Questions

1. Define Object-Oriented Programming (OOP).

.....

.....

.....

2. Name any two advantages of using OOP.

.....

.....

.....

1.9 SUMMARY

This unit introduces C++ and the object-oriented programming paradigm, explaining the limitations of procedural programming that led to the software crisis. It discusses OOP concepts such as classes, objects, encapsulation, inheritance, abstraction, and polymorphism. The unit highlights the advantages and real-world applications of OOP, emphasizing C++’s role in developing reliable, maintainable, and scalable software systems, thereby laying a strong conceptual foundation for advanced programming learning.

1.10 GLOSSARY

- **C++** – A high-level programming language supporting procedural and object-oriented programming.
- **Software Crisis** – Difficulties in developing and maintaining complex software systems.
- **Object** – An instance of a class representing real-world entities.
- **Class** – A blueprint used to create objects.
- **Encapsulation** – Binding data and methods within a single unit.
- **Inheritance** – Acquiring properties of one class into another.
- **Polymorphism** – Ability of methods to take different forms.

1.11 ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 1.1

A. MCQ Answer:

1. d) Compilation

Explanation: Compilation is a process in programming, not a concept of OOP. OOP concepts include encapsulation, inheritance, and polymorphism.

B. True/False Answer:

1. True

Explanation: OOP promotes modularity, code reusability, and maintainability, which addresses issues associated with the software crisis.

C. Short Answer Answers:

1. **Definition of OOP:** Object-Oriented Programming is a programming paradigm that organizes data and functions into objects, allowing interaction between them to design efficient and reusable software.
2. **Two advantages of OOP:**
 - Code reusability through inheritance
 - Improved modularity and maintainability

1.12 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/Us er/UnitUrls?unitId=1466	

NOTE

NOTE

1.13 REFERENCE

- Object-Oriented Programming with C++ – E. Balagurusamy – ISBN: 9781259029936 – 2017
- The C++ Programming Language – Bjarne Stroustrup – ISBN: 9780321563842 – 2013
- C++ Primer – Stanley B. Lippman, José LaJoie, Barbara E. Moo – ISBN: 9780321714114 – 2012
- Programming with C++ – Ashok N. Kamthane – ISBN: 9788131762269 – 2015

1.14 EXERCISE

Part – 1**A. Multiple Choice Questions****1. Which factor mainly led to the software crisis?**

- A. Lack of hardware
- B. Increase in program complexity
- C. Low-level languages
- D. Lack of users

2. Who introduced the C++ programming language?

- A. Dennis Ritchie
- B. James Gosling
- C. Bjarne Stroustrup
- D. Guido van Rossum

3. Which OOP feature groups data and methods into a single unit?

- A. Inheritance
- B. Polymorphism
- C. Encapsulation
- D. Abstraction

4. Which of the following is NOT an advantage of OOP?

- A. Code reusability
- B. Modularity
- C. High complexity
- D. Easier maintenance

Answer with Explanations**1. B. Increase in program complexity**

Explanation: As software grew larger and more complex, procedural approaches failed to manage development, maintenance, and reliability effectively, leading to the software crisis.

NOTE**2. C. Bjarne Stroustrup**

Explanation: Bjarne Stroustrup developed C++ in the early 1980s by extending the C language with object-oriented features.

3. C. Encapsulation

Explanation: Encapsulation binds data and related methods into a single unit, protecting data from outside interference and misuse.

4. C. High complexity

Explanation: OOP reduces complexity by organizing code into reusable and manageable components, not increasing it.

B. Short Answer Type Questions

1. Why is this unit important in learning C++?
2. What is the software crisis?
3. How does OOP help manage complex software systems?
4. What are the basic principles of OOP?
5. Explain the concept of encapsulation with an example.

Part – 2**C. Long Answer Type Questions (200 words)**

1. Explain the software crisis and how C++ helps overcome it.
2. Describe the object-oriented programming paradigm and its key features.
3. What is abstraction in OOP?
4. Differentiate between procedure-oriented and object-oriented programming.
5. Explain inheritance and its benefits.

Part – 3**D. Long Answer Type Questions (300 words)**

1. Discuss the evolution of C++ and its relevance in modern software development.
2. Explain the principles of Object-Oriented Programming and their real-world applications.
3. Discuss the advantages of OOP in software development.
4. Define Object-Oriented Programming (OOP).
5. Write detailed summary of the unit with examples.

UNIT 2

C++ PROGRAMMING ENVIRONMENT

NOTE

UNIT STRUCTURE

- 2.1 Introduction
- 2.2 Objectives
- 2.3 C++ Program Development Environment
- 2.4 The Language
- 2.5 C++ Language Standards
- 2.6 Introduction to Various C++ Compilers
- 2.7 The C++ standard library
- 2.8 Summary
- 2.9 Glossary
- 2.10 Answer to check your progress
- 2.11 Suggested Readings – OER
- 2.12 Reference
- 2.13 Exercise

2.1 INTRODUCTION

Having understood what C++ is and why object-oriented programming is essential, learners now proceed to understand how C++ programs are actually developed and executed. This unit builds upon the conceptual foundation established in the previous unit and focuses on the practical infrastructure required to implement C++ programs.

The unit introduces the C++ program development environment, explaining the roles of editors, compilers, linkers, and loaders in the program execution cycle. Learners study different C++ language standards and understand why standardization is important for portability and compatibility of programs. Various C++ compilers are introduced to familiarize learners with commonly used tools, including traditional environments such as Turbo C++ IDE as well as modern compilation approaches.

The C++ standard library is also introduced, highlighting its importance in providing reusable functions and classes. Learners gain an understanding of how a C++ program is written, compiled, linked, and executed using both integrated development environments (IDEs) and command-line interfaces. This unit ensures that learners are comfortable with setting up a development environment and running programs independently.

Such practical knowledge is essential before learning to write C++ code using language syntax. With this foundation, learners are ready to explore the core elements of the language itself, including program structure, data types, operators, and control statements, which are introduced in the next unit.

NOTE**2.2 OBJECTIVES**

After completing this unit, learners will be able to:

- Identify components of the C++ program development environment.
- Compare different C++ compilers and development tools.
- Execute the steps involved in compiling and running a C++ program.
- Examine the role of C++ language standards and standard libraries.
- Demonstrate program execution using both IDE and command-line methods.

2.3 C++ PROGRAM DEVELOPMENT ENVIRONMENT

The first step in programming in C++ is to set up your development environment. This involves installing a compiler, an “Integrated Development Environment” (IDE) like Turbo C++, Visual Studio or Code: Blocks.

2.4 THE LANGUAGE**Definition of a “Programming Language”**

A programming “language” is a “set of instructions” that are used to communicate with a computer. This language allows you to write/design code that the computer can execute to perform specific tasks. Programming languages can vary widely in their syntax, complexity, and capabilities.

Types of Programming Languages

- 1 High-Level Languages:** These languages are easier to read and write. Examples include C, C++, Python and Java.
- 2 Low-Level Languages:** These languages are closer to machine code and require more technical knowledge to understand. Examples include Assembly and C.
- 3 Compiled vs. Interpreted Languages:**
 - **Compiled Languages:** Translated into machine code before execution (e.g., C, C++).
 - **Interpreted Languages:** Translated during execution (e.g., Python, JavaScript).

Purpose of Programming Languages

Programming languages allow developers to create software applications, automate tasks, manipulate data, and build websites, among other functions. Understanding the purpose of a language helps in selecting the right one for specific tasks.

2.5 C++ LANGUAGE STANDARDS

NOTE

C++ has evolved through several standards, each introducing new features and improving existing functionalities.

Evolution of C++ Standards

1. **C++98(1998):** The first standardized version of C++. Introduced features like templates and the Standard Template Library (STL).
2. **C++03(2003):** A bug fix release for C++98, clarifying some ambiguities and fixing minor issues.
3. **C++11(2011):** A major update that introduced features such as auto keyword, range-based for loops, lambda expressions, and nullptr.
4. **C++14(2014):** A small update that improved some C++11 features and added new functionalities such as generic lambdas.
5. **C++17(2017):** Introduced features like optional values, string view, and filesystems making programming easier.
6. **C++20(2020):** A significant update that added concepts, ranges, and coroutines, further enhancing the language's capabilities.
7. **C++23(2023):** Latest standard with further enhancements. Focused on improving usability and extending features from C++20.
8. **C++26 (Expected):** Next expected standard focusing on incremental updates and potential new features to keep up with modern programming needs.

2.6 INTRODUCTION TO VARIOUS C++ COMPILERS

Here's a list of popular C++ compilers, their owners, and the latest versions as of 2024:

1. GCC (GNU Compiler Collection)

- It is owned by Free Software Foundation (FSF) and the latest version is GCC 13.2 (Released in July 2023)

2. Clang

- It is owned by LLVM Project, under the LLVM Foundation and the latest version is Clang 16.0.6 (Released in August 2023)

3. MSVC (Microsoft Visual C++)

- It is owned by Microsoft and the latest version is MSVC 19.38 (Part of Visual Studio 2022, Version 17.7, Released in August 2023)

NOTE

4. Intel C++ Compiler (ICC/ICX)

- It is owned by Intel and the latest version is Intel one API DPC++/C++ Compiler 2023.2 (Released in June 2023)

5. Borland C++/Embarcadero C++Builder

- It is owned by Embarcadero Technologies and the latest version is C++Builder 11.3 Alexandria (Released in March 2023)

6. IBM XL C/C++ Compiler

- It is owned by IBM and the latest version is IBM XL C/C++ for AIX 16.1.1 (Released in April 2023)

7. Oracle Developer Studio C++ Compiler

- It is owned by Oracle and the latest version is Oracle Developer Studio 12.6 (Released in November 2019)

8. PGI/NVIDIA HPC C++ Compiler

- It is owned by NVIDIA and the latest version is NVIDIA HPC SDK 23.5 (Released in May 2023)

These compilers support various C++ standards and are used across different platforms and environments for C++ development.

2.7 THE C++ STANDARD LIBRARY

Understanding the “C++ Standard Library”

“The C++ Standard Library” is a powerful set of classes and functions/methods that you can use to accomplish common programming tasks. A well-structured library can save time and effort, allowing you to focus on designing your application rather than implementing every detail from scratch.

Components of the Standard Library:

- **Containers:** These are data structures that store objects. Examples include vectors, lists, and maps.
- **Algorithms:** Pre-defined functions that perform actions on containers, such as sorting, searching, and manipulating data.
- **Iterators:** These allow accessing elements of a container in a sequential manner without exposing the underlying structure.
- **Input/Output (I/O) Library:** This enables you to read and write data, whether from console input or files.
- **“Standard Template Library” (STL):** A collection of template classes and functions providing general-purpose types and algorithms.

NOTE**Activity 2.1**

Exploring the C++ Toolchain

Objective: To execute a basic C++ program using IDE and command-line tools.

Required Materials: Computer, C++ compiler/IDE

Prerequisites: Awareness of C++ language basics

Steps:

1. Write a simple C++ program.
2. Compile and run it using an IDE.
3. Compile and run the same program via command line.
4. Compare outputs and errors.

Post-Activity Reflection: Which method offers better control and why?

Expected Outcome: Learners can independently compile and execute C++ programs.

☑ Check Your Progress 2.1**A. Multiple Choice Question (MCQ)**

1. Which of the following is considered the standard library in C++?
 - a) iostream
 - b) stdio.h
 - c) conio.h
 - d) graphics.h

B. True/False

1. All C++ compilers strictly follow the same C++ language standard.

C. Short Answer Questions

1. What is the C++ standard library?

.....

2. Name any two popular C++ compilers.

.....

2.8 SUMMARY

This unit explains how C++ programs are developed and executed using appropriate tools and environments. It introduces the components of the C++ development process, including editors, compilers, and standard libraries. The unit highlights the importance of C++ language standards for portability and

NOTE

compares popular compilers used in modern development. Learners gain practical knowledge of program compilation and execution using IDEs and command-line tools. This foundation prepares learners to understand C++ syntax and core programming constructs in subsequent units.

2.9 GLOSSARY

- **Compiler** – Software that translates C++ source code into machine code.
- **IDE** – Integrated Development Environment combining editor, compiler, and debugger.
- **C++ Standard** – A formal definition of C++ language rules and features.
- **STL** – Standard Template Library providing containers and algorithms.
- **Container** – A data structure that stores objects.
- **Algorithm** – A predefined function for processing data.
- **Iterator** – An object used to traverse container elements.
- **Executable** – A program file ready for execution.

2.10 ANSWER TO CHECK YOUR PROGRESS**Check Your Progress 2.1****A. MCQ Answer:**

1. a) **iostream**

Explanation: The C++ standard library provides classes and functions such as `iostream` for input/output, `vector`, `string`, and other standard utilities.

B. True/False Answer:

1. **False**

Explanation: Different C++ compilers may implement the language standard differently, though they aim to follow the ISO C++ standard as closely as possible.

C. Short Answer Answers:

1. **C++ Standard Library:** It is a collection of classes and functions provided by C++ to perform common tasks like input/output, data manipulation, and algorithms, which helps in writing efficient programs.
2. **Two popular C++ compilers:**
 - GCC (GNU Compiler Collection)
 - Microsoft Visual C++ (MSVC)

2.11 SUGGESTED READINGS – OER**NOTE**

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/Us er/UnitUrls?unitId=1467	

2.12 REFERENCE

- C++ Primer – Stanley B. Lippman, José LaJoie, Barbara E. Moo – ISBN: 9780321714114 – 2012
- The C++ Programming Language – Bjarne Stroustrup – ISBN: 9780321563842 – 2013
- Programming with C++ – Ashok N. Kamthane – ISBN: 9788131762269 – 2015
- Object-Oriented Programming with C++ – E. Balagurusamy – ISBN: 9781259029936 – 2017

2.13 EXERCISE**Part – 1****A. Multiple Choice Questions**

- Which component translates C++ source code into machine-readable code?**
 - Editor
 - Compiler
 - Loader
 - Debugger
- Why are C++ language standards important?**
 - To increase program length
 - To ensure portability and compatibility
 - To restrict compiler development
 - To eliminate libraries
- Which of the following is an interpreted language?**
 - C
 - C++
 - Python
 - Assembly

NOTE

4. Which C++ standard introduced lambda expressions?

- A. C++03
- B. C++98
- C. C++11
- D. C++17

5. Which component of the C++ Standard Library stores data objects?

- A. Algorithms
- B. Containers
- C. Iterators
- D. I/O Library

Answer with Explanations:

1. B. Compiler

Explanation: The compiler converts high-level C++ code into machine code before execution.

2. B. To ensure portability and compatibility

Explanation: Standardization ensures that C++ programs behave consistently across different platforms and compilers.

3. C. Python

Explanation: Python is interpreted at runtime, whereas C and C++ are compiled before execution.

4. C. C++11

Explanation: C++11 introduced major features including lambda expressions, auto keyword, and nullptr.

5. B. Containers

Explanation: Containers such as vectors and lists store collections of objects.

B. Short Answer Type Questions

1. What is a programming language?
2. What is the role of an IDE in C++ development?
3. Differentiate between compiled and interpreted languages.
4. What is a compiler? Name some popular C++ compilers.
5. Explain the role of an IDE in C++ programming.

Part – 2

C. Long Answer Type Questions (200 words)

1. Explain the C++ program development environment.

2. Describe the evolution of C++ language standards.
3. What is the structure of a basic C++ program?
4. Describe the use of header files in C++.
5. What is the difference between syntax errors and logical errors?

Part – 3**D. Long Answer Type Questions (300 words)**

1. Discuss various C++ compilers and their significance in program development.
2. Explain the importance of the C++ Standard Library in software development.
3. Explain the process of debugging in C++.
4. Write a simple C++ program and explain each part.
5. What are the steps involved in compiling and executing a C++ program?

UNIT 3

BASIC C++ SYNTAX AND CONSTRUCTS

NOTE

UNIT STRUCTURE

- 3.1 Introduction
- 3.2 Objectives
- 3.3 Prototype of Main () Function
- 3.4 I/O Operator
- 3.5 Manipulator
- 3.6 Comments
- 3.7 Data Types
- 3.8 Turbo C++ Ide
- 3.9 Creating, Compiling and Running A C++ Program Using Data And Through Command Line
- 3.10 Elements of C++ Language
- 3.11 Structure of A C++ Program
- 3.12 C++ Tokens
- 3.13 Type Conversion in Expressions
- 3.14 Sequential Statements
- 3.15 Mathematical Functions
- 3.16 Branching Statements
- 3.17 Looping Statements
- 3.18 Nested Loops
- 3.19 Summary
- 3.20 Glossary
- 3.21 Answer to Check Your Progress
- 3.22 Suggested Readings – OER
- 3.23 Reference
- 3.24 Exercise

3.1 INTRODUCTION

After becoming familiar with the C++ development environment, learners now focus on understanding the language itself. This unit builds directly on the ability to create and execute programs and introduces the fundamental syntax and constructs of C++. Learners study the structure of a C++ program, including the prototype of the `main()` function, header files, namespaces, and program organization. The unit explains C++ tokens, keywords, identifiers, constants, and variables, along with fundamental data types and type conversion in expressions. Input and output operations using stream operators and manipulators are introduced to enable interaction with users.

The unit also covers comments for documentation and readability. Control structures form a major part of this unit, including sequential execution, branching statements such as `if` and `switch`, looping constructs such as `for`, `while`, and `do-while`, and nested loops. Mathematical functions from standard libraries are also discussed. By the end of this unit, learners can write complete C++ programs that solve simple problems using logic and control

NOTE

flow. These programming basics prepare learners to manage larger sets of data efficiently, leading directly to the study of arrays in the next unit.

3.2 OBJECTIVES

After completing this unit, learners will be able to:

- Construct a complete C++ program using proper program structure.
- Apply data types, operators, and expressions in program development.
- Implement input and output operations using stream operators.
- Develop programs using branching and looping constructs.
- Analyze program flow using nested control structures.

3.3 PROTOTYPE OF MAIN () FUNCTION

main () in C++ acts as the entry point for program execution. The prototype of main () depends on how arguments are handled by it.

Following are the common prototypes for main () function:

1. **No arguments main ():** This is the simplest form of main () in which takes no parameters but returns an integer value. The return value indicates whether the program succeeded or failed.

Syntax

```
int main ();
```

Example Program:

```
#include <iostream.h>|  
  
int main () {  
    cout << "Welcome in C++" << endl;  
    return 0;  
}
```

OUTPUT:

```
Welcome in C++
```

Explanation: cout object and << operator which are defined in iostream header are used to print output on standard output device (monitor). So, string Welcome in C++ written within double quotes print on the screen as output of program.

2. **With arguments main ():** This form of main () allows the program to receive "command line arguments" from the "operating system's" terminal.

Syntax

```
"int main (int argc, char *argv []);"
```

NOTE**Parameters:**

- **int argc:** These refer to the number of command-line arguments passed to the program, including its name (at index 0).
- **char *argv []:** It is an array of strings representing “command-line arguments”. Here,

argv [0] : the name of the program

argv [1] to argv[argc-1]: the additional arguments.

return value of main () represents the exit status of the written program. If return value is 0 it represents successful execution of the program while non zero value represents error or abnormal termination of the program.

Example Program:

```
#include <iostream.h>

int main (int argc, char *argv[]) {
    cout << "Program Name: " << argv [0] << endl;
    cout << "Number of Arguments: " << argc << endl;
    for (int i = 1; i < argc; ++i) {
        cout << "Argument " << i << ": " << argv[i] << endl;
    }
    return 0;
}
```

Command to run program

```
./program_name arg1 arg2 arg3
```

OUTPUT

```
Program Name: ./program_name
Number of Arguments: 4
Argument 1: arg1
Argument 2: arg2
Argument 3: arg3
```

3.4 I/O OPERATOR

Input and Output operators in C++ are used to take input from keyboard and Output data to the monitor.

Input Operator (>>): It is called as extraction or get from operator (>>) and is used to take input from the standard input stream (cin)/Keyboard and stores it in a variable.

Output Operator (<<): It is called as insertion or put to operator (<<) and is used to Output data to the standard Output stream (cout)/Monitor.

Example:

```
#include<iostream.h>

void main ()
```

NOTE

```

{
    int age;

    cout << "Enter your age: "; // Output to monitor
    cin >> age; // Input from keyboard
    cout<<" your age is" <<age;
}

```

Output

Enter your age: 18

Your age is 18

Explanation: First, string "Enter your age:" displayed on screen with the help of cout and <<. Then you have to enter any number (let 18) and with the help of cin and >> number stored in age variable. Again with the help of cout<< ,your age is 18 displayed on screen.

3.5 MANIPULATOR

Manipulators are the functions in C++ used to modify the formatting of input and Output operators in C++.

Commonly used manipulators are

std::endl	used for newline
std::setw()	fused or setting the width of Output
std::precision	used for controlling the precision of floating point numbers.
std::fixed	used for fixed point notation

Example:

```

double pi = 3.14159;

cout << std::fixed << std::setprecision(2) << pi; // Output: 3.14

```

3.6 COMMENTS

Comments in C++ are sentence(s) within the code to explain the purpose of that code. These are not executed by the compiler. They are used in program to make the program code understandable to reader.

Types of comments in C++

- 1. Single line comment:** Used for writing comment in single line. It starts with

//.

Example

```
int a; // This is a variable of type integer
```

NOTE

2. Multiline comment: Used for writing multiple line comments. It starts with

`/*` and ends with `*/`, everything in between is a comment.

Example

`/* It is a multi-line comment`

Which is used to describe something

in multiple lines. `*/`

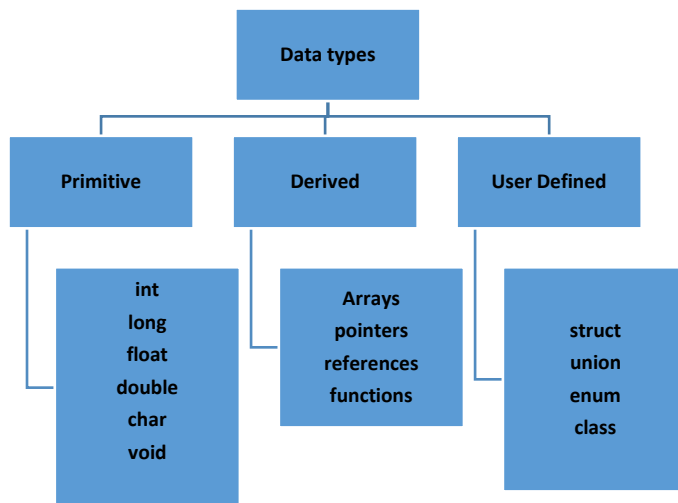
3.7 DATA TYPES

Fig. No. 1.1. Data types in C++

In C++ data types are used to define the type of data a variable can hold. These data types help in storing and manipulating different kind of data.

Basic or primitive or built-in datatypes used in C++ are:

1. **int:** It holds whole numbers (e.g. `int a=45`)
2. **long:** It is used to store larger whole numbers than the `int` data type can handle.
3. **float:** It stores decimal numbers with single precision (e.g. `float b=4.5`)
4. **double:** It stores decimal numbers with double precision (e.g. `float b=4.56473`)
5. **char:** It is used to hold a single character (e.g. `char grade= 'B'`)
6. **void:** It represents the absence of a value. It is mainly used in functions when it doesn't return a value.

Example:

void getdata ()

```

{
    int n;
    cout<< "enter any number";
    cin>>n;
}
  
```

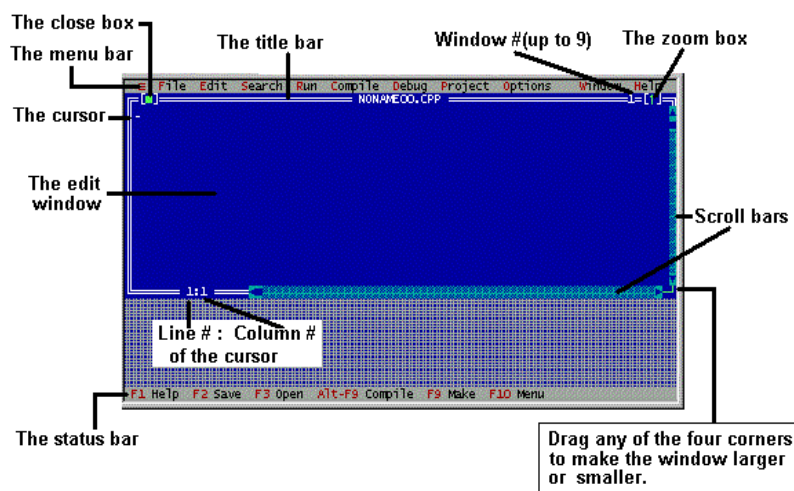
NOTE**3.8 TURBO C++ IDE**

Fig. No. 1.2. Turbo C++ IDE(Source: <https://spot.pcc.edu/~wmorales/cs160/images/cref12c.gif>)

3.9 CREATING, COMPILING AND RUNNING A C++ PROGRAM USING DATA AND THROUGH COMMAND LINE

Steps to Create a C++ Program in Turbo C++

Open Turbo C++ IDE and follow the following steps:

1. Start the Turbo C++ IDE from where it is installed.
2. Create a New File using "File menu" and then select "New" option.
3. Write the C++ program in the editor window.
4. Save the File using File => Save => filename.cpp (Like hello.cpp)
5. Compile the program by pressing Alt + F9 or using "Compile Menu". The IDE will show errors if you mistake in the syntax of the program and that need to be corrected.
6. Run the compiled program by pressing Ctrl + F9 or using "Run Menu".

Executing C++ Program from Command Line

Steps to execute a Turbo C++ program that takes command-line arguments from the Windows terminal/Command Prompt

Step 1: Write the C++ Program

```
#include <iostream.h>

void main(int argc, char* argv[]) {
    // argc is the number of arguments
    // argv is an array of C-strings (char arrays) representing the
    arguments
    if (argc > 1) {
        cout << "Command-line arguments received." << endl;
    }
}
```

NOTE

```

    for (int i = 1; i < argc; ++i) {
        cout << "Argument " << i << ": " << argv[i] << endl;
    }
}
else {
    cout << "No command-line arguments provided." << endl;
}
}

```

Step 2: Compile the Program Using Turbo C++

- i. Open Turbo C++.
- ii. Write or Load the Program and save it with name myprogram.cpp (or any another name)
- iii. Compile the Program:
 - Press Alt + F9 to compile the program.
 - Correct syntax errors, if any.

Step 3: Build the Executable

1. Build the Program:

Press Ctrl + F9 to build the program. This step creates an executable file, usually with name myprogram.exe in the same directory as source code.

Step 4: Run the Executable from Windows Terminal

- i. Open the Windows Command Prompt:

Press Win + R, type cmd, and press Enter.
- ii. Navigate to the Directory (Check the installation location in your computer):

Use the cd command to navigate to the directory where your Turbo C++ executable is located. For example, in my computer it is:

C:\TurboC++\Disk\TurboC3\BIN>
- iii. Run the Program with Command-Line Arguments:

To execute the program with command-line arguments, type the name of the executable followed by the arguments. For example:

myprogram.exe argument1 argument2 argument3

Example 1: To pass hello world as argument, type (Output will appear in command window)

myprogram.exe hello world

NOTE**Output 1**

Command-line arguments received:

Argument 1: hello

Argument 2: world

Example 2: Without argument, type

myprogram.exe

Output 2

No command-line arguments provided.

Step 5: Close Turbo C++

Close Turbo C++ by selecting File => Quit, or simply press Alt + X.

Explanation

-argc: int value, represents the number of command-line arguments passed to the program, including the program name.

-argv[]: Array contains the arguments as C-strings.

Where

argv[0] :program's name

argv[1], argv[2], etc. are the arguments passed from the command line.

3.10 ELEMENTS OF C++ LANGUAGE

C++ is a programming language which builds on C programming language with object-oriented features. It has some key elements that work together to create robust and efficient C++ programs, which are as follows:

- 1. Variables:** These are used to store data in memory locations.

```
int a;
```

Here 'a' is an integer variable used to store integer value.

- 2. Data types:** It defines the type of data. Examples of some built-in data types in C++ are int, float, char.

```
float b=2.5;
```

Here 'b' is a variable of data type float, which is used to store 2.5 value in memory location specified by 'b' variable.

- 3. Operators:** These are symbols that are used to perform operations on values or on variables.

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

Here '+' operator calculate the sum of values stored in 'a' and 'b' variable.

Example of some commonly used operators in C++ are

- Arithmetic operators: +, -, *, /, %
- Relational operators: <, >, <=, >=, !=, ==

NOTE

- Logical operators: &&, ||, !

4. Control Structures: It directs the flow of the C++ program like if, for, while.

```
if(ag>=18)
```

```
    cout<< "You are eligible to give vote";
```

```
else
```

```
    cout<< "You are not eligible to give vote";
```

In above code only one statement will execute depending on the age of person.

5. Functions: These are the block of codes that perform specific tasks.

```
int sum (int a, int b) // This function is used to calculate sum of two
numbers
```

```
{
```

```
    return a+b;
```

```
}
```

6. Class and Objects: These are the fundamental units of object-oriented programming language like C++ (in detail we study in Unit 2).

Class Student

```
class Student
```

```
{
```

```
private:
```

```
    int rollno;
```

```
public:
```

```
void get_rollno ()
```

```
{
```

```
    cout<< "enter your roll no";
```

```
    cin>>rollno;
```

```
}
```

```
}; //end of class
```

Object of Student class

```
Student S;
```

3.11 STRUCTURE OF A C++ PROGRAM

C++ Headers	#include < iostream.h>	Header File
Class definition	class Example	
	{	

NOTE

	};	
	void main ()	Main Function
	{	Start of main ()
Function Body	int a=5, b=6;	Declaration of
Variables	int c=a+b;	Expressions
	cout<<"sum="<<c;	Output
	}	End of main ()

Tab. No. 1.1 Structure of C++ program using addition of 2 numbers

Example of basic C++ "Hello, World!" program.

```
#include <iostream.h>
#include <conio.h>
void main ()
{
    clrscr ();
    cout << "Hello, World!"<<endl<<"Welcome in C++";
    getch ();
}
```

OUTPUT

```
Hello, World!
Welcome in C++
```

Explanation:**1. #include <iostream.h>****Purpose:**

- It tells the compiler to include the standard input-output stream library.
- It provides the necessary functionality for input and output operations in C++, including cout for output and cin for input.

2. #include <conio.h>**Purpose:**

- It includes the console input-output header file.
- It is often used in older C++ programs to provide functions like clrscr() to clear the screen and getch () which waits for a key press.

Imp: conio.h is non-standard and mainly used in DOS-based compilers like Turbo C++. Modern C++ programs typically avoid using conio.h

3. void main ()**Purpose:**

This defines the main function where the program begins execution.

NOTE

Void keyword: represents that the function does not return any value.

Imp: standard signature for the main function in modern C++ is `int main ()`, which should return an integer value to the operating system, typically 0 for successful execution.

4. clrscr ()**Purpose:**

- It is part of the `conio.h` library and is used to clear the output screen.

5. {} (Curly Braces)**Purpose:**

- These braces define the start { and end } of a block of code.
- Here they enclose the body of the main function.

6. cout**Purpose:**

- `cout` is the standard output stream in C++, used to print text or variables to the console/monitor.

7. << (Insertion Operator)**Purpose:**

This operator is used with `cout` to insert or send the data (in this case, the string "Hello, World!") to the output stream/monitor.

8. "Hello, World!"**Purpose:**

- This is a string literal, a sequence of characters enclosed in double quotes. It is the text that will be displayed on the console/monitor.

9. getch ()**Purpose:**

- This function waits for a key press before closing the console window.
- It is part of the `conio.h` library and is used to hold the output screen until a key is pressed.

10. endl: Adds a newline character and flushes the output buffer.

3.12 C++ TOKENS

Tokens are the smallest unit of a C++ program that the compiler recognizes. There are several types of tokens:

- 1. Keywords:** These are reserved or predefined words that have special meaning in C++.

Examples: `int`, `if`, `while`, `for`, `return` etc.

NOTE

- 2. Identifiers:** These are the names given to variables, functions, classes etc. and these are defined by the programmer. It has some rules to write like it must start with a letter or underscore, followed by letters, digits, or underscore.

Example:

```
int area;

void add ();

class Student {    };
```

- 3. Constants (Literals):** These are fixed values that do not change in program.

Example:

```
const float pi=3.14567;

const int maxAge=60;
```

- 4. Operators:** These are symbols/operators that perform operations on variables and values.

Example: +, -, *, /, %, =, ++

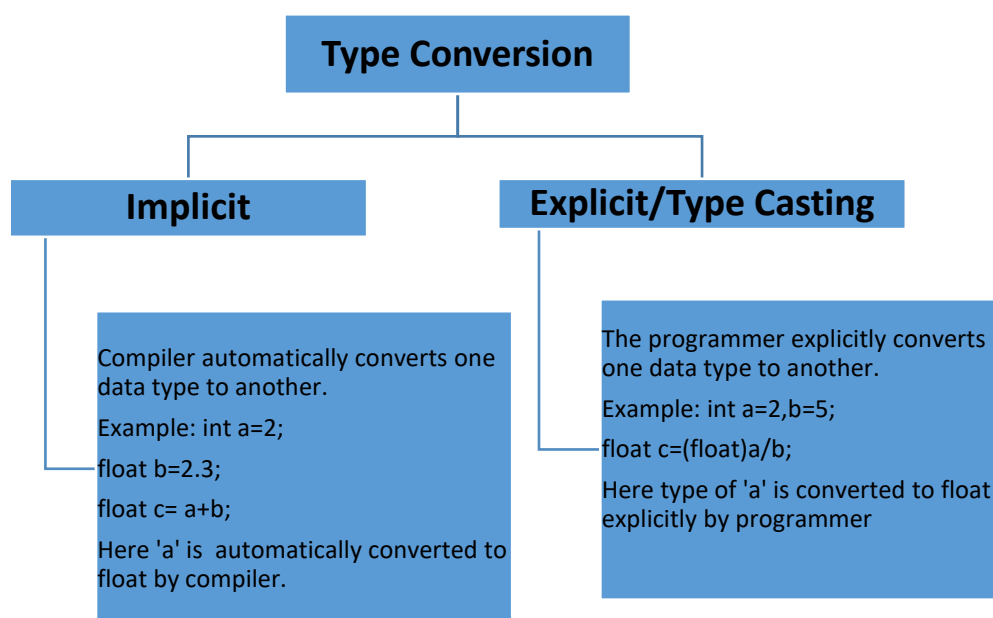
```
int c=a+b;
```

- 5. Punctuation/Symbols:** These are characters that have special meanings like ending a statement or separating elements.

Example: (), {}, [], ;

```
cout<<"Hello";
```

```
int a=2*(2+3);
```

3.13 TYPE CONVERSION IN EXPRESSIONS*Fig. No. 1.3. Type Conversion in expressions*

NOTE**3.14 SEQUENTIAL STATEMENTS**

Sequential statements in C++ program are executed in the same order in which they are written that is one after another.

Example:

```
#include<iostream.h>>

void main ()
{
    int a=2; // 1st sequential statement: variable a is assigned the value 2
    int b=3; // 2nd sequential statement: variable b is assigned the value 3
    int c=a+b; // 3rd sequential statement: sum of a and b is calculated and
    stored in c
    cout<<"sum="<<c; // 4th sequential statement: the sum is printed on
    screen
}
```

OUTPUT

sum=5

In this program each statement is executed in sequential order and it makes the flow straightforward and easy to follow.

3.15 MATHEMATICAL FUNCTIONS

There is a header file in C++ named "math.h" in which there are various mathematical functions defined which are as follows:

- **sqrt(x):** This function returns square root of x. (e.g. sqrt (36) =6)
- **pow (x, y):** This function returns value of x^y . (e.g. pow (2,3) =8)
- **abs(x):** This function returns absolute value of x. (e.g. abs (-3) =3)
- **ceil(x):** This function returns the smallest integer greater than or equal to x. (e.g. ceil (2.3) =3)
- **floor(x):** This function returns the largest integer less than or equal to x. (e.g. floor (2.3) =2)
- **sin(x), cos(x), tan(x):** These functions return the sine, cosine, and tangent of x (in radians).
- **asin(x), acos(x), atan(x):** These are inverse trigonometric functions that takes a value 'x' and returns angle in radians.
- **log(x):** This function returns the natural logarithm of x.
- **exp(x):** This function returns the exponential value of x.

3.16 BRANCHING STATEMENTS

Branching statements in C++ language allows the program to make decisions and execute different set of statements based on conditions.

NOTE

The main branching statements in C++ are:

- 1) if statement
- 2) if else statement
- 3) Nested if else statement
- 4) Ladder if else statement
- 5) switch case statement

1) if statement

If statement executes code inside the if block, if the specified condition is true.

Example Program:

```
#include<iostream.h>

#include<conio.h>

void main ()
{
    clrscr ();
    int a=2;
    if(a==2)
        cout<< "Number is equals to 2";
    getch ();
}
```

Output

Number is equals to 2

Explanation:

The condition `a==2` is true, so the code inside the if block executes, printing "Number is equals to 2."

2) if else statement

In the if else statement, if statement executes a code inside the if block, if the specified condition is true and if it is false, statements associated with else statement will be executed.

Syntax:

```
if (condition) {
    // code to be executed if the condition is true
}
else {
    // code to be executed if the condition is false
}
```

NOTE**Points to Remember:**

- 1 or non-zero value in condition part represents true value
- 0 represents false.
- { } are optional when if else contains single statement. But recommended

to use for easy understanding of scope of if and else.

Example Program:

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

void main ()
{
    clrscr ();
    int a=3;
    if(a==2) {
        cout<< "Number is equals to 2";}
    else {
        cout<< "Number is not equals to 2";}
    getch ();
}
```

Output

Number is not equals to 2

Explanation:

The condition value of 'a' equals to 2 (a==2) is false, so the code inside the else block executed and "Number is not equals to 2" is printed.

3) Nested if else statement

A nested if else statement is an if else statement inside another if else statement. It is used when you want to check multiple conditions in a sequence and each decision is depending on the outcome of the previous one.

Example Program:

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

void main ()
{
    clrscr ();
    int n=5;
```

NOTE

```
if (n>=0)
{
    if(n%2==0)           // if else inside if else statement.
        cout<< "Number is positive and even";
    else
        cout<< "Number is positive and odd";
}
else
{
    if(n%2==0)
        cout<< "Number is negative and even";
    else
        cout<< "Number is negative and odd";
}
getch ();
}
```

Output

Number is positive and odd

Explanation:

The condition $n \geq 0$ is true with outer if statement and $(n \% 2 == 0)$ is false with inner if statement, so the code inside the else block of inner if statement is executed and "Number is positive and odd." is printed on screen.

Here, '%' is modulo operator which returns remainder after division.

Example: When we divide 5 by 2, we get quotient =2 and remainder=1

When we want result as quotient value, we write $5/2(=2)$ and for remainder we write $5\%2(=1)$.

4) Ladder if else statement

A ladder if else statement is a series of if, else if and else statements that are used to check multiple conditions in sequence. The program checks each condition one by one until it finds a true condition and then it executes the corresponding set of statements.

Example Program:

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

void main ()
{
```

NOTE

```

clrscr ();
int n=5;
if (n>=0 && n%2==0)
cout<< "Number is positive and even";
else if (n>=0 && n%2!=0)
cout<< "Number is positive and odd";
else if (n<0 && n%2==0)
cout<< "Number is negative and even";
else
cout<< "Number is negative and odd";
getch ();
}

```

Output

Number is positive and odd

Explanation:

Here, && is a logical 'and' operator. If both conditions combined with && operator are true then it returns true, otherwise it returns false.

In above code condition $n \geq 0$ is true and $(n \% 2 == 0)$ is false so the resultant condition is false in if statement. Now the condition with else if code is checked and as both conditions $n \geq 0$ and $n \% 2$ is not equals to zero ($n \% 2 \neq 0$) are true so resultant condition is true. Therefore, code inside the first else if block is executed and printed "Number is positive and odd."

5) Switch case statement

- Switch-case statement is used to choose between multiple options based on the value of int or char expression/variable.
- It is similar to multiple if-else, but it is faster as it directly jumps to case, which is satisfied instead of checking every case block.
- Each case block contains break; to terminate further execution of switch block.
- Without break statement all case blocks below the one which is satisfied will also get executed.
- default part gets executed when no case will get satisfied.

Syntax:

```

switch (expression)
{

```

NOTE

```
case constant1:
    // code to be executed if expression equals constant1
    break;

case constant2:
    // code to be executed if expression equals constant2
    break;

// you can have any number of case statements

default:
    // code to be executed if none of the cases match

}
```

Example:

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

void main ()
{
    clrscr ();
    int day;

    cout<<"1-Monday\n2-Tuesday\n3-Wednesday\n4-Thursday\n5-
    Friday\n6-Saturday\n7-Sunday\n"; //'\n' and endl in C++ are used
    to change the line

    cout<< "Enter any number to print corresponding day: ";
    cin>>day;

    switch (day)
    {
        case 1: cout << "Monday"; break;
        case 2: cout << "Tuesday"; break;
        case 3: cout << "Wednesday"; break;
        case 4: cout << "Thursday"; break;
        case 5: cout << "Friday"; break;
        case 6: cout << "Saturday"; break;
        case 7: cout << "Sunday"; break;
        default: cout << "Invalid day";
    }
}
```

Output1

NOTE

1-Monday
 2-Tuesday
 3-Wednesday
 4-Thursday
 5-Friday
 6-Saturday
 7-Sunday

Enter any number to print corresponding day: 3

Wednesday

Explanation: As number entered by user is 3 so case 3 of switch statement executed and 'Wednesday' is printed on screen, no other case is executed as break is used to terminate further execution of switch block.

Output2

1-Monday
 2-Tuesday
 3-Wednesday
 4-Thursday
 5-Friday
 6-Saturday
 7-Sunday

Enter any number to print corresponding day: 33

Invalid day

Explanation: As number entered by user is 33 and no such case is available with switch statement so default block executed and 'Invalid day' is printed on screen.

3.17 LOOPING STATEMENTS

Loops in C++, allow a block of code to be executed repeatedly based on given condition. Loops are useful when you need to write same code multiple times.

There are two types of loops: Entry controlled and Exit controlled.

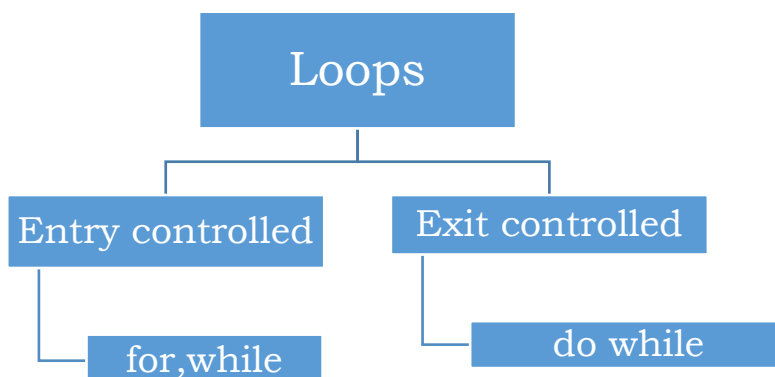


Fig. No. 1.4. Types of Loops

NOTE**Entry controlled Loops**

Entry controlled loops check the condition before executing the code inside the loop.

1. for loop:

for loop is used when number of repetitions/iterations are known in advance.

It is easier to use as compared to other loops as initial/start value, condition and counter can be specified together in a single line separated by semicolon. Condition is checked at the beginning of loop block. So, loop will never execute when condition is false but it continues to execute the code inside the loop as long as given condition remains true.

Syntax:

```
for (initialization; condition; increment/decrement) {
    // code to be executed
}
```

Example 1:

```
for (int i = 1; i < 6; i++) {
    cout << i << " ";
}
```

This will print 1 2 3 4 5

Explanation: From i=1 to 5, condition i<6 is true so the values of i from 1 to 5 printed on screen but at i=6 condition false so loop stops.

Example 2:

Turbo C++ program that uses a for loop to print the multiplication table of a given number.

```
#include<iostream.h>
#include<conio.h>
void main ()
{
    clrscr ();
    int n;
    cout<<" Enter any number";
    cin>>n;
    cout<<" \n Table of entered number "<<n<<"is" <<endl;
    for (int i=1; i<=10; i++)
    {
        cout<<n<<"x" <<i<<" =" <<(n*i) <<endl;
```

NOTE

```

    }
    getch ();
}

```

Output

Enter any number 5

Table of entered number 5 is

```

5x1=5
5x2=10
5x3=15
5x4=20
5x5=25
5x6=30
5x7=35
5x8=40
5x9=45
5x10=50

```

2. while loop:

while loop is used when you don't know the number of iterations in advance. It continues to execute the code inside the loop as long as given condition remains true. As same as **for loop** condition is checked at the beginning of loop block.

Syntax:

```

Initialization;
while (condition) {
    // code to be executed
    Increment/Decrement
}

```

Example:

```

int i = 1;
while (i < 6) {
    cout << i << " ";
    i++; // i++ means i=i+1;
}

```

This will also print: 1 2 3 4 5

Exit controlled Loops

Exit controlled loops check the condition after executing the code inside the loop.

NOTE**3. do while loop**

do while loop is similar to the while loop but it will generate at least one output because the condition is checked after the loop's execution.

Syntax:

```
Initialization;
do {
    // code to be executed
    Increment/Decrement
} while condition;
```

Example:

```
int i = 1;
do {
    cout << i << " ";
    i++;
} while(i<6);
```

This will also print: 1 2 3 4 5

Examples of all loops to print numbers from 5 to 1 in C++.

```
for (i=5; i>=1; i--)
{
    cout<<i<<endl;
}
i=5;
while(i>=1)
{
    cout<<i<<endl;
    i--;
}
i=5;
do
{
    cout<<i<<endl;
    i--;
} while(i>=1);
```

3.18 NESTED LOOPS

Loops within loops are called as nested loops, where inner loop is executed for each iteration of the outer loop.

NOTE**Syntax of nested for loop:**

```

for (initialization; condition; increment/decrement)
{
    for (initialization; condition; increment/decrement)
    {
        // code to be executed
    }
}

```

Example:

```

for (int i = 1; i <= 2; i++) //outer loop
{
    for (int j = 1; j <= 3; j++) //inner loop
    {
        cout << i << " " << j << " ";
    }
    cout << endl;
}

```

Output

1,1 1,2 1,3

2,1 2,2 2,3

Explanation:

Value of i	Value of j	Output
1	1	1,1
1	2	1,1 1,2
1	3	1,1 1,2 1,3
2	1	2,1
2	2	2,1 2,2
2	3	2,1 2,2 2,3

**Activity 3.1**

Control Flow Design.

Objective: To implement decision-making and looping constructs.

Required Materials: Computer, C++ compiler

Prerequisites: Basic program execution knowledge

Steps:

1. Write a program using if-else.
2. Modify it using switch-case.
3. Extend it using loops and nested loops.

NOTE

Post-Activity Reflection: How does control flow affect program logic?

Expected Outcome: Learners can construct complete C++ programs using syntax rules.

☑ Check Your Progress 3.1**A. Multiple Choice Question (MCQ)**

1. Which of the following is the correct way to write a single-line comment in C++?
 - a) /* comment */
 - b) // comment
 - c) # comment
 - d) <!-- comment -->

B. True/False

1. The main() function is mandatory in every C++ program.

C. Short Answer Questions :

1. Define a manipulator in C++. Give an example.

.....

2. What is the difference between while and do-while loops?

.....

3.19 SUMMARY

This unit introduces the fundamental concepts of the C++ programming language. It explains program structure, the main() function, data types, variables, and operators. Input and output operations using stream operators and manipulators are covered for user interaction. The unit extensively discusses control structures, including sequential execution, branching statements, loops, and nested loops. Learners also explore comments, tokens, mathematical functions, and program execution using Turbo C++ and command line. These basics enable students to write logical, structured, and efficient C++ programs, forming a strong foundation for advanced programming concepts.

3.20 GLOSSARY

- **Header File:** File containing predefined functions.

NOTE

- **Main():** Entry point of a C++ program.
- **Variable:** Named memory location.
- **Data Type:** Defines type of data stored.
- **Loop:** Repeated execution block.
- **Branching:** Decision-making statements.
- **Token:** Smallest program unit.
- **Manipulator:** Output formatting function.

3.21 ANSWER TO CHECK YOUR PROGRESS**Check Your Progress 3.1****A. MCQ Answer:**

1. b) // comment

Explanation: In C++, single-line comments start with //. Multi-line comments use /* ... */.

B. True/False Answer:

1. True

Explanation: The main() function is the entry point of every C++ program and is required for execution.

C. Short Answer Answers:

1. **Manipulator:** A manipulator is a special function used with cout or cin to format input/output.
 - Example: endl (inserts a newline), setw(5) (sets field width).
2. **Difference between while and do-while loops:**
 - **While loop:** Condition is checked **before** executing the loop body.
 - **Do-while loop:** Condition is checked **after** executing the loop body, so the loop executes at least once.

3.22 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/User/UnitUrls?unitId=1468	

NOTE

3.23 REFERENCE

- Object-Oriented Programming with C++ – E. Balagurusamy – ISBN: 9781259029936 – 2017
- Programming with C++ – Ashok N. Kamthane – ISBN: 9788131762269 – 2015
- C++: The Complete Reference – Herbert Schildt – ISBN: 9780071506868 – 2003
- Let Us C++ – Yashavant P. Kanetkar – ISBN: 9789389845063 – 2020

3.24 EXERCISE

Part – 1**A. Multiple Choice Questions****1. Which function acts as the entry point of a C++ program?**

- A. start()
- B. main()
- C. begin()
- D. init()

2. Which header file is required for input-output operations in C++?

- A. stdio.h
- B. conio.h
- C. iostream.h
- D. math.h

3. Which operator is used to take input in C++?

- A. <<
- B. >>
- C. =
- D. ::

4. Which manipulator is used to insert a new line?

- A. setw
- B. fixed
- C. endl
- D. precision

5. Which data type stores a single character?

- A. int
- B. float
- C. char
- D. double

Answer with Explanations**1. B. main()**

Explanation: Execution of every C++ program starts from the main() function.

2. C. iostream.h

Explanation: iostream.h provides cin and cout.

3. B. >>

Explanation: >> extracts data from input stream.

4. C. endl

Explanation: endl moves the cursor to the next line.

5. C. char

Explanation: char stores one character.

B. Short Answer Type Questions

1. What is the role of header files in C++?
2. What is an identifier in C++?
3. What is a loop in C++?
4. What are tokens in C++?
5. Define variables and constants with examples.

Part – 2**C. Long Answer Type Questions (200 words)**

1. Explain the structure of a C++ program.
2. Describe branching statements in C++.
3. Explain looping statements in C++.
4. Explain data types available in C++.
5. What are operators? List different types.

Part – 3**D. Long Answer Type Questions (300 words)**

1. Explain data types and variables in C++.
2. Discuss input and output operations in C++.
3. Explain control structures with examples.
4. What are loops? Explain types of loops in C++.
5. Write a program using loops to print numbers from 1 to 10.



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN A/SECT GROUP UNIVERSITY

Approved by: PCI | AICTE | NCTE | UCI | Member of: AU | Joint Committee: UGC | UER | AICTE | Recognized by: UGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:

UNIT 4

ARRAYS AND MULTI-DIMENSIONAL DATA HANDLING

UNIT STRUCTURE

- 4.1 Introduction
- 4.2 Objectives
- 4.3 Arrays - Single-Dimensional Arrays
- 4.4 Two-Dimensional Array (Multi-Dimensional Arrays)
- 4.5 Summary
- 4.6 Glossary
- 4.7 Answer to Check Your Progress
- 4.8 Suggested Readings – OER
- 4.9 Reference
- 4.10 Exercise

4.1 INTRODUCTION

This unit builds upon learners' understanding of variables, data types, and loops from the previous block. While single variables are sufficient for storing individual values, practical programming problems often require handling large collections of data. This unit introduces arrays as a structured method for storing multiple values of the same type under a single name. Learners study single-dimensional arrays, their declaration, initialization, and traversal using loops. The concept of memory representation and indexing is explained to help learners understand how data is stored and accessed.

The unit then extends to two-dimensional and multi-dimensional arrays, which are widely used to represent tables, matrices, and other structured data. Practical examples illustrate how arrays simplify repetitive operations and improve program efficiency. By mastering arrays, learners gain the ability to manage grouped data logically. However, programs that rely solely on arrays can become lengthy and difficult to maintain. To address this challenge, programmers use modular techniques. This realization naturally leads to the study of user-defined functions, which are introduced in the next unit to improve program structure and reusability.

4.2 OBJECTIVES

After completing this unit, learners will be able to:

- Declare and manipulate single-dimensional arrays.
- Implement multi-dimensional arrays for tabular data representation.
- Apply looping constructs to process array elements efficiently.

NOTE

- Analyze memory usage and indexing mechanisms in arrays.
- Develop programs that solve data-handling problems using arrays.

4.3 ARRAYS - SINGLE-DIMENSIONAL ARRAYS

1. Array is a collection of items/elements that have same name and datatype.
2. Elements of the array are identified by their index /subscript number, which starts from 0 and goes to (size of array -1) [in C, C++, Java].
3. It is homogeneous (All elements have same datatype).
4. Compiler allocates contiguous memory location to it.
5. Memory allocation to array at compile time is called as static/compile time array.
6. If memory allocated at runtime, during program execution using malloc/calloc functions in C and new operator in C++, Java, C# that array is called as dynamic/runtime array.

Single Dimension Array

This array has only one dimension/size.

Array Declaration in C++

Datatype arrayname[size];

Example:

```
int a [5];
```

Here,

- **int:** It is the type of data to be stored in the array. We can also use other data types such as char, float, and double.
- **a:** name of the array.
- **5:** It is the size of the array which means only 5 elements can be stored in the array.

Memory allocation of one/ single dimensional array

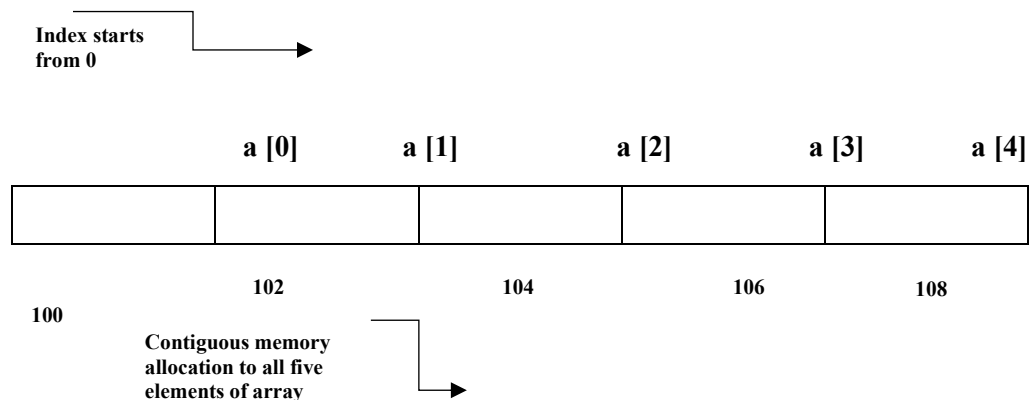


Fig. No. 2.1. Memory allocation of one-dimensional array

NOTE

Size of 1D array=Total memory allocated to integer array of size 5 for 16-bit compiler

=Size of integer datatype*size bytes

=2*5=10 bytes

Initialize Array with Values in C++

```
int a [5] = {1, 2, 3, 4, 5};
```

Initialize Array with Values and without Size in C++

```
int a [ ] = {1, 2, 3, 4, 5};
```

Initialize Array after Declaration (Using Loops)

```
int a [5];
```

```
cout<< "Enter any 5 values";
```

```
for (int i =0; i<5; i++)
```

```
cin>>a[i];
```

Example: Finding Sum of Array Elements

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

```
void main () {
```

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

```
    int sum = 0;
```

```
    for (int i = 0; i < 5; i++) {
```

```
        sum += arr[i];
```

```
    }
```

```
    cout << "Sum of array elements: " << sum << endl;
```

```
}
```

OUTPUT

Sum of array elements: 150

4.4 TWO-DIMENSIONAL ARRAY (MULTI-DIMENSIONAL ARRAYS)

1. Multidimensional Array is having more than one dimension/size.
2. A multi-dimensional array is an array of arrays.

Multidimensional Array Declaration

```
Datatype arrayname [Size1] [Size2] ... [SizeN];
```

Where,

- **Datatype:** Type of data to be stored in the array.
- **arrayname:** Name of the array.

Size1, Size2..., SizeN: Size of each dimension

NOTE

Two-Dimensional Array

1. A two-dimensional array can be visualized as a matrix with rows and columns.
2. Mostly used multidimensional array is two-dimensional array that has two sizes, one for row size and other for column size.

Syntax

Datatype arrayname[row_size] [column_size];

Where,

row_size: Number of rows.

column_size: Number of columns

Example:

```
int a [2][3];
```

Number of elements in 2D array=row_size*column_size
 $=2*3=6$

Memory allocation of two-dimensional array

	Column0	Column1	Column2
Row 0	a [0][0]	a [0][1]	a [0][2]
Row 1	a [1][0]	a [1][1]	a [1][2]

Fig 2.2: Memory allocation of two-dimensional array

Size of given 2D array (Total memory allocated to integer array of size 2x3 for 16-bit compiler)

=Size of integer datatype*row_size*column_size bytes

$=2*2*3=12$ bytes

Example 1: Two-Dimensional Array (Matrix)

```
#include <iostream.h>

void main () {
    int matrix [2][3] = {{1, 2, 3}, {4, 5, 6}};
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
}
```

OUTPUT

1 2 3

4 5 6

NOTE



Activity 4.1

Data Storage Challenge.

Objective: To process structured data using arrays.

Required Materials: Computer, compiler

Prerequisites: Loops and variables

Steps:

1. Create an array for student marks.
2. Compute total and average.
3. Extend to a two-dimensional array.

Post-Activity Reflection: Why are arrays preferred over multiple variables?

Expected Outcome: Learners can apply arrays for efficient data handling.

☑ Check Your Progress 4.1

A. Multiple Choice Question (MCQ)

1. Which of the following correctly declares a one-dimensional integer array of size 5 in C++?
 - a) int arr[5];
 - b) int arr();
 - c) array int[5];
 - d) int[5] arr;

B. True/False

1. A two-dimensional array is used to store data in the form of rows and columns.

C. Short Answer Questions

1. What is a one-dimensional array?

2. What is the main difference between a one-dimensional array and a two-dimensional array?

NOTE

4.5 SUMMARY

This unit introduces arrays as an efficient way to store and manage multiple values of the same data type. It explains single-dimensional and two-dimensional arrays, their declaration, initialization, and memory representation. Learners understand indexing, contiguous memory allocation, and traversal using loops. Practical examples demonstrate how arrays simplify repetitive operations and improve program efficiency. The unit also highlights the importance of arrays in handling large datasets, preparing learners for advanced concepts like user-defined functions and modular programming.

4.6 GLOSSARY

- **Array:** Collection of same data type elements.
- **Index:** Position number of array element.
- **Single-Dimensional Array:** Linear array with one dimension.
- **Two-Dimensional Array:** Array organized in rows and columns.
- **Contiguous Memory:** Continuous memory allocation.
- **Traversal:** Accessing each array element.
- **Static Array:** Compile-time allocated array.
- **Dynamic Array:** Runtime allocated array.

4.7 ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 4.1

A. MCQ Answer:

1. a) `int arr[5];`

Explanation: This is the correct syntax for declaring a one-dimensional array in C++.

B. True/False Answer:

1. **True**

Explanation: A two-dimensional array organizes data in a tabular form using rows and columns.

C. Short Answer Answers:

1. **One-dimensional array:** It is a linear data structure that stores elements of the same data type in contiguous memory locations and is accessed using a single index.
2. **Difference:** A one-dimensional array uses a single index to access elements, while a two-dimensional array uses two indices (row and column) to access elements.

4.8 SUGGESTED READINGS – OER

NOTE

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/Us er/UnitUrls?unitId=1469	

4.9 REFERENCE

- Object-Oriented Programming with C++ – E. Balagurusamy – ISBN: 9781259029936 – 2017
- Programming with C++ – Ashok N. Kamthane – ISBN: 9788131762269 – 2015
- Let Us C++ – Yashavant P. Kanetkar – ISBN: 9789389845063 – 2020
- C++: The Complete Reference – Herbert Schildt – ISBN: 9780071506868 – 2003

4.10 EXERCISE

Part – 1

A. Multiple Choice Questions

1. What is an array in C++?

- A. Collection of different data types
- B. Collection of elements with same name and datatype
- C. A single variable
- D. A function

2. The index of the first element in a C++ array is:

- A. 1
- B. -1
- C. 0
- D. Size of array

3. How is a single-dimensional array declared in C++?

- A. int a;
- B. int a();
- C. int a[5];
- D. array int a;

4. Which memory allocation method assigns memory at compile time?

- A. Dynamic allocation
- B. Runtime allocation
- C. Static allocation
- D. Heap allocation

NOTE**5. A two-dimensional array is best represented as:**

- A. List
- B. Stack
- C. Matrix
- D. Tree

Answer with Explanations**1. B Collection of elements with same name and datatype**

Explanation: An array stores multiple values of the same data type under one name.

2. C 0

Explanation: Array indexing in C++ starts from 0.

3. C int a[5];

Explanation: Arrays are declared using datatype, name, and size.

4. C Static allocation

Explanation: Static arrays are allocated memory during compilation.

5. C Matrix

Explanation: A 2D array contains rows and columns like a matrix.

B. Short Answer Type Questions

1. Define an array.
2. What is array indexing?
3. What is a single-dimensional array?
4. How are multi-dimensional arrays stored in memory?
5. Write a program to add elements of an array.

Part – 2**C. Long Answer Type Questions (200 words)**

1. Explain single-dimensional arrays in C++.
2. Describe memory representation and traversal of arrays.
3. What is an array?
4. How do you declare and initialize arrays in C++?
5. Explain one-dimensional and two-dimensional arrays.

Part – 3**D. Long Answer Type Questions (300 words)**

1. Explain two-dimensional arrays with memory representation.
2. Compare single-dimensional and two-dimensional arrays.
3. Discuss static and dynamic array allocation.
4. Write a program to find the transpose of a matrix.
5. Discuss real-life applications of arrays.

UNIT 5

USER-DEFINED FUNCTIONS AND MODULARITY

NOTE

UNIT STRUCTURE

- 5.1 Introduction
- 5.2 Objectives
- 5.3 User- defined functions
- 5.4 Elements of user-defined functions
- 5.5 Return values and their types
- 5.6 Function calls
- 5.7 Categories of functions
- 5.8 Passing parameters to functions
- 5.9 Recursion
- 5.10 Command line arguments
- 5.11 Storage class specifiers
- 5.12 Summary
- 5.13 Glossary
- 5.14 Answer to Check Your Progress
- 5.15 Suggested readings – OER
- 5.16 Reference
- 5.17 Exercise

5.1 INTRODUCTION

Having learned how to store and process grouped data using arrays, learners now explore modular programming using user-defined functions. This unit builds upon earlier concepts of control structures and data handling. Functions are introduced as independent blocks of code designed to perform specific tasks, reducing redundancy and improving clarity.

Learners study function declaration, definition, and calling mechanisms, along with return values and data types. Parameter passing techniques, including call by value and call by reference, are discussed in detail.

The concept of recursion is introduced to solve problems that can be defined in terms of themselves. Command-line arguments and storage class specifiers are also covered to explain variable scope and lifetime. This unit strengthens structured programming skills and prepares learners to move beyond procedural programming. Once functions and data are grouped together into a single unit, object-oriented programming begins. This transition is the focus of the next unit, where learners are introduced to classes and objects in C++.

NOTE

5.2 OBJECTIVES

After completing this unit, learners will be able to:

- Design user-defined functions for modular program construction.
- Implement parameter passing and return mechanisms in functions.
- Apply recursion to solve repetitive computational problems.
- Examine variable scope using storage class specifiers.
- Develop modular programs using command-line arguments.

5.3 USER- DEFINED FUNCTIONS

Function

- A function is a set of statements which takes some input, does some specific calculation, and produces Output.
- We put some commonly or repeatedly done tasks together to make a function so that instead of writing the same code again and again for different inputs, we can call or use this function.
- In simple terms we can say that a function is a block of code that runs only when it is called.

There are two types of functions:

1. Library functions

- Library functions are also called as built-in or predefined functions.
- These functions are already defined in compiler package.
- So, we don't need to define it in program, we can directly use them by including the header file in C++

Examples: `sqrt ()`, `clrscr ()`, `log ()`, `cos ()` etc.

2. User defined functions

- User defined functions are defined by users to perform some specific tasks.
- These functions are not already defined in compiler package so user have to declare and define it according to their requirement and then use them.

5.4 ELEMENTS OF USER-DEFINED FUNCTIONS

a. Function declaration (or Function Prototype)

Function declaration tells the compiler about the name of function, return type and parameters before the function is actually defined. It is typically placed before `main ()` of the C++ program.

Example: `void sum (int, int);`

`float sum (int, float, int);`

NOTE

b. Return type

Return type denotes the data type of the value that the function returns. It may be int, float, char (any data type). If the function doesn't return any value, then void is used for return type.

Void Sum (int, int);

Here, void indicates that the function sum will not return any value to the calling function.

Float Sum (int, float, int);

Here, float indicates that the function sum will return a decimal value to the calling function.

c. Function name

Function name is the word(identifier) used to call the function. It should be meaningful and reflect the purpose of defined function. In above function declaration sum is the name of function.

d. Arguments (or Parameters)

Arguments/Parameters are the input variables passed to the function. These can be of any data type. Parameters should be specified within the brackets after the name of function.

void sum (int, int);

Here, the function sum has **two arguments**, both are of type integer.

float sum (int, float, int);

Here, the function sum has three arguments: an integer, a float, and another integer.

e. Function definition

It is the code executed when the function is called. The actual task of function implemented in the body of the function. It includes return type, function name, arguments(parameters) and the code of block that performs the actual task of function.

f. Return statement

In a function with a return type, we need to write a return statement at the last line of the function definition.

g. Function call

In the main () or in any function, you call a function by its name followed by brackets with the required parameters so that function will execute and return the desired results.

Tab. No. 2.1 Example of User Defined functions

void sum (int, int);	Function declaration (or Function Prototype)	int sum (int, float, int);
void main ()	Main function (main ()) is calling function)	void main ()

ARRAYS, FUNCTIONS, AND OBJECTS

NOTE

<pre>{ int x=2, y=3;</pre>	x, y, z are actual parameters(arguments)	<pre>{ int x=2, z=3; float y=2.34;</pre>
<pre>sum (x, y);</pre>	Function call	<pre>float t=sum (x, y, z);</pre>
<pre>}</pre>		<pre>cout<<" sum of "<<x<<," "<<y<<"and" "<<z<<" =" "<<t;</pre>
<pre>void sum (int a, int b) { int c=a+b; cout<<" Sum of "<<a<<"and" "<<b<<" =" "<<c; }</pre>	Function Definition sum () is called function a,b,c are formal parameters(arguments)	<pre>int sum (int a, float b, int c) { int z=a+b+c; return z; }</pre>
Sum of 2 and 3=5	OUTPUT	Sum of 2, 2.34 and 3=7.34

Some terms related to functions are:

Calling function: A function which calls another function is called as 'calling function'.

Called function: A function which is called by another function is called as 'called function'.

Actual arguments: Arguments used in calling function are called as 'actual arguments'.

Formal arguments: Arguments used in called function are called as 'formal arguments'.

5.5 RETURN VALUES AND THEIR TYPES

Functions can return values of any valid C++ data type, such as int, float, string, or void (no return).

Example 1: Returning an Integer

```
#include <iostream.h>
int add (int a, int b) {
    return a + b;
}
void main () {
    cout << "Sum: " << add (3, 4) << endl;
```

NOTE

}

Example 2: Returning Void (No Return)

```
#include <iostream.h>

void greet () {
    cout << "Hello, World!" << endl;
}

void main () {
    greet ();
}
```

5.6 FUNCTION CALLS

1. A function is called by using its name and passing appropriate arguments.
2. There are call by value and call by reference methods. For example, refer parameter passing to functions in 2.10 topic.

5.7 CATEGORIES OF FUNCTIONS

1. No arguments and no return value.
2. Arguments but no return value.
3. No arguments but return value.
4. Both arguments and return value.

Tab. No. 2.2 Multiplication of two numbers using all categories of functions

No arguments and no return value.	Arguments but no return value	No arguments but return value	Both arguments and return value
<pre>#include <iostream.h> void multiply () { int a=3, b=4; cout << "Product: " <<(a*b) << endl; } void main () { multiply (); }</pre>	<pre>#include <iostream.h> void multiply (int a, int b) { cout << "Product: " <<(a*b) << endl; } void main () { multiply (3, 4); }</pre>	<pre>#include <iostream.h> int multiply () { return 3 * 4; } void main () { T=multiply (); cout << "Product: " << T<< endl; }</pre>	<pre>#include <iostream.h> int multiply (int a, int b) { return a * b; } void main () { T=multiply (3,4); cout << "Product: " << T<< endl; }</pre>

NOTE

OUTPUT	OUTPUT	OUTPUT	OUTPUT
Product: 12	Product: 12	Product: 12	Product: 12

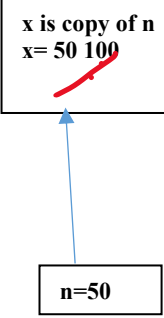
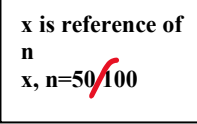
5.8 PASSING PARAMETERS TO FUNCTIONS

Parameters can be passed by value (copies of arguments) or by reference (references to arguments).

Pass by Value: When you pass parameters by value to a function, a copy of the parameters is made (called as formal parameters) and used within the body of the function. If any changes made to the formal parameters inside the function, it does not affect the original parameters (called as actual parameters).

Pass by Reference: When you pass a parameter by reference to a function, you give the function direct access to the original (actual) parameter. If the function changes the formal parameter, it changes the original one (actual parameter), not just a copy.

Tab. No. 2.3 Example of Pass by values and pass by reference technique.

Pass by Value	Pass by Reference
<pre>#include <iostream.h> void changeValue (int x) { x = 100; } void main () { int n = 50; changeValue(n); cout << "Value: " << n << endl; }</pre> 	<pre>#include <iostream.h> void changeValue(int &x) { x = 100; } void main () { int n = 50; changeValue(n); cout << "Value: " << n << endl; }</pre> 
Output Value: 50 Note: No change in the value of n	Output Value: 100 Note: change in the value of n

5.9 RECURSION

Recursion is a technique in which function calls itself inside its body. A function that calls itself is called a recursive function. It's commonly used in tasks that can be broken into similar sub-tasks.

NOTE**Example: Factorial of 5**

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Can also be calculated as:

$$5! = 5 \times 4!$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

$$4! = 4 \times 3!$$

$$4! = 4 \times 3 \times 2 \times 1$$

$$3! = 3 \times 2!$$

$$3! = 3 \times 2 \times 1$$

$$2! = 2 \times 1!$$

$$2! = 2 \times 1$$

$$1! = 1$$



So, we can write

$$n! = n \times (n-1)! \text{ and } 1! = 1 \text{ and } 0! = 1$$

Program: To calculate factorial of any number using recursion.

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

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

```
long int factorial (int n); //function prototype
```

```
void main ()
```

```
{
```

```
clrscr (); //clears the output screen and it is defined on conio.h header file
```

```
int n;
```

```
cout<<" enter any number";
```

```
cin>>n;
```

```
long int f= factorial(n); //function call
```

```
cout<<" factorial of" <<n<<" =" <<f;
```

```
getch (); //it holds the output screen until you press any character from the keyboard
```

```
    //it is also defined in conio.h header file
```

```
}
```

```
long int factorial (int n)// function definition
```

```
{
```

```
if (n==0 || n==1) //terminating condition
```

```
return 1;
```

```
else if(n>1)
```

NOTE

```
return n*factorial(n-1);  
}
```

OUTPUT

```
enter any number 5  
factorial of 5=120
```

5.10 COMMAND LINE ARGUMENTS

Command-line arguments are passed to main () as argc (argument count) and argv (argument vector).

Example: Command-Line Arguments

```
#include <iostream.h>  
  
void main (int argc, char *argv []) {  
    cout << "Program Name: " << argv [0] << endl;  
    for (int i = 1; i < argc; i++) {  
        cout << "Argument " << i << ": " << argv[i] << endl;  
    }  
}
```

5.11 STORAGE CLASS SPECIFIERS

C++ supports various storage class specifiers that define the scope, visibility, and lifetime of variables.

1. auto: Default for local variables.
2. static: Keeps the variable in memory for the lifetime of the program.
3. extern: Declares a global variable defined elsewhere.
4. register: Suggests storing the variable in a CPU register.

Example: auto variable

```
//Auto: By default, all variables are auto and destroyed  
//automatically when their scope ends.  
  
#include<iostream.h>  
  
#include<conio.h>  
  
int a=0; //Global Variable  
  
void main ()  
{  
    clrscr ();  
    auto int a=1; //Local Variable  
}
```

NOTE

```

auto int a=2; //Local Variable
cout<<"a="<<a; // ans: a=2 (Value of local variable of anonymous block)
cout<<"\n"<<::a; //ans: a=0 (Value of global variable)
}
cout<<"\n"<<a; //ans: a=1 (Local variable of main)
getch ();
}

```

Example: Static Variable

```

#include <iostream.h>
void increment () {
    static int counter = 0; // Static variable
    counter++;
    cout << "Counter: " << counter << endl;
}
void main ()
{
    increment ();
    increment ();
    increment ();
}

```

OUTPUT

Counter: 1

Counter: 2

Counter: 3

Example: extern Variable - accessing variable defined outside scope of main function

```

#include<iostream.h>
#include<conio.h>
void main ()
{
    clrscr ();
    extern int a; //informing compiler it's an external variable,
                // do not create new copy
    cout<<"\na="<<a;
}

```

NOTE

```
    getch ();
}

int a=1; //variable outside scope of main
```

OUTPUT

a=1

Example: register Variable

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

void main ()
{
    clrscr (); //To clear screen
    register int a; //Memory from CPU-Register
    register int b;

    asm mov a,5 //a=5, asm means assembly language
    asm mov b,6 //b=6

    cout<<"\na="<<a<<"b="<<b;

    asm add a,4 //a=a+4
    asm sub a,2 //a=a-2

    cout<<"\na="<<a<<"b="<<b;

    getch ();
}
```

OUTPUT

a=5 b=6

a=7 b=6 (The variable a will have the value 7 (5 + 4 - 2), and b will remain 6)



Activity 5.1

❖ Function Decomposition Task.

Objective: To modularize a program using functions.

Required Materials: Computer, compiler

Prerequisites: Arrays and control structures

Steps:

1. Write a program without functions.
2. Break it into meaningful functions.

NOTE

3. Use recursion where applicable.

Post-Activity Reflection: How does modularization improve maintainability?

Expected Outcome: Learners design modular and reusable programs.

☑ Check Your Progress 5.1**A. Multiple Choice Question (MCQ)**

1. Which type of function call occurs when a function calls itself?
 - a) Iteration
 - b) Recursion
 - c) Looping
 - d) Overloading

B. True/False

1. Command line arguments are used to pass data to a program when it is executed.

C. Short Answer Questions

1. What is a user-defined function?

.....

2. Name any two storage class specifiers in C++.

.....

5.12 SUMMARY

This unit introduced user-defined functions as essential components for modular programming in C++. Functions help reduce redundancy and improve readability by dividing programs into logical blocks. The unit explained function declaration, definition, return values, and categories of functions. Parameter passing methods, including call by value and call by reference, were discussed in detail. Recursion was introduced to solve repetitive problems efficiently. Command-line arguments and storage class specifiers explained variable scope and lifetime. Mastery of functions strengthens structured programming skills and prepares learners for object-oriented programming.

5.13 GLOSSARY

- **Function** – Block of code performing a task

NOTE

- **Prototype** – Function declaration
- **Actual Arguments** – Values passed to a function
- **Formal Arguments** – Parameters in function definition
- **Recursion** – Function calling itself
- **Static Variable** – Variable with persistent value
- **Command-line Arguments** – Inputs passed during execution
- **Call by Reference** – Passing variable address

5.14 ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 5.1

A. MCQ Answer:

1. b) Recursion

Explanation: Recursion is the process in which a function calls itself to solve a problem.

B. True/False Answer:

1. True

Explanation: Command line arguments allow input to be provided to a program at the time of execution.

C. Short Answer Answers:

1. **User-defined function:** A user-defined function is a function created by the programmer to perform a specific task and improve program modularity and reusability.

2. Two storage class specifiers:

- Auto
- Static

5.15 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/Us/UnitUrls?unitId=1470	

5.16 REFERENCE

- Object-Oriented Programming with C++ – E. Balagurusamy – ISBN: 9781259029936 – 2017
- Programming with C++ – Ashok N. Kamthane – ISBN: 9788131762269 – 2015
- Let Us C++ – Yashavant P. Kanetkar – ISBN: 9789389845063 – 2020
- C++: The Complete Reference – Herbert Schildt – ISBN: 9780071506868 – 2003

5.17 EXERCISE**Part – 1****A. Multiple Choice Questions****1. What is a user-defined function in C++?**

- A. A function provided by the compiler
- B. A function defined by the operating system
- C. A function written by the user to perform a specific task
- D. A function that cannot return a value

2. Which part of a function tells the compiler about function name and parameters before use?

- A. Function call
- B. Function definition
- C. Function prototype
- D. Return statement

3. Which keyword is used when a function does not return any value?

- A. int
- B. float
- C. return
- D. void

4. In pass-by-value, what happens to actual arguments?

- A. They are modified permanently
- B. They are destroyed
- C. A copy is passed to the function
- D. They are inaccessible

5. Which technique allows a function to modify actual parameters?

- A. Call by value

NOTE

NOTE

- B. Call by name
- C. Call by reference
- D. Recursive call

Answer with Explanations

1. **C** A function written by the user to perform a specific task

Explanation: User-defined functions are created by programmers to perform specific tasks and improve modularity.

2. **C** Function prototype

Explanation: A function prototype (declaration) informs the compiler about function details before calling it.

3. **D** void

Explanation: void specifies that a function does not return any value.

4. **C** A copy is passed to the function

Explanation: In pass-by-value, changes affect only the copied data, not the original variables.

5. **C** Call by reference

Explanation: Call by reference passes the address of variables, allowing direct modification.

B. Short Answer Type Questions

1. What is the purpose of a function in C++?
2. Define function prototype.
3. What is call by reference?
4. Explain function declaration, definition, and call.
5. What are function parameters?

Part – 2**C. Long Answer Type Questions (200 words)**

1. Explain user-defined functions and their advantages.
2. Describe pass-by-value and pass-by-reference with examples.
3. Differentiate between call by value and call by reference.
4. What is recursion?
5. Write a recursive function to find factorial.

Part – 3**D. Long Answer Type Questions (300 words)**

1. Explain the elements of a user-defined function in C++.
2. Explain recursion with an example.
3. Explain storage class specifiers in C++.
4. Explain the concept of modular programming.
5. Discuss advantages of using functions.

UNIT 6

INTRODUCTION TO CLASSES AND OBJECTS

NOTE

UNIT STRUCTURE

- 6.1 Introduction
- 6.2 Objectives
- 6.3 Classes and objects
- 6.4 Structures and classes
- 6.5 Unions and classes
- 6.6 Friend function
- 6.7 Friend classes
- 6.8 Inline function
- 6.9 Scope resolution operator
- 6.10 Static class members- static data members, static member functions
- 6.11 Passing object to functions
- 6.12 Returning objects and object assignment
- 6.13 Array of objects
- 6.14 Pointer to object
- 6.15 Type checking in C++
- 6.16 This pointer
- 6.17 Pointer to derived types
- 6.18 Pointer to class members
- 6.19 References in Classes and Objects
- 6.20 C++'s dynamic allocation operators
- 6.21 Constructors, special characteristics of constructor functions
- 6.23 Default constructor
- 6.24 Parameterized constructors
- 6.25 Copy constructors
- 6.26 Multiple constructors in a class
- 6.27 Constructors with default arguments
- 6.28 Destructors
- 6.29 Summary
- 6.30 Glossary
- 6.31 Answer to Check Your Progress
- 6.32 Suggested readings – OER
- 6.33 Reference
- 6.34 Exercise

6.1 INTRODUCTION

Unit 6 builds directly on the learner's prior understanding of arrays, user-defined functions, and modular programming developed in earlier units. Until this point, programs were written using procedural techniques where data and

NOTE

functions were treated separately. While this approach is effective for small problems, it becomes difficult to manage as program size and complexity increase. This unit marks a significant transition from procedural programming to object-oriented programming, using C++ as the implementation language.

The primary focus of this unit is the introduction of classes and objects, which form the foundation of object-oriented design. Learners study how classes act as blueprints that combine data members and member functions into a single logical unit, and how objects represent real-world entities created from these classes. The unit explains important distinctions between structures, unions, and classes, highlighting access control mechanisms that support data hiding and encapsulation. Advanced features such as friend functions and friend classes are introduced to demonstrate controlled access to private data. The role of inline functions, static data members, and static member functions is also discussed to improve efficiency and shared behavior.

This unit further covers object management concepts essential for real-world applications. Learners explore constructors and destructors, including default, parameterized, copy constructors, and their special characteristics. Dynamic memory allocation using C++ operators, pointers to objects, references, arrays of objects, passing objects to functions, returning objects, and object assignment are explained in detail. These topics enable learners to create flexible, memory-efficient, and reusable class designs.

By the end of this unit, learners gain the ability to design and implement complete object-oriented programs using C++ classes and objects. This strong foundation prepares learners to understand polymorphism, where objects can exhibit multiple behaviors using common interfaces, which is introduced in the next unit. This progression ensures readiness for advanced behavioral abstraction concepts ahead.

6.2 OBJECTIVES

After completing this unit, learners will be able to:

- Design classes that encapsulate data and member functions.
- Create and manage objects using constructors and destructors.
- Implement static members, friend functions, and inline functions.
- Apply dynamic memory allocation for object management.
- Develop programs using arrays of objects and object pointers.

6.3 CLASSES AND OBJECTS

1. A class is a user-defined data type that groups data and functions.
2. Class serves as a blueprint for creating objects that defines the properties (data members) and behaviors (member functions) of objects.
3. In C++, class is a group of similar objects.
4. It is a template from which objects are created.

NOTE

5. It encapsulates data and functions into a single unit.
6. It can have fields, methods, constructors etc.
7. A class is a user-defined data type that we can use in our program.
8. Object is an instance of a class, holding specific values of the data members and accessing the member functions of the class.
9. Class is defined using the class keyword, followed by the class name and curly braces.

Syntax

```
class class_name
{
Access specifier:
Variable declarations;
Access specifier:
function declarations
};
```

Data Members

1. Variables defined inside a class are called as data members of a class.
2. They represent the properties or attributes of the class.

Example:

```
class Car
{
private:
int model_no; //data members
char colour[20];
};
```

Member Functions

1. Functions defined inside a class are called as member functions of a class.
2. They operate on data members and define the behavior of the class.

```
class Car
{
public:
void display_info()
{
cout<<model_no<<" "<<colour<<endl;
}
```

NOTE

```
};
```

Access Specifiers

1. **public:** Members are accessible from outside the class.
2. **private:** Members are only accessible within the class.
3. **protected:** Members are accessible within the class and derived classes.

```
class Car
```

```
{
```

private:

```
int model_no;    //data members
```

```
char colour[20];
```

public:

```
void display_info() //member function
```

```
{
```

```
cout<<model_no<<" "<<colour<<endl;
```

```
}
```

```
};
```

Example 1: Defining a class and creating objects.

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

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

```
class Car {
```

```
public:
```

```
    string brand;
```

```
    string model;
```

```
    int year;
```

```
    void printDetails() {
```

```
        cout << "Brand: " << brand << ", Model: " << model << ", Year: " << year << endl;
```

```
    }
```

```
};
```

```
void main() {
```

```
    Car car1; //object
```

```
    car1.brand = "Toyota";
```

```
    car1.model = "Corolla";
```

```
    car1.year = 2020;
```

```
    car1.printDetails();
```

```
    getch();
}
```

OUTPUT

Brand: Toyota, Model: Corolla, Year: 2020

Example 2: Using multiple objects.

```
#include <iostream.h>
#include <conio.h>
class Student {
public:
    string name;
    int age;
    void displayInfo() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

void main() {
    Student s1, s2;
    s1.name = "Ankit";
    s1.age = 20;
    s2.name = "Seema";
    s2.age = 22;
    s1.displayInfo();
    s2.displayInfo();
    getch();
}
```

OUTPUT

Name: Ankit, Age: 20

Name: Seema, Age: 22

Example 3: Object initialization inside a class constructor.

```
#include <iostream.h>
#include <conio.h>
class Rectangle {
public:
    int length, width;
```

NOTE

```
Rectangle(int l, int w) {
    length = l;
    width = w;
}

int area() {
    return length * width;
}

};

void main() {
    Rectangle rect(5, 10);
    cout << "Area: " << rect.area() << endl;
    getch();
}
```

OUTPUT

Area: 50

6.4 STRUCTURES AND CLASSES

1. A struct (short for "structure") is a user-defined data type that groups together different types of variables under one name.
2. These variables are known as members or fields and it can be of different types, such as integers, floats, or even other structs.
3. A struct is like a blueprint for creating complex data types.

Example: if you're making a program to manage student records, you might want to store each student's name, age, and Percentage. Using a struct, you can group these related pieces of information together.

```
#include <iostream.h>

// Define a struct named 'Student'
struct Student {
    char name[50];
    int age;
    float percentage;
};

int main() {
    // Create a variable of type 'Student'
    struct Student student1;
    // Assign values to the fields of student1
```

NOTE

```

student1.name="Ankit";
student1.age = 20;
student1.percentage = 70;
// Print the student information
cout<<"Name: "<< student1.name<<" , ";
cout<<"Age: "<< student1.age<<" ,";
cout<<"Percentage: "<< student1.percentage;
return 0;
}

```

OUTPUT

Name: Ankit, Age: 20, Percentage: 70

In this example, struct Student groups name, age, and percentage into one unit. This helps organize and manage related data more effectively.

Example 1: Using struct.

```

#include <iostream.h>
#include <conio.h>
struct Point {
    int x, y;
};
void main() {
    Point p1 = {10, 20}; //Creates an instance of Point with x initialized to 10
    and y
                           //to 20.
    cout << "Point x: " << p1.x << ", y: " << p1.y << endl;
    getch();
}

```

OUTPUT

Point x: 10, y: 20

Example 2: Using class.

```

#include <iostream.h>
#include <conio.h>
class Point {
    int x, y;
public:
    Point (int a, int b) {
        x=a;

```

ARRAYS, FUNCTIONS, AND OBJECTS

NOTE

```
        y=b;
    }

    void printPoint () {
        cout << "Point x: " << x << ", y: " << y << endl;
    }
};

void main () {
    Point p1(10, 20);
    p1. printPoint ();
    getch();
}
```

OUTPUT

Point x: 10, y: 20

Example 3: Struct and Class.

```
#include <iostream.h>
#include <conio.h>
struct StructPoint {
    int x, y;
};

class ClassPoint {
    int x, y;
public:
    ClassPoint(int a, int b) : x(a), y(b) {}
    void printPoint() {
        cout << "Class Point x: " << x << ", y: " << y << endl;
    }
};

void main() {
    StructPoint sp = {10, 20};
    cout << "Struct Point x: " << sp.x << ", y: " << sp.y << endl;
    ClassPoint cp(30, 40);
    cp.printPoint();
    getch();
}
```

OUTPUT

Struct Point x: 10, y: 20

Class Point x: 30, y: 40

NOTE

Aspect	Class	Stuct
Default Access	Private	Public
Use	Used for both data and functions	Typically used for holding data
Inheritance	Supports inheritance, class uses private inheritance by default.	Supports inheritance, struct uses public inheritance by default.
Syntax	Defined using keyword `class`	Defined using keyword `struct`
Encapsulation	Allows full encapsulation	Not recommended for encapsulation
Memory	Same memory layout as struct	Same memory layout as class
Usage	Used for more complex data structures with encapsulation and behavior.	Used for plain data structures with simple access.

Tab. No. 2.4 Difference between Structure and Class

6.5 UNIONS AND CLASSES

1. A union allows storing different data types in the same memory location, but only one member can hold one value at a time.
2. A class allows storing different data types independently, with each member storing its own value.

Example 1: Using union.

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

union Data {
    int i;
    float f;
};

void main() {
    Data d;
    d.i = 10;
    cout << "Integer: " << d.i << endl;
```

ARRAYS, FUNCTIONS, AND OBJECTS

NOTE

```
d.f = 3.14;
cout << "Float: " << d.f << endl;
getch();
}
```

OUTPUT

Integer: 10

Float: 3.14

Example 2: Using class.

```
#include <iostream.h>
#include <conio.h>
class Data {
public:
    int i;
    float f;
    void display() {
        cout << "Integer: " << i << "\n Float: " << f << endl;
    }
};
void main() {
    Data d;
    d.i = 10;
    d.f = 3.14;
    d.display();
    getch();
}
```

OUTPUT

Integer: 10

Float: 3.14

Example 3: Union vs Class in terms of memory.

```
#include <iostream.h>
#include <conio.h>
union DataUnion {
    int i;
    double d;
```

NOTE

```
};

class DataClass {
public:
    int i;
    double d;
};

void main() {
    cout << "Size of union: " << sizeof(DataUnion) << endl;
    cout << "Size of class: " << sizeof(DataClass) << endl;
    getch();
}
```

OUTPUT

Size of union: 8

Size of class: 16

Explanation: A union in C++ uses the memory of its largest member. In DataUnion we have an int and double variables. On most systems, an int is 4 bytes and a double is 8 bytes. Therefore, the size of the union will be the size of the largest member, which is double (8 bytes). whereas the **Size of DataClass is 16** bytes because the class includes an int (4 bytes) and a double (8 bytes), with additional padding to align the double properly (resulting in a total of 16 bytes).

Note: The exact sizes can vary based on the compiler and system architecture.

6.6 FRIEND FUNCTION

1. A friend function is not a member of a class it's a normal function but has access to its private and protected members.
2. Declared within the class using the friend keyword.
3. Defined outside the class.
4. Not inherited by derived classes.
5. Can operate on class data from outside the class.
6. It can directly access private data using dot notation (.) on the object passed to it (e.g., C. radius).

Example : Using a friend function.

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

```
Class Circle {
```

NOTE

```
private:
    int radius;

public:
void setRadius() //member function of a class
{
    radius=5;
}

    friend void area (Circle &C) ;//friend function declaration inside the
class
};

void Area (Circle &C) // friend function definition outside the class as it is
not the

                        //member function of a class
{
    float a= 3.14*C. radius*C. radius;
    cout<<" Area of Circle: " <<a;
}

void main()
{
    Circle C;
    C.setRadius (); //member function call
    Area(C); //friend function call
    getch();
}
```

OUTPUT

Area of Circle: 78.5

6.7 FRIEND CLASSES

1. A friend class can access the private and protected members of another class.
2. Declaring a class as a friend gives it access to all the members (private, protected, and public) of the other class.
3. The friend class is declared within the class using the friend keyword.

NOTE

4. Friend classes are used when two classes are tightly coupled and need to interact closely with each other's internal details.
5. A class can be friend of multiple classes.

Example: Demonstrating a friend class.

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

class B; // Forward declaration

class A {
private:
    int valueA;
public:
    A(int a) : valueA(a) {} //default constructor initializing the value of a to 10
    friend class B; // B is a friend of A
};

class B {
public:
    void showA(A & A1) {
        cout << "Value of A: " << A1. valueA << endl;
    }
};

void main() {
    A A1(10);
    B B1;
    B1.showA(A1);
    getch();
}
```

OUTPUT

Value of A: 10

6.8 INLINE FUNCTION

An inline function is a function that is expanded in line when it is invoked. This means the compiler replaces the function call with the function code itself, reducing the overhead of a function call. However, it is a request to the compiler, and the compiler may ignore it if the function is too complex.

NOTE**Example 1: Using an inline function.**

```
#include <iostream.h>
#include <conio.h>
inline int square (int x) {
    return x * x;
}
void main () {
    cout << "Square of 5: " << square (5) << endl;
    getch ();
}
```

OUTPUT

Square of 5: 25

6.9 SCOPE RESOLUTION OPERATOR

Scope: -Visibility or availability of a variable in a program is called as scope. There are two types of scope.

- i) Local scope
- ii) Global scope

Local scope: visibility of a variable is local to the function in which it is declared.

Global scope: visibility of a variable to all functions of a program.

Scope resolution operator in “::”

1. The scope resolution operator (::) is used to access global variable when a local variable with the same name exists .
2. To define member functions of a class outside the class.

Example 1: To access global variable

```
#include<iostream.h>
int a=5; // global variable
void main()
{
    int a=1; //local variable
    cout<<"Local a="<<a<<endl;
    cout<<"Global a="<<::a<<endl;
}
```

OUTPUT

Local a= 1

NOTE

Global a=5

Example2: To define member functions of a class outside the class.

Program to find area of rectangle

```
#include<iostream.h>

class rectangle
{
    int L,B;
    public:
    void get_data();
    void area();
};

void rectangle::get_data()
{
    cout<<"Enter Length of rectangle";
    cin>>L;
    cout<<"Enter breadth of rectangle";
    cin>>B;
}

int rectangle::area()
{
    return L*B;
}

int main()
{
    rectangle r;
    r.get_data();
    cout<<"Area of rectangle is"<<r.area();
    return 0;
}
```

OUTPUT

Enter Length of rectangle 2

Enter breadth of rectangle 3

Area of rectangle is 6

NOTE**6.10 STATIC CLASS MEMBERS- STATIC DATA MEMBERS, STATIC MEMBER FUNCTIONS**

1. A **static data member** is shared among all objects of the class. It is initialized once and retains its value for all objects.
2. A **static member function** can access only static data members and does not require an object to be called.

Example 1: Static data members.

```
#include <iostream.h>
#include <conio.h>
class Counter {
public:
    static int count; // Static data member
    Counter() { //default constructor
        count++;
    }
};
int Counter::count = 0; // Initialization of static data member
void main() {
    Counter c1, c2, c3;
    cout << "Number of objects: " << Counter::count << endl;
    getch();
}
```

OUTPUT

Number of objects: 3

Example 2: Static member function.

```
#include <iostream.h>
#include <conio.h>
class Test {
public:
    static int value; //static data member
    static void display() { // Static member function
        cout << "Value: " << value << endl;
    }
};
int Test::value = 100; // Initialization of static data member
```

NOTE

```
void main() {
    Test::display(); // Call static function without object
    getch();
}
```

OUTPUT

Value: 100

6.11 PASSING OBJECT TO FUNCTIONS

An object can be passed to a function by value or by reference.

Example 1: Passing object by value.

```
#include <iostream.h>
#include <conio.h>
class Rectangle {
public:
    int length, width;
    Rectangle(int l, int w) : length(l), width(w) {}
};
void printArea(Rectangle r) {
    cout << "Area: " << r.length * r.width << endl;
}
void main() {
    Rectangle rect(5, 10);
    printArea(rect); // Object passed by value
    getch();
}
```

OUTPUT

Area: 50

Example 2: Passing object by reference.

```
#include <iostream.h>
#include <conio.h>
class Rectangle {
public:
    int length, width;
    Rectangle(int l, int w) : length(l), width(w) {}
};
```

ARRAYS, FUNCTIONS, AND OBJECTS

NOTE

```
void printArea(Rectangle &r) { // Object passed by reference
    cout << "Area: " << r.length * r.width << endl;
}

void main() {
    Rectangle rect(5, 10);
    printArea(rect);
    getch();
}
```

OUTPUT

Area: 50

6.12 RETURNING OBJECTS AND OBJECT ASSIGNMENT

A function can return an object, which is then assigned to another object or used directly.

Example 1: Returning an object.

```
#include <iostream.h>
#include <conio.h>
class Rectangle {
public:
    int length, width;
    Rectangle(int l, int w) : length(l), width(w) {}
    Rectangle duplicate() {
        return Rectangle(length, width); //returning object
    }
};

void main() {
    Rectangle rect1(5, 10);
    Rectangle rect2 = rect1.duplicate(); // Object assignment
    cout << "Area of rect2: " << rect2.length * rect2.width << endl;
    getch();
}
```

OUTPUT

Area of rect2: 50

6.13 ARRAY OF OBJECTS

1. An array of objects is an array that stores multiple instances of a class.

NOTE

2. It allows managing and accessing multiple objects of the same type using array indexing.
3. Declaration is similar to arrays of basic data types; an array of objects is declared by specifying the class type.
4. Each object in the array can be individually initialized using constructors.

Example: Input Array of Objects from User

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

class Employee {
private:
    string name;
    int salary;
public:
    Employee() {}
    void setData(string n, int s) {
        name = n;
        salary = s;
    }
    void display() {
        cout << "Name: " << name << ", Salary: " << salary << endl;
    }
};

void main() {
    Employee employees[3]; //Array of 3 objects
    string name;
    int salary;
    for (int i = 0; i < 3; ++i) {
        cout << "Enter name and salary of employee " << i+1 << ": ";
        cin >> name >> salary;
        employees[i].setData(name, salary);
    }
    for (int i = 0; i < 3; ++i) {
        employees[i].display();
    }
    getch();
}
```

NOTE

```
}
```

OUTPUT

Enter name and salary of employee 1 Ankit 30000

Enter name and salary of employee 2 Seema 20000

Enter name and salary of employee 3 Anshu 45000

Name: Ankit, Salary: 30000

Name: Seema, Salary: 20000

Name: Anshu, Salary: 45000

6.14 POINTER TO OBJECT

1. A pointer to an object is a pointer that stores the address of an object of a class.
2. It allows access to the members of the object using the pointer.
3. The arrow operator (`^->`) is used to access members of an object through its pointer.
4. Pointers to objects are often used for dynamic memory allocation and efficient memory management.

Example1: Pointer to Object

```
#include <iostream.h>
#include <conio.h>
class Rectangle {
private:
    int length, width;
public:
    Rectangle(int l, int w) : length(l), width(w) {}
    int area() {
        return length * width;
    }
};

void main() {
    Rectangle rect(10, 5);
    Rectangle* ptr = &rect; // Pointer to object
    cout << "Area of Rectangle using pointer: " << ptr->area() << endl;
    getch();
}
```

OUTPUT

Area of Rectangle using pointer: 50

Example 2: Dynamic Object Creation with Pointer

```
#include <iostream.h>
#include <conio.h>
class Circle {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() {
        return 3.14159 * radius * radius;
    }
};
void main() {
    Circle* circlePtr = new Circle(10.0); // Dynamically creating an object
    cout << "Area of Circle: " << circlePtr->area() << endl;
    delete circlePtr; // Deallocate memory
    getch();
}
```

OUTPUT

Area of Circle: 314.159

NOTE

6.15 TYPE CHECKING IN C++

Type checking refers to verifying and ensuring that variables and expressions in a program adhere to the defined types during both compilation and runtime.

Two types of type checking

- 1. Static Type Checking:** In C++, most type checking occurs at compile time, ensuring that variable types are compatible.
- 2. Dynamic Type Checking:** C++ also supports some runtime type checks, especially with inheritance, polymorphism, and 'typeid'

Example 1: Compile-time Type Checking

```
#include <iostream.h>
#include <conio.h>
void main () {
    int a = 5;
```

NOTE

```
double b = 10.5;

// Automatic type checking at compile time
double sum = a + b; // Implicit type conversion of variable a to double
cout << "Sum: " << sum << endl;
getch ();
}
```

OUTPUT

Sum: 15.5

Example 2: `typeid` for Runtime Type Checking

```
#include <iostream.h>
#include <typeinfo>
#include <conio.h>
class Base {};
class Derived : public Base {};
void main() {
    Base* b = new Derived();
    if (typeid(*b) == typeid(Derived)) {
        cout << "The object is of type Derived." << endl;
    }
    else {
        cout << "The object is of another type." << endl;
    }
    delete b;
    getch();
}
```

OUTPUT

The object is of type Derived.

Explanation:

Since b is a pointer to a Derived object, and typeid(*b) is compared with typeid(Derived), the condition typeid(*b) == typeid(Derived) will be true, leading to the output "The object is of type Derived."

6.16 THIS POINTER

1. The `this` pointer is an implicit pointer passed to all non-static member functions of a class.

NOTE

2. It points to the object for which the member function is called.
3. The `this` pointer is used to access the calling object's members.

Example: Basic Usage of `this` Pointer

```
#include <iostream.h>
#include <conio.h>
class Box {
private:
    int length;
public:
    Box(int l) : length(l) {}
    void setLength(int l) {
        this->length = l; // Using this pointer to access length data member
    }
    void display() {
        cout << "Length: " << length << endl; // without using this pointer to access
                                                //length
    }
};
void main() {
    Box box1(10);
    box1.display();
    box1.setLength(20);
    box1.display();
    getch();
}
```

OUTPUT

Length: 10

Length: 20

6.17 POINTER TO DERIVED TYPES

1. A pointer to a derived type is a pointer that can point to objects of a derived class in an inheritance hierarchy.
2. A base class pointer can point to derived class objects, enabling polymorphic behavior.
3. Used in Polymorphism where Base class pointers can be used to invoke virtual functions of derived classes.

NOTE

4. When accessing derived class members, type casting may be required.

Example: Using Virtual Functions with Pointer to Derived Class

```
#include <iostream.h>
#include <conio.h>
class Animal { //Base class
public:
    virtual void sound() {
        cout << "Animal sound" << endl;
    }
};
class Dog : public Animal { //Dog class derived from Animal class
public:
    void sound(){ // function overriding
        cout << "Dog barks" << endl;
    }
};
class Cat : public Animal { //Cat class derived from Animal class
public:
    void sound(){ // function overriding
        cout << "Cat meows" << endl;
    }
};
void main() {
    Animal* animalPtr; //Pointer of base class
    Dog dog; //object of Dog class
    Cat cat; //object of Cat class
    animalPtr = &dog; //Base class pointer pointing to Dog class object
    animalPtr->sound(); // Calls Dog's sound
    animalPtr = &cat; //Base class pointer pointing to Cat class object
    animalPtr->sound(); // Calls Cat's sound
    getch();
}
```

OUTPUT

Dog barks
Cat meows

6.18 POINTER TO CLASS MEMBERS

NOTE

A pointer to a class member is a pointer that stores the address of a member function or member variable of a class.

Explanation:

1. Member Variable Pointers: Used to point to data members.
2. Member Function Pointers: Used to point to member functions.
3. To access a member using a pointer, you must specify both the object and the member using the pointer.

Example 1: Pointer to Member Variable

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

class Box {
public:
    int length; //member variables
    int width;
    Box(int l, int w) : length(l), width(w) {}
};

void main() {
    Box b(10, 5);
    int Box::*ptr = &Box::length; // Pointer to member variable
    cout << "Length: " << b.*ptr << endl;
    getch();
}
```

OUTPUT

Length: 10

Example 2: Pointer to Member Function

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

class Rectangle {
public:
    int length, width;
    Rectangle(int l, int w) : length(l), width(w) {}
    int area() {
        return length * width;
    }
}
```

NOTE

```
    }  
};  
  
void main() {  
    Rectangle rect(10, 5);  
    int (Rectangle::*funcPtr)( ) = &Rectangle::area; // Pointer to member  
function  
    cout << "Area: " << (rect.*funcPtr)() << endl;  
    getch();  
}
```

OUTPUT

Area: 50

6.19 REFERENCES IN CLASSES AND OBJECTS

1. A reference is an alias (another name) for an existing variable.
2. In C++, references are commonly used to pass variables to functions without copying their values.

Example:

```
#include <iostream.h>  
#include <conio.h>  
  
void increment (int& ref) { //ref is the reference variable of x  
    ref++;  
}  
  
void main() {  
    int x = 10;  
    increment(x); // Pass by reference  
    cout << "Value of x: " << x << endl; // x is modified to 11  
    getch();  
}
```

OUTPUT

Value of x: 11

6.20 C++'S DYNAMIC ALLOCATION OPERATORS

1. Dynamic memory allocation in C++ allows memory to be allocated during runtime using the `new` keyword, and deallocated using `delete`.
2. This is particularly useful when the size of an object or array is not known at compile time.

NOTE

3. Dynamic allocation and deallocation operators:

- `new`: Allocates memory on the heap.
- `delete`: Frees memory that was allocated using `new`.
- For arrays, `new [ ]` and `delete [ ]` should be used.

Example 1: Dynamic Memory Allocation of a Single Object

```
#include <iostream.h>
#include <conio.h>
class Circle {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() {
        return 3.14159 * radius * radius;
    }
};

void main() {
    Circle* circlePtr = new Circle(5.0); // Dynamically allocate memory for an
    object
    cout << "Area: " << circlePtr->area() << endl;
    delete circlePtr; // Free memory
    getch();
}
```

OUTPUT

Area: 78.53975

Example 2: Dynamic Array of Objects

```
#include <iostream.h>
#include <conio.h>
class Student {
private:
    string name;
public:
    Student(string n = "") : name(n) {}
    void display() {
        cout << "Name: " << name << endl;
    }
}
```

NOTE

```
    }  
};  
  
void main() {  
    Student* students = new Student[3]; // Dynamic allocation for array of  
objects  
    students[0] = Student("Ankit");  
    students[1] = Student("Seema");  
    students[2] = Student("Anshu");  
    for (int i = 0; i < 3; ++i) {  
        students[i].display();  
    }  
    delete[] students; // Deallocate memory  
    getch();  
}
```

OUTPUT

Name: Ankit

Name: Seema

Name: Anshu

6.21 CONSTRUCTORS, SPECIAL CHARACTERISTICS OF CONSTRUCTOR FUNCTIONS

1. A constructor is a special member function of a class that is automatically called when an object is created.
2. Constructors are used to initialize the objects.
3. It is special because its name is the same name as the class name.
4. It is called as constructor because it constructs the values i.e. provides data for the object of the class.

Special characteristics of Constructor.

- Constructor is a member function of a class, whose name is the same as the class name.
- Constructor gets called automatically when we create the object of the class.
- Constructors do not return value; hence they do not have a return type not even void.
- A Constructor can be overloaded.
- A constructor cannot be declared virtual.

Syntax of Constructors in C++

The prototype of the constructor looks like this:

NOTE

<class-name> (list-of-parameters);

The constructor can be defined inside the class declaration or outside the class declaration

Syntax for Defining the Constructor Within the Class

```
<class-name>(list-of-parameters)
{
    // constructor definition
}
```

Syntax for Defining the Constructor Outside the Class

```
<class-name>::<class-name>(list-of-parameters)
{
    // constructor definition
}
```

Types of Constructors:

1. **Default Constructor:** A constructor that takes no parameters.
2. **Parameterized Constructor:** A constructor that takes arguments to initialize an object with specific values.
3. **Copy Constructor:** A constructor that creates a new object as a copy of an existing object.

6.23 DEFAULT CONSTRUCTOR

Example: Default Constructor

```
#include <iostream.h>
#include <conio.h>
class Box {
private:
    int length;
public:
    Box() : length(0) {} // Default constructor
    void display() {
        cout << "Length: " << length << endl;
    }
};

void main() {
    Box b; // Default constructor is called
    b.display();
    getch();
}
```

NOTE

}

OUTPUT

Length: 0

6.24 PARAMETERIZED CONSTRUCTORS

Example: Parameterized Constructor

```
#include <iostream.h>
#include <conio.h>
class Box {
private:
    int length;
public:
    Box(int l) : length(l) {} // Parameterized constructor
    void display() {
        cout << "Length: " << length << endl;
    }
};

void main() {
    Box b(20); // Parameterized constructor is called
    b.display();
    getch();
}
```

OUTPUT

Length: 20

6.25 COPY CONSTRUCTORS

Example: Copy Constructor

```
#include <iostream.h>
#include <conio.h>
class Box {
private:
    int length;
public:
    Box(int l) : length(l) {} // Parameterized constructor

    Box(const Box& b) { // Copy constructor
```

NOTE

```

        length = b.length;
    }
    void display() {
        cout << "Length: " << length << endl;
    }
};

void main() {
    Box b1(20); // Parameterized constructor
    Box b2(b1); // Copy constructor
    b1.display();
    b2.display();
    getch();
}

```

OUTPUT

Length: 20

Length: 20

6.26 MULTIPLE CONSTRUCTORS IN A CLASS

In C++, constructors can be overloaded (called as constructor overloading), meaning multiple constructors can exist with different parameter lists, allowing for different ways to initialize an object.

Example : Constructor overloading

```

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

class Box {
private:
    int length;
public:
    Box() : length(0) { } // Default constructor
    Box(int l) : length(l) { } // Parameterized constructor

    Box(Box& b) { // Copy constructor
        length = b.length;
    }

    void display() {

```

ARRAYS, FUNCTIONS, AND OBJECTS

NOTE

```
        cout << "Length: " << length << endl;
    }
};

void main() {
    Box b1; //Default constructor called
    Box b2(20); // Parameterized constructor called
    Box b3(b1); // Copy constructor called

    b1.display();
    b2.display();
    b3.display();
    getch();
}
```

OUTPUT

```
Length: 0
Length: 20
Length: 20
```

6.27 CONSTRUCTORS WITH DEFAULT ARGUMENTS

Default arguments are values that a function or constructor uses if no arguments are provided when it's called.

Example: Constructor with Multiple Default Arguments

```
#include <iostream.h>
#include <conio.h>
class Rectangle {
private:
    int length;
    int width;
public:
    Rectangle(int l = 10, int w = 5) : length(l), width(w) {} // Default arguments

    void display() {
        cout << "Length: " << length << ", Width: " << width << endl;
    }
}
```

NOTE

```
};

void main() {
    Rectangle r1;    // Calls constructor with default arguments
    Rectangle r2(15); // Only length is provided
    Rectangle r3(20, 10); // Both arguments provided
    r1.display();
    r2.display();
    r3.display();
    getch();
}
```

OUTPUT

Length:10, Width: 5
 Length:15, Width: 5
 Length:20, Width: 10

6.28 DESTRUCTORS

1. A destructor is also a special member function as a constructor.
2. Destructor destroys the class objects created by the constructor.
3. It is automatically called when an object goes out of scope or is explicitly deleted.
4. It is used to clean up resources that the object might have acquired during its lifetime.

Special Characteristics of Destructor:

- A destructor has the same name as the class, but with a `~` (tilde) prefix.
- A class can have only one destructor, and it cannot be overloaded.
- Destructors do not take arguments and do not return any value.
- If you don't define a destructor, the compiler generates a default one.
- Destructor should be declared in the public section of the program.
- Destructor is called in the **reverse order** of its constructor invocation.

Example 1: Defining a Simple Destructor

```
#include <iostream.h>

class Test {
public:
    Test() { cout<< "\n Constructor executed"; }

    ~Test() { cout<< "\n Destructor executed"; }
};

void main()
```

NOTE

```
{
    Test t; //constructor called
} //Destructor called
```

OUTPUT

Constructor executed

Destructor executed



Activity 6.1

❖ Model a Real-World Object.

Objective: To design a class-based solution.

Required Materials: Computer, compiler

Prerequisites: Functions and structures

Steps:

1. Identify an object (Student, Employee).
2. Define class members.
3. Implement constructors and destructors.

Post-Activity Reflection: How does encapsulation enhance security?

Expected Outcome: Learners can design and implement object-oriented programs.

☑ Check Your Progress 6.1

A. Multiple Choice Question (MCQ)

1. Which of the following is used to access class members using the class name?
 - a) . (dot operator)
 - b) -> (arrow operator)
 - c) :: (scope resolution operator)
 - d) : (colon operator)

B. True/False

1. A friend function can access the private and protected members of a class.

C. Short Answer Questions

1. What is the purpose of a constructor in C++?

.....

.....

.....

2. Define the this pointer in C++.

.....

.....

.....

6.29 SUMMARY

Unit 6 introduces object-oriented programming concepts using C++. It explains classes and objects as the foundation for building structured and reusable programs. The unit covers access control, constructors, destructors, static members, friend functions, and memory management. Learners study dynamic allocation, pointers to objects, arrays of objects, and object passing techniques. Advanced topics such as type checking, references, this pointer, and pointer to derived types are discussed. This unit establishes a solid base for advanced object-oriented features such as polymorphism and inheritance.

6.30 GLOSSARY

- **Class** – Blueprint for objects
- **Object** – Instance of a class
- **Encapsulation** – Binding data and methods
- **Constructor** – Object initialization function
- **Destructor** – Object cleanup function
- **Friend Function** – External function with private access
- **Static Member** – Shared class-level variable
- **Dynamic Allocation** – Runtime memory allocation

6.31 ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 6.1

A. MCQ Answer:

1. **c). (scope resolution operator)**

Explanation: The scope resolution operator is used to define or access class members outside the class using the class name.

B. True/False Answer:

1. **True**

Explanation: Friend functions are granted special access to private and protected members of a class.

C. Short Answer Answers:

1. **Purpose of constructor:** A constructor is a special member function that is automatically called when an object is created. It is used to initialize the data members of the class.
2. **This pointer:** This pointer is an implicit pointer available inside non-static member functions, which holds the address of the current calling object.

NOTE

NOTE

6.32 SUGGESTED READINGS– OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/Use/UnitUrls?unitId=1471	

6.33 REFERENCE

- Object-Oriented Programming in C++ – *E. Balagurusamy* – ISBN: 9781259029936 – 2017
- Programming with C++ – *Ashok N. Kamthane* – ISBN: 9788131762269 – 2015
- The C++ Programming Language – *Bjarne Stroustrup* – ISBN: 9780321563842 – 2013
- C++: The Complete Reference – *Herbert Schildt* – ISBN: 9780071506868 – 2003

6.34 EXERCISE

Part – 1

A. Multiple Choice Questions

1. What is a class in C++?

- A. A built-in function
- B. A variable
- C. A user-defined data type that groups data and functions
- D. A loop structure

2. Which keyword is used to define a class in C++?

- A. struct
- B. object
- C. class
- D. define

3. What is an object in C++?

- A. A datatype
- B. An instance of a class
- C. A function
- D. A pointer

NOTE

4. Which access specifier allows members to be accessed only within the class?

- A. public
- B. protected
- C. private
- D. global

5. Which type of constructor takes arguments?

- A. Default constructor
- B. Copy constructor
- C. Parameterized constructor
- D. Static constructor

Answer with Explanations

1. C. A user-defined data type that groups data and functions

Explanation: A class combines data members and member functions into a single logical unit.

2. C. class

Explanation: The class keyword is used to create a class in C++.

3. B. An instance of a class

Explanation: An object represents a real-world entity created from a class.

4. C. private

Explanation: Private members are accessible only inside the class.

5. C. Parameterized constructor

Explanation: Parameterized constructors initialize objects with user-specified values.

B. Short Answer Type Questions

1. Define object-oriented programming.
2. What is encapsulation?
3. What is a constructor?
4. What is a class in C++?
5. What is an object?

Part – 2**C. Long Answer Type Questions (200 words)**

1. Explain classes and objects in C++.
2. Explain constructors and their types.
3. Explain friend functions and friend classes.
4. Explain data members and member functions.

NOTE

5. What are access specifiers?

Part – 3

D. Long Answer Type Questions (300 words)

1. Discuss the differences between structure, union, and class in C++.
2. Explain dynamic memory allocation and pointers to objects.
3. Explain constructors and destructors with their special characteristics.
4. What is encapsulation in classes?
5. Discuss the importance of classes in OOP.



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN A/SECT GROUP UNIVERSITY

Approved by: PCI | AICTE | NCTE | UCI | Member of: APU | Joint Committee: UGC | UER | AICTE | Recognized by: UGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:

BLOCK 3	POLYMORPHISM IN C++	117-160
Unit 7	Introduction to Polymorphism	119
Unit 8	Function Overloading	127
Unit 9	Operator Overloading	137



Polymorphism is a cornerstone of object-oriented programming, allowing objects to behave differently based on context. In C++, polymorphism enables the same function or operator to operate on different types of data, improving code flexibility, readability, and maintainability. This block provides a thorough understanding of both compile-time (function and operator overloading) and run-time polymorphism (virtual functions), illustrating how C++ supports multiple ways of performing similar operations. Students explore the mechanics of function overloading, constructor overloading, and operator overloading—including unary, binary, and special operators.

They also learn the restrictions and rules governing operator overloading, including the use of friend functions for enhanced access to class members. Practical examples demonstrate the concatenation of strings, overloading the subscript, function call, and I/O operators. The block emphasizes applying polymorphism in real-world programming scenarios, enabling students to write more generalized and reusable code. By the end of this unit, learners will understand how polymorphism enhances object-oriented programming, allowing a single interface to represent different underlying forms of data and behavior.

UNIT 7

INTRODUCTION TO POLYMORPHISM

NOTE

UNIT STRUCTURE

- 7.1 Introduction
- 7.2 Objectives
- 7.3 Introduction to Polymorphism
- 7.4 Types of Polymorphism
- 7.5 Summary
- 7.6 Glossary
- 7.7 Answer to Check Your Progress
- 7.8 Suggested Readings – OER
- 7.9 Reference
- 7.10 Exercise

7.1 INTRODUCTION

Unit 7 builds upon the learner's solid understanding of classes, objects, constructors, destructors, and object management developed in the previous unit. Having learned how data and functions are encapsulated within classes, learners are now ready to explore how objects can exhibit flexible and dynamic behavior. Object-oriented programs often need the same operation to behave differently depending on the context, and this requirement forms the basis of polymorphism. This unit introduces polymorphism as one of the fundamental pillars of object-oriented programming. The term polymorphism means "many forms," and in C++ it refers to the ability of a single function name or interface to represent different behaviors. Learners study the definition, purpose, and importance of polymorphism in designing extensible and maintainable software systems. The unit explains how polymorphism improves code readability, reduces redundancy, and supports scalability. Different types of polymorphism supported by C++ are introduced, with a special focus on compile-time polymorphism. Conceptual explanations and real-world analogies are used to help learners clearly understand how polymorphism operates in programming contexts. Rather than immediately implementing polymorphism, this unit emphasizes conceptual clarity so learners can distinguish polymorphism from other OOP features such as encapsulation and inheritance. Understanding this distinction is essential for applying polymorphism correctly in later stages of programming. By the end of this unit, learners develop a strong theoretical foundation of polymorphism and recognize its role in software design.

This conceptual understanding prepares learners to explore the practical implementation of polymorphism. The next unit focuses on function overloading, which is a key mechanism of compile-time polymorphism in C++, allowing multiple functions with the same name to perform different tasks based on parameters.

NOTE**7.2 OBJECTIVES**

After completing this unit, learners will be able to:

- Explain the role of polymorphism in object-oriented design.
- Differentiate between types of polymorphism in C++.
- Analyze program scenarios requiring polymorphic behavior.
- Relate polymorphism to flexibility and code reusability.
- Prepare polymorphic program designs for implementation.

7.3 INTRODUCTION TO POLYMORPHISM

1. Polymorphism means “one name multiple forms”.
2. Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common base class.
3. In C++, polymorphism is typically achieved via inheritance and is classified into two main types: compile-time (static) polymorphism and run-time (dynamic) polymorphism.

Example:

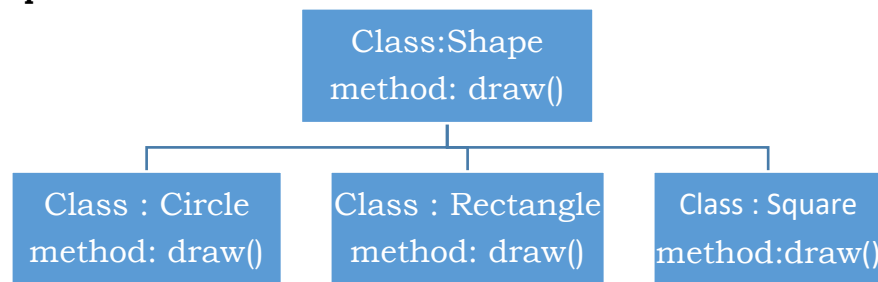


Fig. No. 3.1. Polymorphism

We'll start with a base class called Shape, which will have a method called draw. Then, we'll create three subclasses: Circle, Square and Rectangle. Each subclass will implement its own version of the draw method.

You can run c++ programs online using following online C++ Compiler

<https://www.programiz.com/cpp-programming/online-compiler/>

```

#include <iostream>
using namespace std;
class Shape {
public:
    virtual void draw () {
        cout << "Drawing Shape" << endl;
    }
};

class Circle: public Shape {
public:
    void draw () //function overriding
  
```

NOTE

```

{ cout << "Drawing Circle" << endl;
  }
};
class Rectangle : public Shape {
public:
    void draw () { //function overriding
        cout << "Drawing Rectangle" << endl;
    }
};

class Square: public Shape {
public:
    void draw () { //function overriding
        cout << "Drawing Square" << endl;
    }
};

int main () {
    Shape* shape;
    Circle circle;
    Rectangle rect;
    Square square;
    shape = &circle;
    shape->draw();
    shape = &rect;
    shape->draw();
    shape = &square;
    shape->draw();
    return 0;
}

```

OUTPUT

Drawing Circle
 Drawing Rectangle
 Drawing Square

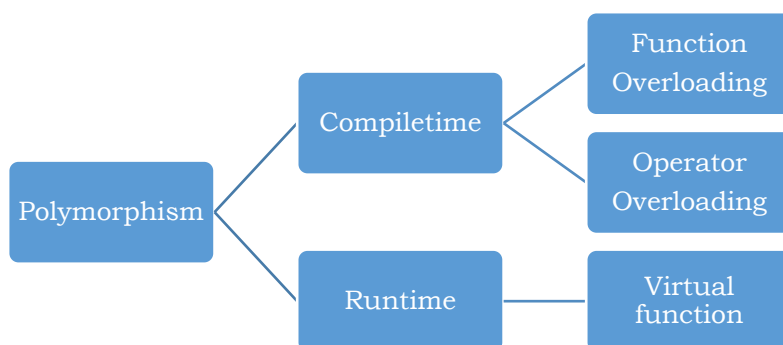
7.4 TYPES OF POLYMORPHISM

Fig. No. 3.2. Types of Polymorphism

NOTE**Activity 7.1**

❖ One Interface, Many Behaviors.

Objective: To design scenarios requiring polymorphism.

Required Materials: Notebook, compiler

Prerequisites: Classes and objects

Steps:

1. Identify a polymorphic scenario.
2. Describe multiple behaviors.
3. Map behaviors to class designs.

Post-Activity Reflection: Why is polymorphism essential for scalability?

Expected Outcome: Learners recognize appropriate polymorphic design situations.

☑ Check Your Progress 7.1**1. A. Multiple Choice Question (MCQ)**

1. What does the term *polymorphism* mean in object-oriented programming?
 - a) Having multiple classes
 - b) Having multiple forms
 - c) Hiding data from users
 - d) Creating a single function

B. True/False

1. Polymorphism allows the same function name to perform different tasks based on the context.

C. Short Answer Questions

1. Define polymorphism in object-oriented programming.

.....

.....

.....

2. Name any two types of polymorphism.

.....

.....

.....

7.5 SUMMARY

Unit 7 introduces polymorphism as a fundamental concept of object-oriented programming. It explains how a single interface can represent multiple behaviors, enhancing flexibility and reusability. The unit focuses on conceptual understanding, distinguishing polymorphism from encapsulation and inheritance. Learners explore the purpose, importance, and types of polymorphism in C++, with emphasis on code scalability and maintainability. This conceptual foundation prepares learners for practical implementation through function overloading in the next unit.

7.6 GLOSSARY

- **Polymorphism** – One interface, multiple behaviors
- **Compile-time Polymorphism** – Behavior resolved during compilation
- **Run-time Polymorphism** – Behavior resolved during execution
- **Virtual Function** – Function supporting dynamic binding
- **Inheritance** – Acquiring properties from a base class
- **Function Overloading** – Same function name, different parameters
- **Dynamic Binding** – Function call resolved at run time

7.7 ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 7.1

A. MCQ Answer:

1. b) Having multiple forms

Explanation: Polymorphism means “many forms,” allowing the same function or operator to behave differently in different situations.

B. True/False Answer:

1. True

Explanation: Polymorphism enables one interface to be used for different actions depending on the situation.

C. Short Answer Answers:

1. **Definition:** Polymorphism is an OOP concept that allows a single function, method, or operator to take multiple forms and perform different tasks.

2. Two types of polymorphism:

- Compile-time (Static) Polymorphism
- Run-time (Dynamic) Polymorphism

NOTE

NOTE**7.8 SUGGESTED READINGS – OER**

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/UnitUrls?unitId=1472	

7.9 REFERENCE

- Basu, S. (2004). E-government and developing countries: An overview. International Review of Law, Computers & Technology, 18(1), 109–132.
- Heeks, R. (2006). Implementing and managing e-government: An international text. London: SAGE Publications.
- Jaeger, P. T., & Thompson, K. M. (2003). E-government around the world: Lessons, challenges, and future directions. Government Information Quarterly, 20(4), 389–394.
- Kumar, P. S. G., & Kaur, A. (2010). Information technology applications in libraries. New Delhi: B. R. Publishing Corporation.
- UNESCO. (2011). E-governance and libraries: Access to government information. Paris: UNESCO.

7.10 EXERCISE**Part – 1****A. Multiple Choice Questions :**

- 1. What does polymorphism mean in object-oriented programming?**
 - A. One object many classes
 - B. One name multiple forms
 - C. One function one task
 - D. One class multiple objects
- 2. Which concept enables different behavior for the same function call?**
 - A. Encapsulation
 - B. Abstraction
 - C. Polymorphism
 - D. Inheritance
- 3. How many main types of polymorphism are supported in C++?**
 - A. One

- B. Two
- C. Three
- D. Four

4. Which mechanism is associated with compile-time polymorphism?

- A. Function overriding
- B. Virtual functions
- C. Function overloading
- D. Inheritance

5. Which keyword enables run-time polymorphism in C++?

- A. static
- B. virtual
- C. inline
- D. extern

Answer with Explanations

1. B One name multiple forms

Explanation: Polymorphism allows a single function name or interface to represent different behaviors.

2. C Polymorphism

Explanation: Polymorphism allows the same operation to behave differently based on context.

3. B Two

Explanation: C++ supports compile-time (static) and run-time (dynamic) polymorphism.

4. C Function overloading

Explanation: Compile-time polymorphism is achieved using function overloading.

5. B virtual

Explanation: Virtual functions support dynamic binding at run time.

B. Short Answer Type Questions

1. Define polymorphism.
2. Why is polymorphism important in software design?
3. What are the two main types of polymorphism in C++?
4. Discuss real-life examples of polymorphism.
5. Differentiate between static and dynamic binding.

NOTE

Part – 2

C. Long Answer Type Questions (200 words)

1. Explain the concept and purpose of polymorphism in object-oriented programming.
2. Explain how polymorphism supports code reusability and flexibility.
3. How is polymorphism implemented in C++?
4. Explain function overloading.
5. Explain operator overloading.

Part – 3

D. Long Answer Type Questions (300 words)

1. Discuss polymorphism as a fundamental pillar of object-oriented programming.
2. Explain compile-time and run-time polymorphism in C++.
3. Differentiate polymorphism from encapsulation and inheritance.
4. Explain the role of polymorphism in scalable software systems.
5. Explain compile-time and runtime polymorphism.

UNIT 8

FUNCTION OVERLOADING AND CONSTRUCTOR OVERLOADING

NOTE

UNIT STRUCTURE

- 8.1. Introduction
- 8.2. Objective
- 8.3. Function overloading
- 8.4. Overloading Constructor Function
- 8.5. Finding the address of an overloaded function
- 8.6. Summary
- 8.7. Glossary
- 8.8. Answers to Check Your Progress
- 8.9. Suggested readings - OER
- 8.10. Reference
- 8.11. Exercise

8.1 INTRODUCTION

This unit builds directly on the conceptual understanding of polymorphism developed in the previous unit. Learners are now prepared to study how polymorphism is implemented at compile time in C++. Function overloading is introduced as one of the most commonly used techniques for achieving polymorphic behavior, allowing multiple functions with the same name to coexist within a program.

The unit explains how function overloading depends on differences in the number, type, or order of parameters, and how the compiler determines which version of the function to invoke. Learners study the rules and restrictions governing function overloading and understand why return types alone cannot distinguish overloaded functions. Practical examples are used to demonstrate how overloading improves code clarity and usability. The unit also covers overloading constructor functions, enabling objects to be initialized in different ways depending on the supplied arguments. In addition, learners explore how the addresses of overloaded functions are resolved and managed by the compiler.

Through this unit, learners gain insight into compile-time decision-making and how C++ supports flexible program design without sacrificing performance. Emphasis is placed on writing clear and meaningful overloaded functions that enhance maintainability rather than introduce confusion.

By the end of this unit, learners are able to design and implement overloaded functions and constructors confidently. This understanding prepares learners to extend the concept of overloading beyond functions to operators. The next unit introduces operator overloading, where built-in operators are redefined to

NOTE

work with user-defined data types, further strengthening polymorphic programming skills

8.2 OBJECTIVES

After completing this unit, learners will be able to:

- Implement function overloading using different parameter lists.
- Design overloaded constructors for flexible object initialization.
- Analyze compiler resolution of overloaded functions.
- Apply overloading rules to avoid ambiguity in programs.
- Develop clean and reusable interfaces using overloaded functions

8.3 FUNCTION OVERLOADING

1. Function overloading allows multiple functions to have the same name but different parameters like type, number, or both within the same scope.
2. It increases code readability and usability by enabling similar operations to be performed using the same function name.
3. Function overloading is resolved at compile time (in statically typed languages), which means the correct function is selected based on the arguments passed at compile time.

Example1: Compile-Time Polymorphism (Function Overloading)

```
#include <iostream>

using namespace std;

class Example {
public:
    void display(int a) {
        cout << "Integer: " << a << endl;
    }

    void display(double a) {
        cout << "Double: " << a << endl;
    }
};

int main() {
    Example obj;
    obj.display(5);
    obj.display(3.14);
}
```

```
return 0;
}
```

OUTPUT

Integer: 5

Double: 3.14

Example2: Function Overloading

```
#include <iostream>
using namespace std;
class Math {
public:
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
};
int main() {
    Math m;
    cout << "Sum of integers: " << m.add(5, 10) << endl;
    cout << "Sum of doubles: " << m.add(5.5, 10.5) << endl;
    return 0;
}
```

OUTPUT

Sum of integers: 15

Sum of doubles: 16

8.4 OVERLOADING CONSTRUCTOR FUNCTION

Constructor overloading allows the creation of multiple constructors with different parameters within the same class.

Example: Constructor Overloading

```
#include <iostream>
using namespace std;
class Box {
private:
```

POLYMORPHISM IN C++

NOTE

```
int length, width;

public:

Box() {          //Default constructor
    length = 0;
    width = 0;
}

Box(int l, int w) { //Parametrized constructor
    length = l;
    width = w;
}

void display() {
    cout << "Length: " << length << ", Width: " << width << endl;
}

};

int main () {
    Box b1;
    Box b2(10, 20);
    cout<<"b1="<<endl;
    b1. display ();
    cout<<"b2="<<endl;
    b2. display ();
    return 0;
}
```

OUTPUT

```
b1=
Length: 0, Width: 0
b2=
Length: 10, Width: 20
```

8.5 FINDING THE ADDRESS OF AN OVERLOADED FUNCTION

In C++, overloaded functions have different signatures, so their addresses can be taken and assigned to function pointers. This is useful when we want to store and call different versions of overloaded functions dynamically.

Example: Function Overloading and Function Pointers

```
#include <iostream>

using namespace std;

void print(int x) {
    cout << "Integer: " << x << endl;
}

void print(double x) {
    cout << "Double: " << x << endl;
}

int main() {
    void (*intPtr)(int) = print;
    void (*doublePtr)(double) = print;

    intPtr(10);    // Calls print(int)
    doublePtr(20.5); // Calls print(double)

    return 0;
}
```

OUTPUT

Integer: 10
Double: 20.5

NOTE



Activity 8.1

Activity Name: Overload for Flexibility

Objective: To implement overloaded functions and constructors.

Required Materials: Computer, compiler

Prerequisites: Functions and classes

Steps:

1. Create multiple functions with same name.
2. Overload constructors.
3. Observe compiler resolution.

Post-Activity Reflection: When can overloading cause ambiguity?

Expected Outcome: Learners apply overloading rules effectively.

NOTE

☑ Check Your Progress 8.1**A. Multiple Choice Question (MCQ)**

1. Which of the following statements is true about constructor overloading?
 - a) A class can have only one constructor
 - b) Constructors can have the same name but must differ in parameters
 - c) Constructors must return a value
 - d) Constructor overloading is not allowed in C++

B. True/False Question

1. Function overloading can be achieved by changing only the return type of a function. (True/False)

C. Very Short Answer Questions

1. Define function overloading in one sentence.

.....

2. What is the main purpose of constructor overloading?

.....

8.4 SUMMARY

Function Overloading is an important concept in Object-Oriented Programming (OOP) that allows a programmer to define more than one function with the same name in the same class or program, but with different parameter lists. The difference in parameters can be in the number of arguments, type of arguments, or order of arguments. This feature makes the code more readable, organized, and flexible because the same function name can be used to perform similar operations in different ways.

Constructor Overloading is a special type of function overloading applied to class constructors. It allows a class to have multiple constructors with different sets of parameters. This makes it easy to create objects in different ways, such as creating objects with default values, partial information, or complete data. Constructor overloading helps in making programs more user-friendly and adaptable.

Finding the Address of an Overloaded Function involves the way the compiler handles multiple functions with the same name. Since overloaded functions share the same name, the compiler uses a process called function signature matching to select the correct function based on the arguments provided

during the function call. Internally, the compiler assigns unique identifiers and memory addresses to each overloaded version, ensuring the correct function is executed.

NOTE

8.5 GLOSSARY

- **Function Overloading** – Defining multiple functions with the same name but different parameter lists within the same scope.
- **Constructor Overloading** – Defining multiple constructors in a class, each with a different parameter list, to allow different ways of object initialization.
- **Parameter List** – The set of variables (number, type, and order) that a function or constructor accepts.
- **Function Signature** – A combination of the function's name and its parameter list that uniquely identifies it for the compiler.
- **Overloaded Function** – A function that shares its name with other functions but has a unique parameter list.
- **Default Constructor** – A constructor that takes no arguments and initializes objects with default values.
- **Parameterized Constructor** – A constructor that takes arguments to initialize an object with specific values.
- **Compiler** – The program that translates source code into machine code and resolves which overloaded function or constructor to execute.
- **Function Address** – The memory location assigned to a function by the compiler, which is used when the function is called.
- **Argument Matching** – The process by which the compiler selects the correct overloaded function based on the type, number, and order of arguments passed.
- **Type Promotion** – When the compiler automatically converts a smaller data type (like `int`) to a larger type (like `float`) to match an overloaded function.
- **Scope** – The region of a program where a function or variable is accessible. Overloaded functions must exist in the same scope to be considered for overloading.
- **Default Values** – Values assigned to function parameters if no argument is provided during a call.
- **Compile-Time Polymorphism** – Another term for function and constructor overloading because the function to be executed is determined at compile time.
- **Signature Conflict** – A situation where two functions or constructors have the same name and identical parameter lists, which is not allowed in overloading.

8.6 ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 8.1

A. MCQ Answer:

1. b) Constructors can have the same name but must differ in parameters

Explanation: Constructor overloading allows multiple constructors in a class with the same name but different parameter lists to initialize objects in different ways.

B. True/False Answer:

1. False

Explanation: Function overloading cannot be achieved by changing the return type alone. The parameter list must differ in number, type, or order.

C. Short Answer Questions

1. Function overloading is defining multiple functions with the same name but different parameter lists in the same scope.
2. Purpose of constructor overloading: To allow objects of a class to be initialized in multiple ways depending on the arguments passed

8.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/UnitUrls?unitId=1473	

8.8 REFERENCE

- C++: The Complete Reference — Herbert Schildt — ISBN-13 9780072226805 / ISBN-10 0072226803 — 2002 (4th Edition)
- Object-Oriented Programming in C++ — Rajesh K. Shukla — ISBN-13 9788126516582 / ISBN-10 8126516582 — 2008
- Object Oriented Programming with C++ — M. T. Somashekara, D. S. Guru, H. S. Nagendraswamy, K. S. Manjunatha — ISBN 9788120344624 — 2nd Edition
- Object Oriented Programming with C++ — E. Balagurusamy — ISBN-13 9789389949186 / ISBN-10 9389949181 — 2020 (8th Edition)
- Object-Oriented Software Construction* — Bertrand Meyer — ISBN 9780136291558 — 2nd Edition (1997)

8.9 EXERCISE**NOTE****Part – 1****A. Multiple Choice Questions****1. Which of the following is true about function overloading?**

- a) Functions must have different names
- b) Functions can have the same name but different parameter lists
- c) Functions must return the same data type
- d) Overloading is not allowed in C++

2. Constructor overloading allows:

- a) Only one constructor per class
- b) Multiple constructors with the same parameter list
- c) Multiple constructors with different parameter lists
- d) Constructors without names

3. Which of the following is not a criterion for function overloading?

- a) Number of parameters
- b) Order of parameters
- c) Return type alone
- d) Type of parameters

4. The process by which the compiler selects the correct overloaded function is called:

- a) Dynamic binding
- b) Function signature matching
- c) Scope resolution
- d) Memory allocation

5. What is the main advantage of constructor overloading?

- a) Reduces memory usage
- b) Allows objects to be initialized in different ways
- c) Functions run faster
- d) Increases compile time

Answer with Explanations**1. b) Functions can have the same name but different parameter lists.**

Explanation: Function overloading is based on differences in parameter lists, not names.

2. c) Multiple constructors with different parameter lists.

Explanation: Constructor overloading allows creating objects in different ways using multiple constructors.

3. c) Return type alone.

Explanation: Overloading cannot be done only by changing return type; parameter lists must differ.

NOTE

4. b) Function signature matching.

Explanation: The compiler matches function calls to the correct overloaded function using the parameter list.

5. b) Allows objects to be initialized in different ways.

Explanation: Constructor overloading provides flexibility in object creation with different initial values.

B. Short Answer Type Questions

1. Define function overloading and give an example.
2. Explain constructor overloading with an example.
3. What is a function signature, and why is it important in overloading?
4. How does the compiler select the correct overloaded function?
5. What are the advantages of using constructor overloading in OOP?

Part – 2**C. Long Answer Type Questions (200 words)**

1. Describe function overloading in detail with at least two examples showing different parameter types and numbers.
2. Explain constructor overloading and how it helps in object initialization, providing an example.
3. Discuss the rules and constraints of function overloading in C++.
4. How does the compiler find the address of an overloaded function? Explain with an example.
5. Compare function overloading and operator overloading in terms of syntax, use, and advantages.

Part – 3**D. Long Answer Type Questions (300 words)**

1. Explain compile-time polymorphism with reference to function overloading and constructor overloading. Include examples to illustrate.
2. Write a detailed note on the advantages of function and constructor overloading in C++ with practical scenarios.
3. Discuss the concept of argument matching and type promotion in overloaded functions with examples.
4. Describe potential errors and signature conflicts that may arise in function and constructor overloading, and how to avoid them.
5. Write a comprehensive example program demonstrating both function overloading and constructor overloading, explaining each part of the code.

UNIT 9

OPERATOR OVERLOADING

IN C++

NOTE

UNIT STRUCTURE

- 9.1. Introduction
- 9.2. Objective
- 9.3. Operator Overloading
- 9.4. Creating a Member Operator Function
- 9.5. Operators (Overloading Unary Operator)
- 9.6. Creating Prefix and Postfix Forms of the Increment (++) and Decrement (--)
- 9.7. Overloading the Shorthand Operators (I.E. +=, == Etc.)
- 9.8. Operator Overloading Restriction (Rules)
- 9.9. Operator Overloading Using Friend Function
- 9.10. Overloading New and Delete Operator
- 9.11. Overloading Some Special Operators- Overloading [] (Subscripting) Operator
- 9.12. Overloading () (Function Call) Operator
- 9.13. Overloading Binary Arithmetic Operators
- 9.14. Concatenating String
- 9.15. Overloading Comma (,) Operator
- 9.16. Overloading the I/O Operators
- 9.17. Summary
- 9.18. Glossary
- 9.19. Answers to Check Your Progress
- 9.20. Suggested Readings - OER
- 9.21. Reference
- 9.22. Exercise

9.1 INTRODUCTION

Building on the principles of polymorphism and function overloading, this unit introduces operator overloading as a powerful feature of C++. Learners have already seen how functions and constructors can be overloaded to perform different operations, and this unit extends that idea to C++ operators. Operator overloading allows user-defined data types such as classes to behave like built-in types, making programs more intuitive and readable.

This unit covers the syntax, rules, and restrictions of operator overloading in detail. Learners study overloading of unary operators, binary arithmetic operators, shorthand operators, and comparison operators. Special attention is given to overloading increment and decrement operators in both prefix and postfix forms. The unit also explains operator overloading using friend functions, highlighting scenarios where non-member access is required.

OBJECT ORIENTED PROGRAMMING IN C++

NOTE

Advanced topics such as overloading the subscript operator, function call operator, comma operator, and input/output operators are discussed. Overloading of special operators like new and delete is also introduced to explain custom memory management.

Through practical examples, learners understand how operator overloading enhances class usability while maintaining clarity and correctness. Emphasis is placed on responsible use of operator overloading to avoid ambiguity and misuse.

By the end of this unit, learners gain confidence in designing expressive class interfaces using overloaded operators. This capability prepares learners to explore inheritance, where behavior and properties are extended across class hierarchies. The next unit introduces the fundamentals of inheritance and access control in C++.

9.2 OBJECTIVES

After completing this unit, learners will be able to:

- Implement unary and binary operator overloading.
- Apply rules and restrictions governing operator overloading.
- Design user-defined types supporting natural operator behavior.
- Use friend functions to overload operators where required.
- Evaluate appropriate use cases for special operator overloading.

9.3 OPERATOR OVERLOADING

C++ has the ability to provide the operators with as special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

The process of overloading involves following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function operator op () in the public part of the class. It may be a member function or a friend function.
3. Here op is the operator to be overloaded.
4. Define the operator function to implement the required operations.

Syntax:

```
return-type classname :: operator op(arglist)
{
    Function body
}
```

9.4 CREATING A MEMBER OPERATOR FUNCTION

An binary operator means, an operator which works on two operands. For example, + is an binary operator, it takes single operand (c+d). So, when

NOTE

overloading a binary operator, it takes one argument (one is object itself and other one is passed argument).

Example: Overloading the + Operator for a Complex Number Class using member function

```
#include <iostream>

using namespace std;

class Complex {
public:
    double real, imag;
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}
    Complex operator + (const Complex& c) {
        return Complex(real + c.real, imag + c.imag);
    }
    void display() {
        cout << "Real: " << real << ", Imaginary: " << imag << endl;
    }
};

int main() {
    Complex c1(3.5, 2.5), c2(1.5, 4.5);
    Complex result = c1 + c2; //call for + operator ()
    result.display();
    return 0;
}
```

OUTPUT

Real: 5, Imaginary: 7

9.5 OPERATORS (OVERLOADING UNARY OPERATOR)

An unary operator means, an operator which works on single operand. For example, ++ is an unary operator, it takes single operand (c++). So, when overloading a unary operator, it takes no argument(because object itself is considered as argument).

Syntax for Unary Operator (Inside a class)

```
return-type operatorsymbol( )
{
    //body of the function
}
```

Syntax for Unary Operator (Outside a class)

```
return-type classname::operator operatorsymbol( )
{
    //body of the function
}
```

NOTE

```
//body of the function
}
```

Example: Overloading Unary Operator –

```
#include<iostream>
using namespace std;
class complex {
float real,img;
public:
complex();
complex(float x, float y);
void display();
void operator-(); //unary - operator
};
complex::complex()
{
real=0;img=0;
}
complex::complex(float x, float y)
{
real=x;
img=y;
}
void complex::display()
{
int imag=img;
if(img<0)
{
imag=-img;
cout<<real<<"-i"<<imag<<endl;
}
else
cout<<real<<"+"i"<<imag<<endl;
}
void complex::operator-()//unary – operator function
{
```

NOTE

```

real=-real;
img=-img;
}
int main()
{
complex c(1,-2);
c.display();
cout<<" After Unary-operation \n";
-c;
c.display();
return 0;
}

```

OUTPUT

1-i2

After Unary – operation

-1+i2

Explanation:

Object c has the imaginary part -2 is negative, so the output of c. display () will be 1-i2. After that the unary - operator is called on c, changing its values: real part becomes -1 and Imaginary part becomes 2. The next call to c.display() will display -1+i2.

9.6 CREATING PREFIX AND POSTFIX FORMS OF THE INCREMENT (++) AND DECREMENT (--)

Unary “operators” like ++ and -- can be overloaded both in prefix and postfix forms.

Example 1: Overloading Prefix ++

```

#include <iostream.h>
using namespace std;
class Counter {
private:
    int count1;
public:
    Counter() : count1(0) {}

```

NOTE

```

Counter operator++() { // Prefix increment
    ++count1;
    return *this;
}

void display() {
    cout << "Count: " << count1 << endl;
}

};

int main() {
    Counter c;
    ++c;
    c.display(); // Output: 1
    return 0;
}

```

OUTPUT

Count: 1

Explanation: This output occurs because the ++c; increments the value of count from 0 to 1, and then c. display () prints the current value of count1(i.e. 1.)

Example 2: Overloading Postfix ++

```

#include <iostream>
using namespace std;
class Counter2 {
private:
    int count;
public:
    Counter2() : count(0) {}

    Counter2 operator++(int) { // Postfix increment
        Counter2 temp = *this; // store the current value of count
        count++;               // Increment the count
        return temp;           // return the original value
    }

    void display() {

```

NOTE

```

        cout << "Count: " << count << endl;
    }
};

int main() {
    Counter2 c;
    c++; // postfix increment
    c.display(); // Output: 1
    return 0;
}

```

OUTPUT

Count: 1

9.7 OVERLOADING THE SHORTHAND OPERATORS (I.E. +=, == ETC.)

Shorthand operators like += and -= can also be overloaded to work with user-defined types.

Example 1: Overloading +=

```

#include <iostream>
using namespace std;
class Point {
private:
    int x, y;
public:
    Point(int a = 0, int b = 0) : x(a), y(b) {}
    Point operator += (const Point& p) {
        x += p.x;
        y += p.y;
        return *this;
    }
    void display() {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Point p1(1, 2), p2(3, 4);

```

NOTE

```

    p1 += p2;
    p1.display(); // Output: (4, 6)
    return 0;
}

```

OUTPUT

(4, 6)

9.8 OPERATOR OVERLOADING RESTRICTION (RULES)

1. We can only overload existing operators.
2. Overloaded operators must have at least one operand that is of user defined type (i.e. class).
3. Basic meaning of an operator cannot be changed.
4. Overloaded operator must follow minimum characteristics of the original operator.
5. If using operator overloading with binary operators through member function left-hand operand should be an object of concerned class.

Certain operators can't be overloaded in C++:

1. . (Member access)
2. ?: (Ternary conditional)
3. .* (Pointer-to-member access)
4. :: (Scope resolution)

For example, you can't overload the dot operator (.) to access class members directly.

9.9 OPERATOR OVERLOADING USING FRIEND FUNCTION

Friend functions are also used to overload operators. Friend functions have access to the private and protected members of the class, and they are especially convenient when we need to overload binary operators where the left operand is not an object of the class.

Example: Overloading the + Operator Using a Friend Function

```

#include <iostream>

using namespace std;

class Complex {
private:
    double real, imag;
public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

```

NOTE

```

friend Complex operator + (Complex& c1, Complex& c2);
void display() {
    cout << "Real: " << real << ", Imaginary: " << imag << endl;
}
};
Complex operator + ( Complex& c1, Complex& c2) {
    Complex C;
    C.real=c1.real+c2.real;
    C.imag= c1.imag + c2.imag;
    return C;
}
int main() {
    Complex c1(1.5, 2.5), c2(2.5, 3.5);
    Complex result = c1 + c2;
    result.display();
    return 0;
}

```

OUTPUT

Real: 4, Imaginary: 6

9.10 OVERLOADING NEW AND DELETE OPERATOR

C++ allows overloading the new and delete operators for custom memory allocation and deallocation routines.

Example 1: Overloading new and delete

```

#include <iostream>
#include <cstdlib>
using namespace std;
class MyClass {
public:
    void* operator new(size_t size) {
        cout << "Overloaded new operator" << endl;
        void* p = malloc(size);
        return p;
    }
    void operator delete(void* p) {
        cout << "Overloaded delete operator" << endl;
    }
}

```

NOTE

```

        free(p);
    }
    MyClass() {
        cout << "Constructor" << endl;
    }

    ~MyClass() {
        cout << "Destructor" << endl;
    }
};

int main() {
    MyClass* obj = new MyClass(); // Calls overloaded new
    delete obj;                  // Calls overloaded delete
    return 0;
}

```

OUTPUT

Overloaded new operator
 Constructor
 Destructor
 Overloaded delete operator

Example 2: Overloading new[] and delete[] for Arrays

```

#include <iostream>
#include <cstdlib>

Using namespace std;

class ArrayClass {
public:
    void* operator new[](size_t size) {
        cout << "Overloaded new[] operator" << endl;
        void* p = malloc(size);
        return p;
    }

    void operator delete[](void* p) {
        cout << "Overloaded delete[] operator" << endl;
        free(p);
    }

    ArrayClass() {

```

NOTE

```

        cout << "Constructor" << endl;
    }
    ~ArrayClass() {
        cout << "Destructor" << endl;
    }
};

int main() {
    ArrayClass* arr = new ArrayClass[3]; // Calls overloaded new[]
    delete[] arr;                       // Calls overloaded delete[]
    return 0;
}

```

OUTPUT

```

Overloaded new[] operator
Constructor
Constructor
Constructor
Destructor
Destructor
Destructor
Overloaded delete[] operator

```

9.11 OVERLOADING SOME SPECIAL OPERATORS- OVERLOADING [] (SUBSCRIPTING) OPERATOR

The [] operator can be overloaded to provide array-like access to objects of a class.

Example: Overloading [] to Access Elements in a Custom Array Class

```

#include <iostream>
using namespace std;
class Array {
private:
    int arr[10];
public:
    Array() {
        for (int i = 0; i < 10; ++i) {
            arr[i] = i * 10;
        }
    }
}

```

NOTE

```

    }

    int& operator[](int index) {
        if (index >= 0 && index < 10)
            return arr[index];
        else {
            cout << "Index out of bounds" << endl;
            exit(0);
        }
    }
};

int main() {
    Array a;

    cout << "Element at index 2: " << a[2] << endl;

    a[2] = 50;

    cout << "Element at index 2 after update: " << a[2] << endl;

    return 0;
}

```

OUTPUT

Element at index 2: 20

Element at index 2 after update: 50

9.12 OVERLOADING () (FUNCTION CALL) OPERATOR

The function call operator () can be overloaded to allow objects of a class to behave like functions.

Example 1: Overloading the Function Call Operator

```

#include <iostream>

using namespace std;

class Functor {
public:
    void operator()(int x) {
        cout << "Functor called with value: " << x << endl;
    }
};

int main() {
    Functor f;

```

NOTE

```
f(100); // Functor object behaves like a function
return 0;
}
```

OUTPUT

Functor called with value: 100

Example 2: Using the Function Call Operator to Simulate Function Objects

```
#include <iostream>
using namespace std;
class Multiply {
private:
    int factor;
public:
    Multiply(int f) : factor(f) {}
    int operator()(int x) {
        return x * factor;
    }
};
int main() {
    Multiply mul(5);
    cout << "5 times 3 = " << mul(3) << endl;
    return 0;
}
```

OUTPUT

5 times 3 = 15

9.13 OVERLOADING BINARY ARITHMETIC OPERATORS

Binary operators like +, -, *, / can be overloaded to perform operations on objects of user-defined types.

Example: Overloading the * Operator for Matrix Multiplication

```
#include <iostream>
using namespace std;
class Matrix {
private:
    int a, b;
public:
    Matrix(int x = 0, int y = 0) : a(x), b(y) {}
```

NOTE

```

Matrix operator * (const Matrix& m) {
    return Matrix(a * m.a, b * m.b);
}

void display() {
    cout << "Matrix(" << a << ", " << b << ")" << endl;
}

};

int main() {
    Matrix m1(2, 3), m2(4, 5);
    Matrix result = m1 * m2;
    result.display(); // Output: (8, 15)
    return 0;
}

```

OUTPUT

Matrix(8, 15)

9.15 OVERLOADING COMMA (,) OPERATOR

The comma operator can be overloaded, though it's rarely used in practice. It evaluates both operands and returns the value of the second one.

The comma operator is typically used to evaluate two expressions in a left-to-right sequence. You can overload this operator to customize its behaviour for user-defined types.

Example1: Overloading the Comma Operator

```

#include <iostream>

using namespace std;

class Test {
private:
    int value;
public:
    Test(int v = 0) : value(v) {}
    Test operator , (const Test& t) {
        return t;
    }
    void display() {
        cout << "Value: " << value << endl;
    }
};

```

NOTE

```
int main() {
    Test t1(10), t2(20);
    Test t3 = (t1, t2); // Comma operator overloaded
    t3.display();      // Output: Value: 20
    return 0;
}
```

OUTPUT

Value: 20

Example2: Overloading the Comma Operator

```
#include <iostream>
using namespace std;
class MyClass {
private:
    int value;
public:
    MyClass(int v = 0) : value(v) {}
    MyClass operator , (const MyClass& obj) {
        return obj; // Returns the right operand
    }
    void display() const {
        cout << "Value: " << value << endl;
    }
};

int main() {
    MyClass obj1(10), obj2(20), obj3(30);
    MyClass result = (obj1, obj2, obj3); // Returns obj3
    result.display(); // Output: Value: 30
    return 0;
}
```

OUTPUT

Value: 30

9.16 OVERLOADING THE I/O OPERATORS

We can overload the stream insertion (<<) and extraction (>>) operators to input and output user-defined types.

NOTE

Overloading the stream insertion (<<) and extraction (>>) operators makes it possible to output and input user-defined types in the same way you handle basic data types like int and float.

Example: Overloading the << Operator for Output

```
#include <iostream>

using namespace std;

class Complex {
private:
    double real, imag;
public:
    Complex(){real=imag=0;}
    Complex(double r , double i ) { real=r; imag=i;}
    friend istream& operator >> (istream& din, Complex& c);
    friend ostream& operator << (ostream& out, const Complex& c);
};

istream& operator >> (istream& din, Complex& c){
    din>>c.real>>c.imag;
    return din;
}

ostream& operator << (ostream& out, const Complex& c) {
    out << "Real: " << c.real << ", Imaginary: " << c.imag;
    return out;
}

int main(){
    Complex a(4.5, 5.5); //Object 1: Calling Parameterized Constructor
    Complex b; //Calling default constructor
    cout<<"Enter value of real imag:";
    cin>>b; //or operator>>(cin,b);
    cout<<"\nValue of a=";
    cout<<a; //or operator(a);
    cout<<"\nValue of b=";
    cout<<b; //or operator(b);
    return 0;
}
```

OUTPUT

Enter value of real imag:2 3

Value of a=Real: 4.5, Imaginary: 5.5

Value of b=Real: 2, Imaginary: 3

NOTE



Activity 9.1

Activity Name: Redefining Operators

Objective: To overload operators for custom data types.

Required Materials: Computer, compiler

Prerequisites: Classes and function overloading

Steps:

1. Define a class with data members.
2. Overload arithmetic operators.
3. Test using objects.

Post-Activity Reflection: What makes operator overloading intuitive or confusing?

Expected Outcome: Learners design expressive class interfaces.

☑ Check Your Progress 9.1

A. Multiple Choice Question (MCQ) :

1. Which of the following is true about overloading the ++ operator in C++?
 - a) Only postfix form can be overloaded
 - b) Only prefix form can be overloaded
 - c) Both prefix and postfix forms can be overloaded separately
 - d) ++ operator cannot be overloaded

B. True/False Question :

1. Operator overloading can be used to change the meaning of operators like :: and sizeof.

C. Short Answer Questions :

1. What is the difference between overloading a **member function operator** and a **friend function operator**?

.....

.....

.....
2. Give one example of a **special operator** that can be overloaded and its use.

.....

.....

.....

NOTE**9.4 SUMMARY**

Operator overloading** in C++ allows programmers to redefine how standard operators work with user-defined types like classes. It enables objects to be manipulated using familiar operators, making the code more intuitive and readable. Operators can be overloaded using **member functions** or **friend functions**, depending on whether access to private members or the left-hand operand is needed. Unary operators like `++`, `--`, `!`, and `~` can be overloaded in **prefix** and **postfix** forms, while binary operators such as `+`, `-`, `\*`, `/`, and `==` can operate on two operands. **Shorthand operators** (`+=`, `-=`, etc.) simplify object operations, and special operators like `[]` and `()` allow objects to behave like arrays or callable functions. Memory management operators `new` and `delete` and I/O operators `<<` and `>>` can also be overloaded. However, some operators like `::` and `sizeof` cannot be overloaded, and overloaded operators should maintain intuitive behavior. Overall, operator overloading enhances object-oriented programming by making user-defined types more flexible, expressive, and easier to use.

9.14 CONCATENATING STRING

+ operator can be overloaded to concatenate strings.

Example: Overloading the + Operator for String Concatenation

```
#include <iostream>

#include <cstring>

using namespace std;

class String {
private:
    char* str;
public:
    String(const char* s = "") {
        str = new char[strlen(s) + 1];
        strcpy(str, s);
    }

    String operator + (const String& s) {
        char* temp = new char[strlen(str) + strlen(s.str) + 1];
        strcpy(temp, str);
        strcat(temp, s.str);
        return String(temp);
    }

    void display() {
        cout << str << endl;
    }
}
```

NOTE

```

    }
    ~String() {
        delete[] str;
    }
};

int main() {
    String s1("Hello ");
    String s2("World!");
    String s3 = s1 + s2;
    s3.display(); // Output: Hello World!
    return 0;
}

```

OUTPUT

Hello World!

9.5 GLOSSARY

- **Operator Overloading** – Defining custom behavior for operators when applied to user-defined types.
- **Member Function** – A function defined inside a class that can access its members.
- **Unary Operator** – An operator that works with a single operand (e.g., ++, --).
- **Binary Operator** – An operator that works with two operands (e.g., +, -).
- **Prefix Operator** – The operator appears before the operand (++obj).
- **Postfix Operator** – The operator appears after the operand (obj++).
- **Shorthand Operators** – Compound assignment operators like +=, -=, *=, ==.
- **Friend Function** – A non-member function that has access to private and protected members of a class.
- **New/Delete Overloading** – Customizing memory allocation and deallocation for class objects.
- **Subscript Operator []** – Allows objects to behave like arrays for element access.
- **Function Call Operator ()** – Enables objects to be invoked as functions.
- **Comma Operator** – Evaluates multiple expressions and returns the last one.
- **I/O Operators << and >>** – Used for input/output overloading for objects.
- **Concatenation** – Joining two strings using the + operator.
- **Operator Overloading Rules** – Guidelines specifying which operators can/cannot be overloaded and how to maintain intuitive usage

NOTE**9.6 ANSWER TO CHECK YOUR PROGRESS****Check Your Progress 9.1****A. Multiple Choice Question (MCQ)**

- 1 Answer: c) Both prefix and postfix forms can be overloaded separately**

Explanation: Both prefix (++obj) and postfix (obj++) forms of the increment operator can be overloaded in C++. The postfix version uses a dummy int parameter to differentiate it from the prefix version.

B. True/False Question

- 2 Answer: False**

Explanation: Some operators, such as :: (scope resolution), ., .*, and sizeof, **cannot** be overloaded in C++.

C. Short Answer Questions

- 3. Answer:**

- **Member function operator:** Overloaded inside the class; the left-hand operand must be an object of the class.
- **Friend function operator:** Overloaded outside the class; can access private members and allows non-class objects as the left-hand operand.

- 4. Answer:**

Example: Subscript operator [] – Allows objects to be accessed like arrays.

Usage Example:

```
class Array {
    int arr[10];
public:
    int& operator[](int index) { return arr[index]; }
};

Array a;
a[0] = 5; // Using overloaded [ ] operator
```

9.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/UnitUrls?unitId=1474	

9.8 REFERENCE

- C++ Primer - Stanley B. Lippman, Josée Lajoie, Barbara E. Moo - 978-0321714114 - 2012
- The C++ Programming Language - Bjarne Stroustrup - 978-0321563842 - 2013
- C++; The Complete Reference - Herbert Schildt - 978-0072226805 - 2002
- Object Oriented Programming with C++ - E. Balagurusamy - (ISBN varies by edition) - 2013
- Effective C++: 55 Specific Ways to Improve Your Programs and Designs - Scott Meyers - 978-0321334879 - 2005

9.9 EXERCISE

Part – 1

A. Multiple Choice Questions

1. Which of the following operators cannot be overloaded in C++?

- a) +
- b) ::
- c) []
- d) ()

2. To differentiate between prefix and postfix increment operator overloading, which is used?

- a) int dummy parameter
- b) bool flag
- c) char type
- d) float parameter

3. Which operator is typically overloaded using a friend function?

- a) ++ (unary)
- b) << (output stream)
- c) () (function call)
- d) [] (subscript)

4. Which of the following statements about operator overloading is true?

- a) Overloaded operators can change the number of operands.
- b) All operators including sizeof can be overloaded.
- c) Overloading should maintain intuitive meaning.
- d) Overloaded operators must always be friend functions.

5. The subscript operator [] is used to:

- a) Access memory dynamically
- b) Access class object elements like an array
- c) Overload arithmetic operations
- d) Overload comparison operators

NOTE

NOTE**Answer with Explanations****1. b) ::**

Explanation: Scope resolution operator cannot be overloaded in C++.

2. a) int dummy parameter

Explanation: Used to differentiate postfix ++ from prefix ++.

3. b) << (output stream)

Explanation: Stream operators often need friend functions to access private members.

4. c) Overloading should maintain intuitive meaning

Explanation: Operators should behave logically to avoid confusion.

5. b) Access class object elements like an array

Explanation: [] allows array-like access to object members.

B. Short Answer Type Questions

1. What is operator overloading and why is it used?
2. Explain the difference between prefix and postfix increment operators.
3. How can friend functions be used for operator overloading?
4. Name two operators that cannot be overloaded in C++.
5. How can the new and delete operators be overloaded?

Part – 2**C. Long Answer Type Questions (200 words)**

1. Explain the process of overloading binary arithmetic operators in C++ with an example.
2. Describe how to overload the subscript [] operator in a class.
3. Explain overloading of the function call () operator and its use.
4. Describe the steps to overload shorthand operators like += and ==.
5. Discuss the rules and restrictions of operator overloading in C++.

Part – 3**D. Long Answer Type Questions (300 words)**

1. Discuss different methods of operator overloading in C++, including member functions and friend functions, with examples for unary and binary operators.
2. Explain how I/O operators (<< and >>) can be overloaded, including syntax and usage examples.
3. Explain how new and delete operators can be overloaded, with example scenarios.
4. Describe how operator overloading improves object-oriented programming and code readability, with examples.
5. Discuss special operators like [], (), and , and explain their overloading along with practical applications



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN A/SECT GROUP UNIVERSITY

Approved by: PCI | AICTE | NCTE | UCI | Member of: APU | Joint Committee: UGC | UER | AICTE | Recognized by: UGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:

BLOCK 4	INHERITANCE IN C++	161-200
Unit 10	Basics of Inheritance	163
Unit 11	Advanced Inheritance Concepts	175
Unit 12	Virtual Functions and Binding	187



Inheritance is a fundamental concept of object-oriented programming that allows a new class to acquire properties and behavior from an existing class. This promotes code reuse, reduces redundancy, and facilitates hierarchical modeling of real-world systems. In C++, inheritance provides flexibility in designing complex programs while maintaining a clean and organized code structure. This block introduces different types of inheritance, including single, multiple, and protected inheritance, emphasizing access control and encapsulation. Students learn how base and derived classes interact, the significance of protected members, and how constructors and destructors behave in an inheritance hierarchy.

The block also covers virtual base classes, virtual functions, and pure virtual functions to illustrate run-time polymorphism in inherited classes. Practical examples demonstrate early versus late binding, parameter passing to base class constructors, and method overriding. By the end of this block, learners will have a clear understanding of designing class hierarchies, controlling access, and using inheritance to extend functionality efficiently while maintaining robust object-oriented design principles.

UNIT 10

FUNDAMENTALS OF INHERITANCE

NOTE

UNIT STRUCTURE

- 10.1. Introduction
- 10.2. Objective
- 10.3. Base class Access control
- 10.4. Inheritance & Protected Members
- 10.5. Protected Base Class Inheritance
- 10.6. Summary
- 10.7. Glossary
- 10.8. Answers to Check Your Progress
- 10.9. Suggested readings - OER
- 10.10. Reference
- 10.11. Exercise

10.1 INTRODUCTION

This unit builds upon the learner's understanding of classes, objects, and polymorphic behavior developed in earlier units. Once programmers know how to define classes and overload functions and operators, the next logical step is to reuse and extend existing class definitions. Inheritance provides a mechanism for creating new classes from existing ones, supporting code reuse and hierarchical program design.

The unit introduces inheritance as a core concept of object-oriented programming and explains how a derived class inherits properties and behaviors from a base class. Learners study inheritance syntax in C++, along with different types of access control such as public, protected, and private inheritance. The role of protected members in inheritance is explained to help learners control accessibility while preserving encapsulation. Real-world examples are used to demonstrate "is-a" relationships and hierarchical modeling.

This unit emphasizes how inheritance reduces redundancy, promotes extensibility, and improves software maintainability. Learners also understand the limitations and appropriate use cases of inheritance, ensuring that it is applied effectively in program design.

By the end of this unit, learners are able to define base classes and derived classes and understand how data and functions are inherited. This foundational knowledge prepares learners to explore more complex inheritance scenarios, such as multiple inheritance, constructor and destructor execution order, and virtual base classes, which are discussed in the next unit.

10.2 OBJECTIVES

After completing this unit, learners will be able to:

NOTE

- Design class hierarchies using inheritance.
- Implement base and derived classes using access specifiers.
- Apply protected members for controlled data access.
- Analyze inheritance relationships using real-world models.
- Develop programs that reuse code through inheritance.

10.3 BASE CLASS ACCESS CONTROL

Base(old) class access control determines how the derived(new) classes can access the members (data and methods) of the base class.

Access specifiers used are:

public: Derived(child) class has full access to the public members of the base(parent) class.

protected: Derived class can access protected members but they are not accessible outside the class hierarchy.

private: Derived class can't access private members of the base class.

10.4 INHERITANCE & PROTECTED MEMBERS

1. The process of creating a new class from an old class is called inheritance or derivation.
2. The old class is known as the base class and new class as the derived class or sub class.
3. The derived class inherits some or all features from the base class.
4. A class can inherit properties from more than one class.
5. Using inheritance, we can achieve reusability which is the important feature of OOPs.
6. Inheritance can be used to extend and modify the capability of the existing class.

Syntax for defining derived class:

Class derived-class-name: visibility-mode base-class-name

{

Members of derived class

};

visibility mode specifies how the features of the base class are inherited by derived class. Visibility mode may be either private/protected/public. The default value is private.

1. private derivation or private inheritance

class XYZ: ABC //private by default

{

Members of XYZ

}; OR

class XYZ: private ABC

{

NOTE

Members of XYZ

};

2. public derivation or public inheritance

class XYZ: public ABC

{

Members of XYZ

};

3. protected derivation or protected inheritance

class XYZ: protected ABC

{

Members of XYZ

};

In inheritance, depending on the visibility mode some features(data members and member functions) of the base are inherited into the derived class which are as follows:

Tab. No. 4.1 Visibility of inherited members

Base class members	Visibility of inherited members		
	private derivation	public derivation	protected derivation
private	Not inherited	Not inherited	Not inherited
public	private	public	protected
protected	private	protected	protected

Example: Program to show visibility of inherited members using public derivation

```
#include<iostream.h>
#include< conio.h>
class Base {
public:
    int x; // public member of Base class
    void displayBase() {
        cout << "Base Class:" << endl;
        cout << "Public Variable: " << x<<endl;
        cout << "Private Variable: " << y<<endl;
        cout << "Protected Variable: " << z<<endl;
    }
private:
    int y; // Private member, not inherited
protected:
    int z; // Protected member, inherited by derived class

    void sety(int a) {
        y = a; // Setting private variable
    }
}; //end of Base class
```

NOTE

```

class Derived: public Base {
public:
    void setz(int a) {
        z = a; // Setting protected variable
    }
    void displayDerived() {
        sety(2); // Initializing private variable of Base class indirectly with the help of sety()
        cout << "Derived Class:" << endl;
        cout << "Public Variable: " << x << endl; // x is accessible
        //cout << "Private Variable: " << y << endl; // Will not work, shows error
        cout << "Protected Variable: " << z << endl; // z is accessible
    }
};

class Derived: public Base {
public:
    void setz(int a) {
        z = a; // Setting protected variable
    }
    void displayDerived() {
        sety(2); // Initializing private variable of Base class indirectly with the help of sety()
        cout << "Derived Class:" << endl;
        cout << "Public Variable: " << x << endl; // x is accessible
        //cout << "Private Variable: " << y << endl; // Will not work, shows error
        cout << "Protected Variable: " << z << endl; // z is accessible
    }
};

int main() {

    Derived d;

    d.x = 1; // This is possible as x is a public member
    d.setz(3);
    d.displayDerived();
    d.displayBase(); // Display from the base class
    return 0; // Wait for user input
}

```

OUTPUT

Derived Class:

Public Variable: 1

Protected Variable: 3

Base Class:

Public Variable: 1

Private Variable: 2

Protected Variable: 3

Protected members of the base class are accessible in derived classes but not outside the class hierarchy. It's useful when you want to hide some details but still allow derived classes to manipulate them. In above program z is the protected member of the base class which is accessible inside the derived class but not in main().

NOTE

10.5 PROTECTED BASE CLASS INHERITANCE

When a base class is inherited as protected (protected derivation), its public members become protected in the derived class (as shown in Table 4.1).

Example 1: Protected Base Class Inheritance

```
#include<iostream.h>
#include<conio.h>
class Base {
public:
    void show() {
        cout << "Base class method\n";
    }
};
class Derived: protected Base {
public:
    void showBase() {
        show(); // Can access Base's public method due to protected inheritance
    }
};
void main() {
    Derived d;
    d.showBase();
    // d.show(); // Error: cannot access Base's public method directly
    getch();
}
```

OUTPUT

Base class method

NOTE

Example 2: Accessing Members from Protected Base Class

```
#include<iostream.h>
#include<conio.h>
class Base {
public:
    int data;
};
class Derived: protected Base {
public:
    void setData(int val) { data = val; }
    void displayData() { cout << "Data: " << data << endl; }
};

void main() {
    Derived D;
    D.setData(10);
    D.displayData();
    getch();
}
```

OUTPUT

Data: 10

**Activity 10.1**

Name: Class Hierarchy Design

Objective: To implement inheritance-based designs.

Required Materials: Notebook, compiler

Prerequisites: Classes and objects

Steps:

1. Identify base and derived classes.
2. Implement access control.
3. Test inherited members.

Post-Activity Reflection: How does inheritance promote reuse?

Expected Outcome: Learners construct basic inheritance hierarchies.

NOTE**☑ Check Your Progress 10.1****A. Multiple Choice Question (1)**

1. Which of the following statements about protected members is correct?
 - a) They can be accessed by external code.
 - b) They are not inherited by derived classes.
 - c) They can be accessed by derived classes but not by external code.
 - d) They are equivalent to private members.

B. True/False Question (1)

1. Private members of a base class can be directly accessed by its derived classes.

(True / False)

C. Short Answer Questions (2)

1. Define protected inheritance and explain its main use.

.....

2. What is the difference between private and protected members in the context of inheritance?

.....

10.4 SUMMARY

Inheritance is a core concept in object-oriented programming that allows a class (derived class) to acquire properties and behaviors of another class (base class). It promotes code reusability, modularity, and simplifies program maintenance. Understanding how access control and member protection affect inheritance is crucial for designing robust systems.

Base Class Access Control determines how the members of a base class—public, private, and protected—are accessible to the derived class. Public members are fully accessible to derived classes and outside code. Private members, however, are inaccessible directly in derived classes, restricting their use to base class functions only. Protected members occupy an intermediate level, remaining hidden from external code but accessible in derived classes. The choice of access specifier in a base class directly influences the visibility and usability of its members in derived classes.

Inheritance & Protected Members emphasizes that protected members are particularly useful when designing a hierarchy where derived classes need controlled access to base class data. While private members enforce strict encapsulation, protected members provide a balance by allowing inheritance while still shielding data from external interference. This ensures derived

NOTE

classes can extend functionality without compromising the integrity of base class data.

Protected Base Class Inheritance refers to the use of the protected inheritance specifier. When a base class is inherited as protected, all its public and protected members become protected in the derived class. This restricts access from outside the derived class hierarchy while maintaining accessibility within the derived classes. Protected inheritance is useful in situations where you want to prevent direct use of base class interfaces by external code but allow derived classes to utilize and extend those members.

Overall, mastering access control and protected inheritance is essential for implementing safe, maintainable, and extensible class hierarchies. Correctly applying these concepts ensures proper encapsulation, controlled exposure of data, and efficient reuse of code through inheritance.

10.5 GLOSSARY

- **Inheritance** – A mechanism in object-oriented programming where a class (derived) acquires properties and behaviors of another class (base).
- **Base Class** – The class whose members and functions are inherited by another class. Also called a parent class.
- **Derived Class** – A class that inherits attributes and methods from a base class. Also known as a child class.
- **Access Specifier** – Keywords that define the visibility of class members. Includes public, private, and protected.
- **Public Members** – Class members accessible everywhere, including derived classes and external code.
- **Private Members** – Class members accessible only within the base class; not directly accessible by derived classes.
- **Protected Members** – Class members accessible within the base class and derived classes, but not by external code.
- **Protected Inheritance** – A type of inheritance where public and protected members of the base class become protected in the derived class.
- **Encapsulation** – The practice of keeping data and methods together in a class and controlling access via access specifiers.
- **Code Reusability** – The ability to use existing code (classes and functions) in new programs without rewriting it.
- **Class Hierarchy** – A structured organization of classes where relationships are established through inheritance.
- **Member Visibility** – The scope in which class members can be accessed, determined by access specifiers and inheritance type.

NOTE

10.6 ANSWER TO CHECK YOUR PROGRESS**Check Your Progress 10.1****A. MCQ Answer:**

1. c) They can be accessed by derived classes but not by external code.

Explanation: Protected members allow controlled access for derived classes while keeping them hidden from outside code, maintaining encapsulation.

B. True/False Answer:

1. Answer: False

Explanation: Private members cannot be directly accessed by derived classes; they are accessible only through base class methods.

C. Short Answer Questions Answers:

1. **Protected inheritance** is a type of inheritance where public and protected members of the base class become protected in the derived class. It is mainly used to allow derived classes access to base class members while preventing external code from using them.
2. **Difference between private and protected members:**
 - o Private members are accessible only within the base class.
 - o Protected members are accessible within the base class and also in derived classes, but not by external code.

10.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/UnitUrls?unitId=1475	

10.8 REFERENCE

- **Object-Oriented Programming with C++** - E. Balagurusamy - 978-0071072830 - 2011 (Fifth Edition)
- **Object-Oriented Programming in C++** - Robert Lafore - 978-0672323089 - 2001 (Fourth Edition)
- **The C++ Programming Language** - Bjarne Stroustrup - 978-0201889544 - 1997 (Third Edition - *Note: Newer editions are available, but this or the 4th edition are definitive references*)
- **C++ How to Program** - Harvey M. Deitel and Paul J. Deitel - 978-0136152507 - 2007 (Sixth Edition - *Note: Newer editions exist*)

10.9 EXERCISE

Part – 1**A. Multiple Choice Questions**

- 1. Which access specifier allows derived classes to access base class members but restricts access from outside code?**
 - a) Public
 - b) Private
 - c) Protected
 - d) Default
- 2. In protected inheritance, the public members of the base class become:**
 - a) Private in derived class
 - b) Protected in derived class
 - c) Public in derived class
 - d) Inaccessible
- 3. Which of the following members is not inherited by a derived class?**
 - a) Public members
 - b) Protected members
 - c) Private members
 - d) All are inherited
- 4. What is the primary benefit of using inheritance in object-oriented programming?**
 - a) Faster execution
 - b) Code reusability
 - c) Hiding all data
 - d) Eliminating classes
- 5. Which keyword is used to declare a derived class in C++?**
 - a) class
 - b) inherit
 - c) extends
 - d) :

Answer with Explanations**1. c) Protected**

Explanation: Protected members of a class are accessible within the class itself and by its derived classes, but they cannot be accessed directly by

external code. This allows controlled inheritance while maintaining encapsulation.

2. b) Protected in derived class

Explanation: In protected inheritance, both public and protected members of the base class become protected members in the derived class. This ensures that derived classes can access them, but they remain hidden from outside code.

3. c) Private members

Explanation: Private members of a base class are not directly inherited by the derived class. They are accessible only through public or protected member functions of the base class. Public and protected members are inherited depending on the type of inheritance.

4. b) Code reusability

Explanation: Inheritance allows a derived class to reuse existing code from a base class without rewriting it. This promotes modularity, reduces redundancy, and makes programs easier to maintain.

5. d) :

Explanation: In C++, the colon (:) is used in the class declaration to indicate inheritance. For example: `class Derived : public Base { ... };`

B. Short Answer Type Questions

1. Define inheritance and explain its importance in object-oriented programming.
2. Differentiate between private, public, and protected members.
3. Explain protected base class inheritance.
4. Why are protected members important in inheritance?
5. What is base class access control and how does it affect derived classes?

Part – 2

C. Long Answer Type Questions (200 words)

1. Explain base class access control in inheritance. Discuss how public, private, and protected members of a base class affect derived classes.
2. Discuss the significance of protected members in inheritance. How do they allow controlled access in derived classes while maintaining data security?
3. Describe different types of inheritance in C++ and how access specifiers affect member visibility.
4. Explain the advantages and limitations of protected base class inheritance.
5. How does inheritance promote code reusability and maintainability in object-oriented programming?

D. Long Answer Type Questions (300 words)

1. Compare and contrast public, private, and protected inheritance with examples. Discuss how member visibility changes in derived classes.
2. Illustrate with an example how protected base class inheritance can be applied in a real-world scenario, such as a banking system or library management system.
3. Discuss the role of inheritance in implementing class hierarchies and how access control helps in maintaining encapsulation.
4. Explain how inheritance affects the design of object-oriented programs, focusing on code reuse, modularity, and flexibility.
5. Write a program example in C++ demonstrating protected inheritance and explain how member visibility is affected in the derived class.

UNIT 11

ADVANCED INHERITANCE CONCEPTS

NOTE

UNIT STRUCTURE

- 11.1. Introduction
- 11.2. Objective
- 11.3. Inheriting Multiple Base Classes
- 11.4. Constructors, Destructors & Inheritance
- 11.5. Passing Parameters to Base Class Constructors.
- 11.6. Granting Access. Virtual Base Classes
- 11.7. Summary
- 11.8. Glossary
- 11.9. Answers to Check Your Progress
- 11.10. Suggested Readings - OER
- 11.11. Reference
- 11.12. Exercise

11.1 INTRODUCTION

Building on the fundamentals of inheritance, this unit explores advanced inheritance mechanisms used in complex software systems. Learners already understand how single inheritance works and how access control affects derived classes. This unit extends that knowledge to multiple inheritance, where a derived class inherits from more than one base class. The unit explains benefits, challenges, and ambiguity problems associated with multiple inheritance. The concept of virtual base classes is introduced to resolve duplication issues in multiple inheritance scenarios. Learners study how C++ ensures that only one copy of shared base class members exists. The unit also examines constructor and destructor behavior in inheritance hierarchies, explaining the order in which base and derived class constructors and destructors are executed. Passing parameters to base class constructors is discussed to help learners understand object initialization in inheritance chains. These advanced concepts are critical for designing robust and efficient class hierarchies. Practical examples reinforce how inheritance behavior affects memory allocation and object lifecycle management. By the end of this unit, learners gain deeper insight into inheritance complexities and confidently manage advanced class relationships. This prepares learners to understand dynamic behavior in inheritance using virtual functions. The next unit introduces virtual functions, pure virtual functions, and the concept of early and late binding in C++.

11.2 OBJECTIVES

After completing this unit, learners will be able to:

NOTE

- Implement multiple inheritance in C++ programs.
- Resolve ambiguity issues using virtual base classes.
- Analyze constructor and destructor execution order.
- Apply parameter passing to base class constructors.
- Design robust inheritance hierarchies for complex systems

11.3 INHERITING MULTIPLE BASE CLASSES

Multiple base classes are involved in multiple and multilevel inheritance.

Multilevel inheritance: This refers to multiple levels of inheritance where a class is derived from another derived class.

Example: Multilevel Inheritance

```
#include<iostream.h>
#include<conio.h>
class Base {
public:
    void showBase() {
        cout << "Base class method\n";
    }
};
class Intermediate: public Base {
public:
    void showIntermediate() {
        cout << "Intermediate class method\n";
    }
};
class Derived: public Intermediate {
public:
    void showDerived() {
        cout << "Derived class method\n";
    }
};

void main() {
    Derived d;
    d.showBase();
    d.showIntermediate();
    d.showDerived();
    getch();
}
```

OUTPUT

Base class method

Intermediate class method

Derived class method

Explanation: In above program, showBase() function of Base class publicly inherited to Intermediate base class and through it publicly inherited to Derived class. Also Derived class inherited showIntermediate() function publicly from Intermediate base class. Therefore, all the functions - showBase(), showIntermediate() and showDerived() are accessed using object of Derived class and dot operator.

Multiple Inheritance:

Multiple inheritance refers to a derived class that inherits/derived from more than one base class.

Example: Multiple Inheritance

```
#include<iostream.h>
#include<conio.h>
class Base1 {
public:
    void showBase1() {
        cout << "Base1 class method\n";
    }
};
class Base2 {
public:
    void showBase2() {
        cout << "Base2 class method\n";
    }
};

class Derived: public Base1, public Base2 {
public:
    void showDerived() {
        cout << "Derived class method\n";
    }
};

void main () {
    Derived D;
    D.showBase1();
    D.showBase2();
```

NOTE

NOTE

```

        D.showDerived();
        getch();
    }

```

OUTPUT

Base1 class method

Base2 class method

Derived class method

Explanation: In above program, Derived class inherits from both Base1 and Base2 classes.

Hence, each base class's methods can be accessed directly from the derived class instance/object (d).

11.4 CONSTRUCTORS, DESTRUCTORS & INHERITANCE

1. Constructors and Destructors are called in a specific order when inheritance is involved.
2. Firstly, the base class constructor is called, followed by derived class constructor.
3. Destruction happens in reverse order.

Example: Constructor & Destructor Call Order

```

#include<iostream.h>
#include<conio.h>
class Base {
public:
    Base() {
        cout << "Base class constructor\n";
    }
    ~Base() {
        cout << "Base class destructor\n";
    }
};
class Derived : public Base {
public:
    Derived() {
        cout << "Derived class constructor\n";
    }
    ~Derived() {
        cout << "Derived class destructor\n";
    }
};

```

NOTE

```
void main() {
    Derived D;
    getch();
}
```

OUTPUT

```
Base class constructor
Derived class constructor
Derived class destructor
Base class destructor
```

Explanation:

When an object of derived class created, base class constructor is called first, followed by the derived class constructor and when the object of derived class goes out of scope, derived class destructor is called first, followed by the base class destructor.

11.5 PASSING PARAMETERS TO BASE CLASS CONSTRUCTORS.

A derived class can pass parameters the base class constructor using the initializer list.

Example 1: Passing Parameters to Base Class Constructor

```
#include<iostream.h>
#include<conio.h>
class Base {
public:
    Base(int a) {
        cout << "Base class constructor, a = " << a << endl;
    }
};
class Derived: public Base {
public:
    Derived(int x, int y): Base(x) { //Derived constructor initializes the Base class with
                                   //the value of x
        cout << "Derived class constructor, y = " << y << endl;
    }
};
void main() {
    Derived obj (5, 10);
    getch();
}
```

OUTPUT

```
Base class constructor, a = 5
```

NOTE

Derived class constructor, y = 10

Example 2: Multiple Parameters to Base and Derived Constructors

```
#include<iostream.h>
#include<conio.h>
class Base {
public:
    Base(int a, int b) {
        cout << "Base class constructor, a = " << a << ", b = " << b << endl;
    }
};
class Derived : public Base {
public:
    Derived(int x, int y, int z) : Base(x, y) {
        cout << "Derived class constructor, z = " << z << endl;
    }
};
void main() {
    Derived obj(3, 4, 7);
    getch();
}
```

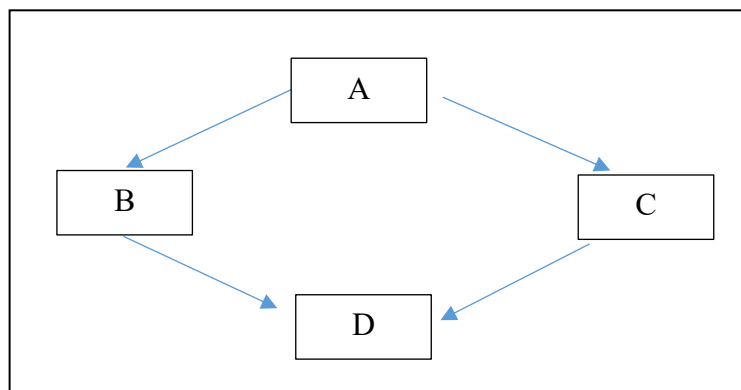
OUTPUT

Base class constructor, a = 3, b = 4

Derived class constructor, z = 7

11.6 GRANTING ACCESS. VIRTUAL BASE CLASSES

1. A virtual base class is a base class that is declared with the keyword "virtual" in a derived class's inheritance list.
2. Virtual base classes are used to solve the Diamond Problem of multiple inheritance. This ensures that only single copy of the base class member is inherited.



NOTE

Diamond Problem: When a derived class inherits from two classes that are inherits from the same base class then there can be ambiguity about which base class's members are being referred to. This is known as the diamond problem.

Example 1: Virtual Base Class to solve Diamond Problem

```
#include<iostream.h>
#include<conio.h>
class A {
public:
    int a;
};
class B : virtual public A {
};
class C : virtual public A {
};
class D : public B, public C {
public:
    void display() {
        cout << "Value of a: " << a << endl;
    }
};

void main() {
    D obj;
    obj.a = 10;
    obj.display();
    getch(); }
```

OUTPUT

Value of a: 10

Explanation: Class A contains an integer member 'a'. Class B and C both inherit from A. Here Class A is declared virtual base class in both classes B and C, so that there will be only one instance of A inherited when B and C are inherited by D. Class D has a display () method that accesses the member 'a' from A. Main function creates an object of D, assigns a value 10 to a, and calls display () to print it.

**Activity 11.1**

Activity Name: Solving Inheritance Ambiguity

Objective: To manage complex inheritance structures.

Required Materials: Computer, compiler

Prerequisites: Single inheritance

Steps:

1. Implement multiple inheritance.

NOTE

2. Introduce virtual base classes.

3. Observe constructor order.

Post-Activity Reflection: What challenges arise in multiple inheritance?

Expected Outcome: Learners resolve inheritance complexities.

☑ Check Your Progress 11.1**A. Multiple Choice Question (MCQ)**

1. Which of the following ensures that only one copy of a base class exists when multiple inheritance is used?

- a) Protected access
- b) Virtual base class
- c) Constructor chaining
- d) Destructor overriding

B. True / False Question (T/F)

1. In multiple inheritance, ambiguity occurs when two base classes have a common member name.

C. Short Answer Questions

1. Explain the difference between public, private, and protected inheritance.

.....

2. How does a derived class call a parameterized constructor of its base class?

.....

11.4 SUMMARY

Inheritance in object-oriented programming allows a class to derive properties and behaviors from another class. Advanced inheritance concepts extend this idea to handle complex scenarios, ensuring code reusability and maintainability. **Multiple inheritance** enables a derived class to inherit from more than one base class, combining their functionalities. However, it can lead to ambiguity when multiple base classes have members with the same name, which is resolved using **scope resolution operators** or **virtual inheritance**. **Constructors and destructors** play a critical role in inheritance. A derived class constructor can invoke base class constructors to ensure proper initialization of inherited attributes, while destructors are called in reverse order to clean up resources efficiently. Passing parameters to base class

NOTE

constructors allows customization of inherited members during object creation, often using **initializer lists**. **Granting access** through access specifiers (public, protected, private) determines which base class members can be used by derived classes and external code. Public members are accessible everywhere, protected members are accessible within derived classes, and private members are restricted to the base class itself. To handle ambiguities in multiple inheritance and avoid duplication of base class members, **virtual base classes** are used. Virtual inheritance ensures that only one copy of a shared base class exists, preventing the “diamond problem.” Understanding these mechanisms is crucial for designing robust class hierarchies. Advanced inheritance improves code modularity, allows method overriding, and ensures proper resource management across complex applications. By mastering constructors, destructors, parameter passing, access control, and virtual inheritance, developers can leverage the full potential of object-oriented design while maintaining clarity, efficiency, and maintainability in their programs.

11.5 GLOSSARY

- **Inheritance** – Mechanism by which a class derives properties and methods from another class.
- **Derived Class** – The class that inherits from one or more base classes.
- **Base Class** – The class whose properties and methods are inherited.
- **Multiple Inheritance** – Inheriting from more than one base class.
- **Constructor** – Special method for initializing objects.
- **Destructor** – Special method for cleaning up resources before object destruction.
- **Initializer List** – Syntax used to pass parameters to base class constructors.
- **Access Specifier** – Keywords (public, protected, private) controlling member accessibility.
- **Virtual Base Class** – Base class that prevents multiple copies in multiple inheritance.
- **Scope Resolution Operator (::)** – Operator used to resolve ambiguity in multiple inheritance.
- **Method Overriding** – Redefining a base class method in a derived class.
- **Diamond Problem** – Ambiguity occurring in multiple inheritance when two base classes share a common ancestor.

11.6 ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 11.1

A. MCQ Answer:

1. b) Virtual base class

Explanation: Virtual base classes prevent duplication of a base class in multiple inheritance, avoiding the diamond problem.

NOTE**B. True/False Answer:****1. Answer: True**

Explanation: Ambiguity arises in multiple inheritance if two base classes have members with the same name, which can be resolved using the scope resolution operator or virtual inheritance.

C. Short Answer Questions Answers:

- 1. Public inheritance** – Public and protected members of the base class retain their access in the derived class.

Protected inheritance – Public and protected members of the base class become protected in the derived class.

Private inheritance – Public and protected members of the base class become private in the derived class.

- 2. Calling parameterized base class constructor** – A derived class can call a parameterized base class constructor using an **initializer list** in its constructor. Example:

```
class Base {
    int x;
public:
    Base(int a) { x = a; }
};

class Derived : public Base {
    int y;
public:
    Derived(int a, int b) : Base(a) { y = b; }
};
```

11.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/UnitUrls?unitId=1476	

11.8 REFERENCE

- **Inside the C++ Object Model** - Stanley B. Lippman - 978-0201834542 - 1996
- **Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions** - Herb Sutter - 978-0201615629 - 1999
- **More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions** - Herb Sutter - 978-0201704345 - 2002

- **C++ FAQs** - Marshall Cline, Greg Lomow, Mike Girou - 978-0201309838 - 1998
- **C++ Primer** - Stanley B. Lippman, Josée Lajoie, Barbara E. Moo - 978-0321714114 - 2012 (Fifth Edition)

11.9 EXERCISE

Part – 1

A. Multiple Choice Questions

- Which access specifier allows derived classes to access base class members but restricts access from outside code?**
 - Public
 - Private
 - Protected
 - Default
- In multiple inheritance, which problem occurs when two base classes have a common ancestor?**
 - Ambiguity Problem
 - Diamond Problem
 - Scope Resolution Problem
 - Constructor Problem
- How can a derived class call a parameterized constructor of its base class?**
 - Using a destructor
 - Using an initializer list
 - Using virtual functions
 - Using operator overloading
- What is the role of a virtual base class?**
 - To allow multiple copies of base class
 - To prevent multiple copies of base class
 - To override base class functions
 - To call private members of base class
- Which of the following is true about destructors in inheritance?**
 - Only base class destructors are called
 - Derived class destructors are called before base class destructors
 - Base class destructors are called before derived class destructors
 - Destructors are not called automatically

Answer with Explanations

1. c) Protected

Explanation- Protected members can be accessed within derived classes but not by external code.

2. b) Diamond Problem

Explanation- Occurs in multiple inheritance when two base classes share a common ancestor.

NOTE

3. b) Using an initializer list

Explanation- Parameterized constructors of the base class can be invoked via initializer lists in the derived class.

4. b) To prevent multiple copies of base class

Explanation- Virtual base classes avoid duplication in the inheritance hierarchy.

5. b) Derived class destructors are called before base class destructors

Explanation- Destructors execute in reverse order of construction to safely free resources

B. Short Answer Type Questions

1. Explain multiple inheritance with an example.
2. What is a virtual base class, and why is it used?
3. How are base class constructors called from a derived class?
4. Define access specifiers and their role in inheritance.
5. Describe the order of execution of constructors and destructors in inheritance.

Part – 2

C. Long Answer Type Questions (200 words)

1. Explain the concept of passing parameters to base class constructors and its significance.
2. Discuss the advantages and disadvantages of multiple inheritance.
3. Describe the use of access specifiers in inheritance with examples.
4. Explain constructors and destructors in inheritance with proper sequencing.
5. Discuss how virtual base classes prevent ambiguity in multiple inheritance.

Part – 3

D. Long Answer Type Questions (300 words)

1. Analyze the challenges of multiple inheritance and how virtual base classes solve them.
2. Explain with examples how derived class constructors can initialize base class members.
3. Describe in detail the lifecycle of objects in inheritance, including constructor and destructor calls.
4. Discuss the role of access control in designing robust inheritance hierarchies.
5. Evaluate real-world scenarios where advanced inheritance concepts improve code reusability and maintainability.

UNIT 12

VIRTUAL FUNCTIONS AND DYNAMIC BINDING

NOTE

UNIT STRUCTURE

- 12.1. Introduction
- 12.2. Objective
- 12.3. Virtual Function
- 12.4. Pure Virtual Functions
- 12.5. Early Vs. Late Binding
- 12.6. Summary
- 12.7. Glossary
- 12.8. Answers to Check Your Progress
- 12.9. Suggested readings - OER
- 12.10. Reference
- 12.11. Exercise

12.1 INTRODUCTION

This unit builds on advanced inheritance concepts to introduce runtime polymorphism in C++. Learners already understand compile-time polymorphism and class hierarchies, and are now ready to study how function calls are resolved at runtime. This unit introduces virtual functions as a mechanism to ensure that the correct function implementation is executed based on the type of object pointed to, rather than the type of pointer used.

The unit explains the concept of dynamic binding and contrasts it with early binding. Learners study how pure virtual functions are used to create abstract base classes and define interfaces. These concepts are essential for designing flexible and extensible systems. The role of base class pointers and derived class objects in achieving runtime polymorphism is demonstrated through examples.

By mastering virtual functions, learners complete the core object-oriented programming concepts in C++. This understanding prepares them to shift focus from program design to interaction with users. The next unit introduces the C++ input-output system and stream handling fundamentals

12.2 OBJECTIVES

After completing this unit, learners will be able to:

1. Implement virtual functions to achieve runtime polymorphism.
2. Design abstract base classes using pure virtual functions.
3. Differentiate between early and late binding mechanisms.

NOTE

4. Apply base class pointers for dynamic function calls.
5. Evaluate program behavior using dynamic binding principles

12.3 VIRTUAL FUNCTION

1. A virtual function is a function in a base class that is implemented to be overridden in derived classes.
2. The virtual keyword allows for runtime polymorphism and allowing you to call derived class methods through base class pointers or references.
3. When a virtual function is called using base class pointer then the program determines which function to execute at runtime based on the type of object not on the type of pointer.
4. Virtual function is declared by using the virtual keyword in the base class.
5. Derived classes can override the virtual function to provide specific implementations.

Example: Virtual Function

```
#include<iostream.h>
#include<conio.h>
class Base {
public:
    virtual void show() {    //virtual function
        cout << "Base class show function\n";
    }
};
class Derived : public Base {
public:
    void show(){    // function overriding
        cout << "Derived class show function\n";
    }
};
void main() {
    Base* basePtr; //base class pointer
    Derived d;

    basePtr = &d;    //baseptr pointing to derived class object
    basePtr->show(); // Calls Derived class show function
    getch();
}
```

OUTPUT

Derived class show function

12.4 PURE VIRTUAL FUNCTIONS

NOTE

1. A virtual function without definition is called as pure virtual function. It must be implemented in derived classes.
2. It is declared using `= 0`.
3. A class which contains at least one pure virtual function becomes an abstract class and we can't able to create objects of that class. Therefore, derived classes must provide implementation to all pure virtual functions to create objects.
4. If a derived class doesn't implement all inherited pure virtual functions of base class then it will also become an abstract class.

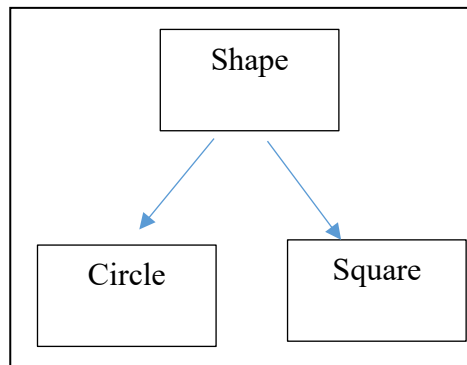


Fig. No. 4.2. Abstract class(Shape)

Example: Pure Virtual Function

```

#include<iostream.h>
#include<conio.h>
class Shape {
public:
    virtual void area() = 0; // Pure virtual function
};

class Circle : public Shape {
public:
    void area(){ //function overriding
        cout << "Area of Circle\n";
    }
};

class Square : public Shape {
public:

```

NOTE

```

void area() {           //function overriding
    cout << "Area of Square\n";
}
};

void main() {
    Circle c;
    Square s;
    c.area();
    s.area();
    getch();
}

```

OUTPUT

```

Area of Circle
Area of Square

```

12.5 EARLY VS. LATE BINDING

1. In C++, **binding** means the process of associating a function call with its definition.
2. Binding can occur at different times during program execution, leading to two types of binding:
 - a) **Early Binding (Static Binding):** The call to a function is resolved at compile time. In this method, the compiler determines what function to call based on the type of the pointer or reference.
 - b) **Late Binding (Dynamic Binding):** The function call gets resolved at runtime, thanks to virtual functions. Here, instead of the type of the pointer or reference deciding what function to call, it actually relies on the actual object type a pointer or reference points to.

Example 1: Early Binding (without Virtual Functions)

```

#include<iostream.h>

#include<conio.h>

class Base {
public:
    void show() {
        cout << "Base class show function\n";
    }
}

```

NOTE

```

};

class Derived : public Base {
public:
    void show() {
        cout << "Derived class show function\n";
    }
};

```

```

void main() {
    Base* basePtr;
    Derived d;
    basePtr = &d;
    basePtr->show(); // Calls Base class show() (Early Binding) as basePtr
pointer
                    //is of Base class.

    getch();
}

```

OUTPUT

Base class show function

Example 2: Late Binding (with Virtual Functions)

```

#include<iostream.h>
#include<conio.h>
class Base {
public:
    virtual void show() {    //virtual function
        cout << "Base class show function\n";
    }
};

class Derived : public Base {
public:
    void show() {            //function overriding
        cout << "Derived class show function\n";
    }
};

```

NOTE

```
void main() {
    Base* basePtr;
    Derived d;
    basePtr = &d; //basePtr of Base class points to object of Derived class
    basePtr->show(); // Calls Derived class show() (Late Binding)
    getch();
}
```

OUTPUT

Derived class show function



Activity 12.1

Activity Name: Runtime Decision Making

Objective: To demonstrate runtime polymorphism.

Required Materials: Computer, compiler

Prerequisites: Inheritance and polymorphism

Steps:

1. Create base and derived classes.
2. Use virtual functions.
3. Call methods using base pointers.

Post-Activity Reflection: How does dynamic binding enhance flexibility?

Expected Outcome: Learners implement runtime polymorphic behavior.

☑ Check Your Progress 12.1

A. Multiple Choice Question (MCQ)

1. Which of the following statements about virtual functions is TRUE?
 - a) Virtual functions are resolved at compile time.
 - b) A derived class cannot override a virtual function.
 - c) Virtual functions support runtime polymorphism.
 - d) Virtual functions cannot be declared in an abstract class.

B. True/False Question :

1. A pure virtual function can have a body (implementation) in the base class.

C. Short Answer Questions :

1. Define late binding and explain its importance in object-oriented programming.

.....

.....

.....

NOTE

2. What is the purpose of an abstract class, and why can't it be instantiated?

.....

12.4 SUMMARY

In object-oriented programming, **virtual functions** are member functions in a base class that can be redefined in derived classes to achieve **runtime polymorphism**. Unlike normal functions, virtual functions enable the program to decide at runtime which function to invoke, depending on the type of the object being pointed to, rather than the type of the pointer. This mechanism is central to achieving **dynamic binding**, allowing flexibility and extensibility in complex programs. A virtual function is declared using the virtual keyword in the base class, and it can be overridden in derived classes.

Pure virtual functions are a specialized form of virtual functions that have no definition in the base class and are declared by assigning 0 in their declaration. They serve as a blueprint for derived classes, compelling them to provide their own implementation. Classes containing at least one pure virtual function are called **abstract classes**, and they cannot be instantiated directly. Pure virtual functions are crucial in designing interfaces and enforcing consistent behavior across derived classes.

The concepts of **early binding** and **late binding** are closely related to virtual functions. **Early binding** occurs when the function to be called is determined at compile time, typically for non-virtual functions. In contrast, **late binding** (or dynamic binding) defers this decision to runtime, which is essential when dealing with pointers or references to base classes pointing to derived objects. Late binding enables polymorphic behavior, allowing a single interface to handle multiple types of objects seamlessly. Understanding these concepts helps developers write flexible, maintainable, and reusable code, which is a hallmark of effective object-oriented programming.

12.5 GLOSSARY

- **Virtual Function** – A base class function that can be overridden in derived classes to enable runtime polymorphism.
- **Dynamic Binding** – The process of resolving a function call at runtime rather than compile time.
- **Pure Virtual Function** – A virtual function with no definition in the base class, declared with =0.
- **Abstract Class** – A class containing at least one pure virtual function; cannot be instantiated.
- **Polymorphism** – The ability of different objects to respond to the same function call in their own way.
- **Early Binding** – Function call resolution at compile time.

NOTE

- **Late Binding** – Function call resolution at runtime, usually via virtual functions.
- **Base Class** – The class from which other classes derive properties and behavior.
- **Derived Class** – A class that inherits from a base class and may override virtual functions.
- **Override** – Providing a new definition for a base class's virtual function in a derived class.
- **Interface** – A class with pure virtual functions that defines behavior for derived classes.
- **Runtime Polymorphism** – Achieving polymorphism during program execution using virtual functions.

12.6 ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 12.1

A. MCQ Answer:

1. c) Virtual functions support runtime polymorphism.

Explanation: Virtual functions allow the program to determine at runtime which function to call based on the actual object type, enabling runtime polymorphism.

B. True/False Answer:

1. True

Explanation: A pure virtual function can have a body in the base class, but derived classes are still required to override it.

C. Short Answer Guidelines:

1. Late Binding: Late binding (dynamic binding) occurs when the function call is resolved at runtime, rather than compile time. It is crucial for polymorphism, allowing the same base class pointer to call derived class functions depending on the actual object type.

2. Abstract Class: An abstract class contains at least one pure virtual function. It cannot be instantiated directly and is used to define a common interface for derived classes, enforcing implementation of certain functions in all derived classes.

12.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/UnitUrls?unitId=1477	

12.8 REFERENCE

- **The C++ Programming Language** - Bjarne Stroustrup - ISBN 978-0-321-56384-2 (For 4th Edition, updated for C++11) - 2013
- **C++ Primer** - Stanley B. Lippman, Josee Lajoie, and Barbara E. Moo - ISBN 978-0-321-71411-4 (For 5th Edition) - 2012
- **Effective C++: 55 Specific Ways to Improve Your Programs and Designs** - Scott Meyers - ISBN 978-0-321-33487-9 (For 3rd Edition) - 2005
- **Thinking in C++, Volume 1: Introduction to Standard C++** - Bruce Eckel - ISBN 0-13-979973-6 (For 2nd Edition) - 2000
- **Object-Oriented Programming with C++** - E. Balagurusamy - ISBN 978-1259029936 (For 6th Edition) - 2013

12.9 EXERCISE

Part – 1

A. Multiple Choice Questions

1. Which keyword is used to declare a virtual function in C++?
 - a) override
 - b) virtual
 - c) abstract
 - d) pure
2. What happens when a base class pointer points to a derived class object and calls a virtual function?
 - a) Base class function is always called
 - b) Derived class function is called at runtime
 - c) Compile-time error occurs
 - d) None of the above
3. Which type of class cannot be instantiated?
 - a) Concrete class
 - b) Abstract class
 - c) Static class
 - d) Derived class
4. A pure virtual function is declared using:
 - a) =0
 - b) virtual only
 - c) override
 - d) abstract

NOTE

NOTE

5. Late binding refers to:

- a) Function call resolved at compile time
- b) Function call resolved at runtime
- c) Function call resolved by compiler errors
- d) Function call resolved using friend function

Answer with Explanations

1 b) virtual

Explanation: The virtual keyword allows a function to be dynamically bound at runtime, enabling polymorphism.

2 b) Derived class function is called at runtime

Explanation: Virtual functions enable runtime resolution; the derived class version overrides the base class function.

3 b) Abstract class

Explanation: Abstract classes have at least one pure virtual function and cannot be instantiated directly.

4 a) =0

Explanation: A pure virtual function is declared in the base class using =0, forcing derived classes to implement it.

5 b) Function call resolved at runtime

Explanation: Late binding (dynamic binding) allows the program to decide which function to call during execution.

B. Short Answer Type Questions

1. Define virtual function and explain its significance in C++.
2. What is a pure virtual function? Give an example.
3. Explain runtime polymorphism with an example.
4. Differentiate between early binding and late binding.
5. What is an abstract class? Why is it used in object-oriented programming?

Part – 2

C. Long Answer Type Questions (200 words)

1. Explain how virtual functions enable runtime polymorphism in C++. Include a code example.
2. Describe pure virtual functions and abstract classes. How do they help in designing interfaces?
3. Discuss the differences between early binding and late binding with examples.

4. Explain the rules and process of overriding a virtual function in derived classes.
5. Illustrate with an example how dynamic binding allows the same code to work with multiple object types.

NOTE**Part – 3****D. Long Answer Type Questions (300 words)**

1. Explain the concept of **virtual functions** in C++ and discuss how they enable **runtime polymorphism**. Illustrate with a detailed example.
2. What are **pure virtual functions** and **abstract classes**? Discuss their role in designing object-oriented programs and creating reusable interfaces.
3. Compare and contrast **early binding** and **late binding**. Explain the advantages and disadvantages of both with suitable examples.
4. Describe the process of **overriding virtual functions** in derived classes. Include rules, best practices, and an example scenario demonstrating dynamic binding.
5. Explain how **dynamic binding** allows a single base class pointer or reference to work with multiple derived class objects. Discuss its significance in real-world object-oriented programming designs.



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN AICTE GROUP UNIVERSITY

Approved by: PCI | AICTE | NCTE | BCI | Member of: ANU | Joint Committee: JUGC | BCI | AICTE | Recognized by: UGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:

BLOCK 5	C++ I/O AND STREAM MANIPULATION	201-236
Unit 13	C++ I/O System Basics	203
Unit 14	Formatting Output	213
Unit 15	Custom Manipulators	227



Input and output operations are critical for interacting with users and external systems in C++ programs. The C++ I/O system provides a robust framework for managing streams of data through predefined classes, functions, and manipulators. This block focuses on understanding the C++ I/O model, including standard streams, stream classes, and formatting capabilities. Students learn how to use input/output operators effectively and how to format output using manipulators and ios member functions.

Advanced techniques, such as setting precision, width, and fill characters, are discussed to create user-friendly and well-formatted output. Additionally, the block introduces the creation of custom manipulators to enhance flexibility in formatting and streamline repetitive tasks. By mastering these techniques, learners can produce professional-quality output, manage file and console input efficiently, and gain better control over program behavior. The knowledge of streams and manipulators is essential not only for basic I/O operations but also for handling more advanced C++ programming tasks involving file handling, reporting, and structured data display.

UNIT 13

C++ I/O SYSTEM BASICS

NOTE

UNIT STRUCTURE

- 13.1. Introduction
- 13.2. Objective
- 13.3. The C++ I/O System Basics
- 13.4. Predefined Streams in C++
- 13.5. Summary
- 13.6. Glossary
- 13.7. Answers to Check Your Progress
- 13.8. Suggested readings - OER
- 13.9. Reference
- 13.10. Exercise

13.1 INTRODUCTION

Unit 13 builds upon the learner's comprehensive understanding of C++ programming and object-oriented concepts developed in the earlier units. By this stage, learners are familiar with classes, objects, constructors, destructors, inheritance, polymorphism, and dynamic binding. They have learned how to design structured and reusable programs, but real-world applications require effective interaction between programs and users or external systems. This interaction is achieved through input and output operations, making knowledge of the C++ I/O system essential.

This unit introduces the fundamentals of the C++ input-output system and explains how data flows between programs and external sources such as the keyboard, screen, and files. Learners study the concept of streams as sequences of bytes and understand how C++ uses stream classes to manage input and output operations. The unit covers the basic architecture of the C++ I/O system, emphasizing the role of stream objects and buffer management. Predefined streams such as standard input, standard output, and standard error are discussed to demonstrate how C++ handles console-based interaction. Learners also explore the use of input and output operators for reading and writing data of various types. The importance of stream states and basic error handling is introduced to ensure reliable program execution.

Through this unit, learners gain the ability to write programs that accept user input and display meaningful output in a controlled manner. The unit emphasizes practical understanding rather than formatting details, ensuring that learners first master core stream operations. This foundational knowledge is necessary before learning advanced formatting techniques.

By the end of this unit, learners are able to use the C++ I/O system confidently to support interactive and user-friendly applications. This understanding prepares learners to control the appearance and organization of output using formatting tools.

NOTE

13.2 OBJECTIVES

After completing this unit, learners will be able to:

- Apply C++ stream concepts for input and output operations.
- Use predefined streams for console-based interaction.
- Implement input-output operations using stream operators.
- Examine stream states and error handling mechanisms.
- Develop interactive C++ programs using stream-based I/O

13.3 THE C++ I/O SYSTEM BASICS

The C++ i/o system is built upon streams, which handle input and output operations. Streams allow data to flow between the program and various I/O devices, such as the console, files, or other programs.

Example : Basic Input and Output

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

void main() {
    int age;

    cout << "Enter your age: ";

    cin >> age;

    cout << "You are " << age << " years old." << endl;

    getch();
}
```

OUTPUT

Enter your age: 18

You are 18 years old.

13.4 PREDEFINED STREAMS IN C++

C++ includes predefined streams to handle input(getting data from the user or a file) and output(showing data on the screen or writing to a file).The most commonly used streams are:

- cin: For reading input from the user (keyboard).
- cout: For printing output to the screen.
- cerr: For printing error messages.
- clog: For printing log messages (buffered).

Example: C++ program Using Predefined streams

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

NOTE

```

int main() {
    int num; // Variable to store user input

    // Log message: Program is running
    clog << "Program started. Please enter a number.\n";

    // Get input from the user using cin
    cout << "Enter a number: "; // Prompt user

    cin >> num; // Read the number

    // Check if the input failed (e.g., user entered something that's not a
    number)
    if (cin.fail()) {
        cerr << "Error: Invalid input. Please enter a valid number.\n"; //
        Print error
        //message

        return 1; // Exit the program with an error status
    }

    // Log message: Successfully received input
    clog << "Successfully received input: " << num << "\n";

    // Perform a calculation (let's square the number)
    int result = num * num;

    // Output the result using cout
    cout << "The square of " << num << " is: " << result << "\n";

    // Log message: Program completed
    clog << "Program completed successfully.\n";

    return 0; // Indicate successful completion
}

```

Explanation:

1. **clog:** It is used to print log messages and to keep track of the program's progress. In this program, we print messages when the program starts, after the input is received, and when the program completes.

NOTE

2. **cout:** It is used to print prompts and results to the user. The program asks the user to input a number and then prints the result (the square of the number).
3. **cin:** It is used to read input from the user. We use it to read an integer value.
4. **cerr:** It is used for printing error messages. If the user enters something that is not a valid number, the program will print an error message and exit.

OUTPUT

Case 1: Normal Input (Valid Number):

Program started. Please enter a number.

Enter a number: 5

Successfully received input: 5

The square of 5 is: 25

Program completed successfully.

Case2: Error Input (Invalid Number):

Program started. Please enter a number.

Enter a number: abc

Error: Invalid input. Please enter a valid numb



Activity 13.1

Activity Name: Interactive Program Design

Objective: To implement input-output operations.

Required Materials: Computer, compiler

Prerequisites: Basic C++ syntax

Steps:

1. Read user input using streams.
2. Display processed output.
3. Handle invalid input.

Post-Activity Reflection: Why are streams preferred over traditional I/O?

Expected Outcome: Learners write interactive C++ programs.

NOTE**☑ Check Your Progress 13.1****A. Multiple Choice Question (MCQ):**

1. Which manipulator is used to control the number of decimal places displayed for floating-point numbers?

- a) setw
- b) setprecision
- c) endl
- d) flush

B. True/False (T/F):

1. The clog stream in C++ is unbuffered and is used to display error messages immediately.

C. Short Answer Questions:

1. What is the difference between cin and cout in C++?

.....

2. Define a stream in C++ and explain its purpose.

.....

13.4 SUMMARY

The **C++ I/O system** is a fundamental part of the language, enabling communication between a program and external devices such as the keyboard, screen, or files. Input and output in C++ are handled through a rich set of library functions and classes provided by the **iostream** library. The core concepts of the C++ I/O system revolve around **streams**, which represent a continuous flow of data either into a program (**input**) or out of a program (**output**).

C++ provides **predefined streams** for basic input and output operations. The standard input stream cin is used to read data from the keyboard, while the standard output stream cout sends data to the console. For error messages, cerr is used, which is an unbuffered stream ensuring immediate output, and clog provides a buffered stream primarily used for logging purposes. These streams are objects of classes derived from the **ios** base class, such as istream for input and ostream for output.

C++ I/O also supports manipulators and formatting options that help control how data is displayed, including **width**, **precision**, **fill characters**, and

NOTE

alignment. Understanding these basic I/O concepts is crucial for efficient interaction with the user and external files. Mastery of predefined streams, along with stream manipulators and error handling mechanisms, forms the foundation for advanced file handling, formatted I/O, and robust program design in C++.

13.5 GLOSSARY

- **Stream** – A sequence of characters flowing into or out of a program.
- **cin** – Standard input stream for reading data from the keyboard.
- **cout** – Standard output stream for displaying data on the console.
- **cerr** – Standard error stream for displaying error messages (unbuffered).
- **clog** – Standard log stream for messages (buffered).
- **iostream** – C++ header file containing input/output stream definitions.
- **istream** – Base class for input streams.
- **ostream** – Base class for output streams.
- **Manipulator** – Special function to format stream output (e.g., endl, setw).
- **Buffering** – Temporary storage of data in memory before processing or output.
- **Precision** – Number of digits displayed for floating-point numbers.
- **Width** – Number of character spaces allocated for output.
- **Formatted I/O** – Displaying input or output in a controlled, readable format.
- **Unbuffered Stream** – Stream that writes data immediately without storing.
- **Base Class** – Parent class from which other classes inherit properties.

13.6 ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 13.1**A. MCQ Answer:****1. b) setprecision**

Explanation: This manipulator sets the number of decimal places to display for floating-point numbers.

B. True/False Answer:**1. False**

Explanation: Clog is a buffered stream, used for logging messages. cerr is the unbuffered stream used for immediate error messages.

NOTE**C. Short Answer Questions:****1. Difference between cin and cout:**

- cin is the standard input stream used to take input from the keyboard.
- cout is the standard output stream used to display output on the console.

2. Stream in C++:

- A stream is a flow of data to or from a program.
- It represents input (data coming into the program) or output (data going out of the program) and simplifies reading and writing operations from devices like the keyboard, console, or files.

13.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/User/UnitUrls?unitId=1478	

13.8 REFERENCE

- **C++ Primer (5th Edition)** - Stanley B. Lippman, Josée Lajoie, Barbara E. Moo - 978-0321714114 - 2012
- **C++ Primer Plus (6th Edition)** - Stephen Prata - 978-0321776402 - 2011
- **Programming: Principles and Practice Using C++ (2nd Edition)** - Bjarne Stroustrup - 978-0321992789 - 2014
- **The C++ Programming Language (4th Edition)** - Bjarne Stroustrup - 978-0321563842 - 2013
- **Accelerated C++: Practical Programming by Example** - Andrew Koenig, Barbara E. Moo - 978-0201703535 - 2000

13.9 EXERCISE**Part – 1****A. Multiple Choice Questions****1. Which header file is required for standard input/output operations in C++?**

- a) <stdio.h>
- b) <conio.h>

NOTE

c) <iostream>

d) <fstream>

2. Which C++ stream is used for unbuffered error output?

a) cout

b) cin

c) cerr

d) clog

3. Which manipulator is used to insert a newline and flush the output buffer?

a) setw

b) endl

c) setprecision

d) flush

4. The standard input stream object in C++ is:

a) cout

b) cin

c) cerr

d) clog

5. Which stream is primarily used for logging purposes?

a) cin

b) cout

c) cerr

d) clog

Answer with Explanations

1. c) <iostream>

Explanation- It provides definitions for cin, cout, cerr, and clog.

2. c) cerr

Explanation- It is an unbuffered stream used to immediately display error messages.

3. b) endl

Explanation- Adds a newline and flushes the output buffer.

4. b) cin

Explanation- Standard input stream for reading data from the keyboard.

5. d) clog

Explanation- Buffered stream used for logging messages during program execution.

B. Short Answer Type Questions

1. Define a stream in C++ and explain its types.

2. What is the difference between cerr and clog streams?

NOTE

3. How does cin handle input from the keyboard?
4. Explain the role of manipulators in C++ I/O.
5. What is the importance of buffering in output streams?

Part – 2**C. Long Answer Type Questions (200 words)**

1. Explain the predefined C++ streams (cin, cout, cerr, and clog) with suitable examples.
2. Describe the formatting capabilities in C++ I/O using manipulators like setw, setprecision, and endl.
3. Explain the concept of input buffering and output buffering in C++ with examples.
4. Write a short program to demonstrate reading multiple inputs using cin and displaying them with cout.
5. Discuss the differences between formatted and unformatted I/O in C++ with examples.

Part – 3**D. Long Answer Type Questions (300 words)**

1. Discuss the C++ I/O system in detail, covering streams, their classes (istream, ostream), and the role of base class ios.
2. Explain the differences between cin, cout, cerr, and clog streams and their practical applications.
3. Describe manipulators in C++ and how they help in formatting output with examples of setw, setfill, and setprecision.
4. Explain how C++ handles errors in input/output streams, including fail state and good state checking.
5. Write a C++ program to demonstrate reading from standard input and writing to both cout and cerr streams with proper formatting.

UNIT 14

FORMATTING OUTPUT USING STREAMS AND MANIPULATORS

NOTE

UNIT STRUCTURE

- 14.1. Introduction
- 14.2. Objective
- 14.3. Formatting using the ios members
- 14.4. Clearing Format Flags
- 14.5. An Overloaded form of setf()
- 14.6. Examining the Formatted Flags
- 14.7. Using width(), precision(), fill()
- 14.8. Using Manipulators to format I/O
- 14.9. Summary
- 14.10. Glossary
- 14.11. Answers to Check Your Progress
- 14.12. Suggested readings – OER
- 14.13. Reference
- 14.14. Exercise

14.1 INTRODUCTION

Unit 14 builds directly on the learner's understanding of the basic C++ input-output system acquired in the previous unit. Learners are already familiar with stream concepts, predefined streams, and the use of input and output operators for basic interaction. While simple input and output operations are sufficient for small programs, real-world applications often require well-organized, readable, and precisely formatted output. This unit addresses that requirement by introducing formatting techniques available in the C++ I/O system. The primary focus of this unit is controlling the appearance of output using ios class members and predefined manipulators. Learners study format flags and understand how they influence numerical representation, alignment, and data presentation. Techniques for setting and clearing format flags are explained to demonstrate how formatting states can be changed dynamically during program execution. The unit also introduces commonly used formatting functions such as width(), precision(), and fill(), enabling learners to control field width, numerical accuracy, and filler characters.

Predefined manipulators are discussed as convenient tools for formatting output without directly modifying format flags. Learners understand how manipulators simplify stream formatting and improve code readability. Emphasis is placed on producing consistent and professional output, which is especially important in reports, tables, and user interfaces. Through examples, learners see how formatted output enhances user experience and improves program clarity.

NOTE

By the end of this unit, learners gain the ability to present data in a structured and visually appealing manner using standard C++ formatting techniques. However, predefined manipulators may not always meet specific formatting requirements. To address this limitation, programmers can define their own formatting behavior.

14.2 OBJECTIVES

After completing this unit, learners will be able to:

- Apply format flags to control output presentation.
- Implement width, precision, and fill formatting techniques.
- Use predefined manipulators to format output efficiently.
- Analyze formatting states during stream operations.
- Design well-structured and readable program outputs

14.3 FORMATTING USING THE IOS MEMBERS

When you print data, you can **format** it to make it look nice and different. The `ios` class provides some settings to control how the output appears.

In C++, formatting of streams is controlled using flags. Flags can be set, cleared, or toggled to control how the data is formatted when it is output.

Common Format Flags:

- **`ios::left`**: Aligns output to the left.
- **`ios::right`**: Aligns output to the right (default for numeric types).
- **`ios::dec`**: Formats integers as decimal.
- **`ios::hex`**: Formats integers as hexadecimal.
- **`ios::oct`**: Formats integers as octal.
- **`ios::scientific`**: Formats floating-point numbers in scientific notation.
- **`ios::fixed`**: Formats floating-point numbers in fixed-point notation.

Example: Formatting Numbers

```
#include <iostream.h>
#include <iomanip> // includes manipulators such as setw(), fixed, scientific etc.
#include <conio.h>
void main() {
    double num = 123.456;
    cout << "Fixed: " << fixed << num << endl;
    cout << "Scientific: " << scientific << num << endl;
    cout << "Left aligned: " << left << setw(10) << num << endl;
    getch();
}
```

OUTPUT

```
Fixed: 123.456000
Scientific: 1.234560e+02
Left aligned: 123.456
```

Explanation:

The fixed manipulator ensures that `num(123.456)` is printed with 6 decimal places, regardless of its actual precision.

NOTE

The scientific manipulator converts the number(123.456) to scientific notation(1.234560e+02.

The setw(10) sets the field width to 10 characters, and **left** ensures that the number is printed starting from the left side of the field. Since num (123.456) has 7 characters, it will be padded with 3 spaces on the right to make the total width 10 characters.)

Example 2 : Conversion of decimal to hexadecimal and octal format

```
#include<iostream.h>
#include<iomanip>
int main() {
    int num = 255;
    cout << hex << num << endl; // Output in hexadecimal format
    cout << oct << num << endl; // Output in octal format
    return 0;
}
```

OUTPUT

```
ff
377
```

Explanation:

Here, **hex** converts the number(255) to hexadecimal format (lowercase letters by default) that is ff and **oct** converts the number(255) to octal format that is 377.

14.4 CLEARING FORMAT FLAGS

Clearing format flags is useful when you want to undo specific formatting changes and return to the default formatting behavior.

To reset formatting flags, we use **unsetf(flag)** -This clears specific format settings on a stream (e.g., ios::fixed).

Example: Clearing Format Flags

```
#include <iostream.h>
#include <iomanip>
#include<conio.h>
void main() {
    cout << fixed << 123.456 << endl;
    cout.unsetf(ios::fixed); // Clear the fixed format
    cout << 123.456 << endl;
    getch();
}
```

OUTPUT

```
123.456000
123.456
```

NOTE**Explanation:**

First, **fixed** was applied to the output, so 123.456 is printed as 123.456000 with **6 decimal places** by default (fixed-point format).

After calling **cout.unsetf(ios::fixed);**, the **fixed format** is cleared, so the number is printed in **normal floating-point format**. This typically prints 123.456 (without extra decimals), as it uses the default behavior of the output stream.

14.5 AN OVERLOADED FORM OF SETF()

In C++, **setf()** is a member function of the **std::ios** class (and its derived classes like **std::ostream**, **std::istream**, etc.), and it is used to set various flags that control how data is formatted when being output to streams like **std::cout**.

The **setf()** function sets formatting flags, allowing more control over multiple flags at once.

Example: Setting Flags Using setf()

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

void main () {
    // Set formatting flags
    cout.setf(ios::showpos); // Show positive sign for positive numbers
    cout.setf(ios::fixed, ios::floatfield); // Use fixed-point notation for floating-point
                                   // numbers

    // Output a floating-point number with the set formatting
    cout << 123.45 << endl;

    getch();
}
```

OUTPUT

```
+123.450000
```

Explanation:

Normally, positive numbers are displayed without a sign, but with **ios::showpos** flag, you'll see a + for positive values.(+123.45). Also you see output is 123.450000 because of the **ios::fixed** flag, which ensures that the output has a fixed number of digits after the decimal point. **ios::floatfield** flag is used to select the floating-point format(**ios::fixed** or **ios::scientific**).

14.6 EXAMINING THE FORMATTED FLAGS

In C++, you can retrieve and examine the currently active formatting flags set on an output stream (such as **std::cout**) using member function **flags()** of the **std::ios** class. This member function returns a **fmtflags** object representing the

current state of the format flags, so you can then use bitwise operations to determine whether some flags are set.

With the `setf()` function you can change format flags, and you could use the `flags()` function to restore or inspect flags after changing them.

Example: Checking Format Flags

```
#include <iostream>

#include <conio.h>

void main() {

    ios::fmtflags f = cout.flags (); // Get current flags

    cout.setf(ios::scientific); // // Set the format to scientific notation

    // check if the scientific notation flag is set

    if ((cout.flags() && ios::scientific) == ios::scientific) {

        cout << "Scientific notation is set" << endl;

    }

    cout.flags(f); // Restore original flags

    getch();

}
```

OUTPUT

Scientific notation is set

Explanation:

In this program, `flags()` function returns and saves the current flags of `std::cout` into the variable `f`. The `setf()` function sets the output format to scientific notation. In if statement we used a bitwise AND operation (`&`) to check if the scientific flag is set in the current format flags. If the result is equal to `ios::scientific`, it means the scientific flag is active. The program then prints "Scientific notation is set".

14.7 USING WIDTH(), PRECISION(), FILL()

In C++, `setw()`, `setfill()`, and `setprecision()` are formatting manipulators which may be used to control the width, fill characters, and precision of numbers and strings when outputting them using `std::cout`. These come under the `<iomanip>` library, which provides functions for input and output manipulation.

NOTE

NOTE**Example 1: Using width() and fill()**

```
#include <iostream.h>
#include<iomanip.h> // for setw(), setfill()
#include<conio.h>

void main() {
    // we are setting the width of the output field to 10 characters and filling with '*'
    cout << "Name: " << setw(10) << setfill('*') << "Amit" << endl;

    getch(); // Wait for user input
}
```

OUTPUT

Name: *****Amit|

Explanation:

Here, "Amit" is printed with a total width of 10 characters, so 6 * characters are added before "Amit" to fill the width.

Example 2: Using precision()

```
#include<iostream.h>
#include <iomanip.h> // for setprecision(), fixed
#include <conio.h> “
void main() {
    double num = 123.456789;
    // Set the output to fixed-point notation and set precision to 4 decimal places
    cout << fixed << setprecision(4) << num << endl;
    getch();
}
```

OUTPUT

123.4568

Explanation:

The number 123.456789 is printed with 4 digits after the decimal point (123.4568) i.e. the number is rounded to 4 decimal places.

14.8 USING MANIPULATORS TO FORMAT I/O

In C++, manipulators are special functions or objects that you can use to control the formatting of output when using std::cout (or other I/O streams).

Key I/O Manipulators

Let's discuss a few of the most common manipulators used for formatting output:

1. **setw(n)** – It sets the field width for output.

NOTE

2. **setfill(c)** – It sets the fill character for padding when the output is smaller than the specified width.
3. **setprecision(n)** – It sets the precision for floating-point numbers, i.e., the total number of digits after the decimal point.
4. **fixed** - Forces the output to be printed in fixed-point notation for floating-point numbers.
5. **scientific** – It forces the output to be in scientific notation for floating-point numbers.
6. **left** and **right** - Control the alignment of output, either left-aligned or right-aligned.
7. **endl** – It inserts a newline and flushes the output buffer.

Example: Using Manipulators

```
#include<iostream.h>
#include<iomanip.h> // included for manipulators like setw(), setfill() etc.
void main() {
    // 1. Using setw(), setfill(), and alignment manipulators
    cout << left << setw(15) << "Item" << right << setw(10) << "Price" << endl;
    cout << left << setw(15) << "Book" << right << setw(10) << "Rs100" << endl;
    cout << left << setw(15) << "Pen" << right << setw(10) << "Rs15" << endl;
    cout << endl; // Extra line break for separation
    // 2. Using setprecision() with fixed and scientific notation
    double num = 123.456789;
    cout << "Using setprecision(2) and fixed: " << fixed << setprecision(2) << num << endl;
    cout << "Using setprecision(4) and scientific: " << scientific << setprecision(4) << num << endl;
    // 3. Using setfill() to set custom fill character and setw()
    cout << "Custom fill character: " << setw(10) << setfill('*') << "Hello" << endl;
    cout << endl; // For clean separation
    // 4. Using endl to insert a newline
    cout << "This is a line." << endl;
    cout << "Another line after endl." << endl;
    getch();
}
```

OUTPUT

```
Item      Price
Book      RS100
Pen       Rs15
```

Using setprecision(2) and fixed: 123.46

NOTE

Using `setprecision(4)` and `scientific`: `1.2346e+02`

Custom fill character: `*****Hello`

This is a line.

Another line after `endl`.

Explanation:

In this program, "**Item**" is printed left aligned in the field of 15 characters, so it is followed by 11 spaces. "**Price**" is printed right_aligned in a field of 10 characters, so it is preceded by 5 spaces. Similarly "Book" and "Pen" are aligned to the left, while "Price", "Rs100" and "Rs15" are aligned to the right.

Here, **`setprecision(n)`** sets the total number of digits after the decimal point (or total significant digits), **`fixed`** forces floating-point output in fixed-point notation (123.46). **`Scientific`** forces floating-point output in scientific-notation (1.2346e+02). The total width for "Hello" is 10 characters, and "Hello" is left-aligned so remaining 5 spaces are filled with *.

`endl`: This manipulator introduces a newline character

("\\n"). It also flushes the output buffer, meaning all buffered output is written immediately to the console. That way you can be certain that output appears instantly.

**Activity 14.1**

Activity Name: Output Presentation Lab

Objective: To format output professionally.

Required Materials: Computer, compiler

Prerequisites: Basic I/O streams

Steps:

1. Apply width, precision, fill.
2. Use manipulators.
3. Compare formatted and unformatted output.

Post-Activity Reflection: How does formatting improve usability?

Expected Outcome: Learners control output presentation effectively

☑ Check Your Progress 14.1**A. Multiple Choice Question (MCQ):**

Which manipulator sets the character used to fill empty spaces in formatted output?

- a) `setw`
- b) `setfill`
- c) `setprecision`
- d) `fixed`

NOTE

B. True/False Question (T/F):

The `ios::right` flag aligns the output to the left within the field width.

C. Short Answer Questions:

1. What is the difference between `setw()` and `width()` in C++?

.....

2. How do you display a floating-point number in scientific notation using C++ manipulators?

.....

14.4 SUMMARY

The unit “**Formatting Output Using Streams and Manipulators**” explores techniques in C++ to control the appearance of data output on the console. It primarily deals with stream formatting using both **member functions of the ios class** and **manipulators**. Initially, the unit explains formatting using **ios members**, such as setting flags to display data in different bases (decimal, octal, hexadecimal), controlling justification (left, right, or internal), and showing signs for numbers. It emphasizes the importance of **clearing format flags** to reset any previously set formatting using `ios::resetiosflags()` or `ios::fmtflags`. An overloaded form of the `setf()` function is introduced, which allows combining multiple formatting flags in a single statement, enhancing code readability and flexibility. The unit further illustrates how to **examine the formatted flags** to understand the current state of the stream. To control spacing and precision, functions like `width()`, `precision()`, and `fill()` are discussed, enabling developers to align output, set the number of decimal places, and fill empty spaces with desired characters. Finally, the unit highlights the use of **manipulators** such as `setw`, `setprecision`, `setfill`, `fixed`, `scientific`, `left`, `right`, and `boolalpha`, which provide a more readable and convenient way to format input/output streams without directly using ios member functions. Overall, mastering these tools allows programmers to present data clearly, professionally, and according to specific formatting requirements.

14.5 GLOSSARY

- **Stream** – A sequence of data elements made available over time, such as `cin` or `cout`.
- **Manipulator** – A function that can be used to modify the formatting of stream I/O, e.g., `setw`, `setprecision`.
- **ios::flags** – Member of `ios` class used to control stream formatting options.
- **setf()** – Function to set formatting flags for a stream.
- **resetiosflags()** – Function to clear previously set formatting flags.

NOTE

- **width()** – Function to set the minimum number of characters for output.
- **precision()** – Sets the number of digits displayed after the decimal point.
- **fill()** – Fills unused spaces in output with a specific character.
- **left, right, internal** – Justification flags controlling text alignment in output.
- **Fixed** – Displays floating-point numbers in fixed-point notation.
- **Scientific** – Displays floating-point numbers in scientific notation.
- **boolalpha** – Manipulator to display true or false instead of 1 or 0.
- **hex, dec, oct** – Format flags for hexadecimal, decimal, and octal number systems.
- **formatted flags** – Current settings in a stream that determine how data is displayed.
- **overloaded setf()** – Version of setf() allowing multiple flags to be set at once.

14.6 ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 14.1**A. MCQ Answer:****1. b) Setfill**

Explanation: The setfill(ch) manipulator sets the character ch used to fill empty spaces in formatted output.

B. True/False Answer:**1. False**

Explanation: ios::right aligns output to the right, not the left.

C. Short Answer Answers:**1. Difference between setw() and width():**

- setw(n) is a **manipulator** used in a single output statement to set the field width temporarily.
- width(n) is an **ios member function** that sets the field width for the next output only, similar to setw(), but used differently in code.

2. Displaying floating-point numbers in scientific notation:
Use the manipulator scientific.

Example:

```
double num = 1234.567;
cout << scientific << num;
```

This outputs the number in exponential form, e.g., 1.234567e+03.

14.7 SUGGESTED READINGS – OER

NOTE

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/UnitUrls?unitId=1479	

14.8 REFERENCE

1. **Standard C++ IOSTreams and Locales: Advanced Programmer's Guide and Reference** – Angelika Langer, Klaus Kreft – ISBN: 978-0321585585 – Published Year: 2000
2. **C++: The Complete Reference** – Herbert Schildt – ISBN: 978-0070532465 – Published Year: 2015
3. **The C++ Programming Language** – Bjarne Stroustrup – ISBN: 020112078X – Published Year: 2013
4. **C++ Primer** – Stanley B. Lippman, Josée Lajoie, Barbara E. Moo – ISBN: 978-0321714114 – Published Year: 2012
5. **C++ in a Nutshell** – Ray Lischner – ISBN: 059600298X – Published Year: 2003

14.9 EXERCISE

Part – 1

A. Multiple Choice Questions

1. Which manipulator is used to set the width of the output field?

- a) setprecision
- b) setw
- c) setfill
- d) fixed

2. What does the precision() function control?

- a) Number of characters in a string
- b) Number of decimal places for floating-point numbers
- c) Field width of output
- d) Stream justification

3. Which flag displays numbers in hexadecimal format?

- a) dec
- b) oct
- c) hex

NOTE

d) fixed

4. Which manipulator is used to display boolean values as true or false?

- a) boolalpha
- b) noboolalpha
- c) setw
- d) setfill

5. What does ios::left do?

- a) Aligns output to the right
- b) Aligns output to the left
- c) Aligns numbers internally
- d) Fills empty space with zeros

Answer with Explanations

1. b) setw

Explanation- setw(n) sets the width of the next output field to n characters.

2. b) Number of decimal places

Explanation- precision() sets how many digits are displayed after the decimal point for floating-point numbers.

3. c) hex

Explanation- hex displays integer values in hexadecimal format.

4. a) boolalpha

Explanation- This manipulator converts boolean values to true or false instead of 1 or 0.

5. b) Aligns output to the left

Explanation- ios::left justifies text to the left within the field width.

B. Short Answer Type Questions

- 1. Explain the purpose of setf() in C++ streams.
- 2. How does fill() differ from setw()?
- 3. What is the difference between fixed and scientific manipulators?
- 4. How can you reset all formatting flags in a stream?
- 5. Give an example of using multiple formatting flags with setf().

Part – 2

C. Long Answer Type Questions (200 words)

- 1. Discuss the use of width(), precision(), and fill() to format output in C++. Include examples.

2. Explain the concept of formatted flags in streams and how `setf()` and `resetiosflags()` are used to manage them.
3. Describe the purpose of manipulators in C++ and compare their advantages over using `ios` member functions directly.
4. Write a program snippet that formats output to show a table of numbers with proper alignment and precision.
5. Explain how justification flags (`left`, `right`, `internal`) affect the formatting of text and numbers.

Part – 3**D. Long Answer Type Questions (300 words)**

1. Explain in detail how C++ streams can be formatted using both `ios` members and manipulators. Include examples of numeric and text formatting.
2. Discuss the importance of formatting output in real-world applications. Explain how `setw`, `setprecision`, `setfill`, and justification flags can improve readability of console output.
3. Write a detailed explanation with code showing how to combine multiple flags using `setf()` and how to reset them.
4. Compare and contrast manipulators like `fixed`, `scientific`, and `boolalpha` with `ios` member functions in terms of ease of use and readability.
5. Explain the internal working of stream formatting flags and how they can be examined and modified to create customized output.

UNIT 15

CREATING CUSTOM MANIPULATORS

NOTE

UNIT STRUCTURE

- 15.1. Introduction
- 15.2. Objective
- 15.3. Creating your own Manipulators
- 15.4. Summary
- 15.5. Glossary
- 15.6. Answers to Check Your Progress
- 15.7. Suggested readings - OER
- 15.8. Reference
- 15.9. Exercise

15.1 INTRODUCTION

Unit 15 builds upon the learner's understanding of the C++ I/O system and formatting techniques acquired in the previous units. Learners are already familiar with predefined streams, basic input-output operations, format flags, and standard manipulators used to control output presentation. While these built-in tools are sufficient for many applications, complex and repetitive formatting requirements often arise in real-world software systems. In such cases, relying solely on predefined manipulators can lead to repetitive code and reduced readability. This unit addresses this limitation by introducing the concept of custom, or user-defined, manipulators in C++.

The primary focus of this unit is to explain how programmers can create their own manipulators to extend the functionality of the C++ I/O system. Learners study the structure and syntax required to define custom manipulators and understand how they interact with stream objects. The unit explains how custom manipulators help encapsulate formatting logic into reusable components, improving code organization and maintainability. Examples illustrate how custom manipulators can control spacing, alignment, and output styles in a consistent manner across programs. Learners also understand the advantages of using manipulators over repeated formatting statements, particularly in large applications where consistency and clarity are essential.

This unit emphasizes practical implementation and reinforces earlier concepts such as stream handling, function design, and reusability. Learners see how custom manipulators integrate seamlessly with existing stream operations and follow the same usage pattern as standard manipulators.

By the end of this unit, learners gain the ability to design and implement custom manipulators that enhance program flexibility and professionalism. This unit consolidates knowledge from the entire block on ++ I/O and

NOTE

formatting, and effectively concludes the course. Learners complete their C++ journey with the skills required to write interactive, well-structured, and professionally formatted applications, preparing them for advanced programming tasks and real-world software development challenges.

15.2 OBJECTIVES

After completing this unit, learners will be able to:

- Design custom manipulators for reusable output formatting.
- Implement user-defined manipulators using stream mechanisms.
- Integrate custom manipulators with standard stream operations.
- Evaluate the effectiveness of custom manipulators in large programs.
- Develop professional-quality output formatting solutions in C++.

15.3 CREATING YOUR OWN MANIPULATORS

Creating custom manipulators in C++ is a strong technique to make code readable, re-useable, and extensible. Manipulator is basically used with an output stream such as `std::cout` to change the output or formatting in C++.

Steps to create a Custom Manipulator:

Define the Manipulator Function: A manipulator is essentially a function that modifies an output stream and returns a reference to the stream.

Using Manipulator in the Output Stream: Once defined, you can use manipulator along with `<<` operator, very much like you would use standard manipulators like `std::endl` or `std::setw`.

Return the Output Stream: The function of manipulator should return the output stream to allow its use in a chain of output operations.

Example: Custom Manipulator to Add a Newline

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

// Custom manipulator function
ostream& custom_newline(ostream &out) {
    out << "\n-\n"; // Insert a newline and a hyphen
    return out; // Return the stream to allow chaining
}

void main() {
    cout << "Hello" << custom_newline << "World!";
    getch();
}
```

OUTPUT

```
Hello
-
World!
```

Explanation:

In this program, `custom_newline` is a function that modifies the output stream `std::ostream`. It outputs `"\n-\n"`, which creates a newline and a hyphen. Finally the function returns the stream (out), enabling us to chain it with other output operations.

**Activity 15.1**

Activity Name: Build Your Own Manipulator

Objective: To design reusable formatting tools.

Required Materials: Computer, compiler

Prerequisites: Stream formatting concepts

Steps:

1. Identify a repetitive formatting need.
2. Design a custom manipulator.
3. Integrate it into a program.

Post-Activity Reflection: How do custom manipulators improve code quality?

Expected Outcome: Learners create reusable and professional formatting solutions.

☑ Check Your Progress 15.1**A. Multiple Choice Question (MCQ):**

1. Custom manipulators in C++ are mainly used for:
 - a) Memory management
 - b) Stream formatting
 - c) File deletion
 - d) Thread control

B. True/False

Custom manipulators can be created only for output streams.

C. Short Answer

1. What is a parameterized manipulator?

.....

2. Name one benefit of using custom manipulators in C++.

.....

NOTE

15.4 SUMMARY

Creating custom manipulators in C++ allows programmers to extend and personalize the behavior of input and output streams beyond the standard manipulators like `endl` and `setw`. Custom manipulators are special functions that work with stream objects to control formatting, display, and input/output behavior in a more flexible way. They are especially useful when repetitive formatting is required, such as displaying currency symbols, adding prefixes or suffixes to output, controlling spacing, or changing number formats. Custom manipulators can be created in two main ways: parameterless manipulators and parameterized manipulators. Parameterless manipulators are simple functions that take and return a reference to an ostream or istream, while parameterized manipulators involve additional arguments and are usually implemented using helper classes or functions. By creating custom manipulators, programmers can make their code more readable and maintainable because formatting logic is separated from the main program logic. This improves consistency in output and reduces errors caused by repeated formatting code. Custom manipulators also support the idea of reusability, as they can be used in multiple parts of a program once defined. Understanding how to design and implement these manipulators helps programmers build more professional-looking console applications and gives them greater control over how data is presented and read from streams. Overall, creating your own manipulators enhances the power of C++ stream operations and makes input/output handling more structured, efficient, and user-friendly.

15.5 GLOSSARY

- **Manipulator** – A special function used to control formatting of input and output streams.
- **Custom Manipulator** – A user-defined manipulator created to perform specific formatting tasks.
- **Stream** – An object used for input and output operations in C++, such as `cin` and `cout`.
- **ostream** – Output stream class used to send data to output devices.
- **istream** – Input stream class used to receive data from input devices.
- **Parameterless Manipulator** – A manipulator that does not take any extra arguments.
- **Parameterized Manipulator** – A manipulator that accepts arguments to control formatting.
- **Formatting** – The way data is arranged and displayed in output.
- **Function Overloading** – Defining multiple functions with the same name but different parameters.
- **Reusability** – The ability to use the same code in multiple places without rewriting it.

15.6 ANSWER TO CHECK YOUR PROGRESS

Check Your Progress 15.1

A. MCQ Answer:

1. b) Stream formatting

Explanation: Custom manipulators are specifically designed to control how data is displayed or read through C++ streams. They allow programmers to define their own formatting rules, such as adding symbols, changing spacing, or modifying the way numbers and text appear in the output.

B. True/False Answer:

1. False

Explanation: Custom manipulators can be created for both output streams (ostream) and input streams (istream). This means they can be used not only to format output shown with cout but also to control how input is read using cin.

C. Short Answer Questions Answers:

1. A parameterized manipulator is a type of user-defined manipulator that takes one or more arguments to control its behavior. Unlike simple manipulators, it can accept values such as width, precision, or custom symbols. This makes it more flexible because the programmer can change the formatting dynamically while using the stream.
2. One major benefit of using custom manipulators is improved code readability and reusability. Instead of repeating the same formatting code multiple times, a manipulator can be written once and reused throughout the program, making the code cleaner, easier to maintain, and less prone to errors.

15.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectionline.com/User/UnitUrls?unitId=1480	

15.8 REFERENCE

- **C++ Cookbook** – Wayne Murphy – ISBN: 978-0596007612 – Published Year: 2006
- **C++ in a Nutshell** – Ray Lischner – ISBN: 059600298X – Published Year: 2003

NOTE

- **C++ Primer** – Stanley B. Lippman, Josée Lajoie, Barbara E. Moo – ISBN: 978-0321714114 – Published Year: 2012
- **Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference** – Angelika Langer, Klaus Kreft – ISBN: 978-0321585585 – Published Year: 2000

15.9 EXERCISE

Part – 1**A. Multiple Choice Questions**

- 1. What is the main purpose of creating a custom manipulator in C++?**
 - a) To increase program execution speed
 - b) To control input and output formatting
 - c) To manage memory allocation
 - d) To handle exceptions
- 2. A parameterless custom manipulator usually works with which stream type?**
 - a) vector
 - b) fstream
 - c) ostream or istream
 - d) string
- 3. Which operator is most commonly overloaded when creating custom manipulators?**
 - a) +
 - b) –
 - c) << and >>
 - d) *
- 4. Parameterized manipulators generally require the use of:**
 - a) Macros only
 - b) Helper classes or functions
 - c) Global variables only
 - d) Pointers only
- 5. Custom manipulators help in improving:**
 - a) Hardware performance
 - b) Code readability and maintainability
 - c) Hard disk speed
 - d) Network bandwidth

Answer with Explanations**1. b) To control input and output formatting**

Explanation: Custom manipulators are designed specifically to format input and output streams.

2. c) ostream or istream

Explanation: Manipulators operate directly on stream objects such as output and input streams.

3. c) << and >>

Explanation: These operators are used to insert data into and extract data from streams.

4. b) Helper classes or functions

Explanation: Parameterized manipulators usually need helper logic to pass and manage arguments.

5. b) Code readability and maintainability

Explanation: They make code cleaner by separating formatting logic from business logic.

B. Short Answer Type Questions

1. What is a custom manipulator in C++?
2. Mention one difference between standard and custom manipulators.
3. What is a parameterless manipulator?
4. Why are custom manipulators useful in large programs?
5. Name any one standard C++ manipulator.

Part – 2**C. Long Answer Type Questions (200 words)**

1. Explain the concept of custom manipulators and their advantages in C++.
2. Describe the steps involved in creating a parameterless custom manipulator.
3. Explain how parameterized manipulators are different from parameterless manipulators.
4. Discuss the role of operator overloading in creating custom manipulators.
5. Explain how custom manipulators improve code readability with suitable examples.

Part – 3**D. Long Answer Type Questions (300 words)**

1. Explain in detail how to create and use a custom parameterless manipulator in C++ with an example.
2. Describe the complete process of designing a parameterized custom manipulator with a suitable example.
3. Discuss the internal working of stream objects when custom manipulators are applied.

NOTE

4. Explain the differences between standard manipulators and user-defined manipulators in detail.
5. Write a detailed note on the importance of custom manipulators in professional C++ application development.



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN A/SECT GROUP UNIVERSITY

Approved by: PCI | AICTE | NCTE | UCI | Member of: APU | Joint Committee: UGC | UER | AICTE | Recognized by: UGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:
