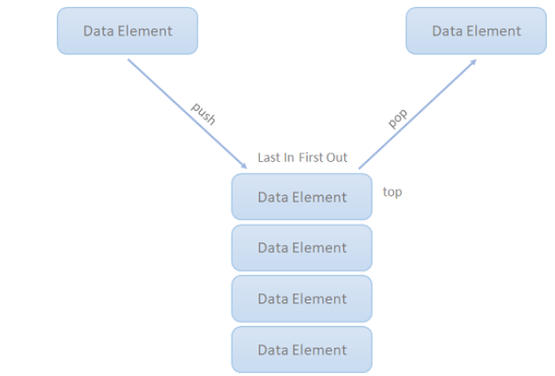
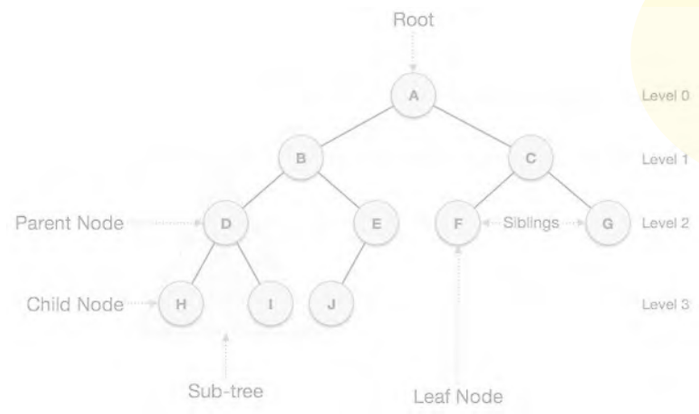
10

Data Structure & OOP Concept Using C++





Government of Nepal
Ministry of Education, Science and Technology
Curriculum Development Centre
Sanothimi, Bhaktapur
Phone : 5639122/6634373/6635046/6630088
Website- https://www.moecdc.gov.np
Email- info@moecdc.gov.np

Technical and Vocational Stream
Learning Resource Material

Data Structure & OOP Concept Using C++

(Grade 10)
Computer Engineering

Government of Nepal
Ministry of Education, Science and Technology
Curriculum Development Centre
Sanothimi, Bhaktapur

Publisher: Government of Nepal
Ministry of Education, Science and Technology
Curriculum Development Centre
Sanothimi, Bhaktapur

Layout by Khados Sunuwar

Preface

The curriculum and curricular materials have been developed and revised on a regular basis with the aim of making education objective-oriented, practical, relevant and job oriented. It is necessary to instill the feelings of nationalism, national integrity and democratic spirit in students and equip them with morality, discipline, self-reliance, creativity and thoughtfulness. It is essential to develop linguistic and mathematical skills, knowledge of science, information and communication technology, environment, health and population and life skills in students. It is also necessary to bring the feeling of preserving and promoting arts and aesthetics, humanistic norms, values and ideals. It has become the need of the present time to make them aware of respect for ethnicity, gender, disabilities, languages, religions, cultures, regional diversity, human rights and social values to make them capable of playing the role of responsible citizens with applied technical and vocational knowledge and skills. This learning resource material for computer engineering has been developed in line with the Secondary Level computer engineering Curriculum with an aim to facilitate the students in their study and learning on the subject by incorporating the recommendations and feedback obtained from various schools, workshops, seminars and interaction programs attended by teachers, students, parents and concerned stakeholders.

In bringing out the learning resource material in this form, the contribution of the Director General of CDC Mr. Yubaraj Paudel and members of the subject committee Dr. Baburam Dawadi, Dr. Sarbim Sayami, Mrs. Bibha Sthapit, Mrs. Trimandir Prajapati is highly acknowledged. This learning resource material is compiled and prepared by Mr. Bimal Thapa, Mr, Bikesh Shrestha, Mr. Nabin Maskey. The subject matter of this material is edited by Mr. Badrinath Timsina and Mr. Khilanath Dhamala. Similarly, the language is edited by Mr. Binod Raj Bhatta. CDC extends sincere thanks to all those who have contributed to developing this material in this form.

This learning resource material contains a wide coverage of subject matters and sample exercises which will help the learners to achieve the competencies and learning outcomes set in the curriculum. Each chapter in the material clearly and concisely deals with the subject matters required for the accomplishment of the learning outcomes. The Curriculum Development Centre always welcomes creative and constructive feedback for the further improvement of the material.

2082 BS

Curriculum Development Centre
Sanothimi, Bhaktapur

Table of Content

Guidelines to Teachers

A. Facilitation Methods

The goal of this course is to combine the theoretical and practical aspects of the contents needed for the subject. The nature of contents included in this course demands the use of practical or learner focused facilitation processes. Therefore, the practical side of the facilitation process has been focused much. The instructor is expected to design and conduct a variety of practical methods, strategies or techniques which encourage students engage in the process of reflection, sharing, collaboration, exploration and innovation new ideas or learning. For this, the following teaching methods, strategies or techniques are suggested to adopt as per the course content nature and context.

Brainstorming

Brainstorming is a technique of teaching which is creative thinking process. In this technique, students freely speak or share their ideas on a given topic. The instructor does not judge students' ideas as being right or wrong, but rather encourages them to think and speak creatively and innovatively. In brainstorming time, the instructor expects students to generate their tentative and rough ideas on a given topic which are not judgmental. It is, therefore, brainstorming is free-wheeling, non-judgmental and unstructured in nature. Students or participants are encouraged to freely express their ideas throughout the brainstorming time. Whiteboard and other visual aids can be used to help organize the ideas as they are developed. Following the brainstorming session, concepts are examined and ranked in order of importance, opening the door for more development and execution. Brainstorming is an effective technique for problem-solving, invention, and decision-making because it taps into the group's combined knowledge and creative ideas.

Demonstration

Demonstration is a practical method of teaching in which the instructor shows or demonstrates the actions, materials, or processes. While demonstrating something the students in the class see, observe, discuss and share ideas on a given topic. Most importantly, abstract and complicated concepts can be presented into visible form through demonstration. Visualization bridges the gap between abstract ideas and concrete manifestations by utilizing the innate human ability to think visually. This enables students to make better decisions, develop their creative potential, and obtain deeper insights across a variety of subject areas.

Peer Discussion

Peer conversation is a cooperative process where students converse with their peers to exchange viewpoints, share ideas, and jointly investigate subjects that are relevant or of mutual interest. Peer discussion is an effective teaching strategy used in the classroom to encourage critical thinking, active learning, and knowledge development. Peer discussions encourage students to express their ideas clearly, listen to opposing points of view, and participate in debate or dialogue, all of which contribute to a deeper comprehension and memory of the course material. Peer discussions also help participants develop critical communication and teamwork skills by teaching them how to effectively articulate their views, persuasively defend their positions, and constructively respond to criticism.

Peer conversation is essential for professional growth and community building outside of the classroom because it allows practitioners to share best practices, work together, and solve problems as a group. In addition to expanding their knowledge horizon and deepening their understanding, peer discussions help students build lasting relationships and a feeling of community within their peer networks.

Group Work

Group work is a technique of teaching where more than two students or participants work together to complete a task, solve a problem or discuss on a given topic collaboratively. Group work is also a cooperative working process where students join and share their perspectives, abilities, and knowledge to take on challenging job or project. Group work in academic contexts promotes active learning, peer teaching, and the development of collaboration and communication skills. Group work helps individuals to do more together than they might individually do or achieve.

Gallery Walk

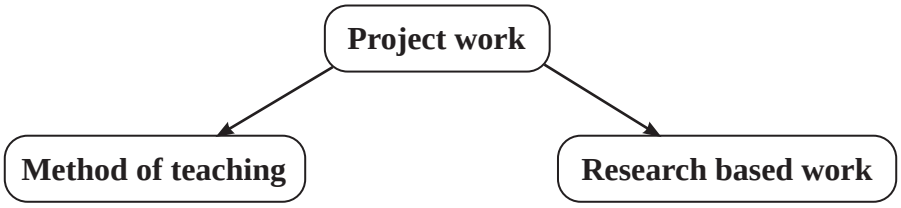
Gallery walk is a critical thinking strategy. It creates interactive learning environment in the classroom. It offers participants or students a structured way to observe exhibition or presentation and also provides opportunity to share ideas. It promotes peer-to-peer or group-to-group engagement by encouraging participants to observe, evaluate and comment on each other's work or ideas. Students who engage in this process improve their communication and critical thinking abilities in addition to their comprehension of the subject matter, which leads to a deeper and more sophisticated investigation of the subjects at hand.

Interaction

The dynamic sharing of ideas, knowledge, and experiences between people or things is referred to as interaction, and it frequently takes place in social, academic, or professional settings. It includes a broad range of activities such as dialogue, collaboration or team work, negotiation, problem solving, etc. Mutual understanding, knowledge sharing, and interpersonal relationships are all facilitated by effective interaction. Interaction is essential for building relationships, encouraging learning, and stimulating creativity in both in-person and virtual contexts. Students can broaden their viewpoints, hone their abilities, and jointly achieve solutions to difficult problems by actively interacting with others.

Project Work

Project work is a special kind of work that consists of a problematic situation which requires systematic investigation to explore innovative ideas and solutions. Project work can be used in two senses. First, it is a method of teaching in regular class. The next is: it is a research work that requires planned investigation to explore something new. This concept can be presented in the following figure.



Project work entails individuals or teams working together to achieve particular educational objectives. It consists of a number of organized tasks, activities, and deliverables. The end product is important for project work. Generally, project work will be carried out in three stages. They are:

- Planning
- Investigation
- Reporting

B. Instructional Materials

Instructional materials are the tools and resources that teachers use to help students. These resources/materials engage students, strengthen learning, and improve conceptual comprehension while supporting the educational goals of a course or program. Different learning styles and preferences can be accommodated by the variety of instructional

resources available. Here are a few examples of typical educational resource types:

- Daily used materials
- Related Pictures
- Reference books
- **Slides and Presentation:** PowerPoint slides, keynote presentations, or other visual aids that help convey information in a visually appealing and organized manner.
- **Audiovisual Materials:** Videos, animations, podcasts, and other multimedia resources that bring concepts to life and cater to auditory and visual learners.
- **Online Resources:** Websites, online articles, e-books, and other web-based materials that can be accessed for further reading and research.

Maps, Charts, and Graphs: Visual representations that help learners understand relationships, patterns, and trends in different subjects.

Real-life Examples and Case Studies: Stories, examples, or case studies that illustrate the practical application of theoretical concepts and principles.

C. Assessment

Formative Test

Classroom discussions: Engage students in discussions to assess their understanding of concepts.

Quizzes and polls: Use short quizzes or polls to check comprehension during or after a lesson.

Homework exercises: Assign tasks that provide ongoing feedback on individual progress.

Peer review: Have students review and provide feedback on each other's work.

Summative Test

Exams: Conduct comprehensive exams at the end of a unit or semester.

Final projects: Assign projects that demonstrate overall understanding of the subject.

Peer Assessment

Group projects: Evaluate individual contributions within a group project.

Peer feedback forms: Provide structured forms for students to assess their peers.

Classroom presentations: Have students assess each other's presentations.

Objective Test

Multiple-choice tests: Use multiple-choice questions to assess knowledge.

True/False questions: Assess factual understanding with true/false questions.

Matching exercises: Evaluate associations between concepts or terms.

Portfolio Assessment

Compilation of work: Collect and assess a variety of student work samples.

Reflection statements: Ask students to write reflective statements about their work.

Showcase events: Organize events where students present their portfolios to peers or instructors.

Observational Assessment

Classroom observations: Observe students' behavior and engagement during class.

Performance observations: Assess practical skills through direct observation.

Field trips: Evaluate students' ability to apply knowledge in real-world settings.

Basic Introduction of Data Structure

Unit 1

1.1 Introduction to Data Structure

In C++, a data structure is a way of organizing and storing data so that it can be accessed and modified efficiently. Data structures provide a means to manage large amounts of data in a structured format, enabling complex operations on data to be performed easily. They are fundamental in solving problems in computer science and play a significant role in software development.

Important of Data Structures

i. They help organize and store data efficiently.

ii. They improve the performance of algorithms by optimizing operations like searching, sorting, and modifying data.

iii. Different data structures suit different types of problems, so choosing the right one can simplify and speed up solutions.

1.2 Advantages of Data Structure

Data structures in C++ provide a range of benefits, allowing for efficient data management, performance optimization, and improved readability and maintainability of code. Here are some key advantages:

i. **Efficient Data Management:** Data structures like arrays, vectors, and lists allow for systematic storage, enabling efficient retrieval, insertion, deletion, and updating of data.

ii. **Performance Optimization:** Choosing the right data structure reduces time complexity and optimizes performance. C++ STL containers are highly optimized for this purpose.

iii. **Code Reusability and Modularity:** STL offers reusable, adaptable data structures through templates, enhancing code readability and maintainability by separating logic from implementation.

iv. **Memory Management:** Dynamic data structures like linked lists optimize memory usage by allocating and deallocating memory as needed, reducing

waste compared to static arrays.

v. **Enhanced Data Security and Integrity:** C++ data structures maintain data integrity through logical organization and strict data typing, while encapsulation ensures controlled access to data.

1.3 Terms Used in Data Structure

In data structures, especially when organizing data for storage and retrieval, terms like data, group item**,** record**,** entity**,** attribute (or field), and file are commonly used. Here's an explanation of each term and its role in organizing data:

i. **Data:** Raw facts or values that can be processed to derive meaningful information (e.g., "25", "John", "3.14"). Stored in data structures for manipulation and problem-solving.

ii. **Group Item:** A collection of related data items logically grouped together (e.g., "Address" containing "Street", "City", "State", "Zip Code"). Represented as structures or classes in C++.

iii. **Record:** A collection of fields describing an entity (e.g., a student record with "ID", "Name", "Age", "Grade"). Implemented in C++ using structs or classes.

iv. **Entity:** A distinct real-world object or concept with identifiable attributes (e.g., a student, book, or employee). Modeled using data structures in C++ (e.g., a "Student" class).

v. **Attribute (or Field):** A specific characteristic or property of an entity (e.g., "Title", "Author", "ISBN" for a "Book"). Defined as data members in C++ structs or classes.

vi. **File:** A collection of records stored on a storage medium, accessed or modified as needed (e.g., a text file with employee records). File handling in C++ uses streams (ifstream and ofstream) for persistent data storage.

1.4 Need of Data Structure

Data structures are needed for various reasons, primarily to manage, organize, and optimize the way data is stored, accessed, and manipulated. Here's an overview of why data structures are crucial:

i. **Efficient Data Storage and Organization:** Data structures organize data systematically, simplifying storage, retrieval, and use of related data items.

ii. **Optimized Access and Retrieval:** Efficient access methods like arrays

(index-based) and hash tables (fast lookups) enable quick retrieval of data, especially for large datasets.

iii. **Better Memory Management:** Dynamic structures (e.g., linked lists, trees) allocate memory on-demand, reducing waste and improving efficiency.

iv. **Improved Performance:** Choosing the right data structure optimizes time complexity for operations (e.g., insertion, deletion, searching), enhancing application performance.

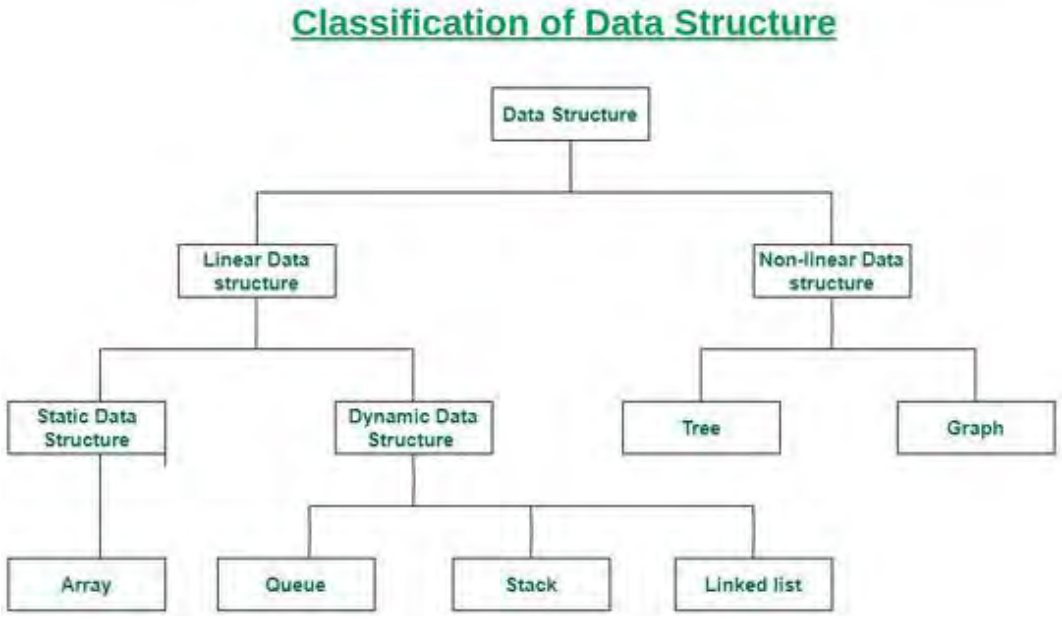
v. **Supports Complex Relationships and Data Modeling:** Data structures like trees (for hierarchies) and graphs (for networks) represent complex relationships, enabling the creation of applications for real-world data.

vi. **Data Integrity and Security:** Data structures help maintain integrity by organizing data and restricting access, preventing unintended modifications through encapsulation (e.g., classes, structs).

vii. **Simplifies Problem Solving:** Predefined data structures (e.g., lists, queues, stacks) offer templates that simplify development, allowing focus on higher-level logic and reducing errors.

viii. **Reduces Complexity:** Data structures organize data logically, making code easier to read, maintain, and debug, especially as applications grow.

1.5 Classification of Data Structure



Data structures can be classified into several categories based on their organization, relationship, and operations. Here's a general classification of data structures:

1. Linear Data Structures

A data structure that preserves a linear connection among its data elements is known as a Linear Data Structure. The arrangement of the data is done linearly, where each element consists of the successors and predecessors except the first and the last data element. However, it is not necessarily true in the case of memory, as the arrangement may not be sequential.

Based on memory allocation, the linear data structures are further classified into two types:

1. **Static Data Structures:** The data structures having a fixed size are known as static data Structures. The memory for these data structures is allocated at the compiler time, and their size cannot be changed by the user after being compiled; however, the data stored in them can be altered. The Array is the best example of the static data structure as they have a fixed size, and its data can be modified later.
2. **Dynamic Data Structures:** The data structures having a dynamic size are known as dynamic data structures. The memory of these data structures is allocated at the run time, and their size varies during the run time of the code. Moreover, the user can change the size as well as the data elements stored in these data structures at the run time of the code. Linked Lists, Stacks, and Queues are common examples of dynamic data structures

Types of Linear Data Structures

The following is the list of linear data structures that we generally use:

1. Arrays

An **Array** is a data structure used to collect multiple data elements of the same data type into one variable. Instead of storing multiple values of the same data types in separate variable names, we could store all of them together into one variable. This statement doesn't imply that we will have to unite all the values of the same data type in any program into one array of that data type. But there will often be times when some specific variables of the same data types are all related to one another in a way appropriate for an array.

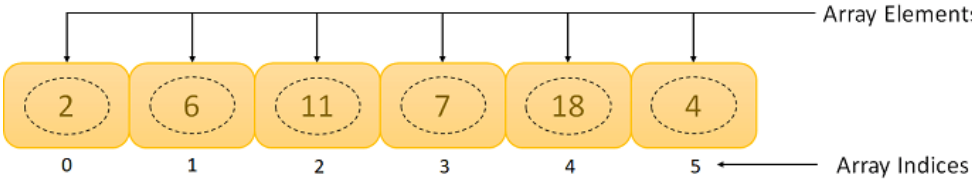


Figure 1: An array

Arrays can be Classified into Different Types

1. **One-Dimensional Array:** An array with only one row of data elements is known as a one-dimensional array. It is stored in ascending storage location.
2. **Two-Dimensional Array:** An array consisting of multiple rows and columns of data elements is called a Two-Dimensional Array. It is also known as a Matrix.
3. **Multidimensional Array:** We can define Multidimensional Array as an Array of Arrays. Multidimensional Arrays are not bounded to two indices or two dimensions as they can include as many indices are per the need.

Some Applications of Array

1. We can store a list of data elements belonging to the same data type.
2. Array acts as an auxiliary storage for other data structures.
3. The array also helps store data elements of a binary tree of the fixed count.
4. Array also acts as a storage of matrices.

2. Linked Lists

A Linked List is another example of a linear data structure used to store a collection of data elements dynamically. Data elements in this data structure are represented by the Nodes, connected using links or pointers. Each node contains two fields, the information field consists of the actual data, and the pointer field consists of the address of the subsequent nodes in the list. The pointer of the last node of the linked list consists of a null pointer, as it points to nothing. Unlike the Arrays, the user can dynamically adjust the size of a Linked List as per the requirements.

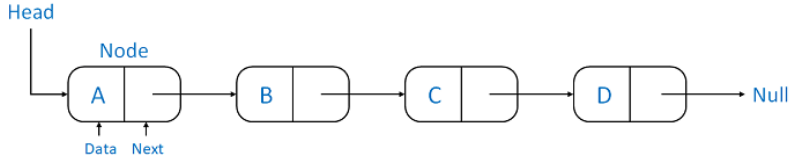


Figure 2: Linked List

Linked Lists can be Classified into Different Ttypes

1. **Singly Linked List:** A singly linked list is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.
2. **Doubly Linked List:** A doubly linked list consists of an information field and two pointer fields. The information field contains the data. The first pointer field contains an address of the previous node, whereas another pointer field contains a reference to the next node. Thus, we can go in both directions (backward as well as forward).
3. **Circular Linked List:** The circular linked list is similar to the Singly Linked List. The only key difference is that the last node contains the address of the first node, forming a circular loop in the circular linked list.

Some Applications of Linked Lists

1. The linked lists help us implement stacks, queues, binary trees, and graphs of predefined size.
2. Linked lists also allow polynomial implementation for mathematical operations.
3. We can use circular linked list to implement operating systems or application functions that Round Robin execution of tasks.
4. Circular linked list is also helpful in a slide show where a user requires to go back to the first slide after the last slide is presented.
5. Doubly linked list is utilized to implement forward and backward buttons in a browser to move forward and backward in the opened pages of a website.

3. Stacks

A **Stack** is a Linear Data Structure that follows the **LIFO** (Last In, First Out) principle that allows operations like insertion and deletion from one end of the Stack, i.e., Top. Stacks can be implemented with the help of contiguous memory, an Array, and non-contiguous memory, a Linked List. Real-life examples of Stacks are piles of books, a deck of cards, piles of money, and many more.

The primary operations in the stack are as follows

1. **Push:** Operation to insert a new element in the Stack is termed as Push Operation.

2. **Pop:** Operation to remove or delete elements from the Stack is termed as Pop Operation.

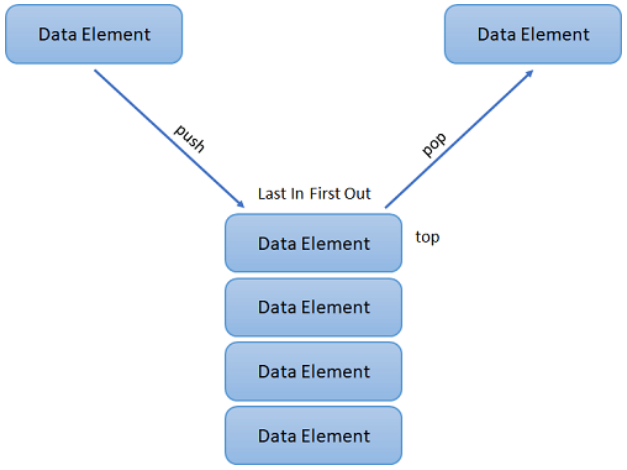


Figure 3: Stack Operation

Some Applications of Stacks

1. The Stack is used as a temporary storage structure for recursive operations.
2. Stack is also utilized as auxiliary storage structure for function calls, nested operations, and deferred/postponed functions.
3. We can manage function calls using Stacks.
4. Stacks allow us to check the expression's syntax in the programming environment.
5. Stacks can be used to reverse a String.
6. Stacks are helpful in solving problems based on backtracking.
7. Stacks are also used in operating system functions.
8. Stacks are also used in UNDO and REDO functions in an edit.

4. Queues

A queue is a linear data structure similar to a Stack with some limitations on the insertion and deletion of the elements. The insertion of an element in a Queue is done at one end, and the removal is done at another or opposite end. Thus, we can conclude that the Queue data structure follows FIFO (First In, First Out) principle to manipulate the data elements. Implementation of Queues can be done using Arrays, Linked Lists, or Stacks. Some real-life examples of Queues are a line at the ticket counter, an escalator, a car wash, and many more.

The following are the primary operations of the Queue:

1. **Enqueue:** The insertion or Addition of some data elements to the queue is called Enqueue. The element insertion is always done with the help of the rear pointer.
2. **Dequeue:** Deleting or removing data elements from the queue is termed dequeue. The deletion of the element is always done with the help of the front pointer.

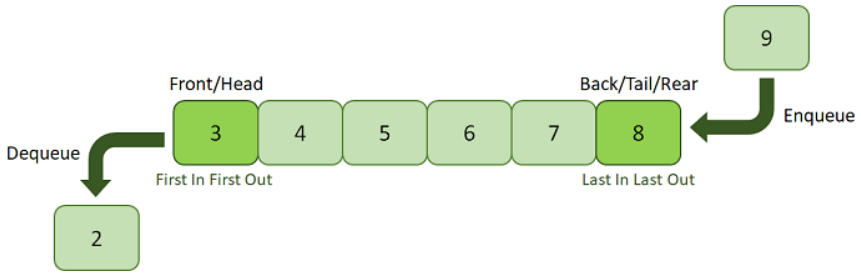


Figure 4: A queue

Some Applications of Queues

1. Queues are generally used in the breadth search operation in Graphs.
2. Queues are also used in Job scheduler operations of operating Systems, like a keyboard buffer queue to store the keys pressed by users and a print buffer queue to store the documents printed by the printer.
3. Queues are responsible for CPU scheduling, job scheduling, and disk scheduling.
4. Priority Queues are utilized in file-downloading operations in a browser.
5. Queues are also used to transfer data between peripheral devices and the CPU.
6. Queues are also responsible for handling interrupts generated by the user applications for the CPU.

Non-Linear Data Structures

Non-linear data structures are data structures where the data elements are not arranged in sequential order. Here, the insertion and removal of data are not feasible in a linear manner. There exists a hierarchical relationship between the individual data items.

Types of Non-Linear Data Structures

The following is the list of non-linear data Structures that we generally use:

1. Trees

A Tree is a non-linear data structure and a hierarchy containing a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

The Tree data structure is a specialized method to arrange and collect data in the computer to be utilized more effectively. It contains a central node, structural nodes, and sub-nodes connected via edges. We can also say that the tree data structure consists of roots, branches, and leaves connected.

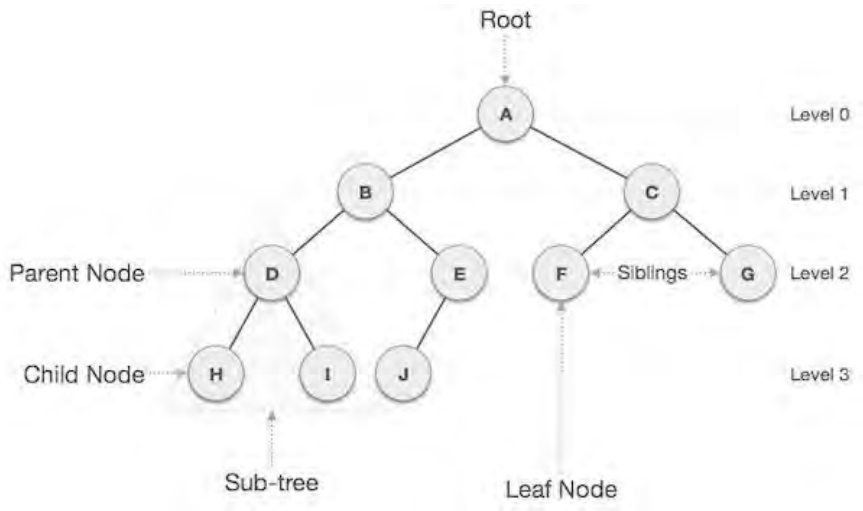


Figure 5: A tree

Trees can be classified into different types

1. **Binary Tree:** A tree data structure where each parent node can have at most two children is termed a Binary Tree.
2. **Binary Search Tree:** A binary search tree is a tree data structure where we can easily maintain a sorted list of numbers.
3. **AVL Tree:** An AVL tree is a self-balancing binary search tree where each node maintains extra information known as a balance factor whose value is either -1, 0, or +1.
4. **B-Tree:** A B-Tree is a special type of self-balancing binary search tree where each node consists of multiple keys and can have more than two children.

Some Applications of Trees

1. Trees are also used to implement the navigation structure of a website.
2. Trees are also helpful in decision-making in gaming applications.
3. Trees are responsible for implementing priority queues for priority-based OS scheduling functions.
4. Trees are also responsible for parsing expressions and statements in the compilers of different programming languages.
5. We can use Trees to store data keys for indexing for database management system (DBMS).
6. Trees are also used in the path-finding algorithm implemented in artificial Intelligence (AI), robotics, and video games applications.

2. Graphs

A Graph is another example of a non-linear data structure comprising a finite number of nodes or vertices and the edges connecting them. The Graphs are utilized to address problems of the real world in which it denotes the problem area as a network such as social networks, circuit networks, and telephone networks. For instance, the nodes or vertices of a graph can represent a single user in a telephone network, while the edges represent the link between them via telephone.

The graph data structure, G is considered a mathematical structure comprised of a set of vertices, V and a set of edges, E as shown below: G = (V,E)

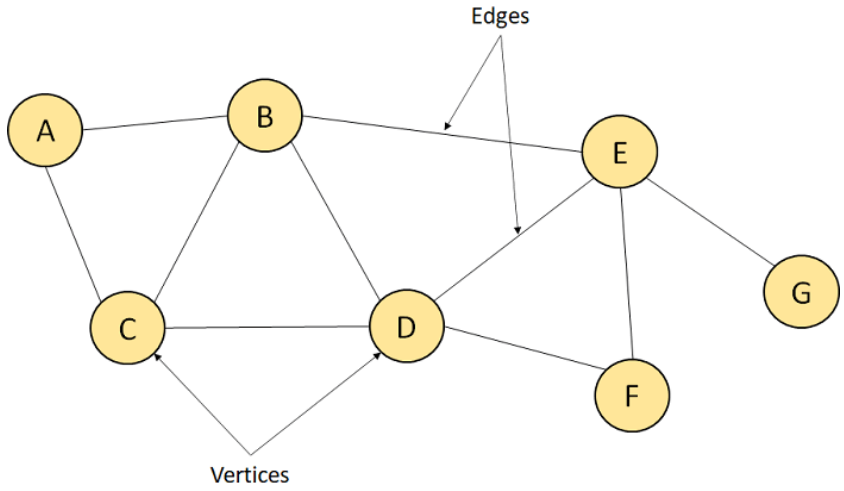


Figure 6: A graph

The above figure represents a Graph having seven vertices A, B, C, D, E, F, G, and ten edges [A, B], [A, C], [B, C], [B, D], [B, E], [C, D], [D, E], [D, F], [E, F], and [E, G].

Depending upon the Position of the Vertices and Edges, the Graphs can be Classified into Different Types

1. **Null Graph:** A graph with an empty set of edges is termed a null graph.
2. **Simple Graph:** A graph with neither self-loops nor multiple edges is known as a simple graph.
3. **Multi Graph:** A graph is said to be Multi if it consists of multiple edges but no self-loops.
4. **Non-Directed Graph:** A graph consisting of non-directed edges is known as a Non-Directed Graph.
5. **Directed Graph:** A graph consisting of the directed edges between the vertices is known as a directed graph.

Some Applications of Graphs

1. Graphs help us represent routes and networks in transportation, travel, and communication applications.
2. Graphs are used to display routes in GPS.
3. Graphs also help us represent the interconnections in social networks and other network-based applications.
4. Graphs are utilized in mapping applications.
5. Graphs are also used to make document link maps of the websites in order to display the connectivity between the pages through hyperlinks.
6. Graphs are also used in robotic motions and neural networks.

1.6 Basic Operations of Data Structures

i. **Traversal:** Traversing a data structure means accessing each data element exactly once so it can be administered. For example, traversing is required while printing the names of all the employees in a department.

ii. **Search:** Search is another data structure operation which means to find the location of one or more data elements that meet certain constraints. Such a data element may or may not be present in the given set of data elements.

For example, we can use the search operation to find the names of all the employees who have the experience of more than 5 years.

iii. **Insertion:** Insertion means inserting or adding new data elements to the collection. For example, we can use the insertion operation to add the details of a new employee the company has recently hired.

iv. **Deletion:** Deletion means to remove or delete a specific data element from the given list of data elements. For example, we can use the deleting operation to delete the name of an employee who has left the job.

v. **Sorting:** Sorting means to arrange the data elements in either Ascending or Descending order depending on the type of application. For example, we can use the sorting operation to arrange the names of employees in a department in alphabetical order or estimate the top three performers of the month by arranging the performance of the employees in descending order and extracting the details of the top three.

vi. **Selection:** Selection means selecting a particular data from the available data. We can select any particular data by specifying conditions inside the loop.

vii. **Update:** The Update operation allows us to update or modify the data in the data structure. We can also update any particular data by specifying some conditions inside the loop, like the Selection operation.

Exercise

Choose the correct answer from the given alternatives.

1. What is the primary goal of a data structure?

   a. To manage data efficiently.
   b. To write clean code.
   c. To debug programs.
   d. To use less memory.

2. Which data structure uses the Last In First Out (LIFO) principle?

   a. Queue
   b. Stack
   c. Array
   d. Linked List

3. In a queue, elements are added at the __ and removed from the __.

   a. Front, rear
   b. Rear, front
   c. Front, middle
   d. Rear, middle

4. Which of the following is a non-linear data structure?

   a. Stack
   b. Queue
   c. Graph
   d. Array

5. The operation of processing each element in a data structure is called.....

   a. Sorting
   b. Searching
   c. Traversal
   d. Merging

6. A graph is a collection of.....

   a. Nodes and Edges
   b. Vertices and array
   c. Tress and Cycles
   d. Linked list and arrays

7. What is a file in data structures?

   a. A collection of related data elements.
   b. A storage location for data.
   c. A data type.
   d. A mathematical function.

8. Which data structure uses FIFO (First In First Out)?

   a) Stack
   b. Queue
   c) Trees
   d. Linked List

9. An array is.....

   a. Dynamic in size
   b. Fixed in size
   c. Always sorted
   d. A graph

10. Which of the following is a non-linear data structure?

   a. Stack
   b. Queue
   c. Graph
   d. Array

Write short answer to the following questions.

1. What are the advantages of data structure?
2. What are some common terms used in data structure?
3. Why data structure is needed?
4. How are data structures classified?
5. What are the different types of linear data structures?
6. What are the different types of non-linear data structures?
7. What are the common operations performed on data structures?

Write long answer to the following questions.

1. Explain the concept of data structure and its significance in computer programming.
2. Describe the different types of data structures used in computer programming.
3. Discuss the advantages of using data structures in computer programming.
4. Describe the different types of data structures based on their organization and implementation.
5. Explain the concept of linear data structures and discuss the different types of linear data structures.
6. Explain the concept of non-linear data structures and discuss the different types of non-linear data structures.
7. Discuss the various operations that can be performed on data structures.

Concept of OOP Using C++

Unit 2

2.1 Introduction to Object Oriented Programming

Object-Oriented Programming (OOP) is a programming language that organizes software design around objects and data, rather than actions and logic. This approach emphasizes the concept of objects, which can contain both data (attributes or properties) and methods (functions or behaviors). OOP is widely used in modern programming languages such as Java, C++, Python, Ruby, and C#.

The primary goal of OOP is to increase modularity, reusability, and maintainability of code. By organizing code into discrete objects, each of which represents a specific concept or entity, OOP makes it easier to manage complex systems and build scalable, flexible software solutions. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Object-Oriented Programming (OOP) in C++ introduces a new way of structuring programs based on objects, which are instances of classes. This paradigm allows you to model real-world entities and their behaviors in the form of classes and objects.

2.2 Features of OOP

C++ is a powerful, high-performance programming language that fully supports Object-Oriented Programming (OOP). It integrates OOP features with the traditional procedural programming style of C, allowing developers to write programs in a cleaner, reusable, and maintainable way. Down below are the key features of OOP in C++:

1. Classes and Objects

- **Class:** A class is a blueprint for creating objects (instances). It defines the attributes (data members) and behaviors (methods) that the objects of the class will have.
- **Object:** An object is an instance of a class. Each object has its own set of attributes and can use the methods defined by the class.

Example

```
#include <iostream>
using namespace std;
class Car {
public:
   string brand;
   string model;
   void display_info() {
      cout << "Brand: " << brand << ", Model: " << model << endl;
   }
};
int main() {
   // Creating an object of the Car class
   Car myCar;
   myCar.brand = "Toyota";
   myCar.model = "Corolla";
   myCar.display_info();  // Output: Brand: Toyota, Model: Corolla
   return 0;
}
```

Explanation: Car is a class that has attributes brand and model, and a method display_info(). The object my Car is an instance of the class Car, and we use it to call the display_info() method.

2. Encapsulation

It is one of the fundamental concepts of Object-Oriented Programming (OOP). It refers to the concept of bundling the data (attributes) and the methods (functions) that operate on the data into a single unit, i.e., a class. Encapsulation also involves restricting direct access to some of the object's internal data, ensuring that the data is only accessed and modified through well-defined functions (methods).

In simpler terms, encapsulation is like putting your data inside a box (the class) and giving controlled access to the data via special functions (methods), rather than letting other parts of the program modify the data directly.

3. Abstraction

It is another key concept in Object-Oriented Programming (OOP). It involves hiding the complexity of the system and only exposing the essential parts. In simpler terms, abstraction allows you to focus on what an object does rather than how it does it.

In C++, abstraction is typically achieved using abstract classes and pure virtual functions. This allows you to define a "template" or interface for a group of classes without worrying about the specific details of how each class performs the actions.

A class that contains at least one **pure virtual function** is called an abstract class. Abstract classes cannot be instantiated directly; they are meant to be inherited by other classes. Pure virtual function is a function declared in an abstract class that has no implementation in the base class and must be overridden in the derived class.

4. Inheritance

It is one of the core concepts of Object-Oriented Programming (OOP). It allows to create a new class (called a **derived class**) based on an existing class (called a **base class**). The derived class **inherits** the attributes (data members) and methods (member functions) of the base class, allowing you to reuse and extend the functionality of the base class.

You can extend or modify the behavior of a base class in a derived class. The access to the inherited members of the base class depends on the access specifier(public, protected, private) used when inheriting.

5. Polymorphism

It is one of the key features of Object-Oriented Programming (OOP). It allows one interface to be used for different data types making it flexible and easy to use. The word "polymorphism" comes from Greek, meaning "many shapes"—it allows methods to behave differently based on the object that is invoking them.

In simpler terms, polymorphism enables objects of different classes to be treated as objects of a common base class, while still allowing them to call methods that are specific to their actual derived class. With polymorphism, we don't need to worry about the specific class of an object. We can treat different objects (of different derived classes) in a uniform way by using a common base class pointer or reference.

2.3 Application

Object-Oriented Programming (OOP) is widely used in software development due to its ability to model real-world entities, promote code reusability, and improve maintainability. OOP is particularly beneficial for building large, complex software systems where modularity, scalability, and ease of maintenance are important.

Here are some common **applications of OOP** across various fields:

1. Software Development

OOP is the foundation of many modern programming languages, frameworks, and libraries used in software development. Some key applications include:

a. Game Development

In game development, OOP is used to model the various game entities like players, enemies, weapons, and levels. Each object (e.g., a character or weapon) can have properties (attributes like health, damage) and methods (actions like attacking, moving) that define how the game works.

Example In a role-playing game (RPG), classes can represent characters, weapons, spells, and items. Each character object can have methods like attack(), defend(), and level_up(), while different types of weapons (e.g., sword, bow) can be subclasses of a generic Weapon class.

b. Simulation Systems

OOP is widely used in simulations where entities interact with each other. For example, simulating traffic, weather, or population dynamics can be done by creating classes for cars, weather systems, or individual agents.

Example A traffic simulation might have a vehicle class, which has subclasses like Car and Truck. These objects interact with a TrafficLight class, with methods for changing light states, and a Road class that tracks vehicles.

2. Web Development

In web development, OOP is used to build and maintain complex websites and applications.

a. Web Frameworks

Many modern web frameworks (like **Django, Ruby on Rails**, and **ASP.NET**) use OOP principles to manage web application components, such as users, forms, databases, and views.

Example In Django, models are Python classes that represent database tables, and each instance of a model represents a record in the database. The User class might have attributes like username, password, and methods like authenticate() and login().

b. Content Management Systems (CMS)

Content management systems like **WordPress** and **Joomla** are based on OOP concepts. In these systems, different types of content (posts, pages, comments) are modeled as objects with specific attributes and methods.

Example A Post class in WordPress might have methods like publish(), edit(), or delete(), and properties like author, content, and published date.

3. Mobile Application Development

OOP plays a crucial role in mobile app development, where user interfaces, behaviors, and interactions are modeled as objects.

a. Android and iOS Apps

Both **Android** (using Java or Kotlin) and **iOS** (using Swift or Objective-C) use OOP principles to model the various components of mobile applications, such as views, controllers, data models, and network operations.

4. Database Management Systems (DBMS)

Object-Oriented Database Management Systems (OODBMS) store data as objects, similar to how OOP models entities in software. Unlike relational databases that store data in tables, OODBMS stores data as objects, making it more aligned with OOP principles.

5. Artificial Intelligence (AI) and Machine Learning

OOP is often used in AI and ML systems, where objects represent different

components of the algorithm, data, or process flow.

a. AI Models and Agents

In AI systems, agents (e.g., a chess-playing agent or a robotic agent) are often represented as objects, where the agent has properties (like position, state, or strategy) and methods (like move(), evaluate(), or decide()).

Example In a game like chess, you might have a ChessPiece class, with subclasses for King, Queen, Knight, etc. Each class would have methods for movement and capturing pieces.

6. E-Commerce Systems

E-commerce applications benefit from OOP by modeling products, customers, orders, payments, and inventories as objects.

a. Product and Order Management

In an online store, you might have classes like Product, Order, ShoppingCart, and Payment. These classes could interact with each other to represent the flow of a customer's purchase.

Example The Order class could have methods like addItem(), removeItem(), and checkout(), while the Payment class could handle transactions with methods like processPayment() or refund().

7. Embedded Systems

OOP is increasingly being used in embedded systems to model devices, sensors, and controllers, making embedded systems more modular and easier to maintain.

Example In an embedded system for controlling a smart home, you could have a sensor class with subclasses for temperatureSensor, motion sensor, and gight sensor. Each sensor could have methods for read data () and send data () to communicate with other devices.

2.4 Structured VS Object Oriented Programming

Structured Programming and **Object-Oriented Programming (OOP)** are two different paradigms used in software development. Each has its own approach to organizing and solving programming problems. Here's a detailed comparison of the two:

Feature	Structured Programming (SP)	Object-Oriented Programming (OOP)
Focus	Sequence of steps (procedure).	Objects and classes (data + behavior).
Code Organization	Divided into functions and procedures.	Organized into objects and classes.
Data Handling	Data is separate from functions.	Data is encapsulated within objects.
Key Principle	Top-down approach, focus on tasks/functions.	Bottom-up approach, focus on objects/entities.
Modularity	Functions are reused, but no data encapsulation.	Encapsulation of data with methods.
Reusability	Achieved by reusing functions.	Achieved by inheritance, polymorphism, and classes.
Maintainability	Can become difficult as the program grows.	Easier to maintain due to modularity and encapsulation.
Real-World Modeling	Does not directly model real-world entities.	Directly models real-world entities through objects.
Code Flexibility	Harder to extend and modify as complexity grows.	Easier to extend and scale with new classes/objects.
Example of Design	Function-driven design.	Class-driven design with objects.

2.5 Tokens and Character Sets

In C++, **tokens** and **character sets** are essential concepts in understanding how the language's syntax works.

1. Tokens in C++

A token is the smallest unit of meaningful code in C++. Tokens are categorized based on their role in the language's grammar. There are six primary types of tokens in C++:

a. Keywords

Keywords are reserved words that have a special meaning in C++ and cannot be used as identifiers (e.g., variable names). Examples: int, class, etc. These

keywords define the structure and behavior of the program.

b. Identifiers

Identifiers are names used to identify variables, functions, classes, and other user-defined items. Identifiers must begin with a letter (a-z, A-Z) or an underscore (_) and may contain letters, numbers, and underscores.

c. Literals

Literals represent constant values that appear directly in the code. There are different types of literals:

- **Integer literals:** 42, -100
- **Floating-point literals:** 3.14, 2.71828f
- **Character literals:** 'a', '1'
- **String literals:** "Hello, World!"
- **Boolean literals:** true, false
- **Null pointer literal:** nullptr

d. Operators

Operators perform operations on variables and values. They can be arithmetic, logical, relational, bitwise, etc. Examples:

- **Arithmetic:** +, -, *, /
- **Relational:** ==, !=, <, >
- **Logical:** &&, ||, !
- **Bitwise:** &, |, ^, ~

e. Punctuation (Separators)

These tokens are used to separate statements, blocks of code, and other syntactic elements. Examples:

- **Semicolon** ; (terminates a statement)
- **Comma** , (separates variables or arguments)
- **Period** . (used to access members of an object)
- **Parentheses** () (used in function calls and expressions)
- **Curly brackets** {} (used for code blocks)

- **Square brackets** [] (used for array subscripts)
- **Colon** : (used in various constructs like case labels, inheritance)

f. Comments

Comments are used for documenting code and are ignored by the compiler. There are two types of comments:

- **Single-line comment:** // This is a comment
- **Multi-line comment:** /* This is a comment */

2. Character Sets in C++

Character set is the combination of English language (Alphabets and White spaces) and math's symbols (Digits and Special symbols). Character Set means that the characters and symbols that a C++ Program can understand and accept. These are grouped to form the commands, expressions, words, c-statements and other tokens for C++ Language.

Special Characters				
+	>	/	[	\
!	;	"	]	{
<	*	.	%	}
:	^	,	~	#
-	(	=	-	|
?	)	'	&	

There are mainly four categories of the character set .

1. Alphabets

Alphabets are represented by A-Z or a-z. C- Language is case sensitive so it takes different meaning for small and upper case letters. By using this character set C statements and character constants can be written very easily. There are total 26 letters used in C-programming.

2. Digits

Digits are represented by 0-9 or by combination of these digits. By using the digits numeric constant can be written easily. Also numeric data can be assigned to the C-tokens. There are total 10 digits used in the C-programming.

3. Special Symbols

All the keyboard keys except alphabet, digits and white spaces are the special symbols. These are some punctuation marks and some special symbols used for special purpose.

There are total 30 special symbols used in the C-programming. Special symbols are used for C-statements like to create an arithmetic statement +, -, * etc., to create relational statement <, >, <=, >=, == etc. , to create assignment statement =, to create logical statement &&, II etc. are required.

4. White Spaces

White spaces has blank space, new line return, Horizontal tab space, carriage ctrl, Form feed etc. are all used for special purpose. Also note that Turbo-C compiler always ignore these white space characters in both high level and low level programming.

2.6 Data Types and Format Specific

Data Types

In C++, **data types** define the kind of data that variables can store. C++ offers a rich set of built-in data types that fall into several categories, including **primitive data** types (simple types), **derived data types**, and **user-defined data types**. Understanding these data types is essential for managing memory and ensuring type safety in your program.

1. Primitive Data Types

These are the built in data types in C++. They include integer types, floating-point types, character types, and boolean types.

a. Integer Types

Integer types are used to represent whole numbers. C++ provides several types for integers, depending on the size and whether they are signed or unsigned. Examples of the integer types are given down below:

- **int:** A basic integer type. Typically 4 bytes (32 bits), but this depends on the system. Example: int x = 10;
- **short:** A smaller integer, typically 2 bytes. Example: short s = 100;
- **long:** A larger integer, typically 4 or 8 bytes. Example: long l = 1000000;

b. Floating-Point Types

Floating-point types are used to represent real numbers (numbers with fractional parts). Example of floating point integer are:

- **float:** A single-precision floating-point number, typically 4 bytes Example: float f = 3.14f;
- **double:** A double-precision floating-point number, typically 8 bytes. Example: double d = 3.14159265359;

c. Character Types

Character types represent individual characters or small integers (ASCII values). Example of character types are:

- **char:** A single character, typically 1 byte. Can store characters like 'a', 'b', 'A', etc. Example : char c = 'A';
- **signed char** and **unsigned char:** Variants of char with explicitly signed or unsigned behavior.

  Example: signed char sc = -50;

  Example: unsigned char uc = 250;

d. Boolean Type

The bool type is used to store truth values (true or false).

- **bool:** Represents a boolean value, which can be either true or false.

  Example: bool is true = true;

2. Derived Data Types

These are types derived from the primitive data types, including arrays, pointers, references, functions, and more. It is made by using primitive data type and should be defined by user.

a. Array

An array is a collection of elements of the same type.

Example: int arr[5] = {1, 2, 3, 4, 5};

b. Pointer

A pointer holds the memory address of another variable.

- **Example** int x = 10; int* p = &x;

  Here, p is a pointer to an integer, and it holds the address of x.

3. User-Defined Data Types

These types are defined by the programmer to provide more specific data structures for the program.

a. Struct (Structure)

A struct is a user-defined data type that allows grouping different types of variables together.

b. Class

A class is similar to a struct, but it can include methods (functions) and private/public access control. It is the foundation for object-oriented programming in C++.

c. Enum (Enumeration)

An enum is a user-defined type consisting of a set of named integer constants.

Format Specific

In C++, **formatting** typically refers to how data is printed to the console or output stream using various formatting options. The language provides powerful ways to control the display of different types of data, especially for numeric types, strings, and characters. These formatting options are most commonly used with iostream streams like std::cout.

C++ offers several ways to format output, including **stream manipulators**, **printf-style formatting**, and **I/O flags**. Let's dive into the different formatting methods:

1. Using iostream Stream Manipulators (C++ Style)

C++ provides a set of **stream manipulators** that allow you to format output in a more readable and type-safe manner.

a. std::setw(int width) — Set Field Width

The setw() manipulator sets the minimum width of the next printed field. If the value is smaller than the specified width, it will be padded with spaces (by default, to the left).

Example

```
#include <iostream>
#include <iomanip>  // For std::setw and other manipulators
using namespace std;

int main() {
   int num = 42;
   cout << setw(5) << num << endl; // Output: "   42" (3 spaces before 42)
   return 0;
}
```

b. std::setfill(char ch) — Fill Character

The setfill() manipulator specifies the character used for padding when the output width is greater than the data width.

- Example

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

int main() {
   int num = 42;
   cout << setw(5) << setfill('*') << num << endl; // Output: "**42"
   return 0;
}
```

c. std::left, std::right, std::internal — Align Text

These manipulators control the alignment of text in a field:

- **std::left**: Left-aligns the text.
- **std::right**: Right-aligns the text.

- **std::internal**: Aligns the text in the center with padding on both sides.
- Example

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
   int num = 42;
   cout << setw(6) << left << num << endl;   // Left-align: "42    "
   cout << setw(6) << right << num << endl;  // Right-align: "    42"
   return 0;
}
```

d. std::fixed and std::scientific — Floating Point Format

These manipulators control the way floating-point numbers are printed:

- **std::fixed**: Forces fixed-point notation (i.e., shows a fixed number of decimal places).
- **std::scientific**: Forces scientific notation (e.g., 1.23e+02).

Example

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
   double pi = 3.14159;
   cout << fixed << setprecision(2) << pi << endl;  // Output: "3.14"
   cout << scientific << setprecision(2) << pi << endl;  // Output: "3.14e+00"
   return 0;
}
```

e. std::setprecision(int n) — Control Decimal Precision

The setprecision() manipulator sets the number of significant digits for floating-point numbers.

Example

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

int main() {
    double pi = 3.141592653589793;
    cout << setprecision(4) << pi << endl; // Output: "3.142"
    return 0;
}
```

2. Using printf-style Formatting (C-Style)

While C++ provides stream manipulators for formatting, you can also use **C-style formatting** with the printf function (from <cstdio>). This style of formatting uses format specifiers to control the output.

a. Common Format Specifiers

- %d or %i — Integer
- %f — Floating-point number
- %c — Character
- %s — String
- %x — Hexadecimal (lowercase)
- %X — Hexadecimal (uppercase)
- %o — Octal
- %p — Pointer (address)
- %e — Scientific notation

Example

```
#include <cstdio>

int main() {
```

```
    int num = 42;
    double pi = 3.14159;
    char ch = 'A';
    printf("Integer: %d\n", num);          // Output: Integer: 42
    printf("Floating-point: %.2f\n", pi);  // Output: Floating-point: 3.14
    printf("Character: %c\n", ch);         // Output: Character: A
    return 0;
}
```

b. Width and Precision

You can specify the width and precision of the printed output using printf format specifiers:

- **Width**: Specifies the minimum width for the printed value.
- **Precision**: For floating-point values, this specifies the number of digits after the decimal point.

Example

```
#include <cstdio>
int main() {
    double pi = 3.14159265359;
   printf("Width: %-10.2f|\n", pi);  // Left-aligned with width 10 and 2 decimal places
     printf("Width: %10.2f|\n", pi);    // Right-aligned with width 10 and 2 decimal
places
    return 0;
}
```

2.7 Basic Input/ Output

In C++, basic **input** and **output (I/O)** operations are primarily handled through the **standard input** and **standard output** streams, using the cin and cout objects, respectively. These objects are part of the <iostream> library, which is included by default in many C++ programs.

1. Output in C++: std::cout

The cout (character output stream) is used to send output to the console.

Basic Syntax

```
#include <iostream>  // For input and output stream objects
int main() {
   cout << "Hello, World!" <<endl;
   return 0;
}
```

- **cout**: This is the object used to output data to the console.
- <<: The insertion operator is used to send data to the cout stream.
- **endl**: Inserts a newline character (\n) and flushes the output buffer. This is equivalent to using "\n", but it has the added effect of flushing the stream, which can be useful in certain cases, such as when working with buffered output.

Example Printing Variables

You can also print variables using cout:

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

int main() {
   int num = 10;
   double pi = 3.14159;
   char letter = 'A';
   cout << "Integer: " << num << endl;
   cout << "Pi: " << pi << endl;
   cout << "Character: " << letter << endl;
   return 0;
}
```

2. Input in C++: cin

The cin (character input stream) is used to read input from the user (usually from the keyboard).

Basic Syntax

```
#include <iostream>  // For input and output stream objects
int main() {
   int age;
   cout << "Enter your age: ";
   cin >> age;  // Read user input and store it in 'age'
   cout << "You entered: " << age << std::endl;
   return 0;
}
```

- **cin**: This is the object used to read input from the user.
- >>: The extraction operator is used to read data from cin and store it in a variable.

Example Reading and Printing Multiple Inputs

```
#include <iostream>
using namespace std;
int main() {
   string name;
   int age;
   cout << "Enter your name: ";
   cin >> name;  // Reads a single word (up to the first space)

   cout << "Enter your age: ";
   cin >> age;  // Reads an integer value

   cout << "Name: " << name << ", Age: " << age << endl;
   return 0;
}
```

3. Basic Example Input and Output Together

Here's an example where we take multiple inputs from the user and print them in a formatted way:

```
#include <iostream>
#include <string>
#include <iomanip>  // For std::setw and other formatting manipulators
using namespace std;
int main() {
   string name;
   int age;
   double height;

   // Input from user
   cout << "Enter your name: ";
   getline(cin, name);

   cout << "Enter your age: ";
   cin >> age;
      cout << "Enter your height (in meters): ";
   cin >> height;
      // Output formatted data
   cout << "\n-- User Information --" << endl;
   cout << "Name: " << name << endl;
   cout << "Age: " << age << " years" << endl;
   cout << "Height: " << fixed << setprecision(2) << height << " meters" << endl;
   return 0;
}
```

2.8 Basic Program Structure

The basic structure of a **C++ program** includes several essential components, such as **headers**, a **main function**, and possibly other functions or variables. C++ follows a structured format where the program execution begins from the main() function.

Here is an overview of the basic components and their order in a typical C++ program:

1. Program Structure Breakdown

a. Preprocessor Directives

Preprocessor directives are commands that are processed before the compilation of the program begins. They typically include header files, which provide declarations and definitions for commonly used functions and objects.

```
#include <iostream>   // Header for input/output streams
```

- **#include**: This directive tells the preprocessor to include the specified file in the program before the actual compilation. In this case, we include <iostream> to use standard input-output features.

b. Namespace Declaration

In C++, the **standard library** (such as std::cout, std::cin, etc.) is encapsulated in a namespace called std. You can use std:: explicitly or declare that you're using the standard namespace to avoid the need for std:: prefix using namespace std; // Avoids needing to prefix 'std::' for every standard library use

c. The Main() Function

The main() function is where the execution of the program begins. It serves as the entry point for a C++ program. The **return type** of main() is typically int, which indicates that the function returns an integer value to the operating system upon termination.

```
int main() {
   // Code for execution goes here


   return 0;  // Return an integer (usually 0 for successful execution)
}
```

- **int main()**: This is the main function that all C++ programs start execution from.
- **return 0;**: A return statement that indicates successful execution to the operating system. The value 0 is commonly used to signify success. Other values can indicate errors.

d. Statements and Declarations

Inside the main() function (or any function), you can declare variables, perform calculations, control program flow, and output results.

- **Declarations**: Defining variables or objects before using them.
- **Statements**: Lines of code that execute certain actions (e.g., assigning values to variables, printing output, etc.).

2. Basic C++ Program Example

Let's look at a complete simple program to illustrate this structure:

```
#include <iostream>  // Include the iostream header for input/output
using namespace std;  // Use the standard namespace to avoid std:: prefix
// The main function - program execution starts here
int main() {
    // Declare variables
    int num1, num2, sum;
    // Get user input
    cout << "Enter two numbers: ";  // Output message
    cin >> num1 >> num2;          // Input values into num1 and num2
    // Calculate sum
    sum = num1 + num2;
    // Output the result
    cout << "The sum of " << num1 << " and " << num2 << " is " << sum << endl;
    return 0;  // Return 0 to indicate successful execution
}
```

2.9 Control Statements

Control statements in C++ are used to control the flow of execution in a program. They allow you to make decisions, repeat operations, or alter the execution path based on specific conditions. There are several types of control statements in C++:

- **Decision-making statements**: if, else-if, switch
- **Looping statements**: for, while, do-while
- **Jump statements**: break, continue, return

1. Decision-Making Statements

Decision making statements are also known as conditional statements or branching statements, are an essential part of programming language. They allow you to control the flow of your program by executing different blocks of code based on certain conditions. Conditional Statements helps you build logic into your programs, enabling them to make decision and respond to different situations.

In C++, **decision-making statements** allow the program to choose a path of execution based on certain conditions. These conditions evaluate whether an expression is true or false, and the program makes decisions accordingly. The main decision-making statements in C++ are:

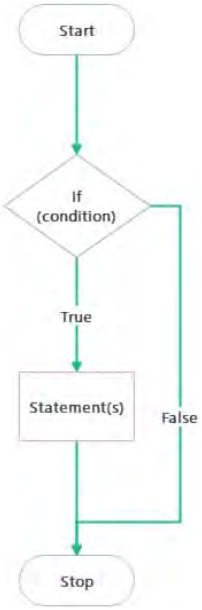
a. if Statement

The if statement executes a block of code if the specified condition is true. It allows you to perform different actions in your program depending on whether a particular condition evaluated to true or false.

Syntax

```
if (condition) {
   // Code to execute if condition is true
}
```

Flowchart



Example

```
int num = 10;
if (num > 0) {
    cout << "The number is positive." << endl;
}
```

In this example, the code inside the if block will be executed because the condition (num > 0) is true.

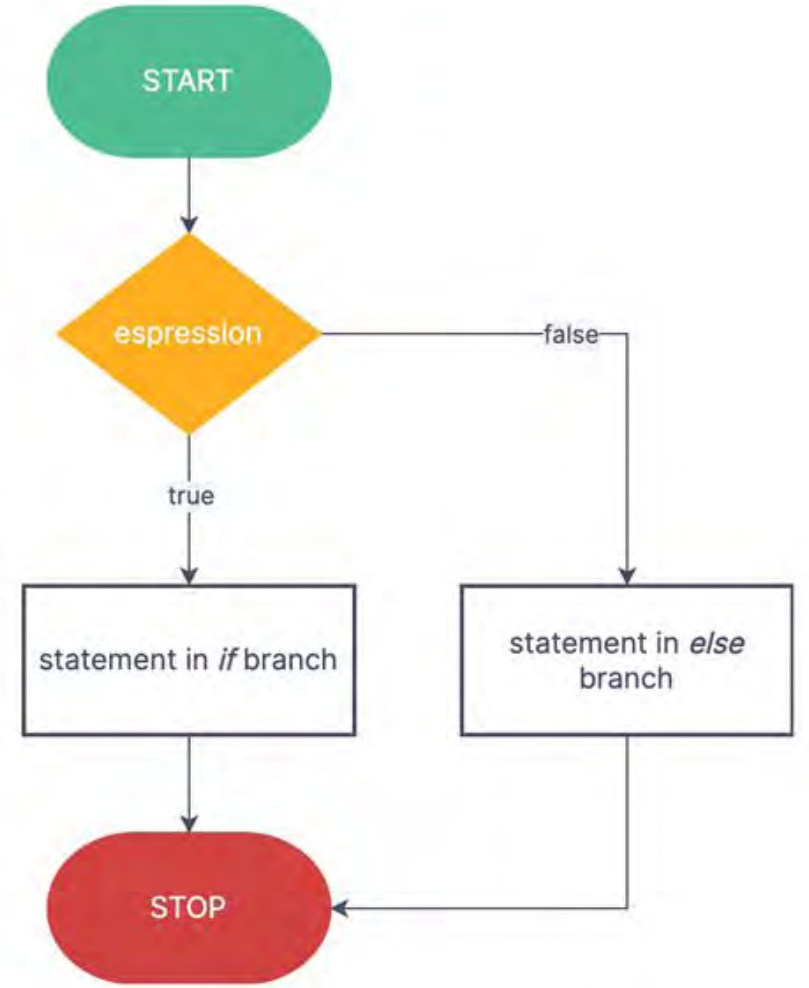
b. if-else Statement

The if-else statement provides an alternative block of code to execute if the condition is false. The else statement is used in conjunction with an if statement to specify a block of code that should be executed when the condition in the if statement is false.

Syntax

```
if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}
```

Flowchart



Example

```
int num = -5;
if (num > 0) {
    cout << "The number is positive." << endl;
} else {
    cout << "The number is not positive." << endl;
}
```

Here, since num is not greater than 0, the program will print "The number is not positive."

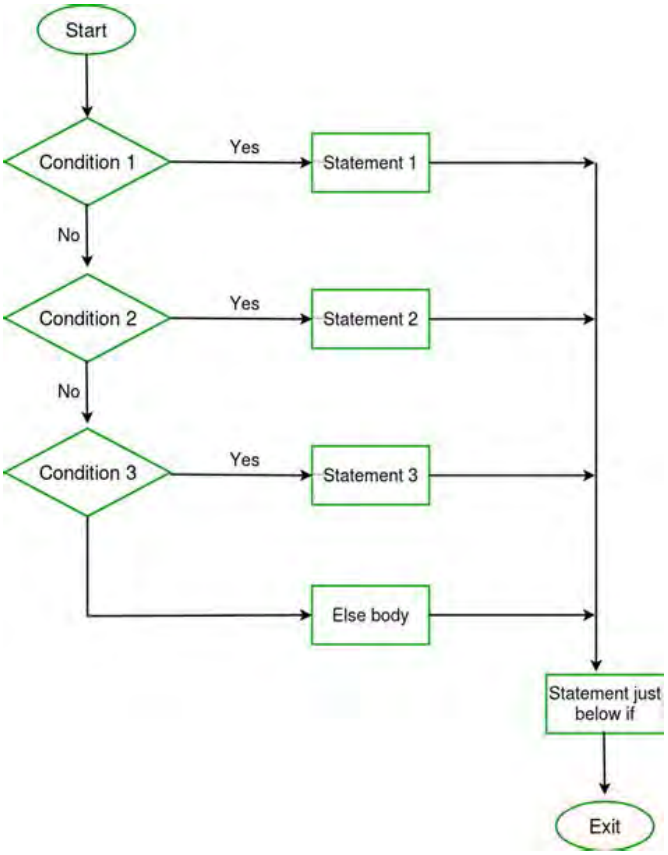
c. Else-if Statement

The else-if statement allows checking multiple conditions sequentially. It is used when there are more than two conditions to check.

Syntax

```
if (condition1) {
   // Code to execute if condition1 is true
} else if (condition2) {
   // Code to execute if condition2 is true
} else if (condition3) {
   // Code to execute if condition3 is true
} else {
   // Code to execute if none of the above conditions are true
}
```

Flowchart



Example

```
int num = 0;
if (num > 0) {
   cout << "The number is positive." << endl;
} else if (num < 0) {
   cout << "The number is negative." << endl;
} else {
   cout << "The number is zero." << endl;
}
```

Here, since num is 0, the program will print "The number is zero."

d. Switch Statement

The switch statement is an alternative to multiple if-else conditions when you need to compare a single variable against multiple values. It is used for more readable code when dealing with several possible values of a variable. It is used for multi-branch decision-making based on the value of an expression. It allows you to select one of many code blocks to be executed depending on the value of the expressions.

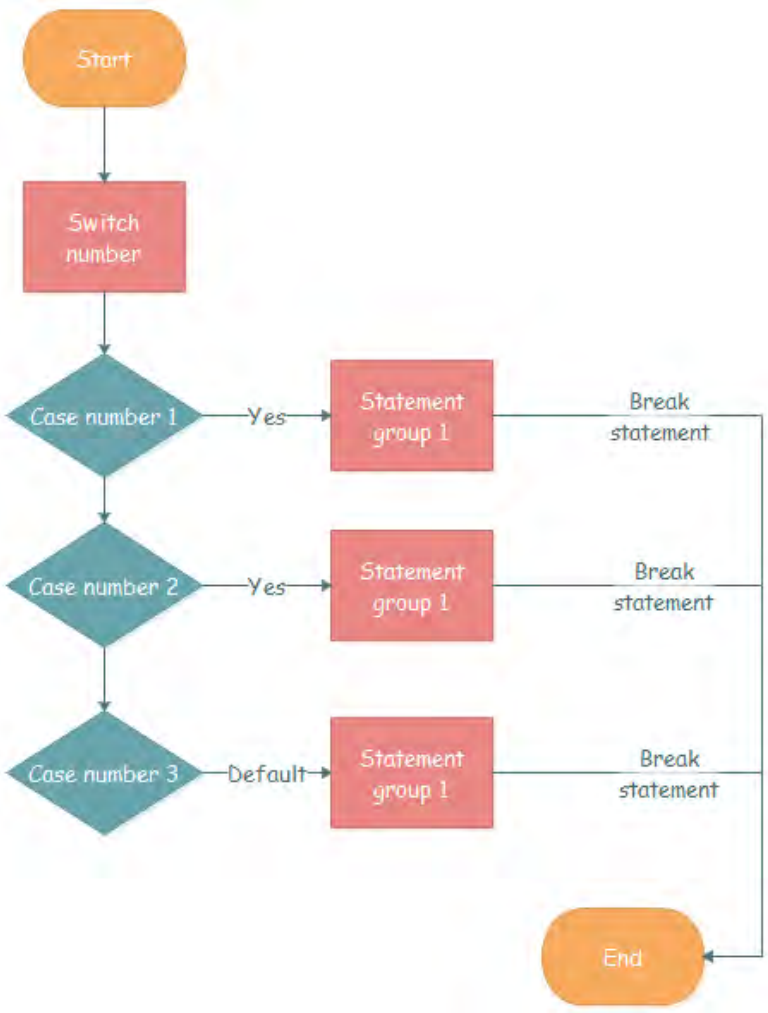
Syntax

```
switch (expression) {
   case value1:
      // Code to execute if expression == value1
      break;
   case value2:
      // Code to execute if expression == value2
      break;
   case value3:
      // Code to execute if expression == value3
      break;
   default:
      // Code to execute if no case matches
```

}

- The break statement is used to exit the switch block after a case is executed.
- The default case is optional, and it is executed if none of the case conditions match.

Flowchart



Example

```
int day = 3;
switch (day) {
   case 1:
```

```
        cout << "Monday" << endl;
        break;
    case 2:
        cout << "Tuesday" << endl;
        break;
    case 3:
        cout << "Wednesday" << endl;
        break;
    case 4:
        cout << "Thursday" << endl;
        break;
    case 5:
        cout << "Friday" << endl;
        break;
    case 6:
        cout << "Saturday" << endl;
        break;
    case 7:
        cout << "Sunday" << endl;
        break;
    default:
        cout << "Invalid day!" << endl;
}
```

In this example, since day = 3, the output will be "Wednesday."

Summary of Decision-Making Statements:

- **if**: Executes a block of code if the condition is true.

- **if-else**: Executes one block of code if the condition is true, and another block if the condition is false.
- **else-if** : Used for multiple conditions.
- **switch**: Used to compare a single expression to multiple values (cases).

Example Program Using Multiple Decision-Making Statements

```
#include <iostream>
using namespace std;
int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;
    // if-else statement
    if (num > 0) {
        cout << "The number is positive." << endl;
    } else if (num < 0) {
        cout << "The number is negative." << endl;
    } else {
        cout << "The number is zero." << endl;
    }
    // switch statement
    switch (num) {
        case 1:
            cout << "The number is one." << endl;
            break;
        case 2:
            cout << "The number is two." << endl;
```

```
            break;
        default:
            cout << "The number is neither one nor two." << endl;
    }
    return 0;
}
```

In this program, the user enters a number, and based on that input:

- The program will tell whether the number is positive, negative, or zero (using if-else).
- The program will also check if the number is 1 or 2 using a switch statement.

2. Looping Statement

In C++, looping statements allow you to execute a block of code multiple times based on certain conditions. They are essential for automating repetitive tasks, iterating over data structure, and controlling program flow. There are three primary types of loops in C++: for, while, and do-while. Below is an overview and example of each type:

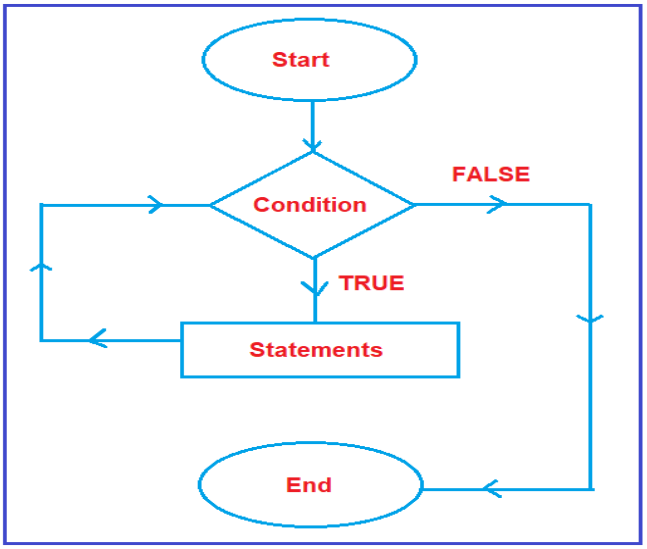
a. For Loop

The for loop is used to repeatedly execute a block of code a specified number of times. The for loop is typically used when the number of iterations is known beforehand. It includes initialization, condition checking, and increment/decrement in a single line.

Syntax

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

Flowchart



Example

```
#include <iostream>
using namespace std;
int main() {
   // Loop from 1 to 5
   for (int i = 1; i <= 5; i++) {
      cout << "Iteration " << i << endl;
   }
   return 0;
}
```

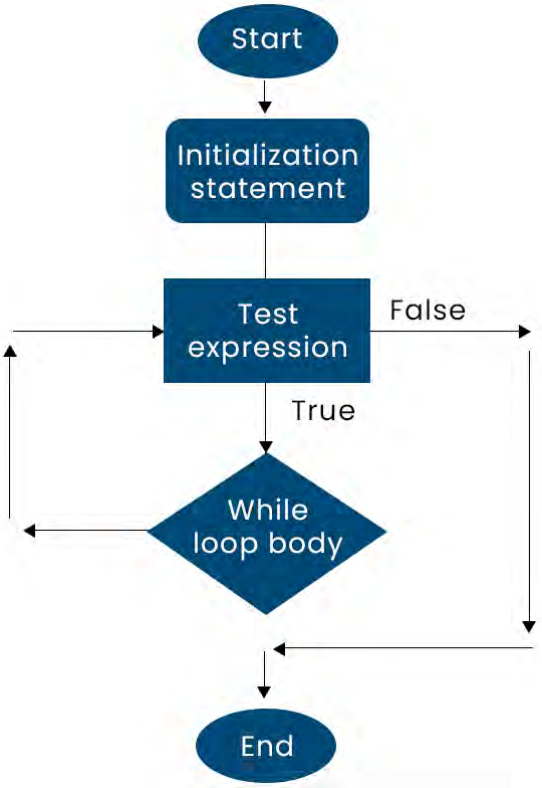
b. While Loop

The while loop is used to repeatedly execute a block of code as long as a specified condition remains true. It is a pre-test loop, which means that the condition is checked before the code block is executed. If the condition is false from the beginning, the code block inside the code will not execute at all.

Syntax

```
while (condition) {
   // Code to be executed
}
```

Flowchart



Example

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    // Loop while i is less than or equal to 5
    while (i <= 5) {
        cout << "Iteration " << i << endl;
        i++;  // Increment i after each iteration
    }
    return 0;
}
```

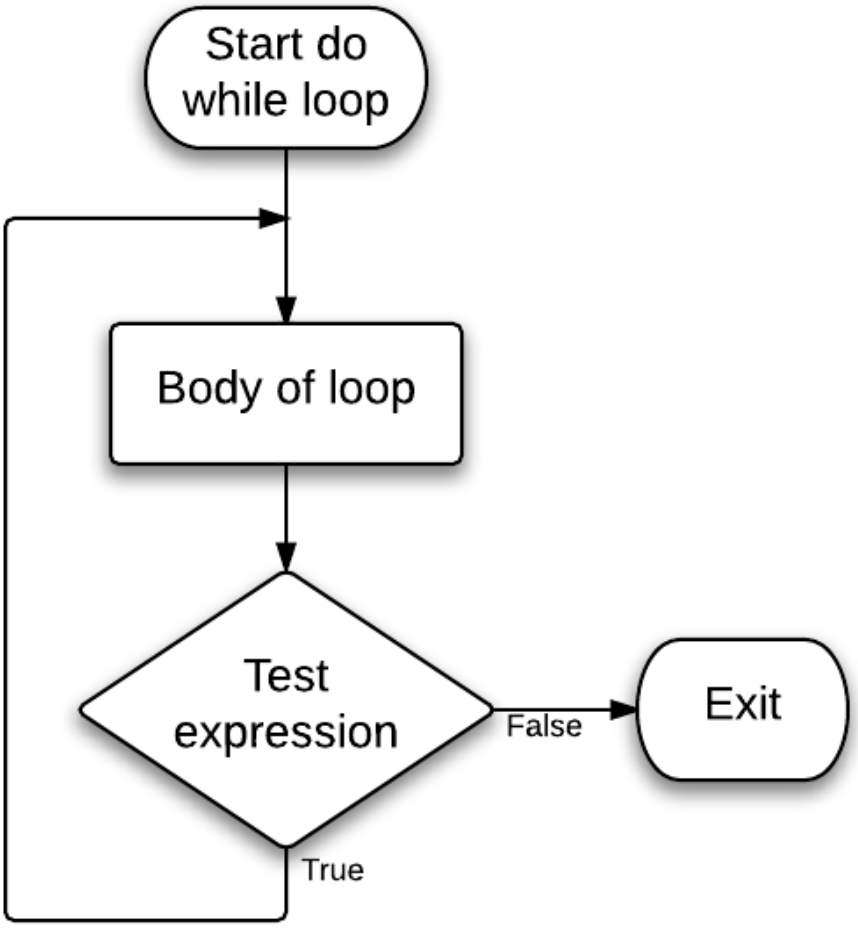
c. do-While Loop

The do-while loop guarantees that the code block will be executed at least once, regardless of the condition. It checks the condition after executing the code block. Unlike the while loop, the do-while loop guarantees that the code block is executed at least once before checking the condition. It is a post-test loop because the condition is checked after the code block is executed..

Syntax

```
do {
   // Code to be executed
} while (condition);
```

Flowchart



Example

```
#include <iostream>
using namespace std;
int main() {
   int i = 1;
   // Loop at least once and continue while i is less than or equal to 5
   do {
      cout << "Iteration " << i << endl;
      i++;
   } while (i <= 5);
   return 0;
}
```

3. Statements

In C++, **jump statements** are used to control the flow of execution by jumping to a different part of the program. These statements allow yo to manipulate the loop execution, skip certain iteration and exit the loops prematurely when the specified conditions are met. The primary jump statements in C++ are break, continue, goto, and return. Each serves a different purpose in altering the flow of control within loops, conditional blocks, or functions.

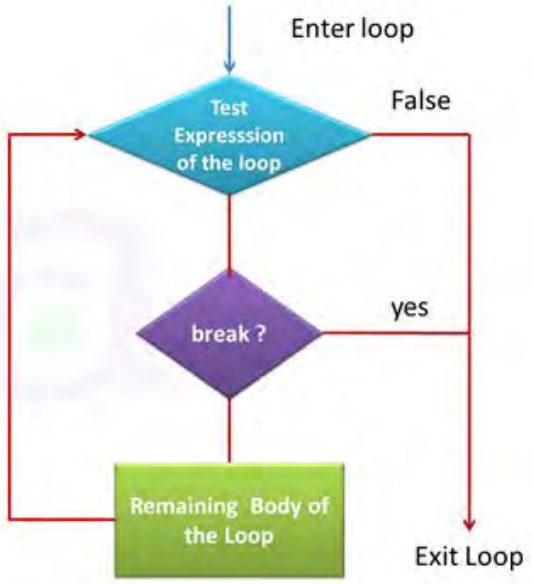
a. Break Statement

The break statement is used to exit from a loop or a switch statement prematurely, regardless of whether the loop or switch condition has been satisfied.

Usage

- **In loops**: Exits the loop entirely.
- **In switch statements**: Exits the switch case.

Flowchart



Example (Loop)

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break;  // Exit the loop when i equals 5
        }
        cout << i << " ";
    }
    return 0;
}
```

Output

1 2 3 4

Example (Switch)

```
#include <iostream>
using namespace std;
int main() {
   int x = 2;
   switch(x) {
      case 1:
         cout << "Case 1" << endl;
         break;
      case 2:
         cout << "Case 2" << endl;
         break;  // Break here exits the switch
      default:
         cout << "Default case" << endl;
   }
   return 0;
}
```

Output

Case 2

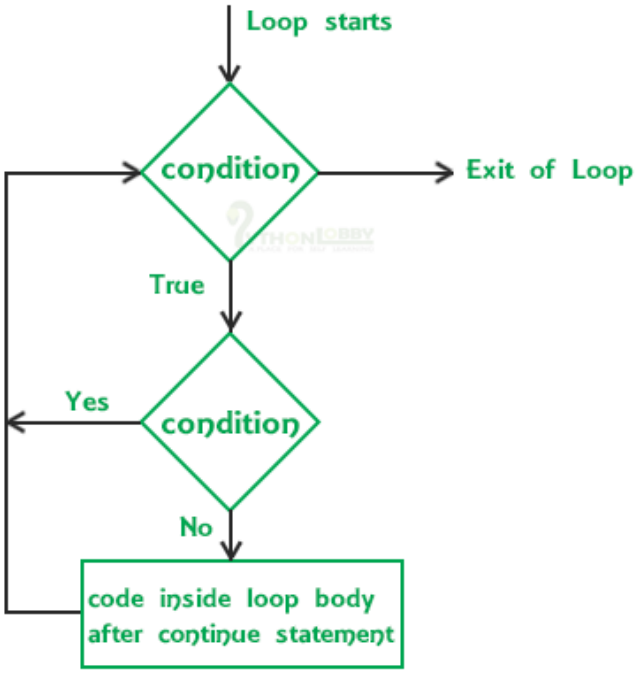
b. Continue Statement

The continue statement is used inside loops to skip the rest of the current iteration and move to the next iteration of the loop.

Usage

- **In loops**: Skips the current iteration and proceeds with the next iteration of the loop.

Flowchart



Example

```
#include <iostream>
using namespace std;
int main() {
   for (int i = 1; i <= 5; i++) {
      if (i == 3) {
         continue;  // Skip the iteration when i equals 3
      }
      cout << i << " ";
   }
   return 0;
}
```

Output

1 2 4 5

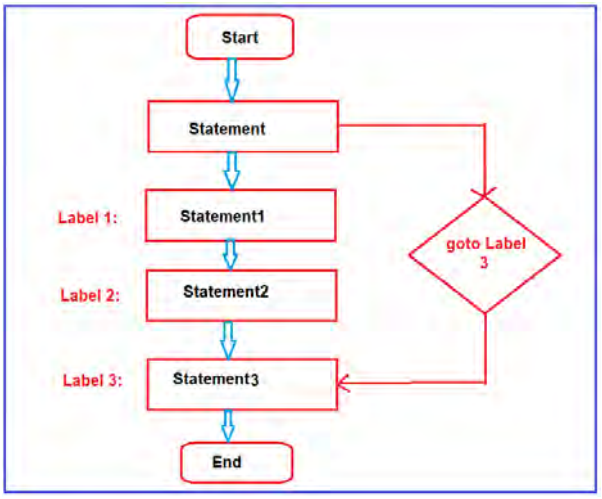
c. Goto Statement

The goto statement is used to jump to a labeled statement in the program. It is generally discouraged because it makes the code harder to understand and maintain, but it can be useful in certain situations where structured control flow (like loops and conditionals) isn't sufficient.

Usage

- **In any part of the program**: Jumps to a specific label defined elsewhere in the code.

Flowchart



Example

```
#include <iostream>
using namespace std;
int main() {
   int i = 1;
   loop_start:
      if (i > 5) {
         goto end;  // Jump to 'end' label when i exceeds 5
      }
      cout << i << " ";
```

```
        i++;
        goto loop_start;  // Jump back to 'loop_start' label
    end:
        cout << "\nEnd of loop";
    return 0;
}
```

Output

1 2 3 4 5

End of loop

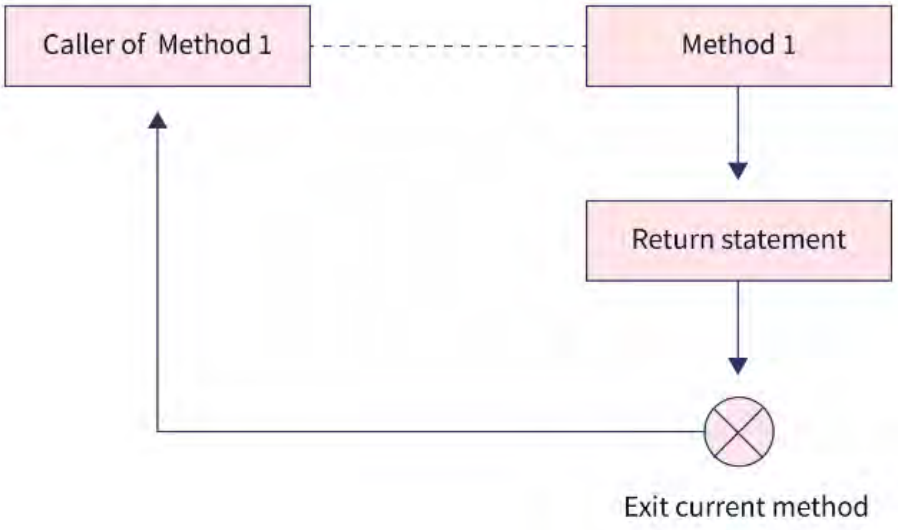
d. Return Statement

The return statement is used to exit from a function and optionally return a value to the calling function. It is primarily used in non-void functions to return a value, but in main(), it is used to indicate the program's exit status.

Usage

- **In functions**: Exits the function and optionally returns a value.
- **In main()**: Returns an integer exit code to the operating system (0 for successful execution).

Flowchart



Example

```
#include <iostream>
using namespace std;
int add(int a, int b) {
    return a + b;  // Return the sum of a and b
}

int main() {
    int result = add(3, 4);
    cout << "The sum is: " << result << endl;
    return 0;  // Indicates successful execution
}
```

Output

```
The sum is: 7
```

Exercise

Choose the correct answer from the given alternatives.

1. Which programming paradigm emphasizes the use of functions?

   a. Object-oriented programming
   c. Functional programming
   b. Declarative programming
   d. Event-driven programming

2. Which programming paradigm is based on the concept of a static machine?

   a. Object-oriented programming
   c. Functional programming
   b. Declarative programming
   d. Procedural programming

3. The ability of an object to expose its internal data and behavior to other objects is called:

   a. Objects
   c. Encapsulation
   b. Object
   d. Abstraction

4. The ability of an object to protect its internal data and behavior from other objects is called:

   a) Objects
   c. Polymorphism
   b) Encapsulation
   d. Abstraction

5. The ability of an object to take on many forms is called:

   a. Object
   c. Polymorphism
   b. Objects
   d. Inheritance

6. Which programming paradigm is based on the concept of instructions executed in order?

   a. Object-oriented programming
   c. Functional programming
   b. Declarative programming
   d. Procedural programming

7. What is polymorphism in object-oriented programming?

   a. The ability of an object to expose its internal properties and behavior to other objects
   b. The ability of an object to take many forms

c. A sequence of characters that represents a single unit of syntax

d. A variable that holds a value

8. What is a token in programming?

a. A sequence of characters that represents a single unit of syntax

b. A variable that holds a value

c. A programming construct that encapsulates data and behavior

d. A programming language keyword

9. Which data type is used to represent whole numbers in programming?

a. Float

b. Double

c. Integer

d. Character

10. Which data type is used to represent true or false values in programming?

a. Float

b. Double

c. Integer

d. Boolean

Write short answer to the following questions.

1. What is Object-Oriented Programming (OOP)?
2. What are the key features of OOP?
3. What are some applications of OOP?
4. What is the difference between structured and object-oriented programming?
5. What are tokens and character sets in programming, and why are they important in programming?
6. What is basic input/output in programming?
7. What is the basic structure of a program in programming?
8. What are control statements and how are they used in programming?
9. What is encapsulation and why is it important in OOP?
10. How does inheritance work in OOP and what are its benefits?
11. What is polymorphism and how is it implemented in OOP?
12. What are some examples of OOP programming languages?
13. What are some best practices for OOP programming?

14. What are the advantages of OOP over other programming paradigms?

15. What is an object in OOP and how is it different from a class?

Write long answer to the following questions.

1. What is Object-Oriented Programming (OOP)? Explain the main principles of OOP and how they differ from other programming paradigms.

2. What are the key features of OOP? Discuss each feature in detail and provide examples of how they are used in programming.

3. What are some applications of OOP? Explain how OOP is used in software development and provide examples of OOP-based systems.

5. What are the difference between structured and object-oriented programming?

6. What are tokens and character sets in programming languages, and provide examples of different types of tokens and character sets.

7. What are data types and why are they important in programming? Discuss the different data types used in programming, their classification, and how they are used in programming.

8. What is basic input/output in programming? Explain the input/output operations in programming and provide examples of how they are used in different programming languages.

9. What is the basic structure of a program in programming? Discuss the different components of a basic program, including headers, functions, and statements.

10. What are control statements and how are they used in programming? Explain the different types of control statements, including if-else statements, for loops, while loops, and switch statements, and provide examples of their use in programming.

11. What are some best practices for OOP programming? Discuss the importance of encapsulation, inheritance, and polymorphism in OOP, and provide examples of best practices for designing and implementing OOP-based systems.

Project Work

1. Write a program that uses for loop to print the numbers from 1 to 10.

2. Write a program that uses while loop to print the even numbers from 2 to 10.

3. Write a program that uses do-while loop to repeatedly ask the user to enter a number and then print out whether the number is even or odd until the user enters 0.

4. Write a program that uses for loop to calculate the sum of the first 10 natural numbers.
5. Write a program that uses while loop to calculate the product of the first 5 natural numbers.
6. Write a program that uses do-while loop to repeatedly ask the user to enter a number and then print out whether the number is even or odd until the user enters 0.
7. Write a program that takes an integer input from the user and checks if it's positive, negative, or zero using if-else ladder statement.
8. Write a program that takes character input from the user and checks if it's a vowel, consonant, digit, or special character using if-else ladder statement.
9. Write a program that takes character input from the user and checks whether it is a vowel or consonant using switch statement.
10. Write a program that takes letter grade as input from the user and displays the corresponding GPA value using a switch statement.
11. Write a program that takes user input of a number and checks if it's divisible by 5, 3, 2, or not using a switch statement.
12. Write a program to check if a given year is a leap year or not using if-else statement.
13. Write a program to check if a given number is even or odd using if-else statement.
14. Write a program to find the largest of three numbers using if-else statement.
15. Write a program to calculate the grade of a student based on their marks using if-else statement.

Class and Objects

Unit 3

3.1 Introduction to Class and Objects

Class

In C++, a class is a user-defined data type that can hold both data (attributes) and functions (methods) that operate on that data. It serves as a blueprint for creating objects. A class defines the properties (variables) and behaviors (functions) that the objects of that class will have.

Class Declaration Syntax

```
class class_name{

//Access Specifiers

        //Data Member

        //Member Function

}
```

Key Points:

1. **Access Modifiers**
   - **private**: Members are not accessible outside the class.
   - **public**: Members are accessible outside the class.
   - **protected**: Members are accessible to derived classes.
2. **Methods: Functions defined inside the class to manipulate the data members.**
3. **Data Members: Variables that hold the data of an object.**

Objects

In C++, an **object** is an instance of a class. It is a concrete representation of the class, where the class defines the properties (data members) and behaviors (member functions), and the object holds specific values for those properties. When you create an object, the constructor of the class is called, and the object is initialized based on

the constructor's logic.

Object Initialization and Declaration Syntax:

1. Declaring and Initializing an Object

An object can be declared and initialized in various ways:

```
// Syntax to declare an object

ClassName objectName;              // Default object
ClassName objectName(arguments);   // Object with parameters passed to constructor
```

Where:

- ClassName: The name of the class.
- objectName: The name of the object being created.
- arguments: Arguments passed to a constructor if it's a parameterized constructor.

2. Types of Initialization

- **Default Initialization** (using default constructor): If the class has a default constructor (no parameters or all parameters have default values), an object can be initialized like this:

  ```
  ClassName object Name;
  ```

- **Parameterized Initialization** (using parameterized constructor): If the class has a constructor that accepts parameters, you can initialize the object with values during declaration:

  ```
  ClassName objectName(arg1, arg2);
  ```

3.2 Access Specifier

In C++, **access specifier** define the visibility and accessibility of class members (both data members and member functions). These access control mechanisms ensure that the class can enforce encapsulation by controlling how and where its members can be accessed or modified.

C++ provides three primary access specifiers:

1. Public Access Specifier

- Members declared as public are accessible from anywhere in the program (both

inside and outside the class).

- Used for functions or variables that are intended to be accessed directly by other parts of the program.
- These access specifier are written using public keyword.

Syntax for Public Access Specifiers:

```
class ClassName {

public:
   // public member
};
```

Here is the example to illustrate public access specifier

```
#include<iostream>
using namespace std;
// class definition
class Circle
{
   public:
      double radius;
      double  compute_area()
      {
         return 3.14*radius*radius;
      }
   };
// main function
int main()
{
   Circle obj;
    // accessing public datamember outside class
```

```
    obj.radius = 5.5;
    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

Output

```
Radius is: 5.5
Area is: 94.985
```

2. Private access specifier

- Members declared as private are **not** accessible from outside the class. They can only be accessed by the class's own member functions or friends.
- Used to protect sensitive data or implementation details from being accessed or modified directly by other parts of the program.
- These access specifier are written using private keyword.

Syntax for Private Access Specifiers:

```
class ClassName {
private:
    // private members
};
```

Here is the example to illustrate private access specifier

```
#include <iostream>
using namespace std;
class Circle {
private:
    double radius;  // Private data member
public:
    // Setter function to assign value to private member
    void setRadius(double r) {
```

```
        radius = r;
    }
    // Public function to compute area
    double compute_area() {
        return 3.14 * radius * radius;
    }
};
int main() {
    Circle obj;
    // Setting radius using setter function
    obj.setRadius(1.5);
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

Output

```
Area is: 7.065
```

3. protected access specifer:

- Members declared as protected are not accessible outside the class, but they can be accessed by **derived classes** (i.e., classes that inherit from the base class).
- This is useful when you want to allow derived classes to access or modify certain data members or methods while keeping them hidden from the outside world.

Syntax

```
class ClassName {
protected:
    // protected members
};
```

Here is the example to illustrate protected access specifier

```
#include <iostream>
```

```
using namespace std;
class Shape {
protected:
    double length, width;  // Protected members
public:
    // Constructor to initialize protected members
    void setValues(double l, double w) {
        length = l;
        width = w;
    }
};
// Derived class
class Rectangle : public Shape {
public:
    // Function to compute area (can access protected members)
    double compute_area() {
        return length * width;
    }
};
int main() {
    Rectangle rect;
    // Setting values using public method
    rect.setValues(5.0, 3.0);
    // Calling function to compute area
    cout << "Area of Rectangle: " << rect.compute_area() << endl;
    return 0;
}
```

Output

Area of Rectangle: 15

Why Use Access Specifiers?

Access specifiers are essential for encapsulation, a key object-oriented principle. By controlling access to class members, you can:

- Hide the internal implementation details.
- Ensure that objects remain in a valid state by restricting how data is modified.
- Provide well-defined interfaces (through public functions) for interacting with the object.

Access specifiers help you control the visibility and accessibility of your class members, providing a level of protection and ensuring the integrity of the object's state.

3.3 Declaration of Class and Objects

In C++, **classes** and **objects** are fundamental concepts of **object-oriented programming (OOP)**.

- **Class**: A blueprint or template for creating objects, defining their properties (data members) and behaviors (member functions).
- **Object**: An instance of a class, where the class's properties and behaviors are instantiated with specific values.

1. Declaration of a Class in C++

A class in C++ is declared using the class keyword, followed by the class name and a body enclosed in curly braces {}. The body of the class contains data members (variables) and member functions (methods).

Syntax for Declaring a Class

```
class ClassName {
   // Access specifier (public, private, protected)
      // Data members (attributes)
public:
   // Member functions (methods)
   void setData(int, string);  // Method to set values
   void display() const;  // Method to display data
```

```
private:
    // Data members (attributes)
    int data1;
    string data2;
};
```

2. Declaration and Initialization of Objects

Once a class is defined, you can create (instantiate) objects of that class. When you declare an object, the constructor of the class is called, and the object is initialized accordingly.

Syntax for Declaring and Initializing Objects:

```
ClassName objectName;              // Default initialization
ClassName objectName(arguments);   // Parameterized initialization
```

Example Declaring a Class and Creating Objects

```
#include <iostream>
using namespace std;
// Class declaration
class Car {
private:
    string brand;
    int year;
public:
    // Function to set values
    void setCar(string b, int y) {
        brand = b;
        year = y;
    }
    // Function to display values
    void showCar() {
```

```
        cout << "Car Brand: " << brand << ", Year: " << year << endl;
    }
};
int main() {
    Car car1, car2; // Creating objects of class Car
    // Setting values using function
    car1.setCar("Toyota", 2020);
    car2.setCar("Honda", 2022);
    // Displaying values
    car1.showCar();
    car2.showCar();
    return 0;
}
```

Output

```
Car Brand: Toyota, Year: 2020
Car Brand: Honda, Year: 2022
```

Difference Between Class and Objects

Feature	Class	Object
Definition	Class is a blueprint or template for creating objects.	Object is an instance of a class.
Purpose	It is defines properties (attributes) and behaviors (methods).	It represents a specific entity with those properties and behaviors.
Memory Allocation	No memory is allocated until an object is created.	Memory is allocated when an object is instantiated from a class.
Example Syntax	class Car { ... };	Car myCar.
Accessibility	Members can have access specifiers: public, private, or protected.	Accesses public members directly but follows encapsulation rules.

Type	A user-defined data type	A variable of that data type
Lifespan	Exists in the program code; loaded at compile time.	Exists during runtime and has a specific lifecycle (until it goes out of scope or deleted).
Identity	Provides the structure for multiple objects.	Unique identity for each instance created.

3.4 Class Methods and Data Members

In C++, **class methods** and **data members** are the building blocks of object-oriented programming (OOP). These components define the behavior and attributes of an object, respectively.

1. Data Members

Data members are **variables** that hold the state or attributes of an object. They represent the properties of the object, and each object of the class has its own copy of the data members (unless they're declared static).

Syntax for Data Members

```
class ClassName {
   // Access specifier: private, protected, or public
private:
   // Data members (variables)
   int data1;
   double data2;
   string name;

public:
   // Methods (functions)
};
```

- **Private Data Members**: These are typically used to protect the internal state of an object. They can only be accessed and modified through member functions

(getters and setters).

- **Public Data Members**: These can be accessed directly from outside the class, but this is generally not recommended because it violates encapsulation.

2. Member Functions (Methods)

Member functions (methods) define the behavior or actions that an object can perform. These functions are tied to a specific object, and they can access and manipulate the data members of the class.

Example of Class with Data Members and Member Functions

```
#include <iostream>
using namespace std;
class Person {
private:
   string name;  // Data member for storing the name
   int age;      // Data member for storing the age
public:
   // Member function to set name and age
   void setData(string n, int a) {
      name = n;
      age = a;
   }
   // Member function to display person details
   void displayData() {
      cout << "Name: " << name << "\nAge: " << age << endl;
   }
};
int main() {
   Person person1;  // Create an object of Person class
   // Set data for person1
   person1.setData("Alice", 25);
```

```
    // Display person1 details
    person1.displayData();
    return 0;
}
```

Output

```
Name: Alice
Age: 25
```

3.5 Concept of Constructors and Destructors

A constructor is a special member function of a class and shares the same name as of class, which means the constructor and class have the same name. Constructor is called by the compiler whenever the object of the class is created, it allocates the memory to the object and initializes class data members by default values or values passed by the user while creating an object. Constructors don't have any return type because their work is to just create and initialize an object.

Basic syntax of constructor

```
class class_name{
 private:
 // private members
 public:
 // declaring constructor
 class_name({parameters})
 {
  // constructor body
 }
};
```

1. Basics of Constructors

- **Name**: The constructor has the same name as the class.
- **No Return Type**: Constructors do not have a return type (not even void).
- **Automatic Call**: They are automatically called when an object is created.

- **Overloading**: Constructors can be overloaded, meaning a class can have multiple constructors with different parameter lists.

2. Types of Constructors

There are several types of constructors in C++:

a) Default Constructor

Default constructor is also known as a zero-argument constructor, as it doesn't take any parameter. It can be defined by the user if not then the compiler creates it on his own. Default constructor always initializes data members of the class with the same value they were defined.

Syntax:

```
class class_name{
  private:
  // private members
  public:
  // declaring default constructor
  class_name()
  {
    // constructor body
  }
};
```

Example

```
#include <iostream>
using namespace std;
class Person{
  // declaring private class data members
private:
  string name;
  int age;
```

```
public:
 // declaring constructor
 Person()
 {
    cout<<"Default constructor is called"<<endl;
    name = "student";
    age = 12;
 }
 // display function to print the class data members value
 void display()
 {
    cout<<"Name of current object: "<<name<<endl;
    cout<<"Age of current object: "<<age<<endl;
 }
};
int main()
{
   // creating object of class using default constructor
   Person obj;
   // printing class data members
   obj.display();
   return 0;
}
```

Output

```
Default constructor is called
Name of current object: student
Age of current object: 12
```

b) Parameterized Constructor

A **parameterized constructor** takes arguments and allows different initial values for objects.

Syntax

```
class class_name{
 private:
 // private members
 public:
 // declaring parameterized constructor
 class_name(parameter1, parameter2,...)
 {
  // constructor body
 }
};
```

Example

```
#include <iostream>
using namespace std;
class Person{
 // declaring private class data members
private:
 string name;
 int age;
public:
 // declaring parameterized constructor of three different types
 Person(string person_name)
 {
    cout<<"Constructor to set name is called"<<endl;
```

```
    name = person_name;
    age = 12;
  }
  Person(int person_age)
  {
    cout<<"Constructor to set age is called"<<endl;
    name = "Student";
    age = person_age;
  }
  Person(string person_name, int person_age)
  {
    cout<<"Constructor for both name and age is called"<<endl;
    name = person_name;
    age = person_age;
  }
  // display function to print the class data members value
  void display()
  {
    cout<<"Name of current object: "<<name<<endl;
    cout<<"Age of current object: "<<age<<endl;
    cout<<endl;
  }
};
int main()
{
   // creating objects of class using parameterized constructor
   Person obj1("First person");
```

```
    // printing class data members for first object
    obj1.display();
    Person obj2(25);
    // printing class data members for second object
    obj2.display();
    Person obj3("Second person",15);
    // printing class data members for third object
    obj3.display();
    return 0;
}
```

Output

```
Constructor to set name is called
Name of current object: First person
Age of current object: 12
Constructor to set age is called
Name of current object: Student
Age of current object: 25
Constructor for both name and age is called
Name of current object: Second person
Age of current object: 15
```

c) Copy Constructor

A **copy constructor** initializes an object using another object of the same class. It's used for deep copying if an object contains pointers or dynamically allocated resources.

Syntax

```
class class_name{
  private:
  // private members
```

```
  public:
  // declaring copy constructor
  class_name(const class_name& obj)
  {
    // constructor body
  }
};
```

Example

```
#include <iostream>
using namespace std;
class Person{
  // declaring private class data members
private:
  string name;
  int age;
public:
  Person(string person_name, int person_age)
  {
     cout<<"Constructor for both name and age is called"<<endl;
     name = person_name;
     age = person_age;
  }
  Person(const Person& obj)
  {
     cout<<"Copy constructor is called"<<endl;
     name = obj.name;
     age = obj.age;
```

```
    }
    // display function to print the class data members value
    void display()
    {
        cout<<"Name of current object: "<<name<<endl;
        cout<<"Age of current object: "<<age<<endl;
        cout<<endl;
    }
};
int main()
{
    // creating objects of class using parameterized constructor
    Person obj1("First person",25);
    // printing class data members for first object
    obj1.display();
    // creating copy of the obj1
    Person obj2(obj1);
    // printing class data members for second object
    obj2.display();
    return 0;
}
```

Output

```
Constructor for both name and age is called
Name of current object: First person
Age of current object: 25
Copy constructor is called
Name of current object: First person
Age of current object: 25
```

Destructor

In C++, a **destructor** is a special member function of a class that is automatically invoked when an object goes out of scope or is explicitly deleted. Destructors are mainly used for cleanup, such as releasing resources (e.g., memory, file handles) that the object may have acquired during its lifetime. Here's an overview of destructors, their syntax, usage, and behavior:

Syntax of Destructors

```
class class_name{
  private:
  // private members
  public:
  // declaring destructor
  ~class_name()
  {
    // destructor body
  }
};
```

Example of Destructor

```
#include <iostream>
using namespace std;
class class_name{
  // declaring private class data members
private:
  int a,b;
public:
  // declaring Constructor
  class_name(int aa, int bb)
  {
```

```
        cout<<"Constructor"<<endl;
        a = aa;
        b = bb;
        cout<<"Value of a: "<<a<<endl;
        cout<<"Value of b: "<<b<<endl;
        cout<<endl;
    }
    // declaring destructor
    ~class_name()
    {
        cout<<"Destructor"<<endl;
        cout<<"Value of a: "<<a<<endl;
        cout<<"Value of b: "<<b<<endl;
    }
};
int main()
{
    // creating objects of class using parameterized constructor
    class_name obj(5,6);
    return 0;
}
```

Output

```
Constructor
Value of a: 5
Value of b: 6
Destructor
Value of a: 5
Value of b: 6
```

Important Points about the Destructor

- Destructors are the last member function called for an object and they are called by the compiler itself.
- If the destructor is not created by the user then compile creates or declares it by itself.
- A Destructor can be declared in any section of the class, as it is called by the compiler so nothing to worry about.
- As Destructor is the last function to be called, it should be better to declare it at the end of the class to increase the readability of the code.
- Destructor is just the opposite of the constructor as the constructor is called at the time of the creation of the object and allocates the memory to the object, on the other side the destructor is called at the time of the destruction of the object and deallocates the memory.

Exercise

Choose the correct answer from the given alternatives.

1. What is the basic unit of object-oriented programming?

   a. Object
   b. Class
   c. Function
   d. Variable

2. What is the access specifier in C++?

   a. A keyword used to define visibility of class members
   b. A keyword used to define loops
   c. A keyword used to define data types
   d. A keyword used to define conditional statements

3. Which access specifier in C++ makes class members accessible only from within the class?

   a. Public
   b. Private
   c. Protected
   d. None of the above

4. Which access specifier in C++ makes class members accessible from the class and its derived classes?

   a. Public
   b. Private
   c. Protected
   d. None of the above

5. Which keyword is used to declare a class in C++?

   a. Class
   b. Define
   c. Type
   d. Category

6. Which keyword is used to declare an object in C++?

   a. Class
   b. Define
   c. Type
   d. None of the above

7. Which operator is used to access the members of a class?

   a. Dot (.)
   b. Arrow (->)
   c. Double colon (::)
   d. None of the above

8. Which method is used to initialize an object in C++?

a. Constructor
c. Virtual method
b. Destructor
d. None of the above

9. Which method is used to free the memory allocated to an object in C++?

a. Constructor
c. Virtual method
b. Destructor
d. None of the above

10. Which type of constructor does not take any arguments?

a. Parameterized constructor
c. Copy constructor
b. Default constructor
d. None of the above

11. Which type of constructor takes arguments?

a. Parameterized constructor
c. Copy constructor
b. Default constructor
d. None of the above

12. Which type of constructor is used to create a copy of an object?

a. Parameterized constructor
c. Copy constructor
b. Default constructor
d. None of the above

13. Which operator is used to create a dynamic object in C++?

a. New
c. Both a and b
b. Delete
d. None of the above

14. Which operator is used to free the memory allocated to a dynamic object in C++?

a. New
c. Both a and b
b. Delete
d. None of the above

15. Which function is called automatically when an object is destroyed?

a. Constructor
c. Virtual method
b. Destructor
d. None of the above

16. Which function is called automatically when an object is created?

a. Constructor
c. Virtual method
b. Destructor
d. None of the above

Write short answer to the following questions.

1. What is a class in C++ and how is it different from a structure?
2. What are the advantages of using object-oriented programming in C++?
3. What are the three access specifiers in C++ and how do they affect class members?
4. What is the purpose of a constructor in C++ and how is it declared?
5. What is the difference between a default constructor and a parameterized constructor in C++?
6. How is a destructor declared in C++ and when is it called?
7. What are data members in a class and how do they differ from member functions?
8. How can you access data members and member functions of a class in C++?
9. What is the difference between public, private, and protected access specifiers in C++?
10. How do you declare an object of a class in C++?
11. What is a copy constructor in C++ and how is it different from a regular member function?
12. How do you declare a default constructor in C++?
13. How is a destructor in C++ declared and when is it called?
14. How do you declare a parameterized constructor in C++?
15. How do you declare a copy constructor in C++?
16. What are data members and member functions in a C++ class?
17. What is the role of the public access specifier in C++ classes?

Write long answer to the following questions.

1. Explain the concept of object-oriented programming and how it is implemented in C++ with the help of classes and objects.
2. Discuss the various access specifiers in C++ and how they affect the accessibility of class members.
3. Define a class in C++ and explain the syntax for declaring data members and member functions in C++. Explain with an example.
4. What are class methods in C++? Explain with an example.

5. How does the concept of data encapsulation help in object-oriented programming? Explain with an example.
6. Explain the difference between public, private, and protected access specifiers in C++.
7. What is a constructor in C++? Explain its syntax and how it is used to initialize objects.
8. Explain the concept of a destructor in C++ and how it is used to free memory allocated to an object.
9. Discuss the various types of constructors and destructors in C++ and their syntax with the help of examples.
10. Explain the importance of constructors and destructors in C++ and also discuss how they contribute to the overall functionality of a program.

Project Work

1. Write a program that demonstrates the use of public, private, and protected access specifiers in a class. Declare a class "Student" with private data members like "name" and "age". Declare a public method "display" to display the name and age of the student. Demonstrate the use of public, private, and protected access specifiers in the program.
2. Write a program that declares a class named "Circle" with data members like "radius", "diameter", and "circumference". Declare methods like "calculate Radius", "calculateDiameter", and "calculateCircumference" to calculate the radius, diameter, and circumference of the circle. Finally, create an object of the "Circle" class and display the values of its data members.
3. Write a program that demonstrates the use of public, private, and protected access specifiers in a class. Declare a class "Student" with private data members like "name" and "age". Declare a public method "display" to display the name and age of the student. Demonstrate the use of public, private, and protected access specifiers in the program.
4. Write a program that declares a class named "Circle" with data members like "radius", "diameter", and "circumference". Declare methods like "calculateRadius", "calculateDiameter", and "calculateCircumference" to calculate the radius, diameter, and circumference of the circle. Finally, create an object of the "Circle" class and

display the values of its data members.

5. Write a program to create a class for a "Person" with data members like "name" and "age". Declare a constructor to initialize the data members of the class. Finally, create an object of the "Person" class and display its data members.
6. Write a program that demonstrates the use of objects in a class. Declare a "Rectangle" class with data members like "length" and "breadth". Declare a method "calculateArea" to calculate the area of the rectangle. Finally, create objects of the "Rectangle" class and display the area of the rectangle.
7. Write a program that creates a constructor and a destructor for a class. Declare a class "Person" with data members like "name" and "age". Declare a constructor to initialize the data members of the class. Also, declare a destructor to destroy the object of the class. Finally, create an object of the "Person" class and display its data members.
8. Write a program to create a class for a student, with data members for roll number, and marks in three subjects. Also, create methods for student details and calculating the total marks.
9. Write a program to create a class for a bank account, with data members for account number, account holder name, and account balance. Also, create methods for depositing and withdrawing money, and for displaying account details.

Unit 4 Abstraction and Encapsulation

4.1 Introduction to Abstraction

Abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and ignoring the details. Data abstraction refers to the process of providing only essential information about the data to the outside world, ignoring unnecessary details or implementation.

It has two types

i. **Data abstraction –** This type only shows the required information about the data and ignores unnecessary details.

ii. **Control Abstraction –** This type only shows the required information about the implementation and ignores unnecessary details.

4.2 Advantages of Abstraction

i. Simplifies interfaces and reduces complexity.

ii. Promotes modularity and independent development.

iii. Improves code maintainability and flexibility.

iv. Increases reusability and encourages code reuse.

v. Enhances security through encapsulation.

vi. Supports polymorphism and extensibility, allowing for scalable design.

vii. Facilitates efficient teamwork by enabling developers to work on different parts independently.

4.3 Achieve Abstraction Using C++ Program

Abstraction can be achieved by using different method, which can be discussed below:

i. Abstraction Using Classes

We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can

decide which data member will be visible to the outside world and which is not. For example:

```
#include <iostream>
using namespace std;
class Addition {
private:
   int num1, num2;  // Private data members
public:
   void setNumbers(int a, int b) {  // Public method to set values
      num1 = a;
      num2 = b;
   }
   int getSum() {  // Public method to return sum
      return num1 + num2;
   }
};
int main() {
   Addition obj;
   obj.setNumbers(10, 20);  // Setting values using a method
   cout << "Sum: " << obj.getSum() << endl;  // Displaying the sum
   return 0;
}
```

Output

Sum:30

ii. Abstraction in Header Files

One more type of abstraction in C++ can be header files. For example, pow() function available used to calculate the power of a number without actually knowing which algorithm function use to calculate the power. Thus we can say that header files hides all the information or details from the user. For example:

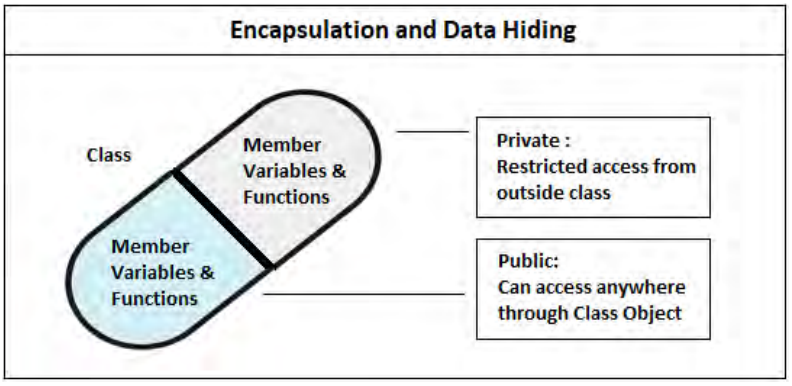
```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
float pie=3.14, r, area;
cout<< “Enter the radius:”;
cin>>r;
area= pie * pow(r,2);
cout << "The area of the circle is: " << area << endl;
return 0;
}
```

Output

```
Enter the radius:5
The area of the circle is: 78.5
```

4.4 Introduction to Encapsulation

Encapsulation is a fundamental concept in object-oriented programming (OOP) and is widely used in C++. It refers to the bundling of data and functions that operate on the data into a single unit called a "class." Encapsulation helps control access to the internal state of an object and provides a way to hide the complexity and details of how an object is implemented.



4.5 Advantages of Encapsulation

i. Encapsulation restricts direct access to data, ensuring it can't be modified or misused.

ii. Getter and setter methods validate data, preventing invalid changes.

iii. Self-contained classes are easier to maintain, modify, and test.

iv. Encapsulated classes can be reused in different programs without changes.

v. Internal details are hidden, simplifying interaction with the class.

vi. Encapsulation provides better control over data modifications, ensuring consistency.

4.6 Achieve Encapsulation Using C++ program

Encapasulation can be achieved with the help of getter and setter methods. For example

```
#include <iostream>
using namespace std;
class sampleData {
private:
   int num; // Private variable
   char ch; // Private variable
public:
   // Getter methods
   int getInt() {
      return num; // Return the value of num
   }
   char getCh() {
      return ch; // Return the value of ch
   }
   // Setter methods
   void setInt(int n) {
      num = n; // Set the value of num
```

```
    }
    void setCh(char c) {
        ch = c; // Set the value of ch
    }
};
int main() {
    sampleData s; // Create an object of sampleData
    s.setInt(100); // Set num to 100
    s.setCh('Z'); // Set ch to 'Z'
    // Output the values using getter methods
    cout << "num = " << s.getInt() << endl;
    cout << "ch = " << s.getCh() << endl;
    return 0;
}
```

Output

```
num = 100
ch = Z
```

Exercise

Choose the correct answer from the given alternatives.

1. What is abstraction in C++?

   a. Hiding data and showing only necessary details.

   b. Hiding both data and implementation details.

   c. Showing all implementation details.

   d. Making all data public.

2. Which of the following is NOT a type of abstraction?

   a. Data Abstraction
   b. Control Abstraction
   c. Functional Abstraction
   d. None of the above

3. Which of the following access specifiers help achieve abstraction in C++?

   a. Public
   b. Private
   c. Both a and b
   d. None of the above

4. What is the main purpose of encapsulation?

   a. To hide implementation details and protect data.
   b. To increase code execution speed.
   c. To make all data public for easy access.
   d. To allow global access to variables.

5. Which keyword is used to define private data members in a class?

   a. Protected
   b. Private
   c. Public
   d. Static

6. How can encapsulation be achieved in C++?

   a. Using getter and setter methods.
   b. Using friend functions.
   c. Using only public variables.
   d. Using global variables.

7. Which function in the <cmath> library is used for calculating power in C++?

   a. Power()
   b. Sqrt()
   c. Pow()
   d. Exp()

8. Which of the following is NOT an advantage of encapsulation?

   a. Ensures data security.
   b. Reduces code complexity.
   c. Increases code redundancy.
   d. Improves code maintainability.

9. Which of the following statements about abstraction is TRUE?

   a. Abstraction is implemented using classes and access specifiers.
   b. Abstraction does not improve code reusability.
   c. Abstraction exposes all data members of a class.
   d. Abstraction is not useful in large projects.

10. Encapsulation is also known as:

   a. Information hiding
   b. Data overloading
   c. Function overloading
   d. Inheritance

Write short answer to the following questions.

1. What is abstraction in C++ and why is it important in object-oriented programming?
2. Explain the two types of abstraction with examples.
3. What are the advantages of abstraction in C++? List any four.
4. How does a class help achieve abstraction in C++?
5. How do header files contribute to abstraction in C++?
6. Define encapsulation and explain its significance in object-oriented programming.
7. How does encapsulation help in data security and integrity?
8. Explain the role of getter and setter methods in encapsulation with an example.

Write long answer to the following questions.

1. Explain in detail the concept of abstraction in C++ with an example program demonstrating its implementation.
2. Compare and contrast abstraction and encapsulation in C++ with suitable examples.
3. Write a C++ program to demonstrate abstraction using a class that represents a bank account. The class should include methods for deposit, withdrawal, and balance inquiry.
4. Discuss the advantages of encapsulation in object-oriented programming. Provide a

C++ program to show how encapsulation is implemented.

5. Explain how abstraction is used in C++ with the help of a header file. Provide an example program.

Project Work

1. Write a C++ program to create a class rectangle that hides the length and breadth as private members and provides public methods to set values and calculate the area.
2. Implement a C++ program that uses the cmath library functions to calculate the square root and power of a number.
3. Create a student class with private data members name, roll number, and marks. Use getter and setter methods to access and modify the values.
4. Develop a C++ program for a simple banking system where a bank account class encapsulates account number, balance, and account holder's name, allowing controlled access via getter and setter methods.
5. Write a C++ program where a class employee has private attributes name and salary, and provides public setter and getter methods to modify and display the values.

Unit 5 Inheritance

5.1 Introduction to Inheritance

Inheritance in C++ is a fundamental concept of Object-Oriented Programming (OOP) that allows one class to acquire the properties and behaviors (methods) of another class. The capability of a class to derive properties and characteristics from another class is called inheritance. It promotes code reuse and establishes a relationship between classes, which helps in creating a hierarchical structure in your programs. The goal of inheritance is to create a relationship between classes, which helps to organize code, reduce redundancy, and promote code reuse.

5.2 Advantages of Inheritance

i. **Code Reusability –** Reduces redundancy by allowing reuse of existing code.

ii. **Maintainability –** Simplifies code maintenance by centralizing updates in the base class.

iii. **Extensibility –** Enables easy addition of new functionalities without modifying existing code.

iv. **Encapsulation –** Promotes data hiding and security by controlling access to class members.

v. **Polymorphism Support –** Allows method overriding for dynamic behavior changes.

vi. **Hierarchy Representation –** Models real-world relationships efficiently.

vii. **Reduced Development Time –** Speeds up development by leveraging existing implementations.

5.3 Base Class and Derived Class

Base Class

A class from which properties are inherited is called base class. It is also known as parent class or super class. It cannot inherit properties and methods of derived class.

Syntax:

```
class base_classname{
//code
}
```

Derived Class

A class from which is inherited from the base class. It is also known as child class or sub class. It can inherit properties and method of base class.

Syntax:

```
class derived_classname:access_mode base_classname{
//code
}
```

5.4 Types of Inheritance

Inheritance in C++ allows a class (derived class) to inherit properties and behavior from another class (base class). There are five main types of inheritance in C++:

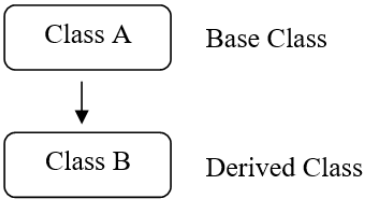
1. Single Inheritance

In single inheritance, a class is allowed to inherit from only one class i.e one sub class is inherited by one base class only.

Syntax

```
class derivedclass_name : access_mode base_class
{
 // body of subclass
};
```

Example




```
class A
{
... .. ...
```

```
};
class B: public A
{
... .. ...
};
```

Implementation

```
#include <iostream>
using namespace std;
class Base {
public:
   void display() {
      cout <<"This is the Base class" << endl;
   }
};
class Derived : public Base {
public:
   void show() {
      cout << "This is the Derived class" << endl;
   }
};
int main() {
   Derived d;
   d.display(); // Accessing Base class function
   d.show();    // Accessing Derived class function
   return 0;
}
```

Output

```
This is the Base class
This is the Derived class
```

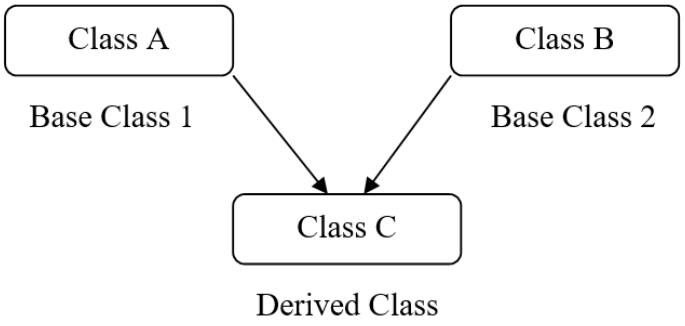
2. Multiple Inheritance

Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one subclass is inherited from more than one base class.

Syntax

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  // body of subclass
};
```

Example




```
class A
{
... .. ...
};
class B
{
... .. ...
};
class C: public A, public B
{
... ... ...
};
```

Implementation

```
#include <iostream>
using namespace std;
class A {
public:
    void displayA() {
        cout << "This is class A" << endl;
    }
};
class B {
public:
    void displayB() {
        cout << "This is class B" << endl;
    }
};
class C : public A, public B {
public:
    void displayC() {
        cout << "This is class C" << endl;
    }
};
int main() {
    C c;
    c.displayA(); // Accessing Class A method
    c.displayB(); // Accessing Class B method
    c.displayC(); // Accessing Class C method
    return 0;
}
```

Output

This is Class A

This is Class B

This is Class C

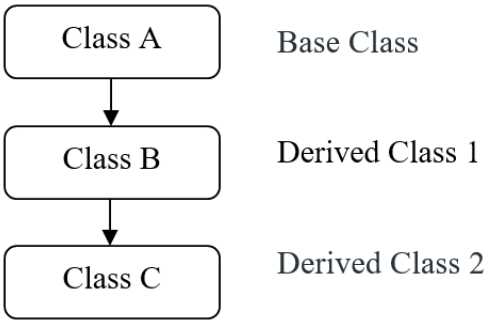
3. Multiple Inheritance

In this type of inheritance, a derived class is created from another derived class and that derived class can be derived from a base class or any other derived class. There can be any number of levels.

Syntax

```
class base_class
{
... .. ...
}
class derived_class1: access_specifier base_class
{
... .. ...
}
class derived_class2: access_specifier derived_class1
{
... .. ...
}
```

Example



```
class A
{
... .. ...
};
class B : public A
{
... .. ...
};
class C: public B
{
... ... ...
};
```

Implementation

```
#include <iostream>
using namespace std;
class A {
public:
   void displayA() {
      cout << "This is class A" << endl;
   }
};
class B : public A {
public:
   void displayB() {
      cout << "This is class B" << endl;
   }
};
```

```
class C : public B {
public:
   void displayC() {
      cout << "This is class C" << endl;
   }
};
int main() {
   C c;
   c.displayA(); // Accessing Class A method
   c.displayB(); // Accessing Class B method
   c.displayC(); // Accessing Class C method
   return 0;
}
```

Output

```
This is Class A
This is Class B
This is Class C
```

4. Hierarchical Inheritance

In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.

Syntax:

```
class base_class
{
... .. ...
}
class derived_class1: access_specifier base_class
{
... .. ...
}
```

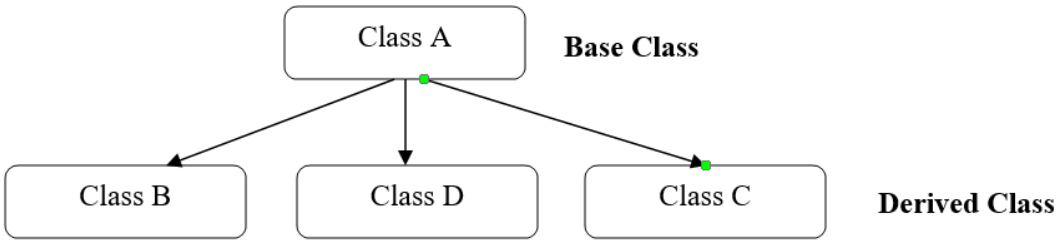
class derived_class2: access_specifier base_class

{

... .

}

Example



class A

{

……..…...

}

class B : public A

{

……..…..…...

}

class C : public A

{

……..…..…...

}

class D : public A

{

……..…..…..…...

}

Implementation

```
#include <iostream>
using namespace std;
class Base {
public:
   void display() {
      cout << "This is the Base class" << endl;
   }
};
class Derived1 : public Base {
public:
   void show1() {
      cout << "This is the first derived class" << endl;
   }
};
class Derived2 : public Base {
public:
   void show2() {
      cout << "This is the second derived class" << endl;
   }
};
int main() {
   Derived1 d1;
   Derived2 d2;
   d1.display(); // Accessing Base class function via first derived class
   d1.show1();   // Accessing first derived class function
   d2.display(); // Accessing Base class function via second derived class
   d2.show2();   // Accessing second derived class function
   return 0;
}
```

Output

This is the Base class

This is the first derived class

This is the Base class

This is the second derived class

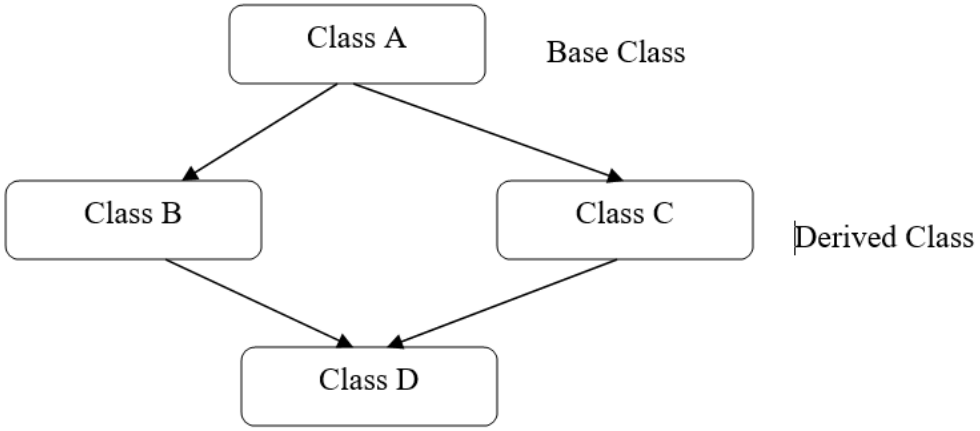
5. Hybrid Inheritance

Hybrid Inheritance is implemented by combining more than one type of inheritance. For Example Combining Hierarchical inheritance and Multiple Inheritance will create hybrid inheritance in C++.

Syntax

```
class base_class
{
 ……
};
class derivedclass_name : access_mode base_class
{
 // body of subclass
};
```

Example



```
#include <iostream>
using namespace std;
class A {
public:
    void displayA() {
        cout << "This is class A" << endl;
    }
};
class B : public A {
public:
    void displayB() {
        cout << "This is class B" << endl;
    }
};
class C : public A {
public:
    void displayC() {
        cout << "This is class C" << endl;
    }
};
class D : public B, public C {
public:
    void displayD() {
        cout << "This is class D" << endl;
    }
};
int main() {
```

```
    D d;
    d.B::displayA(); // Accessing Class A method from class B
    d.C::displayA(); // Accessing Class A method from class C
    d.displayB();    // Accessing Class B method
    d.displayC();    // Accessing Class C method
    d.displayD();    // Accessing Class D method
    return 0;
}
```

Output

```
This is class A
This is class A
This is class B
This is class C
This is class D
```

Exercise

Choose the correct answer from the given alternatives.

1. What is the base class in inheritance?

   a. A class that inherits from another class

   b. A class that is inherited by another class

   c. A function inside a class

   d. None of the above

2. Which type of inheritance allows a class to inherit from multiple base classes?

   a. Single b. Multilevel

   c. Multiple d. Hybrid

3. In C++, which keyword is used to define inheritance?

   a. Class b. Inherit

   c. Extends d. None of the above

4. What is the main advantage of inheritance?

   a. Reduces redundancy b. Slows down execution

   c. Increases complexity d. Removes encapsulation

5. What is hybrid inheritance?

   a. Combination of different types of inheritance

   b. Only single inheritance

   c. Inheritance without base class

   d. A class without any methods

6. Which inheritance model represents a "parent-child" relationship?

   a. Multiple b. Hierarchical

   c. Multilevel d. Hybrid

7. What is used to resolve ambiguity in multiple inheritance?

   a. This pointer b. Virtual base class

   c. Inheritance resolution operator d. Object slicing

8. How many base classes can a derived class inherit from in C++?

   a. Only one
   b. Only two
   c. Any number
   d. None

9. In hierarchical inheritance, how many base classes are there?

   a. One
   b. Two
   c. Three
   d. Unlimited

10. What happens if two base classes have a function with the same name in multiple inheritance?

    a. Compiler error
    b. Ambiguity error
    c. Overloading happens
    d. Function gets deleted

Write short answer to the following questions.

1. What is inheritance in C++?
2. Define base class and derived class with an example.
3. What are the advantages of inheritance?
4. Explain single inheritance with a simple syntax.
5. What is multiple inheritance?
6. How does hierarchical inheritance work?
7. What is the main purpose of hybrid inheritance?
8. Why is polymorphism useful in inheritance?
9. What is the difference between public and private inheritance?
10. How does inheritance help in code reusability?

Write long answer to the following questions.

1. Explain the concept of inheritance in C++ with an example program.
2. Describe different types of inheritance in C++ with suitable examples.
3. What are the advantages and disadvantages of using inheritance in C++?
4. Explain multiple inheritance in detail with a sample C++ program.
5. Discuss hybrid inheritance and demonstrate it with a C++ program.
6. Compare and contrast single inheritance and multiple inheritance with examples.

Project Work

1. Write a C++ program to demonstrate single inheritance.
2. Write a C++ program to implement multiple inheritance.
3. Write a C++ program to show how hierarchical inheritance works.
4. Develop a C++ program for multilevel inheritance with three levels.
5. Write a C++ program to demonstrate hybrid inheritance.

Unit 6 Polymorphism

6.1 Introduction to Polymorphism

Polymorphism is a key concept in object-oriented programming that allows a single function, class, or object to take on multiple forms. The term "polymorphism" is derived from the Greek words "poly," meaning many, and "morph," meaning forms.

In C++, polymorphism provides the flexibility to perform a single action in different ways depending on the context. It is primarily achieved through **function overloading**, **operator overloading**, and **inheritance**. There are two types of polymorphism and they are:

1. Compile-time Polymorphism
2. Run-time Polymorphism

Advantages of Polymorphism

i. **Code Reusability:** Reuse the same code for different types of objects, reducing redundancy.

ii. **Simplifies Maintenance:** Make changes without modifying the existing structure.

iii. **Flexibility:** Functions can work with different types of objects, making the program more adaptable.

iv. **Code Readability:** Write cleaner code with one interface for multiple behaviors.

v. **Supports Extensibility:** Add new features without affecting existing code.

vi. **Modular Programming:** Break the program into smaller, manageable parts.

vii. **Dynamic Binding:** Determine which function to execute at runtime, providing flexibility.

viii. **Abstraction:** Hide complex details and show only necessary functionality.

6.2 Syntax of Polymorphism

Polymorphism in C++ has two main types: **Compile-time polymorphism** and **Run-time polymorphism**. Below is the syntax for both:

1. Compile-Time Polymorphism

Function Overloading Syntax

A single function name can perform different tasks based on the number or type of parameters.

```
class ClassName {
public:
  void functionName(int a) {
    // Code for integer parameter
  }
  void functionName(double b) {
    // Code for double parameter
  }
};
```

Operator Overloading Syntax

Operators can be redefined to perform specific tasks for custom objects.

```
class ClassName {
public:
  ReturnType operator symbol(const ClassName& obj) {
    // Code for adding two objects
  }
};
```

2. Run-Time Polymorphism

Virtual Function Syntax:

A base class function is overridden in the derived class using the virtual keyword.

```
class BaseClass {
public:
  virtual void functionName() {
    // Base class implementation
```

```
    }
};
```

6.3 Types of Polymorphism

1. Compile-Time Polymorphism (Static Binding)

In compile-time polymorphism, the decision about which function or operator to call is made during the program's compilation. It is also called **early binding**.

Types of Compile-Time Polymorphism:

a. Function Overloading

b. Operator Overloading

a) Function Overloading

In function overloading, multiple functions with the same name are defined in a class, but they differ in the number or type of arguments. The compiler determines which function to call based on the arguments passed.

Example

```
#include <iostream>
using namespace std;
class Calculator {
public:
    void add(int a, int b) {
        cout << "Sum of integers: " << a + b << endl;
    }
    void add(double a, double b) {
        cout << "Sum of doubles: " << a + b << endl;
    }
};
int main() {
    Calculator calc;
    calc.add(5, 10);        // Calls the integer version
    calc.add(3.5, 2.5);     // Calls the double version
```

```
    return 0;
}
```

b) Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. Operator overloading is a compile-time polymorphism.

Syntax:

```
returnType operator symbol(arguments)
{
//Code
}
```

where,

returnType - the return type of the function

operator - a special keyword

symbol - the operator we want to overload (+, <, -, ++, etc.)

arguments - the arguments passed to the function

Rules for Operator Overloading

i. Only existing operator can be overloaded and new operators cannot be overloaded.

ii. The overloaded operators must contains at least one operand of the user defined datatypes.

iii. We don't use a friend function to overload certain operators

iv. When unary operators are overloaded through member function they take no explicit argument but if they are overloaded by a friend they take one argument.

v. When binary operators are overloaded through member function they take one explicit argument but if they are overloaded by a friend they take no argument.

Operation that can be Performed

i. Arithmetic Operations: +, -, *, /, %

ii. Logical Operations: && and //

iii. Relational Operations: = = ,>=, =>

iv. Pointer Operations: & and *

v. Memory Management Operators: new, delete[]

Operation that cannot be Performed

i. :: Scope resolution operator

ii. ?: Ternary operator

iii. , Memory Selector

iv. Size of operator

Operator Overloading Consists of two Types

i. Unary Operator

ii. Binary Operator

Unary Operator Overloading

Unary operator overloading operates on only one operand. Example of unary operator are increment (++) and decrement (--).

Example

```
#include <iostream>
using namespace std;
class Complex {
   int a, b;
public:
   Complex() {  // Default constructor
   }
   void getValue() {
      cout << "Enter the two numbers: ";
      cin >> a >> b;
   }
   void operator++() {  // Prefix increment
      ++a;
      ++b;
```

```
    }
    void operator--() {  // Prefix decrement
        --a;
        --b;
    }
    void display() {
        cout << a << " + " << b << "i" << endl;
    }
};
int main() {
    Complex obj;
    obj.getValue();
    ++obj;  // Calls prefix increment
    cout << "Incremented Complex Number:\n";
    obj.display();
    --obj;  // Calls prefix decrement
    cout << "Decremented Complex Number:\n";
    obj.display();
    return 0;
}
```

Output

```
Enter the two numbers: 1 2
Incremented Complex Number:
2 + 3i
Decremented Complex Number:
1 + 2i
```

Binary Operator Overloading

The operators that contains two operand is called binary operator overloading.

Example

```
#include <iostream>
using namespace std;
class Overloading {
   int value;
public:
   void setValue(int temp) {
      value = temp;
   }
   // Operator overloading for +
   Overloading operator+(const Overloading& ob) {
      Overloading t;
      t.value = value + ob.value;
      return t;
   }
   void display() {
      cout << value << endl;
   }
};
int main() {
   Overloading obj1, obj2, result;
   int a, b;
   cout << "Enter two values: ";
   cin >> a >> b;
   obj1.setValue(a);
   obj2.setValue(b);
   result = obj1 + obj2;  // Calls operator+()
```

```
    cout << "Input Values:\n";
    obj1.display();
    obj2.display();
    cout << "Result: ";
    result.display();
    return 0;
}
```

Output

```
Enter two values: 1 2
Input Values:
1
2
Result: 3
```

2. Run-Time Polymorphism (Dynamic Binding)

Runtime polymorphism is also known as dynamic polymorphism or late binding. In runtime polymorphism, the function call is resolved at run time. It is achieved using function overriding and virtual function.

Function Overriding

Function overriding in C++ is a concept in Object-Oriented Programming (OOP) where a derived class provides a specific implementation of a function that is already defined in its base class. This allows dynamic (runtime) polymorphism, enabling late binding.

Example

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

```
  {
    cout << "Class::Base"<<endl;
  }
};
class Derived:public Base
{
  public:
  void show_val() //function overridden from base
  {
    cout << "Class::Derived"<<endl;
  }
};
int main()
{
  Base b;
  Derived d;
  b.show_val();
  d.show_val();
}
```

Output

```
Class::Base
Class::Derived
```

Virtual Functions

A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.

Example

```
#include <iostream>
using namespace std;
class Base {
public:
   virtual void show_val() {
      cout << "Class::Base" << endl;
   }
};
class Derived : public Base {
public:
   void show_val() { // No override keyword
      cout << "Class::Derived" << endl;
   }
};
int main() {
   Base* b;  // Base class pointer
   Derived d; // Derived class object
   b = &d;
   b->show_val(); // Late binding (dynamic dispatch)
   return 0;
}
```

Output

```
Class::Derived
```

Rules for Virtual Functions

The rules for the virtual functions in C++ are as follows:

1. Virtual functions cannot be static.

2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as the derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have a virtual destructor but it cannot have a virtual constructor.

Exercise

Choose the correct answer from the given alternatives.

1. What is polymorphism in C++?
   a. A feature that allows multiple classes to have the same member variables
   b. A feature that allows functions or operators to operate in different ways
   c. A type of variable declaration
   d. A process of data encryption

2. Which of the following is an example of compile-time polymorphism?
   a. Function Overloading
   b. Function Overriding
   c. Virtual Functions
   d. Dynamic Binding

3. What keyword is used to achieve runtime polymorphism in C++?
   a. Overload
   b. Friend
   c. Virtual
   d. Static

4. Which of the following is NOT a type of polymorphism in C++?
   a. Compile-time polymorphism
   b. Run-time polymorphism
   c. Data polymorphism
   d. Operator overloading

5. What does operator overloading allow in C++?
   a. Creating new operators
   b. Changing the precedence of operators
   c. Assigning new functionality to existing operators
   d. Defining new types of operators

6. Function overloading is an example of:
   a. Compile-time polymorphism
   b. Run-time polymorphism
   c. Data abstraction
   d. Encapsulation

7. Which of the following statements about virtual functions is true?
   a. They cannot be overridden in derived classes.
   b. They must be defined inside the derived class.

c. They enable runtime polymorphism.

d. They must always return an integer.

8. What is the output of the following code?

```
#include <iostream>
using namespace std;
class Base {
public:
   virtual void show() { cout << "Base class"; }
};
class Derived : public Base {
public:
   void show() { cout << "Derived class"; }
};
int main() {
   Base* b;
   Derived d;
   b = &d;
   b->show();
   return 0;
}
```

a. Base class

b. Derived class

c. Error in code

d. No output

9. What will happen if a virtual function is not overridden in a derived class?

a. Compilation error occurs

b. The base class version of the function is used

c. The derived class version is always called

d. It causes a segmentation fault

10. Which of the following cannot be overloaded in C++?

a. + (Addition Operator) b. = (Assignment Operator)

c. :: (Scope Resolution Operator) d. << (Stream Insertion Operator)

Write short answer to the following questions.

1. What is polymorphism in C++? Explain its significance in object-oriented programming.
2. Differentiate between compile-time polymorphism and run-time polymorphism with examples.
3. What is function overloading? Provide an example in C++.
4. What are the rules for operator overloading in C++?
5. Explain function overriding with an example.
6. What is a virtual function? How does it support dynamic polymorphism?
7. What are the advantages of polymorphism in C++?
8. Explain the difference between early binding and late binding.
9. What is the purpose of the virtual keyword in C++?
10. Why can't constructors be virtual in C++?

Write long answer to the following questions.

1. Discuss operator overloading in C++ with an example program.
2. Write a program to demonstrate function overloading in C++.
3. Explain function overriding and virtual functions with a suitable program.
4. Differentiate between function overloading and function overriding with examples.
5. Write a C++ program that demonstrates runtime polymorphism using virtual functions.
6. Explain the rules and limitations of operator overloading in C++.
7. Discuss the advantages of polymorphism in software development.
8. Write a program to overload the + operator for a user-defined class.

Project Work

1. Write a C++ program that implement function overloading for an Area class that calculates the area of different shapes.
2. Create a C++ program demonstrating operator overloading for + and - operators.
3. Develop a C++ program that implements function overriding using base and derived classes.
4. Write a C++ program that overloads the increment (++) and decrement (--) operators.
5. Create a C++ program to demonstrate virtual functions.
6. Implement an example of polymorphism using abstract classes and pure virtual functions.

References

Book References

Lafore, Robert – Object-Oriented Programming in C++, Pearson.

Drozdek, Adam – Data Structures and Algorithms in C++, Cengage Learning.

Malik, D.S. – Data Structures Using C++, Cengage Learning

Web References

https://www.geeksforgeeks.orgc

https://www.tutorialspoint.com

https://dribbble.com

https://en.wikipedia.org

https://www.programiz.com

https://www.w3schools.com

https://kthmcollege.ac.in