UNIT 1

Introduction to Programming:

Programming, often referred to as coding, is the process of designing, writing, testing, and maintaining the source code of computer programs. It involves creating instructions that a computer can execute to perform specific tasks. These instructions, written in a programming language, allow developers to communicate with computers and create software that solves problems, automates tasks, and provides entertainment, among other things.

Key Concepts in Programming

1. Programming Languages:

- Programming languages are formal languages used to write instructions for computers. They provide a set of syntax and semantics for developers to create programs. Some popular programming languages include:

  - Python: Known for its simplicity and readability, widely used in web development, data science, and automation.

  - Java: A versatile, platform-independent language commonly used in enterprise applications, mobile development, and more.

  - C++: Known for its performance and efficiency, often used in system software, game development, and applications requiring real-time processing.

  - JavaScript: Primarily used for web development, enabling interactive elements on websites.

  -Ruby, PHP, Swift, C#, etc.**: Other languages that are popular in various domains.

2. Syntax and Semantics:

- Syntax refers to the set of rules that define the structure of valid statements in a programming language. Syntax errors occur when these rules are violated.

- Semantics deals with the meaning behind the syntactical elements and structures. It defines what a program does when executed.

3. Basic Constructs:

- Variables: Named storage locations in memory used to store data that can be modified during program execution.

- Data Types: Define the type of data a variable can hold, such as integers, floating-point numbers, characters, and more.

- Operators: Symbols that perform operations on variables and values, like addition (+), subtraction (-), and comparison (==).

- Control Structures: These include conditional statements (if, else), loops (for, while), and switch cases, which control the flow of execution based on certain conditions.

4. Functions and Procedures:

- Functions and procedures are blocks of code that perform specific tasks and can be reused throughout a program. They help in breaking down complex problems into manageable parts.

5. Algorithms and Data Structures:

- Algorithms are step-by-step procedures or formulas for solving a problem.

- Data Structures are ways of organizing and storing data in a program, such as arrays, linked lists, stacks, queues, and trees.

6. Debugging and Testing:

- Debugging is the process of identifying and fixing errors or bugs in a program. Testing involves running the program with different inputs to ensure it behaves as expected.

7. Integrated Development Environments (IDEs):

- IDEs are software applications that provide comprehensive facilities to programmers for software development. They typically include a code editor, compiler, debugger, and other tools. Examples include Visual Studio, Eclipse, PyCharm, and IntelliJ IDEA.

The Importance of Programming

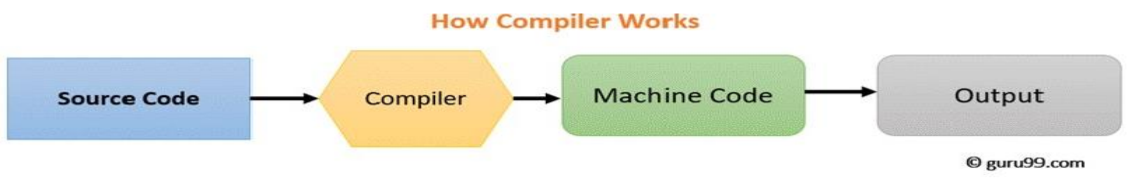
Programming is a fundamental skill in today's technology-driven world. It enables the creation of software applications, websites, games, and other digital tools that are integral to modern life. Learning programming helps develop problem-solving skills, logical thinking, and creativity. It's also a valuable skill in many career fields, including software development, data science, cybersecurity, finance, healthcare, and more.

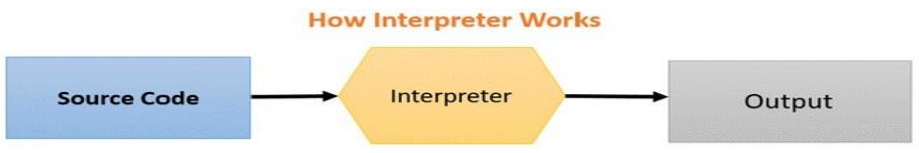
As technology continues to evolve, programming languages and paradigms change, but the core principles of programming—understanding how to solve problems and translate those solutions into code—remain constant.

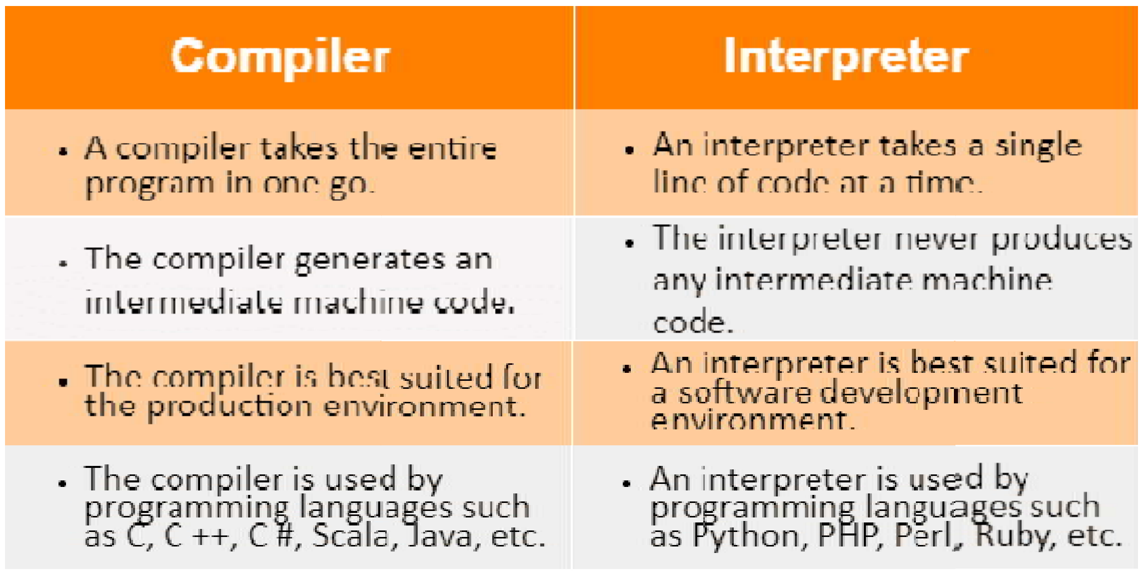
Definition of compiler:A compiler is a special program that translates a programming language's source code into machine code, bytecode or another programming language. The source code is typically written in a high-level, human-readable language such as Java or C,C++,

Definition of interpreter:

An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code. Examples of interpreted languages are Perl, Python R , and Mat lab.







Compiler	Interpreter
• A compiler takes the entire program in one go.	• An interpreter takes a single line of code at a time.
• The compiler generates an intermediate machine code.	• The interpreter never produces any intermediate machine code.
• The compiler is best suited for the production environment.	• An interpreter is best suited for a software development environment.
• The compiler is used by programming languages such as C, C ++, C #, Scala, Java, etc.	• An interpreter is used by programming languages such as Python, PHP, Perl, Ruby, etc.

Program development life cycle

The Program Development Life
development of software programs. The PDLC is similar to the Software Development Life Cycle (SDLC) but is applied at a higher level, to manage the development of multiple software programs or projects. This article focuses on discussing PDLC in detail.

What is PDLC?

The PDLC is an iterative process that allows for feedback and adjustments to be made at each phase, to ensure that the final product meets the needs of the stakeholders and is of high quality.

- Program Development Life Cycle (PDLC) is a systematic way of developing quality software.
- It provides an organized plan for breaking down the task of program development into manageable chunks, each of which must be completed before moving on to the next phase.

Phases of PDLC

1. **Planning:** In this phase, the goals and objectives of the program are defined, and a plan is developed to achieve them. This includes identifying the resources required and determining the budget and schedule for the program.

2. **Analysis:** In this phase, the requirements for the program are defined and analyzed. This includes identifying the stakeholders, their needs and expectations, and determining the functional and non-functional requirements for the program.

3. **Design:** In this phase, the program's architecture and design are developed. This includes creating a detailed design of the program's components and interfaces, as well as determining how the program will be tested and deployed.

4. **Implementation:** In this phase, the program is developed and coded. This includes writing the program's source code and creating any necessary documentation.

5. **Testing:** In this phase, the program is tested to ensure that it meets the requirements and is free of defects.

6. **Deployment:** In this phase, the program is deployed and made available to users.

7. **Maintenance:** After the deployment, the program is maintained by fixing any bugs or errors that are found and updating the program to meet changing requirements.

Steps in PDLC

The program development process is divided into the steps discussed below:

1. Defining the Problem

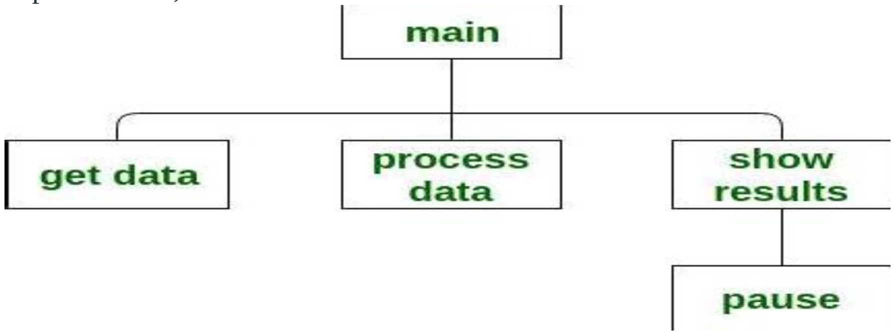
The first step is to define the problem. In major software projects, this is a job for system analyst, who provides the results of their work to programmers in the form of a program specification. The program specification defines the data used in program, the processing that should take place while finding a solution, the format of the output and the user interface.

2. Designing the Program

Program design starts by focusing on the main goal that the program is trying to achieve and then breaking the program into manageable components, each of which contributes to this goal. This approach of program design is called top-bottom program design or modular programming. The first step involve identifying main routine, which is the one of program's major activity. From that point, programmers try to divide the various components of the main routine into smaller parts called modules. For each module, programmer draws a conceptual plan using an appropriate program design tool to visualize how the module will do its assign job. **Program Design Tools:** The various program design tools are described below:

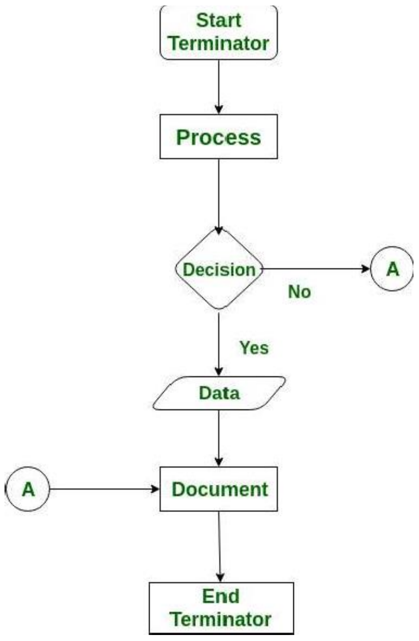
- **Structure Charts:** A structure chart, also called Hierarchy chart, show top-down design of program. Each box in the structure chart indicates a task that program must accomplish. The

Top module, called the Main module or Control module. For example:



- **Algorithms:** An algorithm is a step-by-step description of how to arrive at a solution in the most easiest way. Algorithms are not restricted to computer world only. In fact, we use them in everyday life.

- **Flowcharts:** A flowchart is a diagram that shows the logic of the program. For example:



3. Coding the Program

Coding the program means translating an algorithm into specific programming language. The technique of programming using only well defined control structures is known as Structured programming. Programmer must follow the language rules, violation of any rule causes error. These errors must be eliminated before going to the next step.

4. Testing and Debugging the Program

After removal of syntax errors, the program will execute. However, the output of the program may not be correct. This is because of logical error in the program. A logical error is a mistake that the programmer made while designing the solution to a problem. So the programmer must find and correct logical errors by carefully examining the program output using Test d ata. Syntax error and Logical error are collectively known as Bugs. The process of identifying errors and eliminating them is known as Debugging.

5. Documenting the Program

After testing, the software project is almost complete. The structure charts, pseudocodes, flowcharts and decision tables developed during the design phase become documentation for others who are associated with the software project. This phase ends by writing a manual that provides an overview of the program's functionality, tutorials for the beginner, in-depth explanations of major program features, reference documentation of all program commands and a thorough description of the error messages generated by the program.

6. Deploying and Maintaining the Program

In the final phase, the program is deployed (installed) at the user's site. Here also, the program is kept under watch till the user gives a green signal to it. Even after the software is completed, it needs to be maintained and evaluated regularly. In software maintenance, the programming team fixes program errors and updates the software.

Benefits of PDLC

Provides a structured approach: PDLC provides a structured approach to developing software, which helps to ensure that the program is developed in a logical and organized way.

Facilitates communication: PDLC helps to facilitate communication between different stakeholders, such as developers, project managers, and customers.

Identifies and manages risks: PDLC helps to identify and manage potential risks during the development of the program, allowing for proactive measures to be taken to mitigate them.

Improves quality: PDLC helps to improve the quality of the final product by providing a systematic approach to testing and quality assurance.

Increases efficiency: By following a PDLC, the development process becomes more efficient as it allows for better planning and organization.

Limitations of PDLC

- **Can be time-consuming:** Following a strict PDLC can be time-consuming, and may delay the development of the program.
- **Can be inflexible:** The rigid structure of PDLC may not be suitable for all types of software development projects, and may limit the ability to incorporate changes or new ideas.
- **Can be costly:** Implementing a PDLC may require additional resources and budget, which can be costly for organizations.
- **Can be complex:** PDLC can be complex, and may require a certain level of expertise and knowledge to implement effectively.
- **May not be suitable for smaller projects:** PDLC may not be suitable for smaller projects as it can be an overkill and would not be cost-effective.

Introduction to Procedure-Oriented Programming (POP)

Procedure-Oriented Programming (POP) is a programming paradigm that emphasizes the use of procedures or routines to perform specific tasks. Also known as imperative or procedural programming, this approach is one of the earliest and most traditional methods of software development. In POP, the primary focus is on the sequence of instructions that the computer must follow to achieve a desired outcome.

Key Concepts in Procedure-Oriented Programming

1. **Procedures/Functions:**
   - The core building blocks in POP are procedures or functions. A procedure is a block of code designed to perform a particular task. Functions can take inputs (arguments), perform operations, and return outputs (results).
   - Procedures help break down a program into smaller, manageable parts, making the code more modular and easier to understand.
2. **Modularity:**
   - Modularity in POP is achieved by dividing the program into smaller, self-contained functions or procedures. Each function performs a specific task and can be tested independently.
   - This modularity helps in organizing the code, making it easier to debug, maintain, and update.
3. **Top-Down Approach:**
   - POP often follows a top-down approach, where the overall system is broken down into smaller, more manageable components. The process starts with the high-level design and progressively breaks it down into detailed procedures.
   - This approach makes it easier to understand the system's overall structure before delving into the specifics.
4. **Global Data:**
   - In POP, data is often stored in global variables, which are accessible by all functions within the program. Functions operate on this shared data, which can lead to challenges in managing state and ensuring data consistency.
5. **Control Structures:**
   - POP relies heavily on control structures like loops (for, while), conditionals (if, switch), and case statements to control the flow of the program. These structures dictate the sequence in which instructions are executed.

Advantages of Procedure-Oriented Programming

1. **Simplicity:**
   - POP is straightforward and easy to understand, especially for beginners. It focuses on a clear sequence of instructions to solve specific problems.

2. **Efficiency:**

- The use of functions promotes code reuse. A function can be called multiple times within a program, reducing redundancy and saving development time.

3. **Ease of Implementation:**

   - POP is well-suited for small to medium-sized problems that can be clearly defined and broken down into a sequence of steps.

Disadvantages of Procedure-Oriented Programming

1. **Poor Data Encapsulation:**

   - Since data is often global, it can be accessed and modified from anywhere in the program, leading to potential data integrity issues. It can be challenging to track how data changes throughout the program.

2. **Scalability Issues:**

   - As programs grow in size and complexity, managing global data and ensuring that functions do not interfere with each other becomes difficult. POP can lead to tightly coupled code, making it harder to modify and maintain.

3. **Limited Reusability:**

   - Functions in POP are often designed to operate on specific data structures, limiting their reusability in different contexts or applications.

Examples of Procedure-Oriented Programming Languages

- **C:** One of the most widely used POP languages, known for its efficiency and control over system resources.
- **Fortran:** Often used in scientific and engineering applications for numerical computations.
- **Pascal:** Designed for teaching programming concepts and known for its clear syntax and strong type checking.

Object-Oriented Programming (OOP).

Object-Oriented Programming (OOP) is a programming paradigm centered around the concept of "objects," which can encapsulate both data and methods. This paradigm is designed to model real- world entities and interactions, making it particularly effective for managing complex software systems. OOP emphasizes modularity, code reuse, and abstraction, aiming to make software development more intuitive and manageable.

Key Concepts in Object-Oriented Programming

1. **Objects:**

   - An object is an instance of a class. It represents a real-world entity or concept with a specific state and behavior. Objects combine data (attributes) and functions (methods) that operate on the data.

- Example: In a banking system, an Account object might have attributes like balance and accountNumber, and methods like deposit() and withdraw().

2. **Classes:**

   - A class is a blueprint or template for creating objects. It defines the data attributes and methods that the objects created from the class will have.
   - Example: A class Car might define attributes such as color, model, and engineType, and methods like start(), stop(), and accelerate().

3. **Encapsulation:**

   - Encapsulation is the practice of bundling the data (attributes) and methods (functions) that operate on the data into a single unit or class. It restricts direct access to some of the object's components, providing a controlled interface.
   - **Access Modifiers:** Encapsulation is often implemented using access modifiers like private, protected, and public to control the visibility and accessibility of class members.

4. **Inheritance:**

   - Inheritance allows one class (child or subclass) to inherit attributes and methods from another class (parent or superclass). This promotes code reuse and creates a hierarchical relationship between classes.
   - Example: A class ElectricCar might inherit from the class Car, adding specific attributes like batteryLife and methods like charge().

5. **Polymorphism:**

   - Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single function or method to operate on different types of objects, often through method overriding or method overloading.
   - **Method Overloading:** Multiple methods with the same name but different parameters.
   - **Method Overriding:** Redefining a method in a subclass that was already defined in its superclass.

6. **Abstraction:**

   - Abstraction focuses on hiding the complex implementation details and showing only the essential features of an object. It allows developers to work with high-level concepts without needing to understand the low-level details.
   - Example: An abstract class Shape might define a method draw(), but different shapes like Circle and Rectangle would provide their specific implementations of the draw() method.

Advantages of Object-Oriented Programming

1. **Modularity:**
   - OOP promotes modular design by breaking down software into discrete objects or classes. This modularity makes it easier to manage, develop, and understand complex systems.
2. **Code Reusability:**
   - Through inheritance and composition, OOP allows code to be reused across different parts of a program or across different projects. This reduces redundancy and improves maintainability.
3. **Maintainability:**
   - Encapsulation and abstraction make it easier to manage and update code. Changes in the implementation of a class do not affect other parts of the program as long as the public interface remains consistent.
4. **Scalability:**
   - OOP supports the creation of scalable software systems by allowing new features and functionalities to be added through extensions of existing classes rather than modifying existing code.

Disadvantages of Object-Oriented Programming

1. **Complexity:**
   - OOP can introduce additional complexity, especially for beginners. The concepts of classes, inheritance, and polymorphism can be challenging to grasp initially.
2. **Performance Overhead:**
   - The use of abstraction, dynamic dispatch, and additional object management can introduce performance overhead compared to procedural programming.
3. **Steeper Learning Curve:**
   - Mastering OOP principles and practices can require a more significant investment of time and effort compared to procedural programming.

Examples of Object-Oriented Programming Languages

- **Java:** A widely-used, class-based language known for its portability and use in enterprise applications and Android development.
- **C++:** An extension of C that supports both procedural and object-oriented programming, widely used in system software and game development.
- **Python:** Known for its simplicity and readability, Python supports OOP and is used in various fields, including web development, data science, and artificial intelligence.

- **C#:** A language developed by Microsoft that is used for developing applications on the .NET framework, with strong support for OOP.

S.no.	On the basis of	Procedural Programming	Object-oriented programming
1.	**Definition**	It is a programming language that is derived from structure programming and based upon the concept of calling procedures. It follows a step-by-step approach in order to break down a task into a set of variables and routines via a sequence of instructions.	Object-oriented programming is a computer programming design philosophy or methodology that organizes/ models software design around data or objects rather than functions and logic.
2.	**Security**	It is less secure than OOPs.	Data hiding is possible in object-oriented programming due to abstraction. So, it is more secure than procedural programming.
3.	**Approach**	It follows a top-down approach.	It follows a bottom-up approach.
4.	**Data movement**	In procedural programming, data moves freely within the system from one function to another.	In OOP, objects can move and communicate with each other via member functions.
5.	**Orientation**	It is structure/procedure-oriented.	It is object-oriented.
6.	**Access modifiers**	There are no access modifiers in procedural programming.	The access modifiers in OOP are named as private, public, and protected.
7.	**Inheritance**	Procedural programming does not have the concept of inheritance.	There is a feature of inheritance in object-oriented programming.
8.	**Code reusability**	There is no code reusability present in procedural programming.	It offers code reusability by using the feature of inheritance.
9.	**Overloading**	Overloading is not possible in procedural programming.	In OOP, there is a concept of function overloading and operator overloading.
10.	**Importance**	It gives importance to functions over data.	It gives importance to data over functions.

11.	**Virtual**	In procedural programming, there are no	In OOP, there is an appearance of virtual

	class	virtual classes.	classes in inheritance.
12.	**Complex problems**	It is not appropriate for complex problems.	It is appropriate for complex problems.
13.	**Data hiding**	There is not any proper way for data hiding.	There is a possibility of data hiding.
14.	**Program division**	In Procedural programming, a program is divided into small programs that are referred to as functions.	In OOP, a program is divided into small parts that are referred to as objects.
15.	**Examples**	Examples of Procedural programming include C, Fortran, Pascal, and VB.	The examples of object-oriented programming are - .NET, C#, Python, Java, VB.NET, and C++.

Basic Concepts of Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is centered around the concept of objects and classes, which facilitate the creation of modular, reusable, and maintainable code. Here are the fundamental concepts of OOP:

1. Objects

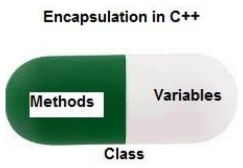
- **Definition:** An object is an instance of a class. It represents a real-world entity or concept that has state and behavior.
- **Attributes:** Objects have attributes (also called properties or fields) that store data. For example, an object of class Car might have attributes like color, model, and engineType.
- **Methods:** Objects have methods (also called functions or operations) that define their behavior. Methods can operate on the object's attributes. For example, a Car object might have methods like start(), stop(), and accelerate().

2. Classes

- **Definition:** A class is a blueprint or template for creating objects. It defines the attributes and methods that its objects will have.
- **Syntax:** Classes typically include a constructor (a special method for initializing objects) and may include destructors (methods for cleanup).
- **Example:** A Person class might include attributes like name, age, and methods like greet() and celebrateBirthday().

3. Encapsulation

- **Definition:** Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on the data into a single unit (class). It also involves restricting access to some of the object's components.
- **Access Modifiers:**
  - **Private:** Members that are not accessible from outside the class.
  - **Protected:** Members that are accessible within the class and its subclasses.
  - **Public:** Members that are accessible from outside the class.
- **Example:** A BankAccount class might have a private attribute balance and public methods deposit() and withdraw() to modify the balance.



4. Inheritance

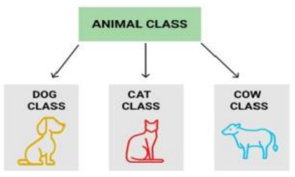
- **Definition:** Inheritance is a mechanism where a new class (child or subclass) inherits attributes and methods from an existing class (parent or superclass). It allows for code reuse and the creation of hierarchical relationships.

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

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

- **Types:**
  - **Single Inheritance:** A subclass inherits from one superclass.
  - **Multiple Inheritance:** A subclass inherits from more than one superclass (supported in some languages like C++ but not in others like Java).
- **Hierarchical Inheritance:** Multiple subclasses inherit from a single superclass.
- **Multilevel Inheritance:** A subclass inherits from another subclass.**Example:** A Dog class can inherit from an Animal class, gaining the eat() and sleep() methods from Animal while adding

its own methods like bark().



4. Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A person at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. C++ supports operator overloading and function overloading.

- ***Operator Overloading***: The process of making an operator exhibit different behaviors in different instances is known as operator overloading.

- ***Function Overloading***: Function overloading is using a single function name to perform different types of tasks. Polymorphism is extensively used in implementing inheritance.

  - **Definition:** Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables methods to be used interchangeably, providing flexibility in code.

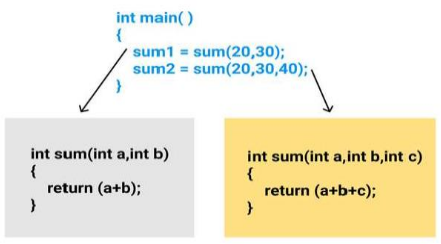
  - **Types:**

    - **Method Overloading:** Multiple methods with the same name but different parameters within the same class.

    - **Method Overriding:** A subclass provides a specific implementation of a method that is already defined in its superclass.

    - **Operator Overloading:** Some languages allow operators to be overloaded to work with user-defined types.

  - **Example:** A method draw() might be implemented differently in Circle and Rectangle classes, but both can be treated as Shape objects and called in the same way.



```
int main( )
{
  sum1 = sum(20,30);
  sum2 = sum(20,30,40);
}
```

```
int sum(int a,int b)
{
   return (a+b);
}
```

```
int sum(int a,int b,int c)
{
   return (a+b+c);
}
```

6. Abstraction

- **Definition:** Abstraction focuses on hiding the complex implementation details and exposing only the necessary features of an object. It simplifies the interaction with complex systems by presenting a clear and simplified interface.
- **Abstract Classes:** Classes that cannot be instantiated directly and are meant to be subclasses. They can include abstract methods (methods without implementation) that must be implemented by derived classes.
- **Interfaces:** In some languages, interfaces define a contract of methods that implementing classes must provide. They allow for the definition of methods without specifying how they are implemented.
- **Example:** An abstract class Shape might define an abstract method draw(), which is implemented differently by subclasses Circle and Rectangle.

Benefits of Object-Oriented Programming

1. **Modularity:**
   - **Definition:** OOP allows software to be divided into discrete, self-contained units called objects. Each object encapsulates its own data and methods.
   - **Benefit:** This modular approach simplifies development and maintenance, as changes to one part of the system are less likely to impact other parts. It also promotes a clear and organized structure.
2. **Code Reusability:**
   - **Definition:** Through inheritance, OOP enables the creation of new classes based on existing ones, allowing code to be reused and extended.
   - **Benefit:** Reduces redundancy and duplication of code, making development faster and more efficient. Code that works with a superclass can also work with any subclass, enhancing flexibility.
3. **Maintainability:**
   - **Definition:** OOP's encapsulation and modularity make it easier to manage and update software.
   - **Benefit:** Changes can be made to individual objects or classes without affecting the entire system. Encapsulation hides the internal details and exposes only necessary parts, simplifying maintenance.
4. **Scalability:**
   - **Definition:** OOP supports the creation of scalable systems by allowing classes to be extended and modified independently.
   - **Benefit:** Makes it easier to add new features and functionalities to a system without disrupting existing code. Inheritance and polymorphism facilitate the integration of

new components.

5. **Abstraction:**

- **Definition:** OOP allows developers to create abstract representations of complex real-world entities.
- **Benefit:** Simplifies interaction with complex systems by exposing only relevant information and hiding implementation details. This abstraction provides a clear interface for interacting with objects.

6. **Flexibility and Extensibility:**
   - **Definition:** OOP supports polymorphism, which allows objects to be treated as instances of their parent class.
   - **Benefit:** Enhances flexibility in code, as objects can be used interchangeably and extended without modifying existing code. It supports dynamic method binding, which allows for more adaptable and reusable code.
7. **Improved Design:**
   - **Definition:** OOP encourages the use of design principles such as SOLID principles (Single responsibility, Open/closed, Liskov substitution, Interface segregation, Dependency inversion).
   - **Benefit:** Results in well-structured and manageable code. These design principles promote better organization, clearer responsibilities, and less coupling between components.

Applications of Object-Oriented Programming

1. **Software Development:**
   - **Application:** OOP is widely used in software development to create robust, scalable, and maintainable applications.
   - **Examples:** Desktop applications (e.g., Microsoft Office), mobile apps (e.g., Android apps), and web applications (e.g., e-commerce platforms).
2. **Game Development:**
   - **Application:** OOP is used to model game entities, interactions, and behaviors. It helps manage complex game worlds and character interactions.
   - **Examples:** Games like “Minecraft” and “The Sims” use OOP principles to manage game objects, characters, and their interactions.
3. **GUI Applications:**
   - **Application:** OOP is used to create graphical user interfaces (GUIs) by modeling user interface components as objects.
   - **Examples:** GUI frameworks such as Java Swing, JavaFX, and Qt utilize OOP principles to build windows, buttons, and other interface elements.

4. **Simulation and Modeling:**

  - **Application:** OOP is employed in simulation and modeling to represent real-world systems and processes.
  - **Examples:** Simulations for traffic management, financial modeling, and scientific research use OOP to create models and simulate scenarios.

5. **Web Development:**
   - **Application:** OOP is used in web development to design and manage web applications and services.
   - **Examples:** Frameworks such as Django (Python) and Ruby on Rails (Ruby) leverage OOP principles to handle web requests, manage data, and build dynamic web applications.

6. **Database Management:**
   - **Application:** OOP is used in object-oriented databases and Object-Relational Mapping (ORM) systems to map database entities to objects in code.
   - **Examples:** Tools like Hibernate (Java) and Entity Framework (.NET) use OOP principles to simplify data access and manipulation.

7. **Embedded Systems:**
   - **Application:** OOP can be used in embedded systems to model and manage hardware components and their interactions.
   - **Examples:** Embedded software for automotive systems, industrial control systems, and consumer electronics often employs OOP principles for modular design.

INTRODUCTION TO C++

History of C++:

- Until 1980, C programming was widely popular, and slowly people started realizing the drawbacks of this language, at the same time a new programming approach that was Object Oriented Programming.
- The C++ programming language was created by Bjarne Stroustrup and his team at Bell Laboratories (AT&T, USA) to help implement simulation projects in an object- oriented and efficient way.
- C++ is a superset of C because; any valid C program is valid C++ program too but not the vice versais not true.
- C++ can make use of existing C software libraries with major addition of “Class Construct”.
- This language was called “C with classes” and later in 1983, it was named “C++” by Rick Mascitii.
- As the name C++ implies, C++ was derived from the C programming language: ++ is the incrementoperator in C.

Characteristics of C++:

- Object-Oriented Programming: It allows the programmer to design applications like a communication between object rather than on a structured sequence of code. It allows a greater reusability of code in a more logical and productive way.
- Portability: We can compile the same C++ code in almost any type of computer & operating system without making any changes.
- Modular Programming: An application’s body in C++ can be made up of several source code files that are compiled separately and then linked together saving time.
- C Compatibility: Any code written in C can easily be included in a C++ program without making any changes.
- Speed: The resulting code from a C++ compilation is very efficient due to its duality as high-leveland low-level language.
- Machine independent: It is a Machine Independent Language.
- Flexibility: It is highly flexible language and versatility.
- Wide range of library functions: It has huge library functions; it reduces the code development timeand also reduces cost of software development.
- System Software Development: It can be used for developing System Software Viz., Operating system, Compilers, Editors and Database.

Basic Syntax of C++

Hello World Program:

```
#include <iostream>

int main() {

    std::cout << "Hello, World!" << std::endl;

    return 0;

}
```

Applications of C++

1. Systems Programming:
   - Operating Systems: C++ is used in the development of operating systems and system-level software due to its low-level capabilities and performance.
   - Device Drivers: Writing drivers for hardware components.
2. Application Development:
   - Desktop Applications: C++ is used for developing cross-platform desktop applications and graphical user interfaces (GUIs).
   - Office Software: Includes productivity tools and utilities.
3. Game Development:
   - Games: C++ is widely used in game development due to its high performance and control over system resources.
4. Embedded Systems:
   - Embedded Software: C++ is used in embedded systems for its efficiency and ability to interact with hardware.
5. Finance and Trading Systems:
   - High-Frequency Trading: C++ is employed in financial applications requiring real-time performance and low latency.
6. Scientific Computing:
   - Simulations and Models: Used in scientific research for simulations, modeling, and computational tasks.

Structure of a C++ Program

The structure of a C++ program is designed to be logical and organized, making it easier to read, maintain, and debug. Here's a breakdown of the key components that typically make up a C++ program:

General Syntax	Example Program

/* General Structure */ Pre-processor Directivesmain ( )
 	
{
 	
Variable Declarations; Executable Statements;
 	
}
	/* A Simple Program to display a Message * /
	#include<iostream.h>
	
	void main( )
	
	{
	
	cout<<" This is the first C++ Program";
	
	}

1. Preprocessor Directives

Preprocessor directives are instructions that are processed before the actual compilation of the code begins.

- The linker section begins with a hash (#) symbol. #include …… is a preprocessor directive.
- It is a signal for preprocessor which runs the compiler.
- The statement directs the compiler to include the header file from the C++ Standard library.

They typically include the following:

- Header File Inclusions: These include necessary libraries and header files.

#include <iostream> // Includes the standard input-output stream library

#include <cmath> // Includes the math library

- Macro Definitions: Used to define constants or macros.

#define PI 3.14159 // Defines a macro for the value of PI

2. Namespace Declaration : Namespaces help avoid name conflicts, especially in large projects or when using multiple libraries.

- Standard Namespace:

using namespace std; // Allows the use of standard library names without std:: prefix

3. Global Declarations and Definitions

These are variables, constants, and functions that are declared outside of any class or function, making them accessible from any part of the program.

- Global Variables:

```
int globalVar = 100;  // A global variable accessible throughout the program
```

4. Class Definitions

Classes define the blueprint for objects, encapsulating data and functions that operate on the data.

- Example Class:

```
class Rectangle {

private:

   double width, height; public:

   Rectangle(double w, double h) : width(w), height(h) {} double

   getArea() {

      return width * height;

   }

   void setWidth(double w) {

      width = w;

   }

   void setHeight(double h) {

      height = h;

   }

};
```

5. Function Definitions

Functions define reusable blocks of code. They can be defined globally, within classes, or as member functions.

Example Function:

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

6. Main Function

The main function is the entry point of a C++ program. Every C++ program must have a main function.

- As the name itself indicates, this is the main function of every C++ program.
- Execution of a C++ program starts with main ( ).
- No C++ program is executed without the main () function.
- The function main () should be written in the lowercase letter and shouldn't be terminated with a semicolon.
- It calls other library functions and user-defined functions.
- There must be one and only main ( ) in every C++ program.

Braces { }:

```
{
        …………..;
        …………..;
}
```

- The statements inside any function including the main ( ) function is enclosed with the opening andthe closing braces.

Main Function Example:

```
int main() {
    greet();  // Call the greet function Rectangle
    rect(10.0, 20.0);
    cout << "Area of Rectangle: " << rect.getArea() << endl; return 0;
    // Return 0 to indicate successful execution
}
```

7. Comments

Comments are used to document the code. They are ignored by the compiler.

- Single-line Comment:

```
// This is a single-line comment
```

- Multi-line Comment:

```
/* This is a multi-line comment that
   spans multiple lines */
```

8. Variable Declarations:

- The declaration is the part of the C++ program where all the variables, arrays, and functions aredeclared with their basic data types.
- This helps the compiler to allocate the memory space inside the computer memory.
- Example:
- int sum, x, y;

9. Executable Statements:

- These are instructions to the computer to perform some specific operations.
- These statements can be expressions, input-output functions, conditional statements, looping statements, function call and so on. They also include comments.
- Every statement end with semicolon ";" except control statements.
- Semicolon ";" also known as Terminator.

- Example:

```
cout<<"Welcome to Computer Science Class";
```

10. Additional Elements

- Error Handling: Mechanisms like exception handling (try, catch, throw) to manage errors.
- Template Definitions: Templates allow writing generic functions and classes.

C++ Syntax

Let's break up the following code to understand it better:

```
#include<iostream>
using name space std;

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

Line 1: #include <iostream> is a **header file library** that lets us work with input and output objects, such as cout (used in line 5). Header files add functionality to C++ programs.

Line 2: using namespace std means that we can use names for objects and variables from the standard library.

Line 3: A blank line. C++ ignores white space. But we use it to make the code more readable.

Line 4: Another thing that always appear in a C++ program is int main(). This is called a **function**. Any code inside its curly brackets {} will be executed.

Line 5: cout (pronounced "see-out") is an **object** used together with the insertion operator (<<) to output/print text. In our example, it will output "Hello World!".

Note: Every C++ statement ends with a semicolon ;.

Note: The body of int main() could also been written as:
int main () { cout << "Hello World! "; return 0; }

Remember: The compiler ignores white spaces. However, multiple lines makes the code more readable.

Line 6: return 0; ends the main function.

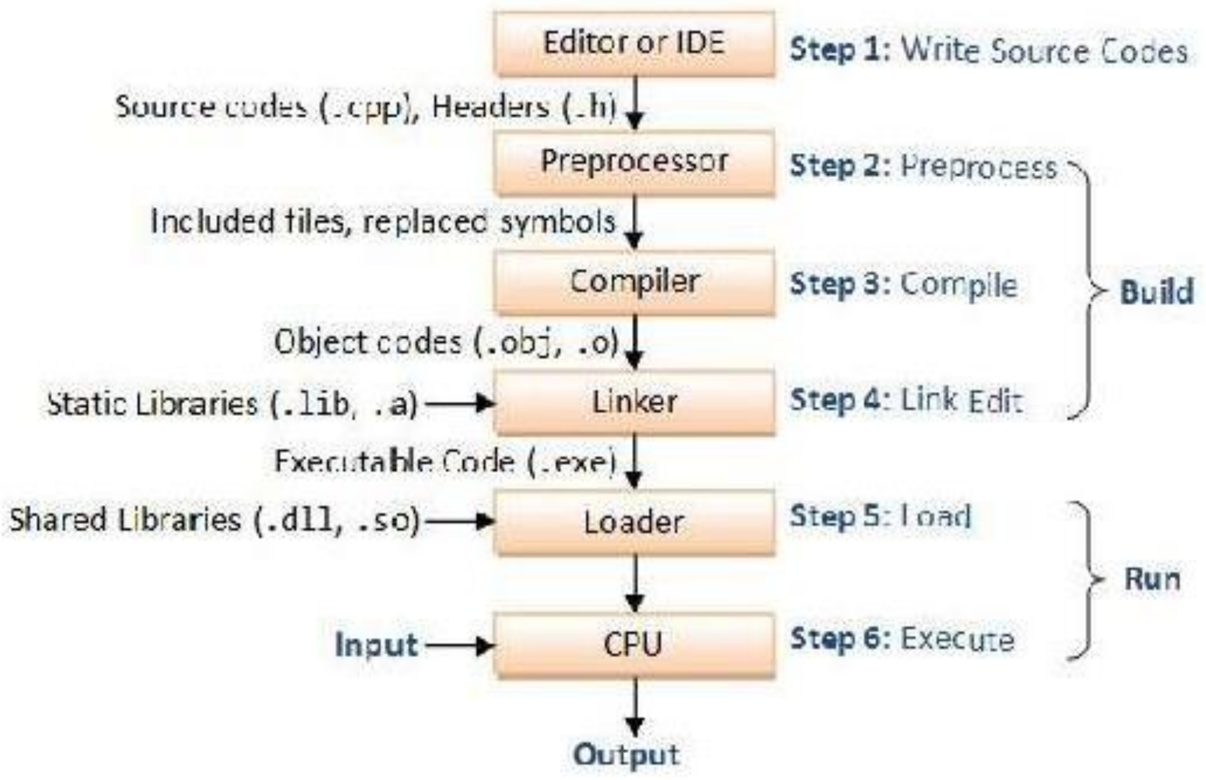
Line 7: Do not forget to add the closing curly bracket } to actually end the main function.

Omitting Namespace

You might see some C++ programs that runs without the standard namespace library. The using namespace std line can be omitted and replaced with the std keyword, followed by the :: operator for some objects:

```
#include <iostream>

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

➤ Translating a C++ program:



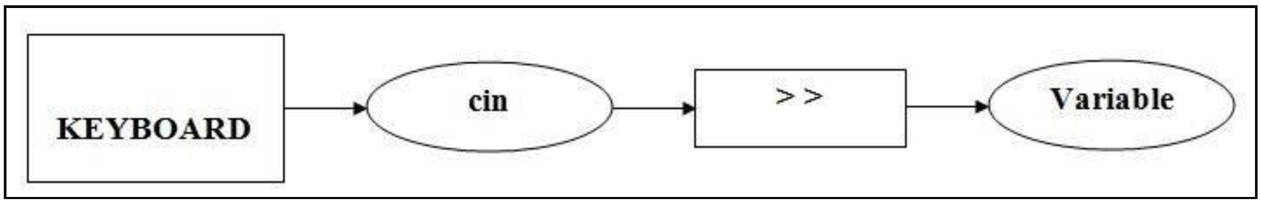
INPUT AND OUTPUT Statements

Introduction

- The input output operations are done using library functions cin and cout objects of the classiostream.

- Using the standard input and output library, we will able to interact with the user by printing
- message on the screen and getting the user's input from the keyboard.
- A stream is an object where a program can either insert/extract characters to/from it.
- The standard C++ library includes the header file iostream, where the standard input and outputstream objects are declared.

Input Operator “>>”:



- The standard input device is usually the keyboard.
- Input in C++ is done by using the "stream extraction" (>>) on the cin stream.
- The operator must be followed by the variable that will store the data that is going to be extractedfrom the stream.

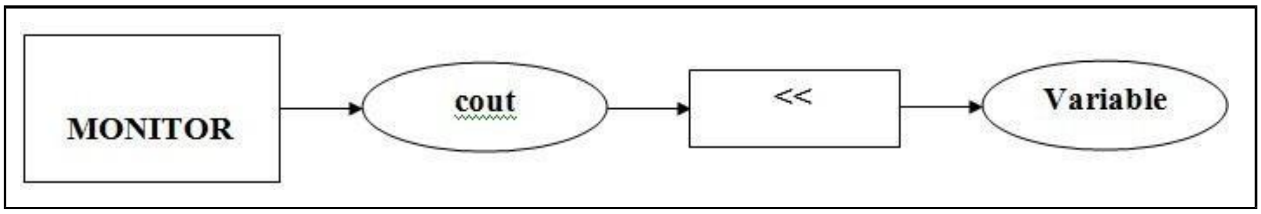
Example:

```
int      age;

cin>>age;
```

- The first statement declares a variable of the type int called age, and the second one waits for aninput from cin (the keyboard) in order to store it in this integer variable.
- cin stands for "console input".
- It can only process the input from the keyboard once the RETURN key has been pressed.
- We must always consider the type of the variable that we are using as a container with cinextraction. For example, if we request an integer we will get an integer.

Output Operator "<<":

- The standard output device is the screen (Monitor).
- Outputting in C++ is done by using the object followed by the "stream insertion" (<<).
- cout stands for console output.



Example:

```
cout<<"Let us learn C++"; // prints Let us learn C++ on the screen.
```

- The << operator inserts the data that follows it into the stream preceding it.

- The sentence in the instruction is enclosed between double quotes (”), because it is constant string
- of characters.
- Whenever we want to use constant strings of characters we must enclose them between double
- quotes (“) so that they can be clearly distinguished from the variables name.

Example:

cout<<”sum”; //prints sum

cout<<sum; //prints the content of the variable sum

- In order to perform a line break on the output we must explicitly insert a new-line character intocout.
- In C++ a new-line character can be specified as ‘\n’ (backslash n), the new-line character is an escape sequence character and helps in formatting the output statement.

- Program: To demonstrate the cout statement:

```
#include<iostream.h>#include<conio.h> void main( )

{

cout<<”C++ is an Obejct Orinted Programming Language”<<”\n”;cout<<”C++ is an case
sensitive language\n”;

getch();

}
```

Cascading of I/O Operators:

- C++ supports the use of stream extraction (>>) and stream insertion (<<) operator many times in asingle input (cin) and output (cout) statements.
- If a program requires more than one input variable then it is possible to input these variables in asingle cin statement using multiple stream extraction operators.
- Similarly, when we want to output more than one result then this can be done using a single coutstatement with multiple stream insertion operators.
- This is called cascading of input output operators.

Example:

```
cout<<"Enter the first number";

cin>>a;

cout<<"Enter the second number"; cin>>b;
```

- Instead of using cin statement twice, we can use a single cin statement and input the two numbersusing multiple stream extraction operators.

```
cout<<"Enter the two number";

cin>>a>>b;
```

- Similarly, we can even output multiple results in a single cout statements using cascading ofstream insertion operators.

```
cout<<"The sum of two number is"<<sum<<endl;
```

Formatted Output (Manipulators) :

- Manipulators are the operators used with the insertion operator << to format the data display. Themost commonly used manipulators are endl and setw.

1. The endl manipulator : The endl manipulator, when used in a output statement , causes a line

feed to be inserted. It has same effect as using new line character "\n".

```
cout<< " JSS college of arts commerce and science"<<endl; cout<< " Ooty road,
Mysuru";
```

2. The setw( ) Manipulator : The setw( ) manipulator sets the width of the field assign for the output. It takes the size of the field (in number of character) as a parameter. The output will be right justified. Example the code :

```
cout<<setw(6)<<"R" ;
```

Generates the following output on the screen (each underscore represent a blank space)

```
_ ______ R
```

In order to use this manipulator, it is must to include header file iomanip.h

C++ Character Set:

- Character Set means the valid set of characters that a language can recognizes.
- The character set of C++ includes the following:

Alphabet s	Upper letters case	A, B, C, D .... X, Y, Z	
	Lower letters case	a, b, c, d................x, y, z	
Digits	0,1,2,3………9		
Special Characters	, comma	. Period	` Apostrophe
	: Colon	; Semicolon	? Question mark
	! Exclamation	_ Underscore	| Pipeline
	{Left brace	} Right Brace	# Hash
	[Left bracket	] Right Bracket	^ Caret
	(Left parenthesis	) Right parenthesis	& ampersand
	/ Slash	\ Back slash	~ Tilde
	+ Plus sign	- Minus Sign	< Less Than
	* Asterisk	% Percentage	> Greater Than

C++ Tokens:

- The smallest individual unit in a program is known as token.
- These elements help us to construct statements, definitions, declarations, and so on, which in turnhelps us to construct complete program.
- Tokens used in C++ are:
  1. Identifier
  2. Reserved Keywords
  3. Constants or Literals
  4. Punctuators
  5. Operators

C++ Variables

Variables are containers for storing data values.

In C++, a variable is a named storage location in memory that holds a value. Variables are fundamental to programming as they allow the storage and manipulation of data. Each variable has a type that determines the kind of data it can hold, such as integers, floating-point numbers, characters, etc.

In C++, there are different types of variables (defined with different keywords), for example:

- int - stores integers (whole numbers), without decimals, such as 123 or -123
- double - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- string - stores text, such as "Hello World". String values are surrounded by double quotes
- bool - stores values with two states: true or false
- Declaring (Creating) Variables
- To create a variable, specify the type and assign it a value:

Syntax : type variableName = value;

Where type is one of C++ types (such as int), and variableName is the name of the variable (such as **x** or **myName**). The equal sign is used to assign values to the variable.

To create a variable that should store a number, look at the following example: Create a

variable called **myNum** of type int and assign it the value **15**:

```
int myNum = 15; cout
<< myNum;
```

You can also declare a variable without assigning the value, and assign the value later: int

```
myNum;
myNum                                                    = 15;
cout << myNum;
```

Note that if you assign a new value to an existing variable, it will overwrite the previous value:

```
int myNum = 15;  // myNum is 15
myNum = 10;  // Now myNum is 10
cout << myNum;  // Outputs 10
```

Other Types

A demonstration of other data types:

```
int myNum = 5;               // Integer (whole number without decimals)
double myFloatNum = 5.99;    // Floating point number (with decimals)
char myLetter = 'D';         // Character
```

string myText = "Hello"; // String (text)
bool myBoolean = true; // Boolean (true or false)

Display Variables

The cout object is used together with the << operator to display variables. To

combine both text and a variable, separate them with the << operator: int myAge

= 35;
cout << "I am " << myAge << " years old.";

Add Variables Together

To add a variable to another variable, you can use the + operator: int x =

5;
int y = 6;
int sum = x + y; cout
<< sum;

C++ Declare Multiple Variables

Declare Many Variables

To declare more than one variable of the same type, use a comma-separated list: int x = 5,

y = 6, z = 50;
cout << x + y + z;

One Value to Multiple Variables

You can also assign the same value to multiple variables in one line: int x, y,

z;
x = y = z = 50;
cout << x + y + z

C++ Identifiers

All C++ **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

In C++, identifiers are names used to identify variables, functions, classes, objects, arrays, labels, and other user-defined items. Identifiers play a crucial role in giving meaningful names to various elements in the code, making it more readable and maintainable.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Note: It is recommended to use descriptive names in order to create understandable and maintainable code:

```
// Good
int minutesPerHour = 60;

// OK, but not so easy to understand what m actually is
int m = 60;
```

The general rules for naming variables are:

- Names can contain letters, digits and underscores
- Names must begin with a letter or an underscore (_)
- Names are case-sensitive (myVar and myvar are different variables)
- Names cannot contain whitespaces or special characters like !, #, %, etc.
- Reserved words (like C++ keywords, such as int) cannot be used as names

In C++, an identifier is a name used to identify a variable, function, class, object, module, or any other user-defined item. Identifiers are essential for naming and differentiating various elements within a program, making the code more readable and maintainable.

Rules for Naming Identifiers

1. Characters Allowed:
   - Identifiers can contain letters (both uppercase and lowercase), digits (0-9), and underscores (_).
   - The first character must be a letter or an underscore; it cannot be a digit.
2. Case Sensitivity:
   - Identifiers in C++ are case-sensitive, meaning Variable and variable would be considered different identifiers.
3. No Reserved Words:
   - Identifiers cannot be the same as C++ keywords or reserved words, such as int, float, if, else, etc.
4. No Special Characters:
   - Identifiers cannot include special characters like @, #, $, %, etc., except for the underscore (_).
5. Length:
   - While there's technically no limit to the length of an identifier, it is good practice to keep them reasonably short for ease of reading and maintenance. However, C++ compilers may have limits on the maximum number of characters that are significant.

Best Practices for Naming Identifiers

1. Meaningful Names:
   - Use meaningful names that convey the purpose of the variable or function. For example, use age instead of a, or calculateTotal instead of ct.
2. Use Camel Case or Underscores:
   - For multi-word identifiers, use camel case (e.g., totalAmount) or underscores (e.g., total_amount) for better readability.
3. Consistent Naming Conventions:

   - Stick to a consistent naming convention throughout the codebase. For example, if you use camel case for variables, maintain that style across all variable names.
4. Avoid Starting with Underscores:
   - Although legal, avoid starting identifiers with an underscore, especially followed by a capital letter (e.g., _Identifier), as these are typically reserved for implementation-specific names or are used in the C++ Standard Library.
5. Avoid Single-Letter Identifiers:
   - Except for loop counters (like i, j, k), avoid using single-letter identifiers, as they are not descriptive.

Examples of Valid Identifiers

- age
- _total
- sumOfNumbers
- calculateArea
- temperature_in_Celsius
- Examples of Invalid Identifiers
- 123abc (starts with a digit)
- int (reserved keyword)
- total-amount (contains a special character -)
- class (reserved keyword)

Identifiers and Scope

The scope of an identifier determines where it can be accessed in the program. For instance, a variable defined within a function (local variable) can only be accessed within that function. In contrast, a global variable can be accessed from any part of the program.

Example:

```
int globalVar = 10;  // Global identifier void

someFunction() {

    int localVar = 20;  // Local identifier

}

int main() {

    // globalVar can be accessed here
    // localVar cannot be accessed here return

    0;

}
```

Keywords

Keywords in C++ are reserved words that have special meanings defined by the language. These words are integral to the syntax and cannot be used as identifiers (such as variable names, function names, or class names) in the program. C++ keywords are predefined and used for various purposes like defining data types, controlling flow, handling exceptions, and more.

Here's a categorized list of C++ keywords along with brief descriptions:

Data Types

- int: Defines an integer type.
- char: Defines a character type.
- float: Defines a floating-point type.
- double: Defines a double-precision floating-point type.
- bool: Defines a Boolean type, representing true or false.
- void: Specifies that a function does not return a value or defines a pointer with no type.

Modifiers

- signed: Specifies that a variable can hold negative or positive values (default for integers).
- unsigned: Specifies that a variable can only hold non-negative values.
- long: Increases the size of the data type.
- short: Reduces the size of the data type.

Control Flow

- if: Introduces a conditional statement.
- else: Specifies the alternative block of code to execute if the if condition is false.
- switch: Allows multi-way branching based on the value of an expression.
- case: Defines a branch within a switch statement.
- default: Specifies the default block of code in a switch statement when no case matches.

- for: Defines a loop with an initialization, condition, and increment/decrement.
- while: Defines a loop that continues while a condition is true.
- do: Defines a loop that executes at least once before checking a condition.
- break: Exits from the nearest enclosing loop or switch statement.
- continue: Skips the remaining statements in the current loop iteration and proceeds to the next iteration.
- goto: Transfers control to a labeled statement (not recommended for use due to readability and maintenance issues).

Storage Classes

- auto: (C++11) Deduces the type of the variable automatically.
- register: Suggests that a variable be stored in a CPU register for faster access.
- static: Preserves the value of a variable between function calls or restricts the visibility of a function/variable within its translation unit.
- extern: Specifies that the variable is defined elsewhere, typically in another file.
- mutable: Allows a member of an object to be modified even if the object is declared as const.

Type Qualifiers

- const: Specifies that a variable's value cannot be changed.
- volatile: Indicates that a variable may be changed unexpectedly, preventing compiler optimizations.
- Function and Variable Qualifiers
- inline: Suggests that a function should be expanded inline, rather than through the normal function call mechanism.
- virtual: Specifies that a method can be overridden in a derived class.
- explicit: Prevents the compiler from using implicit conversions for a constructor or conversion operator.
- friend: Grants a function or another class access to the private and protected members of the class.

Exception Handling

- try: Begins a block of code that will test for exceptions.

- catch: Specifies a block of code to execute when a specific exception type is thrown.

- throw: Used to throw an exception.

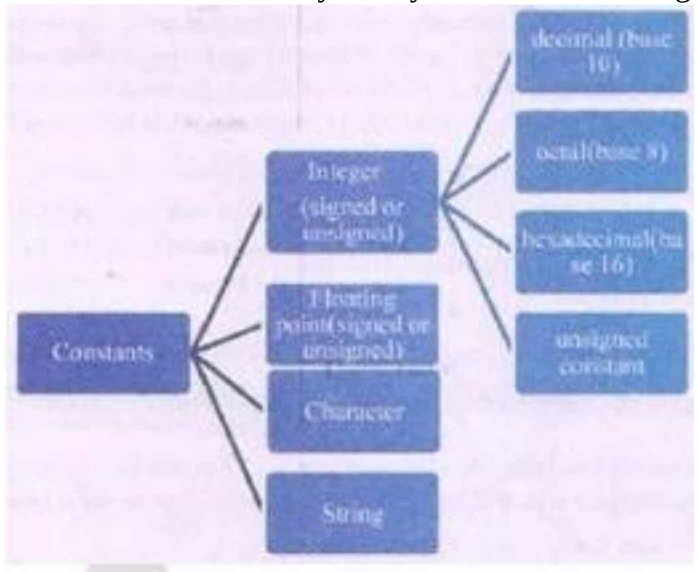
Miscellaneous

- new: Allocates memory dynamically on the heap.
- delete: Deallocates memory allocated by new.
- this: A pointer to the current object instance.
- return: Exits a function and optionally returns a value.
- namespace: Defines a scope to prevent name conflicts.
- using: Imports names from a namespace into the current scope.
- typeid: Returns information about the type of an object.
- typename: Used in template programming to specify a type.
- operator: Used to define or overload operators.
- sizeof: Returns the size of a data type or object.
- dynamic_cast: Safely casts a pointer or reference to a base class to a pointer or reference to a derived class.
- static_cast: Performs a non-polymorphic cast.
- const_cast: Adds or removes const to/from a variable.
- reinterpret_cast: Converts any pointer type to any other pointer type, even if the types are not related.
- enum: Defines an enumeration, a distinct type consisting of a set of named values.
- struct: Defines a structure, a user-defined data type.
- union: Defines a union, a data structure that can store different data types in the same memory location.
- class: Defines a class, a blueprint for objects.
- private, protected, public: Access specifiers for class members.
- template: Used to define generic classes or functions.
- decltype: (C++11) Determines the type of an expression.
- nullptr: (C++11) A null pointer constant.
- static_assert: (C++11) A compile-time assertion.

- noexcept: (C++11) Specifies that a function does not throw exceptions.

Constants:

- A constant are identifiers whose value does not change during program execution.
- Constants are sometimes referred to as literal
- A constant or literal my be any one of the following:



- Integer Constant
- Floating Constant
- Character Constant
- String Constant

Integer Constant:

- An integer constant is a whole number which can be eitherpositive or negative.
- They do not have fractional part or exponents.
- We can specify integer constants in decimal, octal or hexadecimal form.

Decimal Integer Constant: It consists of any combination of digits taken from the set 0 to 9.For example:

```
int a = 100;                    //Decimal Constant

int b = -145                    // A negative decimal constant

int c = 065                     // Leading zero specifies octal constant, not
decimal
```

Octal Integer Constant: It consists of any combination of digits taken from the set 0 to 7. However the first digit must be 0, in order to identify the constant as octal number.

For example:

```
int a = 0374;                   //Octal Constant
```

int b = 097; // Error: 9 is not an octal digit.

Hexadecimal Integer Constant: A Sequence of digits begin the specification with 0X or 0x,followed by a sequence of digits in the range 0 to 9 and A (a) to F(f).

For example:

int a = 0x34; int b = -0XABF;

Unsigned Constant: To specify an unsigned type, use either u or U suffix. To specify a longtype, use either the l or L suffix.

For example:

unsigned a = 328u; //Unsigned value

long b = 0x7FFFFFL; //Long value specified as hex constant unsigned long c = 0776745ul;//Unsigned long values as octal constant

Floating Point Constant:

- Floating point constants are also called as “real constants”.
- These values contain decimal points (.) and can contain exponents.
- They are used to represent values that will have a fractional part and can be represented in two forms(i.e. fractional form and exponent form)
- Floating-point constants have a “mantissa”, which specifies the value of the number, an “exponent”
- which specifies the magnitude of the number, and an optional suffix that specifies the constant’s type.
- The mantissa is specified as a sequence of digits followed by a period, followed by an optional sequence of digits representing the fractional part of the number.
- The exponent, if present, specifies the magnitude of the number as a power of 10.
- Example: 23.46e0 // means it is equal to 23.46 x 10^0 = 23.46 x 1 = 23.46
- It may be a positive or negative number. A number with no sign is assumed to be a positive number.For example, 345.89, 3.142

Character Constants:

- Character constants are specified as single character enclosed in pair of single quotation marks.
- For example char ch = ‘P’; //Specifies normal character constant

- A single character constant such as ‘D’ or ‘r’ will have char data type. These character constants will be assigned numerical values.
- The numerical values are the ASCII values which are numbered sequentially for both uppercase and lowercase letters.
- For example, ASCII value of A is 65, B is 66, …..Z is 90 (uppercase), a is 97, b is 98……. Z is 122 (lowercase), 0 is 48, 1 is 49, …… 9 is 57 (digits).

- There are certain characters used in C++ which represents character constants. These constants start with a back slash (\ ) followed by a character. They are normally called as escape sequence. Someof the commonly used escape sequences are.

Escape Sequence	Meaning	Escape Sequence	Meaning
\’	Single Quote	\”	Double Quote
\?	Question Mark	\\	Back Slash
\0	Null Character	\a	Audible Bell
\b	Backspace	\f	New Page
\n	New Line	\r	Carriage Return
\t	Horizontal Tab	\v	Vertical Tab
\nnn	Arbitrary octal value	\xnn	Arbitrary Hexa Value

- Escape Sequence is a special string used to control output on the monitor and they are represented by a single character and hence occupy one byte.

String Constants:

- A string constant consists of zero or more character enclosed by double quotation marks (“).
- Multiple character constants are called string constants and they are treated as an array of char.
- By default compiler adds a special character called the “Null Character” (\0) at the end of the stringto mark the end of the string.
- For example: char str[15] = “C++ Programming” ;
- This is actually represented as char str[15] = “C++ Programming\0” in the

memory.

C++ Operators:

- An operator is a symbol that tells the compiler to perform specific mathematical or logicalmanipulations.
- C++ is rich in built-in operators and there are almost 45 different operators.
- Operators in C++ are can be divided into the following classes:
- Arithmetic Operator
- Relational Operator
- Logical Operator
- Unary Operator
- Conditional Operator
- Bitwise Operator
- Assignment Operator
- OtherOperator

Operator operates on constants and variables which are called operands. Operators may also be classified on the number of operands they act on either:

Unary	Binary	Ternary
Unary operators operate on only one operand.	The binary operator operates on two operands.	The ternary operator operates on three operands.
Example: ++, - -	+, -, *, /, %, &&, ||	?:

Unary Operators

- Unary operators have only one operand; they are evaluated before any other operation containingthem gets evaluated.
- The following are the list of unary operators.

Operator	Name	Function
!	Logical NOT	If a condition is true then Logical NOT operator will make false.
&	Address-of	Used to give the address of the operand
~	One's Complement	Converts 1 to 0 and 0 to 1
*	Pointer dereference	Used along with the operand to represent the pointer data type.
+	Unary plus	Used to represent a signed positive operand
++	Increment	Used to increment an operand by 1
-	Unary negati n	Used to represent a signed negative operand
- -	Decrement	Used to represent an operand by 1

- **Increment Operator**

Increment operator is used to increasing the value of an integer by one. This is represented by “++”.

Example: a++, a+1

Decrement Operator

Decrement operator is used to decreasing the value of an integer by one. This is represented by “--”.

Example: a--, a-1 Let

a=10 and b=5

a++; //a becomes 11 b--; //b becomes4

Both the increment & decrement operators come in two versions:

Prefix increment/decrement:

- When an increment or decrement operator precedes its operand, it is called prefix increment ordecrement (or pre-increment / decrement).
- In prefix increment/decrement, C++ performs the increment or decrement operation before using thevalue of the operand.
- Example: If sum = 10 and count =20 then
- Sum = sum + (++count);
- First count incremented and then evaluate sum = 31.

Postfix increment/decrement:

- When an increment or decrement operator follows its operand, it is called postfix increment ordecrement (or post-increment / decrement).
- In postfix increment/decrement, C++ first uses the value of the operand in evaluating the expression
- before incrementing or decrementing the operand’s value.
- Example: If sum = 10 and count =20 then
- Sum = sum + (count++);
- First evaluate sum = 30, and then increment count to 21.

Binary Operators

- The binary operators are those operators that operate on two operands. They are as arithmetic,relational, logical, bitwise, and assignment operators.

Arithmetic Operator

- Arithmetic operators are used to performing the basic arithmetic operations such as arithmetic,subtraction, multiplication, division and modulo division (remainder after division).

Operator	Description	Example(a=10, b=20)
+	Adds two operand	a + b = 30
-	Subtracts second operand from the first	a - b = -10
*	Multiply both operand	a * b = 200
/	Divide numerator by denominators	b / a = 2
%	Modulus operators and remainder of after an integer division	b % a = 0

Relational Operator

- Relational Operator is used to comparing two operands given in expressions.
- They define the relationship that exists between two constants.
- For example, we may compare the age of two persons or the price of two items….these comparisons can be done with the help of relational operators.
- The result in either TRUE(1) or FALSE(0).Some of the relational operators are:

Operator	Description	Example (a=10, b=5)
<	Checks if the value of left operand is less than the value of right operand	a < b returns false(0)
<=	Checks if the value of left operand is less than or equal to the value of right operand	a <= b returns false(0)
>	Checks if the value of left operand is greater than the value of right operand	a > b returns true(1)

>=	Checks if the value of left operand is greater than or equal to the value of right operand	a >= b returns

		false(0)
= =	Checks if the value of two operands is equal or not	a = = b returns false(0)
! =	Checks if the value of two operands is equal or not	a != b returns true(1)

Logical Operators

- Logical operators are used to testing more than one condition and make decisions.

  Some of the logicaloperators are

Ope rato r	Mean ing	Description	Example
&&	Logic al AND	If both the operands are non-zero then condition becomes true.	If a=10 and b=5 then, ((a==10) && (b>5)) returns false.
||	Logic al OR	If any of the two operands is non- zero then condition becomes true.	If a=10 and b=5 then, ((a==10) || (b>5)) returns true.
!	Logic al NOT	If a condition is true then the Logical NOT operator will make false.	If a=10 then, !(a==10) returns false.

Bitwise Operators

- A bitwise operator works on bits and performs bit by bit operation.
- Bitwise operators are used in bit level programming.

Operat ors	Meaning of operators

&	Bitwise AND

|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement

- The truth table for bitwise AND (&), Bitwise OR(|), Bitwise XOR (^) are as follows:

A	B	A&B (Bitwise AND)	A | B (Bitwise OR)	A ^ B (Bitwise XOR)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

- Assume A = 60 and B =13; the following operations take place:

Step 1: Converts A and B both to its binary equivalent.

A = 0011 1100

B = 0000 1101

Step 2: Then performs the bitwise and, or and not operation. The result is given below. A & B

= 0000 1100 = 12

A | B = 0011 1101 = 61

A^B = 0011 0001 = 49

~A = 1100 0011 = -60

The Bitwise operators supported by C++ are listed in the following table:

Operator	**Descri ption**	**Exampl e**
&	Binary AND Operator copies a bit to the result if it exists in both operands	(A&B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A|B) will give 61 which is 0011 1101

^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A^B) will give 49 which is 0011 0001
~	Binary Ones complement Operator is unary and has the effect of ‘Flipping’ bits	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A<<2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A>>2 will give 15 which is0000 1111

Assignment Operators

- The most common assignment operator is =. This operator assigns the value on the right side to the lft side.

Example :

var = 5 //5 is assigned to var

a = b; //value of b is assigned to a 5 =

b; // Error! 5 is a constant.

- The assignment operators supported by C++ are listed below:

Operator	Example	Same as
=	a=b	a=b
+=	a+=b	a=a+b
-=	a-=b	a=a-b
*=	a*=b	a=a*b
/=	a/=b	a=a/b
%=	a%=b	a=a%b
<< =	a<<=2	a = a<<2
>>=	a>>=2	a = a>>2
&=	a & = 2	a = a & 2
^=	a ^ = 2	a = a ^ 2
|=	a | = 2	a = a | 2

C++ Shorthand’s:

- C++ Offers special shorthand's that simplify the coding of a certain type of assignment statements.
- The general format of C++ shorthand's is:
- Variable Operator = Expression
- Following are some examples of C++ shorthand's:

x - = 10;	Equivalent to	x = x – 10;
x * = 5;	Equivalent to	x = x * 5;
x / = 2 ;	Equivalent to	x = x / 2;
x % = z;	Equivalent to	x = x % z;

Conditional Operator:

- A ternary operator pair "? :" is available in C++ to construct conditional expressions of the form:
- **exp1? exp2: exp3,** where exp1,exp2, and exp3 are expressions,
- The operator "?:" works as follows: exp1 is evaluated first. If it is true, then the expression exp 2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression.

- **Example:** a=10; b=5;

  x = (a>b) ? a:b;

Special Operator:

Operators	Meaning of operators
sizeof()	It is a unary operator which is used in finding the size of the data type. Example: sizeof(a)
, (comma)	Comma operators are used to linking related expressions together. Example: int a=10, b=5
. (dot) and -> (arrow)	Member Operator used to reference individual members of classes, structure and unions.
● **cast**	Casting Operator convert one data type to another.
&	Address Operator & returns the address of the variable.
*****	Pointer Operator * is pointer to a variable.

- **Sizeof Operator:** Returns the size of a data type or object in bytes.

  int a = 10;
  std::cout << sizeof(a); // Output: size of int (usually 4 bytes) std::cout << sizeof(float); // Output: size of float (usually 4 bytes)

- **Comma Operator:** Used to separate multiple expressions where only one expression is expected.

```
int a = 1, b = 2;
int result = (a = a + b, b = a - b); // result = 3, a = 3, b = 1
```

- **Conditional (Ternary) Operator:** condition ? expr1 : expr2

```
int a = 10; int
b = 20;
int max = (a > b) ? a : b; // max = 20
```

- **Pointer Operators:** & (address-of), * (dereference)

```
int a = 10;
int *ptr = &a; // ptr holds the address of a
int b = *ptr;  // b is now 10, the value pointed to by ptr
```

- **Type Casting:** Converts a variable from one type to another.
    - **C-style:** (type)expression

    ```
    double d = 5.5;
    int i = (int)d; // i = 5
    ```

    - *C++ Casts:*
        - static_cast<type>(expression)
        - dynamic_cast<type>(expression)
        - const_cast<type>(expression)
        - reinterpret_cast<type>(expression)

Example:

```
double d = 5.5;
int i = static_cast<int>(d); // i = 5
```

Precedence of Operators or Hierarchy of Operators In C++:

- An expression is a combination of opcode and operand.
- The operators would be arithmetic, relational, and logical operators.
- If the expression contains multiple operators, the order in which operations carried out is called theprecedence of operators. It is also called as priority or hierarchy.
- The Operators with highest precedence appear at the top of the table and those with the lowest appearat the bottom.

Category	**Operator**	**Associativity**
Postfix	( ) [ ] -> . ++ --	Left to Right
Unary	= - ! ~ ++ -- (type) * & sizeof	Right to Left
Multiplicative	* / %	Left to Right
Additive	+ -	Left to Right
Shift	<< >>	Left to Right

Relational	<<= >>=	Left to Right
Equality	== !=	Left to Right
Bitwise AND	&	Left to Right
Bitwise XOR	^	Left to Right
Bitwise OR	|	Left to Right
Logical AND	&&	Left to Right
Logical OR	||	Left to Right
Conditional	?:	Right to Left
Assignment	= += -= *= /= %=	Right to Left
Comma	,	Left to Right

Expressions

Expressions in C++ are combinations of variables, operators, and function calls that are evaluated to produce a value. Expressions are a fundamental aspect of programming, as they form the basis for performing computations, making decisions, and controlling the flow of a program. The value resulting from the evaluation of an expression can be of any data type, depending on the elements involved.

Types of Expressions

1. **Arithmetic Expressions**
2. **Relational Expressions**
3. **Logical Expressions**
4. **Bitwise Expressions**
5. **Assignment Expressions**
6. **Conditional Expressions**
7. **Combinations of Expressions**

1. Arithmetic Expressions

Arithmetic expressions involve arithmetic operators and evaluate to numerical values.

Example:

```
int a = 5; int
b = 10;
int c = a + b * 2; // c = 25
```

In the example, a + b * 2 is an arithmetic expression that results in 25.

2. Relational Expressions

Relational expressions compare two values using relational operators and evaluate to a boolean value (true or false).

Example:

```
int a = 5; int
b = 10;
bool result = (a < b); // result = true
```

Here, (a < b) is a relational expression that evaluates to true.

3. Logical Expressions

Logical expressions use logical operators to combine multiple boolean expressions and also evaluate to a boolean value.

Example:

```
bool a = true;
bool b = false;
bool result = (a && b) || (!a); // result = false
```

The expression (a && b) || (!a) combines logical operations and evaluates to false.

4. Bitwise Expressions

Bitwise expressions perform operations at the binary level on integer types using bitwise operators.

Example:

```
int a = 5;      // Binary: 0101
int b = 3;      // Binary: 0011
int result = a & b; // result = 1 (Binary: 0001)
```

In this case, a & b performs a bitwise AND operation.

5. Assignment Expressions

Assignment expressions assign a value to a variable and evaluate to the assigned value.

Example:

```
int a;
a = 10; // a is assigned the value 10
```

Here, a = 10 assigns the value 10 to a and the expression itself evaluates to 10.

6. Conditional Expressions

Conditional expressions, also known as ternary expressions, are a shorthand for if-else conditions.

Syntax:

```
condition ? expression1 : expression2;
```

Example:

```
int a = 10;
int b = 20;
int max = (a > b) ? a : b; // max = 20
```

The expression (a > b) ? a : b evaluates to a if a > b is true, otherwise it evaluates to b.

7. Combinations of Expressions

Expressions can be combined and nested to perform complex computations and evaluations.

Example:

```
int a = 10, b = 20, c = 30;
bool result = (a + b > c) && (c - b < a); // result = true
```

This combined expression evaluates multiple sub-expressions and logical operations.

Expression Evaluation

The evaluation of expressions in C++ follows specific rules:

1. **Operator Precedence:** Determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before operators with lower precedence.
2. **Associativity:** Defines the order of evaluation for operators of the same precedence. Operators can be left-associative (evaluated left-to-right) or right-associative (evaluated right-to-left).
3. **Type Conversion:** During the evaluation of expressions, implicit type conversions may occur to match operand types.

Example:

```
int a = 5; float
b = 2.5;
float result = a + b; // Implicit conversion of 'a' to float, result = 7.5
```

Here's a summary of operator precedence and associativity in C++:

Highest Precedence (evaluated first) to Lowest Precedence (evaluated last)

1. **Primary Operators:**
   - () (Function call) [Left-to-right]
   - [] (Array subscript) [Left-to-right]

- . (Member access) [Left-to-right]
- -> (Member access through pointer) [Left-to-right]
- ++ (Post-increment) [Left-to-right]
- -- (Post-decrement) [Left-to-right]

2. *Unary Operators:*
    - ++ (Pre-increment) [Right-to-left]
    - -- (Pre-decrement) [Right-to-left]
    - + (Unary plus) [Right-to-left]
    - - (Unary minus) [Right-to-left]
    - ! (Logical NOT) [Right-to-left]
    - ~ (Bitwise NOT) [Right-to-left]
    - * (Dereference) [Right-to-left]
    - & (Address-of) [Right-to-left]
    - sizeof (Size of) [Right-to-left]
    - new, new[] (Dynamic memory allocation) [Right-to-left]
    - delete, delete[] (Dynamic memory deallocation) [Right-to-left]
    - typeid (Type information) [Right-to-left]
3. *Type Casting Operators:*
    - static_cast, dynamic_cast, const_cast, reinterpret_cast [Right-to-left]
4. *Pointer-to-member Operators:*
    - .* (Pointer to member access) [Left-to-right]
    - ->* (Pointer to member access through pointer) [Left-to-right]
5. *Multiplicative Operators:*
    - * (Multiplication) [Left-to-right]
    - / (Division) [Left-to-right]
    - % (Modulus) [Left-to-right]
6. *Additive Operators:*
    - + (Addition) [Left-to-right]
    - - (Subtraction) [Left-to-right]
7. *Shift Operators:*
    - << (Left shift) [Left-to-right]
    - >> (Right shift) [Left-to-right]
8. *Relational Operators:*
    - < (Less than) [Left-to-right]
    - <= (Less than or equal to) [Left-to-right]
    - > (Greater than) [Left-to-right]
    - >= (Greater than or equal to) [Left-to-right]
9. *Equality Operators:*
    - == (Equal to) [Left-to-right]
    - != (Not equal to) [Left-to-right]
10. *Bitwise AND Operator:*
    - & (Bitwise AND) [Left-to-right]
11. *Bitwise XOR Operator:*
    - ^ (Bitwise XOR) [Left-to-right]
12. *Bitwise OR Operator:*
    - | (Bitwise OR) [Left-to-right]
13. *Logical AND Operator:*
    - && (Logical AND) [Left-to-right]
14. *Logical OR Operator:*

- || (Logical OR) [Left-to-right]

15. *Conditional (Ternary) Operator:*

- ? : (Conditional) [Right-to-left]

16. *Assignment Operators:*
    - = (Assignment) [Right-to-left]
    - += (Add and assign) [Right-to-left]
    - -= (Subtract and assign) [Right-to-left]
    - *= (Multiply and assign) [Right-to-left]
    - /= (Divide and assign) [Right-to-left]
    - %= (Modulus and assign) [Right-to-left]
    - &= (Bitwise AND and assign) [Right-to-left]
    - |= (Bitwise OR and assign) [Right-to-left]
    - ^= (Bitwise XOR and assign) [Right-to-left]
    - <<= (Left shift and assign) [Right-to-left]
    - >>= (Right shift and assign) [Right-to-left]
17. *Comma Operator:*
    - , (Comma) [Left-to-right]

Notes on Operator Precedence and Associativity

- **Higher Precedence:** Operators with higher precedence are evaluated first. For example, multiplication (*) has higher precedence than addition (+), so in the expression 3 + 5 * 2, the multiplication is performed first, resulting in 13 rather than 16.
- **Associativity:** When operators of the same precedence level appear in an expression, associativity determines the order of evaluation. For most operators, associativity is left-to-right, meaning expressions are evaluated from left to right. For example, in a - b - c, the subtraction is performed left-to-right. Some operators, like the assignment and conditional operators, have right-to-left associativity.
- **Parentheses:** Parentheses () can be used to explicitly specify the order of evaluation, overriding the default precedence and associativity rules. For example, in the expression (3 + 5) * 2, the addition is performed first due to the parentheses.

Example:

```
int a = 5, b = 10, c = 15;
int result = a + b * c; // result = 5 + (10 * 15) = 155
int result2 = (a + b) * c; // result2 = (5 + 10) * 15 = 225
```

DATA TYPES

Introduction

- To understand any programming languages we need to first understand the elementary conceptswhich form the building block of that program.
- The basic building blocks include the variables, data types etc.
- C++ provides a set of data types to handle the data that is used by the program

Variable:

- A variable is an object or element and it is allowed change during the execution of the program.
- Variable represents the name of the memory location.

Declaration of a variable:

The syntax for declaring a variable is:

datatype variable_name;

- The variable_name is an identifier. These variables are used to denote constants, arrays, function,structures, classes and files.
- The variables are named storage location whose values can be manipulated during program run.
- Examples:

Some valid variables are:

reg_no, marks, name, student1, dob;

Some invalid variables are:

Double	-	keyword cannot be name of the variable.
Total marks	-	empty spaces are not allowed between
variables names2student	-	variable name should be begin with an alphabet
?result	-	variable should begin with alphabet or underscore only.

Initializing a variable:

- The syntax to initialize a variable is:

data_type variable_name = value;

- Example: Let b be a variable declared of
  the type int. thenint b = 100;
- There are two values associated with a variable known as lvaue and rvalue. It means, for example,let p be a variable declared of the type int. then

```
int    p = 100;
```

- Here, name of the variable is p values assigned to variable is 100 i.e. rvaluememory address location is 2000 i.e. lvalue
- Lvalue is the location value. It holds the memory address location at which the data value is stored.
- Rvalue is the data value. It holds the value assigned to the variable by the

programmer i.e. Rvalueof p = 100.

- C++ compiler allows us to declare a variable at run time. This is dynamic initialization. They canbe initialized anywhere in the program before they are used.
- The access modifier ‘const’ prefixed along with the data type for a variable does not allow thevalue to be changed at all throughout the program.

  cont int a = 100;
- The keyword const becomes an access modifier for the variables.
  - A value can be assigned to lvalue only in an expression.
  - Value can be assigned to a variable using the assignment operator ‘=’.
  - The expression to the left of an assignment operator should always be an lvalue (memorylocation) because that memory location should be available to store the rvalue.
  - Constant identifiers can appear to the right of the assignment operator only since are notlvalues.

Data Types:

- Data Types can be defined as the set of values which can be stored in a variable along with theoperations that can be performed on those values.
- The main aim of C++ program is to manipulate data.
- C++ defines several types of data and each type has unique characteristics.
- C++ data types can be classified as:
- The fundamental data type(built-in data)
- Derived Data type
- User-defined data type
- The simple or fundamental data types are the primary data types which are not composed of anyother data types.
- The simple data types/fundamental data types include int, char, float, double and void.

These are the basic data types provided by C++:

- *Integral Types:*
  - int: Standard integer type.
  - char: Character type, typically used to represent single characters.
  - bool: Boolean type, representing true or false.
  - wchar_t: Wide character type, used for characters of larger character sets.
  - short: Short integer type, typically smaller than int.
  - long: Long integer type, typically larger than int.
  - unsigned: Modifier indicating non-negative integers.
- *Floating-Point Types:*

- float: Single-precision floating-point type.

- double: Double-precision floating-point type.
- long double: Extended precision floating-point type.

Example of Integral and Floating-Point Types:

```
int age = 25; char
initial = 'A';
bool isStudent = true;
float height = 5.9f;
double distance = 12345.6789;
long double preciseValue = 1.234567890123456789L;
```

The int type:

- The **int** type is used to store integers.
- Integers are whole numbers without any fractional parts.
- This includes number such as 1, 45 and -9 are integers.
- 5.2 is not an integer because it contains a decimal point.
- The integer can be positive or negative values and the ranges of number we can store are from
- -32786 to 32767.
- An integer is allocated 2 bytes (16 bits) of memory space.
- The possible operations include addition, subtraction, multiplication, division, remainder etc.
- The General form of an integer declaration is:
    - **int** variable_name;
- **Example:** **int** a, b=5;

The char type:

- It is character data type to store any character from the basic character set.
- Characters are enclosed in single quotation marks (‘). ‘A’, ‘a’, ‘b’, ‘9’, ‘+’ etc. are character
- constants.
- When a variable of type char is declared, the compiler converts the character to its equivalentASCII code.
- A character is allocated only 1 byte (8 bits) of memory space.
- A character is represented in a program by the keyboard **char.**
- The general form of a character declaration is:

char variable_list; **Example:** **char** alpha=’a’;

The float type:

- This represents the number with fractional part i.e. real numbers.
- The **float** type is used to store real numbers.
- Number such as 1.8, 4.5, 12e-5 and -9.66 are all floating point numbers.
- It can also be both positive and negative. The range of numbers we can store from -34e-38 to3.4e38.
- Float is allocated 4 bytes (32 bits) of memory space.
- The general form of a float declaration is:
  - **float** variable_name;
- **Example:** **float** a=5.5;

The double type:

- The double and float are very similar. The float type allows you to store single precision floating point numbers, while the double keyword allows you to store double precision floating point numbers.
- Its size is typically 8 bytes of memory space.
- The range of numbers we can store are from -1.7e308 to 1.7e308.
- The general form of a double declaration is:
  - **double** variable_list;
- **Example:** **double** a = 5.5e-7; //a is equivalent to 5.5x10-7

The void type:

- The void data type has no values and no operations.
- In other words both the set of values and set of operations are empty.
- **Example:** **void** main( )
- In this declaration the main function does not return any value.

The bool type:

- The bool type has logical value true or false. The identifier true has the value 1, and the identifierfalse has the value 0.
- The general form of a bool declaration is:
  - **bool** variable_name;
- **Example:** **bool** legal_age=true;
- The statement legal_age= (age>=21); assigns the value true if age is greater than or equal to 21 orelse it returns the value false.

Type	Size in bytes	Range	Examples

int	2	-32768 to 32767	8, 100, -39
char	1	-128 to 127	‘d’, ‘6’, ‘#’
float	4	3.4 x 10^{-38} to 3.4 x 10^{38} -1	45.345, 0.134, 3.142
double	8	1.7 x 10^{-308} to 1.7 x 10^{308} -1	3.141567788888888 88

Derived data types

- These data types are constructed using simple or fundamental data types.
- This includes arrays, functions, pointers and references.

Derived data types are built from fundamental data types:

- **Arrays:** A collection of elements of the same type, stored in contiguous memory locations.

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

- **Pointers:** Variables that store the memory address of another variable.

  int *ptr = &age;

- **References:** An alias for another variable.

  int &ref = age;

- **Function Types:** Define the type of a function based on its return type and parameters.

  int sum(int, int); // Function type returning int and taking two int parameters

User defined data types

- These data types are also constructed using simple or fundamental data types.
- Some user defined data types include structure, union, class and enumerated.

Enumerated data type:

- AN enumeration is a user defined type consisting of a set of named constants called enumerators.
- enum is a keyword that assigns values 0, 1, 2…… automatically.
- This helps in providing an alternative means for creating symbolic constants.
- The syntax for enum is as follows:
  - enum [tag] { enum – list} ; //for definition for

    enumerated type enumtaddeclarator; //for declaration

    of variable type tag

- Example 1:

- enum choice { very_bad, bad, satisfactory, good, very_good};choice mychoice;
- Example 2:
  - enum MyEnumType { ALPHA, BETA, GAMMA };
  - Here, ALPHA takes the value 0, BETA takes the value of 1, GAMMA takes the value of 2.
- Example 3:
  - enum footsize { small = 5, medium = 7, large = 10};

C++ allows the creation of complex data types using fundamental and derived types:

- **Structures (struct):** A collection of variables under a single name, allowing different data types.

```
struct Person {
    std::string name;
    int age;
    double height;
};
```

- **Classes:** Similar to structures but with additional features like methods, access control, and encapsulation.

```
class Car { public:
    std::string brand;
    int modelYear;
    void display() {
        std::cout << brand << " " << modelYear << std::endl;
    }
};
```

- **Unions:** Like structures, but all members share the same memory location.

```
union Data {
    int intValue; float
    floatValue; char
    charValue;
};
```

- **Enumerations (enum):** A distinct type consisting of a set of named constants.

```
enum Color { RED, GREEN, BLUE };
```

- **Typedefs and using:** Create an alias for another data type.

typedef unsigned long ulong;
using Str = std::string;
int i = BETA;

//giv

e i value of 1 int j = 3 + GAMMA

//giv

e j a value of 5

- On the otherhand, C++ does not support an implicit conversion form int to an enum type. This typeconversion is always illegal.
  - MyEnumType x = 2; //should not be allowed by compiler MyEnumType y = 123; //should not be allowed by compiler
- Note that it does not matter whether the int matches one of the constants of the num type.

Type Conversion:

- Converting an expression of a given type into another type is known as type-casting or typeconversion.
- Type conversions are of two types, they are:
  - Implicit Conversion
  - Explicit Conversion

1. Implicit Type Conversion

Implicit type conversion, also known as automatic type conversion or type promotion, is performed by the compiler without explicit instruction from the programmer. It typically occurs when a value of one data type is assigned to another, or when operands of different types are involved in an operation.

Rules of Implicit Conversion:

- A smaller integer type can be promoted to a larger integer type (e.g., int to long).
- An integer can be promoted to a floating-point type (e.g., int to float or double).
- In mixed expressions, lower precision types are converted to higher precision types (e.g., float to double).

Example:

```
int a = 10; double
b = 5.5;
double result = a + b; // 'a' is implicitly converted to double, result = 15.5
```

In this example, the integer a is automatically converted to a double to match the type of b before the addition.

2. Explicit Type Conversion

Explicit type conversion, or type casting, is performed by the programmer to convert a variable to a different type. This is necessary when implicit conversion is not possible, or when the programmer wants to enforce a specific type conversion.

Syntax for C-Style Casting:

```
(type) expression;
```

Example:

```
double d = 9.7;
int i = (int)d; // i = 9, fractional part is discarded
```

C++ Style Casting (Preferred): C++ provides more explicit and safer casting operators:

1. static_cast: Performs a standard type conversion.
2. dynamic_cast: Used for converting pointers and references to classes in a class hierarchy (used with polymorphic types).
3. const_cast: Adds or removes the const qualifier from a variable.
4. reinterpret_cast: Reinterprets the bit pattern of an object.

Example using static_cast**:**

```
double d = 9.7;
int i = static_cast<int>(d); // i = 9
```

Common Scenarios for Type Conversion

1. **Arithmetic Operations:** In expressions involving mixed data types, smaller types are converted to larger types to prevent data loss.
2. **Function Calls:** When passing arguments to functions, if the types don't match the expected parameter types, implicit conversion may occur.
3. **Assignment:** When assigning a value of one type to a variable of another type, implicit conversion happens if it's allowed by the language rules.
4. **Type Promotion:** Converting a smaller integral type to a larger integral type, or converting a floating-point number with fewer bits to one with more bits.

Precautions with Type Conversion

1. **Data Loss:**
   - Converting from a floating-point type to an integer type may lose the fractional part.
   - Converting from a larger integer type to a smaller one may lead to truncation or overflow.
2. *Precision Issues:*

- Converting a double to a float can result in precision loss.

3. *Runtime Errors:*
    - Incorrect use of dynamic_cast can result in a nullptr if the conversion is not valid.

Example of Potential Data Loss:

int largeNumber = 3000000000; // Outside the range of int, may result in undefined behavior short smallNumber = largeNumber; // Data loss if largeNumber exceeds the range of short

Storage Classes

In C++, storage classes define the scope (visibility), lifetime, and linkage of variables or functions within a program. They determine how and where variables are stored, their default initial values, and how they are linked when declared in different translation units.
Understanding storage classes is crucial for managing resources efficiently and ensuring proper program behavior.

Types of Storage Classes

1. **auto**
2. **register**
3. **static**
4. **extern**
5. **mutable**

1. auto

The auto keyword in modern C++ is primarily used for type inference, allowing the compiler to automatically deduce the type of a variable from its initializer. However, historically in C and early C++, auto was a storage class specifier indicating automatic storage duration. By default, local variables are auto.

Example:

```
auto x = 10;   // Compiler deduces x as int
auto y = 10.5; // Compiler deduces y as double
```

In this context, auto does not specify storage duration but type deduction.

2. register

The register storage class suggests to the compiler that the variable may be frequently used, and thus should be stored in a CPU register for faster access. However, the actual use of a register is not guaranteed, as it depends on the compiler and available hardware resources.

Example:

```
register int counter = 0;
```

With modern optimizers and compilers, the register keyword is largely obsolete and rarely used.

3. static

The static storage class has different meanings depending on the context:

- **Inside a function:** A static variable retains its value between function calls. It has local scope but persists for the duration of the program.

  Example:

  ```
  void foo() {
      static int count = 0;
      count++;
      std::cout << "Count: " << count << std::endl;
  }
  ```

  In this example, count retains its value between successive calls to foo().

- **In global context or namespace:** A static variable or function has internal linkage, meaning it is accessible only within the translation unit (source file) it is defined in. This is useful for encapsulation and avoiding name conflicts.

  Example:

  ```
  static int globalCounter = 0;

  static void updateCounter() {
      globalCounter++;
  }
  ```

  Here, globalCounter and updateCounter are only accessible within the file they are declared in.

4. extern

The extern storage class is used to declare a global variable or function that is defined in another translation unit. It extends the visibility of the variable or function to other files.

Example:

```
// In file1.cpp
extern int globalCounter;

// In file2.cpp
int globalCounter = 0;

void incrementCounter() {
    globalCounter++;
}
```

In this example, globalCounter is defined in file2.cpp and declared as extern in file1.cpp to use the same variable.

5. mutable

The mutable keyword is used in the context of classes. It allows a member of an object to be modified even if the object itself is const. This is useful when you have a logically constant object but need to modify some internal states that do not affect the visible state of the object.

Example:

```
class Example {
public:
    void setData(int d) const { data = d;
    }
    int getData() const {
        return data;
      }
  private:
          mutable int data; // Can be modified even in a const object
};

int main() {
    const Example obj;
    obj.setData(5); // Allowed due to mutable std::cout
    << obj.getData(); // Outputs: 5
}
```

In this example, the data member is mutable, allowing it to be modified even when accessed through a const object.

UNIT 2

CONTROL STATEMENTS

Introduction

- Control statements are statements that alter the sequence of flow of instructions.
- Any single input statement, assignment and output statement is simple statement.
- A group of statement that are separated by semicolon and enclosed within curled braces { and } is called a block or compound statement.
- The order in which statements are executed in a program is called flow of control.

Types of control statements:

- *C++ supports two basic control statements.*
  - Selection statements
  - Iteration statements

Selection Statements:

- This statement allows us to select a statement or set of statements for execution based on some condition.
- It is also known as conditional statement.
- This structure helps the programmer to take appropriate decision.
- The different selection statements,viz.
  - ifstatement
  - if– elsestatement
  - Nested–ifstatement
  - switch statement

if statement:

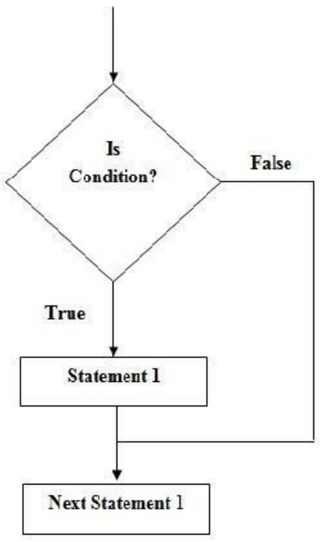
- This is the simplest form of if statement.
- This statement is also called as one-way branching.
- This statement is used to decide whether a statement or set of statements should be executed or not.
- The decision is based on acondition which can be evaluated toTRUE or FALSE.
- The general form of simple–if statement is:

```
if(TestCondition) //This Conditionis true Statement1;
Statement2;
```

- Here, the test condition is tested which results in either a TRUE or FALSE value. If the result of the test condition is TRUE then the Statement 1 is executed. Otherwise, Statement 2 is executed.

  Ex:if(amount >=5000)

  discount=amount*(10/100); net-amount = amount – discount;



EX 1 : Write a C++ program to find the largest, smallest and second largest of three numbers using simple if statement.

```
#include<iostream.h>
#include<conio.h>voi d
main( )
{
        Int a, b, c;
        int largest,smallest,seclargest;
        clrscr( );
        cout<<"Enter the three numbers"<<endl;
        cin>>a>>b>>c;
        largest = a;                                    //Assumefirstnumberaslargest
        if(b>largest)
                largest=b;
        if(c>largest)
                largest=c;

        smallest = a;                                   //Assumefirstnumberassmallest
        if(b<smallest)
                smallest=b;
        if(c<smallest)
                smallest=c;

        seclargest = (a + b + c) – (largest + smallest);
        cout<<"Smallest Number is = "<<smallest<<endl;
        cout<<"Second LargestNumberis="<<seclargest<<endl;
        cout<<"Largest Number is = "<< largest<<endl;  getch();
}
```

EX 2 : WriteaC++program to input the total amount in abill, iftheamount is greater than 1000, a discount of 8% is given. Otherwise, no discount is given. Output the total amount, discount and the final amount. Use simple if statement.

```
#include<iostream.h>
#include<conio.h> void
main( )
{
        Float TAmount, discount, FAmount; clrscr(
        );
        cout<<"EntertheTotal Amount"<<endl;
        cin>>TAmount;
        discount=0;                                         //CalculateDiscount
        if(TAmount>1000)
                Discount=(8/100)*TAmount;
        FAmount=TAmount –Discount                           //CalculateFinalAmount
        cout<<"ToatalAmount="<<TAmount<<endl;
        cout<<"Discount = "<<discount<<endl; cout<<"Final
        Amount = "<< FAmount<<endl; getch( );
}
```

- **if–else statement:**
    - This structure helps to decide whether a set of statements should be executed or another set of statements should be executed.
    - This statement is also called as two-way branching.
    - The general form of if–elsestatementis:

      ```
       if (Test Condition)
      Statement1;
      else
      Statement2;
      ```

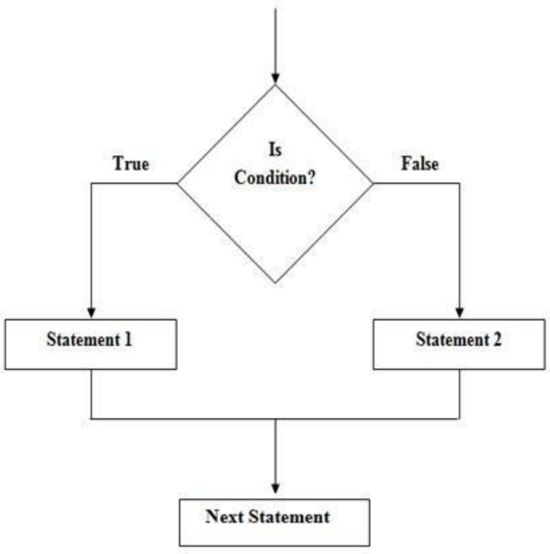
    - Here, the test condition is tested.If the test- condition is TRUE, statement-1 is executed. Otherwise Statement 2 is executed.

      Ex:if(n %2 ==0)

      ```
      cout<<"NumberisEven"; else
      cout<<"NumberisOdd";
      ```

EX 1 : Write a C++ program to check whether a given year is a leap year not,Using if- else statement.

```
#include<iostream.h>
#include<conio.h>voi d
main( )
{
        Int year;
        clrscr( );
        cout<<"Enter the Year in the form YYYY"<<endl;
        cin>>year;
        if(year%4==0&&year%100!=0||year%400==0)
                cout<<year<<"isaleapyear"<<endl;
        else
                cout<<year<<"isnotleapyear"<<endl;
        getch();
}
```



EX 2 : Write a C++ program to accept a character. Determine whether the character is a lower-case or upper-case letter.

```
#include<iostream.h>
#include<conio.h> void
main( )
{
        Char ch;
        clrscr();
        cout<<"Enterthe Character"<<endl; cin>>ch;
        if(ch>='A' &&ch<='Z')
                cout<<ch<<"isanUpper-Case Character"<<endl;
        else
        if(ch>='a'&&ch <='z')
                cout<<ch<<"isanLower-CaseCharacter"<<endl;
        else
                cout<<ch<<"isnotan alphabet"<<endl;
        getch();
}
```

- **Nested if statement:**
    - If the statement of an if statement is another if statement the n such an if statement is called as Nested-if Statement.
    - Nested-ifstatement containsanifstatementwithinanotherif statement.
    - There are two forms of nested if statements.
- **if–else-if statement:**
    - This structure helps the programmer to decide the execution of a statement from multiple statements based on a condition.
    - There will be more than one condition to test.
    - This statement is also called as multiple-way branch.
    - The general form of if–else–if statement is:

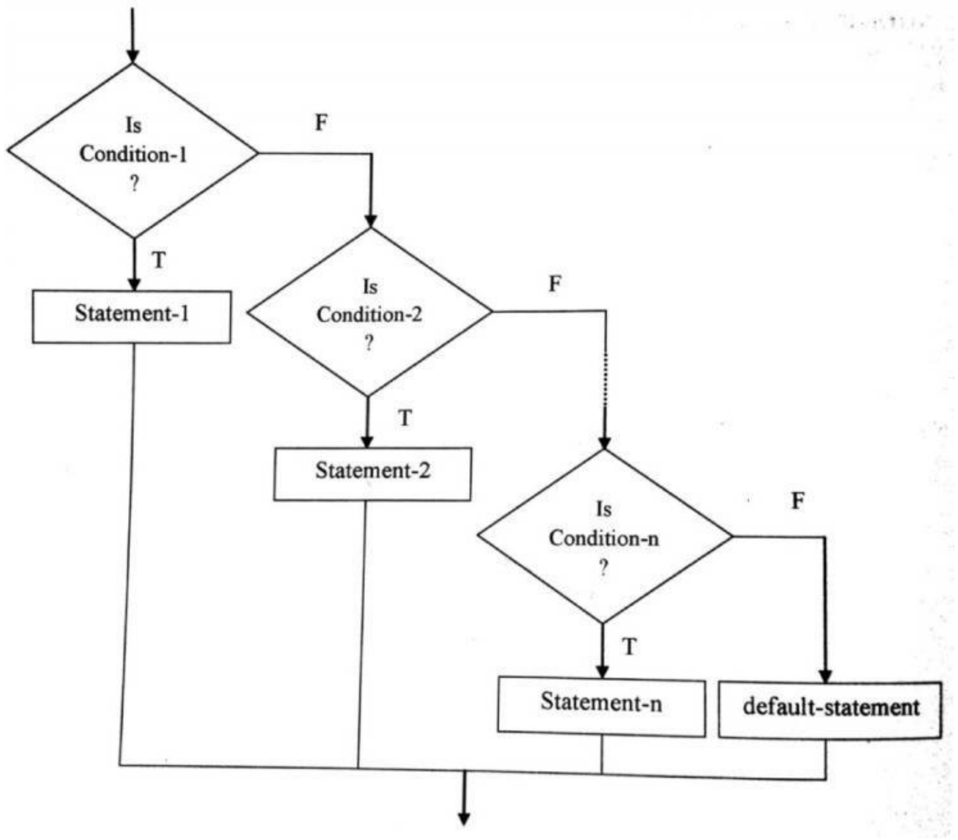
```
if(TestCondition1)
Statement1;
else
if(TestCondition2) Statement2;
elseif(testCondition N)
StatementN;
else
Default Statement;
```

Example:

```
if(marks>=85 )
            PRINT“Distinction”
      else
            if(marks>=60 )
                  PRINT“FirstClass”
            else
                  if(marks>=50 )
                        PRINT“SecondClass”
                        else if(marks>=35 )
                      PRINT“Pass”
                      else
                      PRINT“Fail;
```

- Here, Condition 1 is tested. If it is TRUE, Statement 1 is executed control transferred out of the structure. Otherwise, Condition 2 is tested. If it is TRUE, Statement 2 is executed control is transferred out of the structure and so on.

- If none of the condition is satisfied, a statement called default statement is executed.



EX 1 : Write a C++ program to input the number of units of electricity consumed in a house and calculate the final amount using nested-if statement. Use the following data for calculation.

Units consumed	**Cost**
<30	Rs.3.50 / Unit
>=30 and<50	Rs.4.25 / Unit
>=50 and<100	Rs.5.25 / Unit
>=100	Rs.5.85 / Unit

```
#include<iostream.h>
#include<conio.h>voi d
main( )
{
```

```
int units;
```

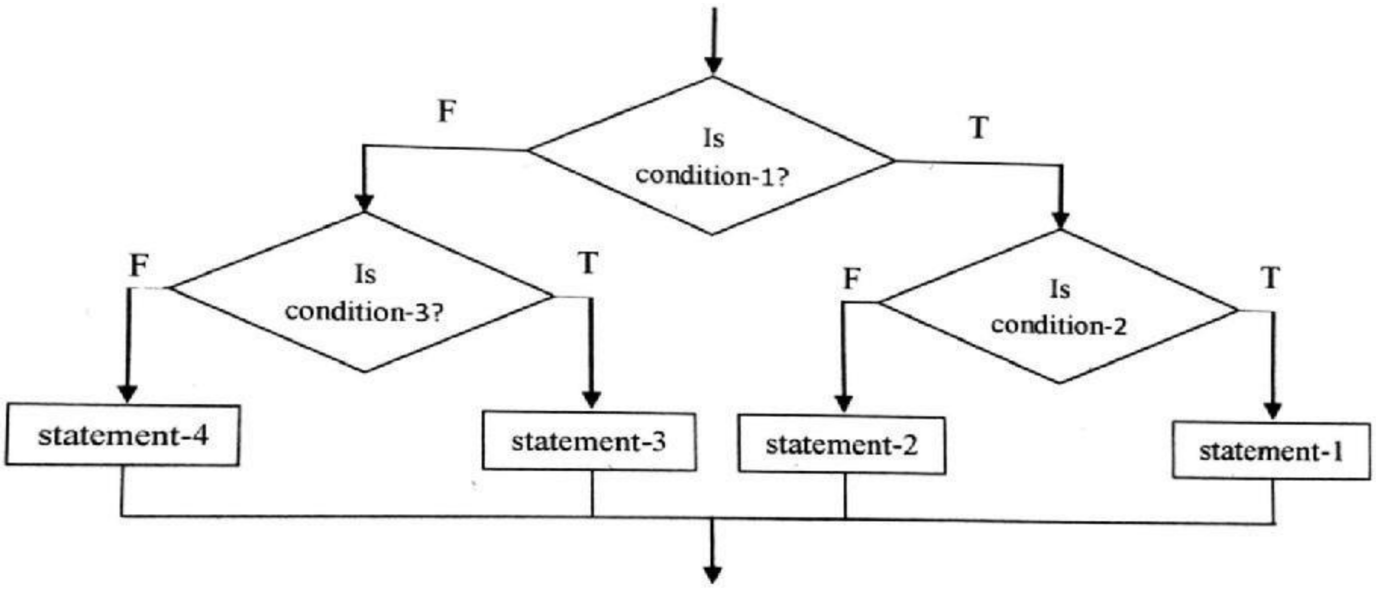
```
        floatBillamount;
        clrscr( );
        cout<<"Enterthenumberofunitsconsumed"<<endl;
        cin>>units;
        if(units<30)
                Billamount=units *3.50 ;
        else
        if(units<50)
                Billamount=29* 3.50 + (units– 29)* 4.25 ;
        else
        if(units<100)
                Billamount=29 *3.50 +20 *4.25 +(units–49)* 5.25 ;
        else
                Billamount=29 * 3.50 + 20 * 4.25 +50* 5.25+(units– 99)* 5.85 ;

        cout<<"TotalUnitsConsumed="<<units<<endl;
        cout<<"Toatl   Amount   =   "<<Billamount<<endl;
        getch( );
}
```

- The general form of if–else-ifstatement is:

```
        if(TestCondition1)
                if(TestCondition2)
                        Statement1;
                else
                        Statement2;
        else
```

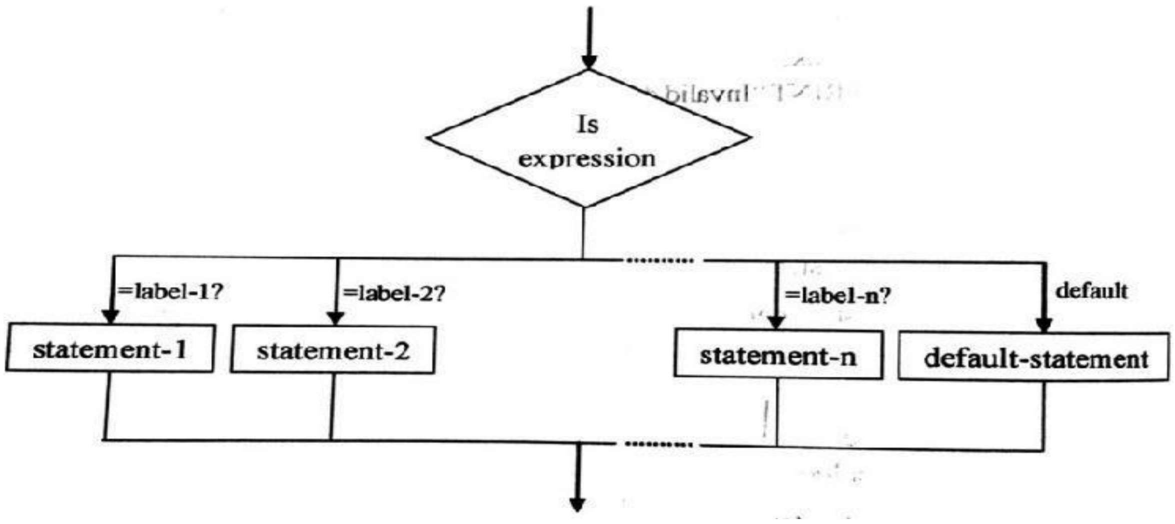
Ex: Tofindthegreatestofthreenumbersa,bandc. if (a>b)

```
        if(a>c)
                OUTPUT a
        else
                OUTPUT c
else
        if(b >c)
                OUTPUT b
        else
                OUTPUT c
```



- **SwitchStatement:**
    - C++ has built in multiple-branch selection statement i.e.switch.
    - If there are more than two alternatives to be selected, multiple selections construct isused.
- The general formof Switch statement is: Switch (Expression)

```
{
        Case Label-1:           Statement1;
                                Break;
        Case Label-2:           Statement1;
                                Break;
                                …………..
        Case Label-N:           StatementN;
                                Break;
        Default         :       Default-Statement;
}
```



- **Ex**:To find the name of the day given the day number

```
switch ( dayno )
{
        Case1:          cout<<"Sunday"<<endl;
                        break;
        Case2:          cout<<"Monday"<<endl;
                        break;
        Case3:          cout<<"Tuesday"<<endl;
                        break;
        Case4:          cout<<"Wednesday"<<endl;
                        break;
        Case5:          cout<<"Thursday"<<endl;
                        break;
        Case6:          cout<<"Friday"<<endl;
                        break;
        Case7:          cout<<"Saturday"<<endl;
                        break;
        default:        cout<<"InvalidDayNumber"<<endl;
}
```

- The switch statement is a bit peculiar with in the C++language because it uses labels instead of blocks.
- This force up to put break statements after the group of statements that we want to execute for

aspecific condition.

- Other wise the remainder statements including those corresponding too there labels also are executed until the end of the switch selective block or a break statement is reached.

Ex : Write a C++ program to input the marks of four subjects. Calculate the total percentage and output the result as either “First Class” or “Second Class” or “Pass Class” or “Fail” using switch statement.

Class	**Range(%)**
FirstClass	Between60%to100%
SecondClass	Between50%to59%
PassClass	Between40%to49%
Fail	Lessthan 40%

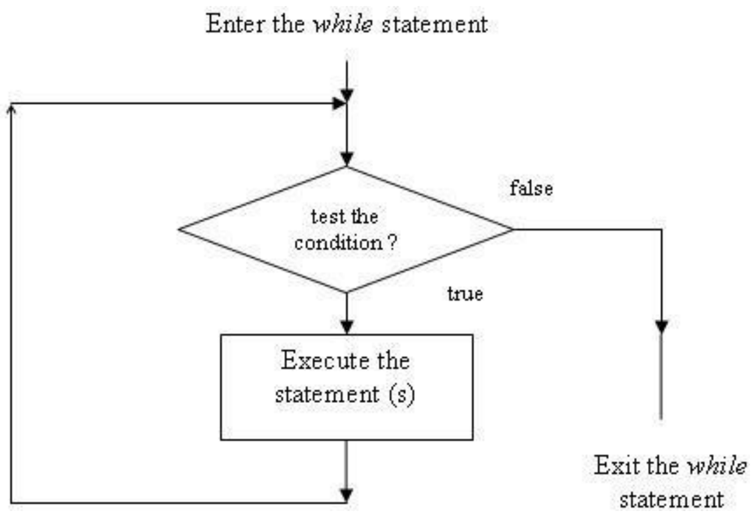
```
#include<iostream.h>
#include<conio.h>
Void main()
{
        Int m1,m2,m3,m4,total,choice; float per;
        clrscr( );
        cout<<”Enter the First subject marks”<<endl; cin>>m1;
        cout<<”Enter the Second subject marks”<<endl;
        cin>>m2;
        cout<<”Enter the Third subject marks”<<endl; cin>>m3;
        cout<<”Enter the Fourth subject marks”<<endl;
        cin>>m4;

        total=m1+m2+m3+m4; per
        = (total / 400) * 100; cout<<”Percentage=
        “<<per<<endl;

        choice=(int)per/10;
        cout<<”The result of the student is: “<<endl;
                switch(choice)
        {
        case 10:
       case 9:
       case 8:
       case 7:  case6:cout<<"FirstClass"<<endl;
       break; case5:
       cout<<"SecondClass"<<endl; break;
       case4:cout<<"Pass Class"<<endl;
       break; default:cout<<"Fail"<<end;


    }
 getch();
}
```

- **Iterative Constructsor Looping**
    - Iterative statements are the statements that are used to repeatedlyexecute a sequence of statements until some condition is satisfied or a given number of times.
    - The process of repeated execution of a sequence of statements until some condition is satisfied is called as iteration or repetition or loop.
    - Iterative statements are also called as repetitive statement or looping statements.
    - There are three types of looping structures in C++.
        - While loop
        - Do while loop
        - For loop
- **while loop:**
    - This is a **pre-tested loop** structure.
    - This structure checks the condition at the beginning of the structure.
    - The set of statements are executed again and again until the condition is true.
    - When the condition becomes false, control is transferred out of the structure.
    - The general form of while structure is while (Test Condition)

```
{
    Statement1
```

Statement2

```
        ……..
        StatementN
    }
    End of While
```


Enter the *while* statement
test the condition ?
false
true
Execute the statement (s)
Exit the *while* statement

- Example:

```
        n =10;
        While(n >0)
        {
            cout<<n<<"\t";
            --n;
        }
        cout<<"End of while loop\n";
        Output:10 9 8 76 5 4 3 21 End of while loop
```

EX 1 : Write a C++ program to find sum of all the digits of a number using while statement. #include<iostream.h>

```
#include<conio.h>voi d
main( )
{
intnum,sum,rem; clrscr( );
cout<<"EntertheNumber"<<endl; cin>>num;
sum =0;
    while(num!=0)
    {
        rem = num % 10;
        sum=sum+rem; num
        = num/10;
    }
    cout<<"Sumofthedigitsis"<<sum<<endl;
    getch();
}
```

EX 2 : Write a C++ program to input principal amount, rate of interest and time period. Calculate compound interest using while statement.

(Hint:Amount=P*(1+R/100)^T,CompoundInterest=Amount–P) #include<iostream.h>

```
#include<conio.h>voi d
main( )
{
        Float pri,amt,priamt,rate,ci; int time,
        year;
        clrscr( ); cout<<”EnterthePrincipalamount,rateofinterestandtime”<<endl;
        cin>>pri>>rate>>time;

        year = 1;
        priamt=pri;

        while(year<=time)
        {
                amt=pri*(1+rate/100); year
                ++;
        }
        ci=amt – priamt;
        cout<<”CompoundInterestis“<<ci<<endl; getch(
        );
}
```

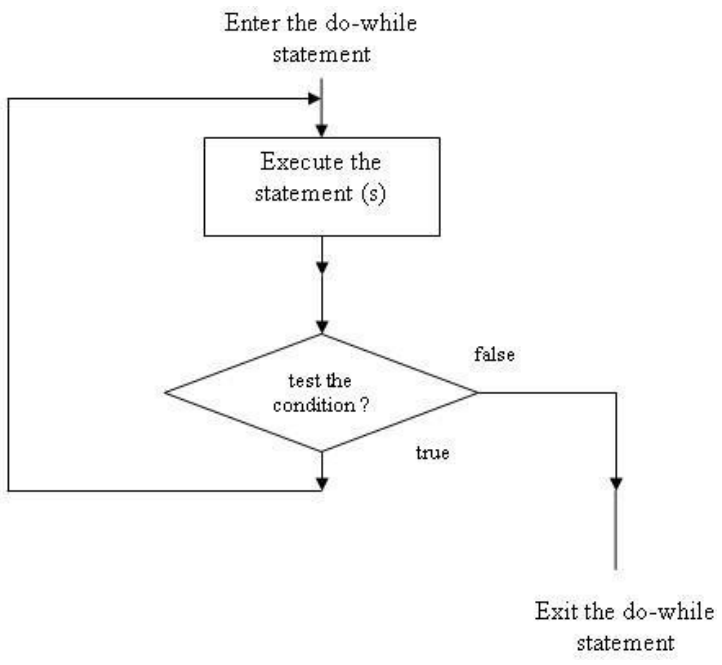
EX 3: Write a C++ program to check whether the given number is power of 2.

```
#include<iostream.h>
#include<conio.h> void
main( )
{
        intnum,m,flag; clrscr(
        );
        cout<<”Enterthe Number”<<endl;
        cin>>num;
        m = num;
```

```
    flag = 1;
    while(num>2)
            if(num%2 ==1)
            {if(flag)
        }
        else
        flag=0;   break;
        num    =num/2;
        else
        cout<<m<<"ispowerof 2 "<<endl;
        cout<<m<<"is notpowerof2 "<<endl; getch();
  }
```

- **Do while statements:**
    - This is a post-tested loop structure.
    - This structure checks the condition attheend ofthestructure.
    - The set of statements are executed again and again until the condition is true.
    - When the condition becomes false, control is transferred out ofthestructure.
    - The general form of while structure is do

```
{
        Statement1
        Statement2
        ……..
        StatementN
}while(Test Condition);
```


Enter the do-while statement
Execute the statement (s)
test the condition ?
false
true
Exit the do-while statement

- Example:

```
        i =2;
        do
        {
              cout<<i<<"\t";
                i =i +2;
                } while ( i<=25);
```

EX 1 Write a C++ program to check whether the given number is an Armstrong Number using do-while statement. **(Hint: 153 = $1^3 + 5^3 + 3^3$)**

```
#include<iostream.h>
#include<conio.h>
void main( )
{  Int num,rem,sum,temp;
        cout<<"Enterthethreedigitnumber"<<endl; cin>>num;
        temp=num;
        sum = 0;  do
        {rem=temp%10;
        sum=sum+rem*rem*rem; temp = temp / 10;
        }while(temp!=0);
        if(sum == num)
                cout<<num<<"isan ArmstrongNumber "<<endl;
        else
                cout<<num<<"isnotan ArmstrongNumber "<<endl;
}
```

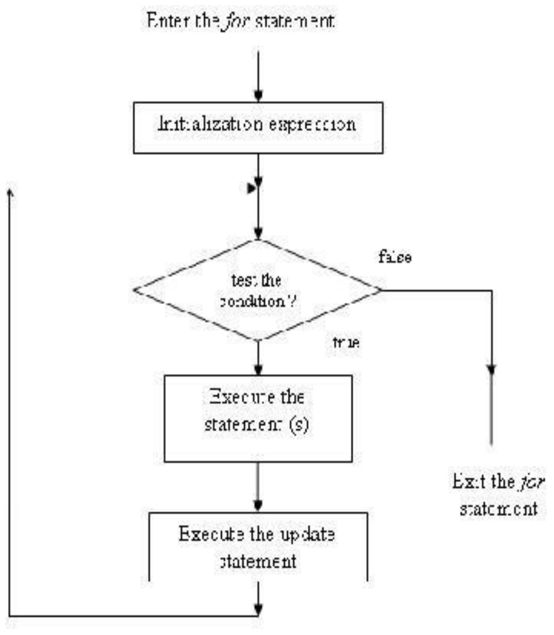
Difference between while and do while loop:

while	**do while**
This is pre-tested looping structure	This is post tested looping structure
It tests the given condition at in itial point Of looping structure	It tests the given condition at the last of Looping structure.
Minimum execution of loop is zero	Minimum execution of loop is once**.**
Syntax: **while(Testcondition)** **{** statement1; statement2; …………….; statementn; **}**	**Syntax:** **do** **{** statement1; statement2; statementn; **}** **while(Testcondition);**
Semi colon is not used.	Semi colon is used.

- *for statement:*
  - This structure is the **fixed execution structure**.
  - This structure is usually used when we know in advance exactly how many times as set of statements is to be repeatedly executed again and again.
  - This structure can be used as increment looping or decrement looping structure.
  - The general form of for structure is as follows:

```
for(Expression 1; Expression 2; Expression 3)
{
        Statement1;
        Statement   2;
        StatementN;
}
```

Where, Expression 1 represents Initialization Expression
2 represents Condition Expression 3 represents Increment/Decrement


Enter the for statement
Initialization expression
test the condition?
false
true
Execute the statement (s)
Execute the update statement
Exit the for statement

- Example:

```
sum =0; for(i=1;i<=10;i++)
        sum =sum +i;
```

- This looping structure works as follows:
  - Initialization is executed. Generally it is an initial value setting for a counter variable and is executed only one time.
  - Condition is checked.if it is TRUE the loop continues,other wise the loop ends and control exists from structure.
  - Statement is executed as usual, is can either a single statement or a block enclosed in curled braces { and }.
  - Atlast, what ever is specified in the increase field is executed and the loop gets back to executed step 2.
  - The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon sign between them must be written compulsorily

- Optionally, using the comma operator we can specify more than one expression in any of the fields included in a for loop.

Ex 1 Write a C++ program to find the factorial of a number using for statement.

(Hint:5!= 5 * 4 *3 * 2 * 1 = 120)

```
#include<iostream.h>
#include<conio.h>
void main( )
{
        Int num,fact,i;
        clrscr( );
        cout<<"Enterthe number"<<endl; cin>>num;
        fact=1;
        for(i =1; i<=num; i++)
                fact=fact*i;
        cout<<"Thefactorialof a"<<num<<"!is: "<<fact<<endl; getch();
}
```

EX 2 : Write a C++ program to generate the Fibonacci sequence up to a limit usingfor statement.

(Hint: 5 = 0 1 1 2 3)

```
#include<iostream.h>
#include<conio.h>vo id
main( )
{
        Int num,first,second,count,third;
        clrscr( );
        cout<<"Enter the number limit for Fibonacci Series"<<endl; cin>>num;
        first = 0;
        second = 1;
        cout<<first<<"\t"<<second;
        third = first + second;
        for(count=2;third<=num; count++)
        {
                cout<<"\t"<<third
                first = second; second
                = third;
                third=first+second;
        }
        cout<<"\nTotal terms= "<<count<<endl;
        getch();
}
```

- **Jump statements or Transfer of control from within loop**

Transfer of control within looping are used to

- Terminate the execution of loop.
- Exiting a loop.
- Half way through to skip the loop.

All these can be done by using break, exit and continue statements.

break statement

The break statement has two uses

- You can use It to terminate a case in the switch statement.
- You can also use it to force immediate termination of a loop like while, do-while and for, by passing the normal loop conditional test.

When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement.

The general form of break statement is:

break;

Example:

```
    for(n=0;n<100; n++)
    {
            cout<<n; if(n==10)break;
    }
```

Program: To test whether a given number is prime or not using break statement.

```
#include<iostream.h>
#include,conio.h>voi d
main( )
{
        Int n,i,status;
        clrscr( );
        cout<<"Enter the number";
        cin>>n;
        status=1;
        for(i=2;i<=n/2;i++)
        {
                if(n%i ==0)
                {
                        status=0
                        cout<<"It is not a prime"<<endl; break;
                }
        }
        if(status)
        cout<<"It is a prime number"<<endl; getch();
}
```

- **exit() function:**
  - This function causes immediate termination of the entire program, forcing a return to the operating system.
  - In effect, exit( ) function acts as if it were breaking out of the entire program.
  - The general form of the exit() functionis:

    ```
    exit( ); or     void exit(intreturn_code);
    ```

    - The return code is used by the operating system and may be used by calling programs.
    - An exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened.

Program:To test whethera givennumber is prime or not using exit() statement.

```
#include<iostream.h>
#include<conio.h>
void main( )
{
        int n, i;
        clrscr();
        cout<<"Enter the number";
        cin>>n;
```

```
        for(i=2;i<=n/2;i++)
        {
                if(n%i ==0)
                {
                        cout<<"It is not a prime"<<endl; exit(0);
                }
        }
        cout<<"Itis aprime number"<<endl; getch();
}
```

- **Continue statement:**
  - The continue statement causes the program to skip the rest of the loop in the current iteration as if end of the statement block has been reached, causing it to jump to start of the following iteration.
  - The continue statement works some what like break statement.
  - Instead of forcingtermination, however continue forces the next iteration of the loop to take place, skipping any code in between.
  - Thegeneralformofthecontinuestatement is:

    continue;

- Example:

```
        for(n=10;n<0;n--)
        {
                if(n==10)continue;
                cout<<n;
        }
```

- **Goto statement:**
  - The goto allows to makes an absolute jump to another point in the program.
  - This statement execution causes an unconditional jump or transfer of control from one statement to the other statement with in a program ignoring any type of nesting limitations.
  - The destination is identified by a label, which is then used as an argument for the goto statement.
  - A label is made of a valid identifier followed by a colon (:).
  - The general form of goto statement is: statement1;

    statement2;

g
o
t
o
l
a
b
e
l
_
n
a
m
e
;
s
t
a
t
e
m
e
n
t
3
;
s
t
a
t
e
m
e
n
t

4
;
label_name:statement5;
statement6;

Program:To print
from 10 to 1 using go
to statements.
#include<iostream.h>
#
i
n
c
l
u
d
e
<
c
o
n
i
o
.
h
>
#
i
n
c
l
u
d
e
<
i

```
o
m
a
n
i
p
.
h
>
v
o
i
d
m
a
i
n
(
)
{
    int n=10;
    l
    o
    o
    p
    :
    c
    o
    u
    t
    <
    <
    ”
    \
    t
”
<
<
n
;
n
-
-
;
i
f
(
n
>
0
)
g
o
t
o
l
o
o
p
;
c
o
u
t
<
<
”
E
n
d
o
```

```
        f
        l
        o
        o
        p
        ”
        ;
        g
        e
        t
        c
        h
        (
        )
        ;
    }
```

UNIT 2

MODULAR PROGRAMING

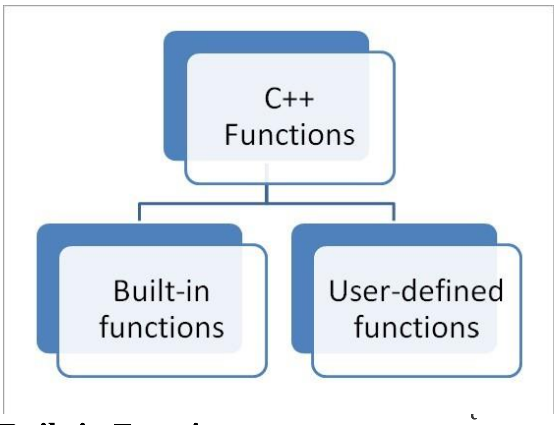
Functions and Its Types :

A function is a set of statements that are put together to perform a specific task. It can be statements performing some repeated tasks or statements performing some specialty tasks like printing etc.

One use of having functions is to simplify the code by breaking it into smaller units called functions. Yet another idea behind using functions is that it saves us from writing the same code again and again. We just have to write one function and then call it as and when necessary without having to write the same set of statements again and again.

Types Of Functions In C++

In C++, we have two types of functions as shown below.



Built-in Functions

Built-in functions are also called libr ry functions. These are the functions that are provided by C++ and we need not write them ourselves. We can directly use these functions in our code.These functions are placed in the header files of C++.

For Example, <cmath>, <string> are the headers that have in-built math functions and string functions respectively.

Example of using built-in functions in a program.

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

int main()
  {
string name;
cout << "Enter the input string:"; getline
(std::cin, name);
cout << "String entered: " << name << "!\n"; int
size = name.size();
cout<<"Size of string : "<<size<<endl;
     }
```

Output:

```
Enter the input string: JSS COLLEGE!
String entered: JSS COLLEGE!
Size of string: 12
```

Here we are using the headers <iostream> and <string>. The data types and other input/output functions are defined in <iostream> library. String functions used like getline, size are a part of the <string> header.

User-Defined Functions

C++ also allows its users to define their own functions. These are the user-defined functions. We can define the functions anywhere in the program and then call these functions from any part of the code. Just like variables, it should be declared before using, functions also need to be declared before they are called.

The general syntax for user-defined functions (or simply functions) is as given below:

```
return_type functionName(param1,param2,….param3)

   {

   Function body;

   }
```

So as shown above, each function has:

- **Return type:** It is the value that the functions return to the calling function after performing a specific task.
- **functionName** : Identifier used to name a function.
- **Parameter List:** Denoted by param1, param2,…paramn in the above syntax. These are the arguments that are passed to the function when a function call is made. The parameter list is optional i.e. we can have functions that have no parameters.
- **Function body:** A group of statements that carry out a specific task.

As already mentioned, we need to 'declare' a function before using it.

Function Declaration

A function declaration tells the compiler about the return type of function, the number of parameters used by the function and its data types. Including the names of the parameters in the function, the declaration is optional. The function declaration is also called as a function prototype.

We have given some examples of the function declaration below for your reference.

int sum(int, int);

Above declaration is of a function 'sum' that takes two integers as parameters and returns an integer value. void

swap(int, int);

This means that the swap function takes two parameters of type int and does not return any value and hence the return type is void.

void display();

The function display does not take any parameters and also does not return any type.

Function Definition

A function definition contains everything that a function declaration contains and additionally it also contains the body of the function enclosed in braces ({}).

In addition, it should also have named parameters. When the function is called, control of the program passes to the function definition so that the function code can be executed. When execution of the function is finished, the control passes back to the point where the function was called.

For the above declaration of swap function, the definition is as given below:

```
void swap(int a, int b){
            b = a + b;
            a = b - a;
            b = b - a;
  }
```

Note that declaration and definition of a function can go together. If we define a function before referencing it then there is no need for a separate declaration.

Let us take a complete programming Example to demonstrate a function.

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

  void swap(int a, int b)  {              //here a and b are formal parameters b
    = a + b;
    a = b - a;
    b = b - a;
cout<<"\nAfter swapping: ";
cout<<"a = "<<a;  cout<<"\tb =
"<<b;
return;
  }
int main()
  {
int a,b;
cout<<"Enter the two numbers to be swapped: "; cin>>a>>b;

cout<<"a = "<<a;
cout<<"\tb = "<<b;
swap(a,b);            //here a and b are actual parameters
  }
```

```
Enter the two numbers to be swapped: 5 3 a = 5
b = 3
After swapping: a = 3 b = 5
```

In the above example, we see that there is a function swap that takes two parameters of type int and returns nothing. Its return type is void. As we have defined this function before function main, which is a calling function, we have not declared it separately.

In the function main, we read two integers and then call the swap function by passing these two integers to it. In the swap function, the two integers are exchanged using a standard logic and the swapped values are printed.

Calling A Function

When we have a function in our program, then depending on the requirement we need to call or invoke this function. Only when the function is called or invoked, the function will execute its set of statements to provide the desired results.

The function can be called from anywhere in the program. It can be called from the main function or from any other function if the program is using more than one function. The function that calls another function is called the "Calling function".

In the above example of swapping numbers, the swap function is called in the main function. Hence the main function becomes the calling function.

Formal And Actual Parameters

We have already seen that we can have parameters for the functions. The function parameters are provided in the function definition as a parameter list that follows the function name. When the function is called we have to pass the actual values of these parameters so that using these actual values the function can carry out its task.

The parameters that are defined in the function definition are called **Formal Parameters**. The parameters in the function call which are the actual values are called **Actual Parameters.**
In the above example of swapping numbers, we have written the comments for formal and actual parameters. In the calling function i.e. main, the value of two integers is read and passed to the swap function. These are the actual parameters.

We can see the definitions of these parameters in the first line of the function definition. These are the formal parameters.

Note that the type of formal and actual arguments should match. The order of formal and actual parameters should also match.

Return Values

Once the function performs its intended task, it should return the result to the calling function. For this, we need the return type of the function. The function can return a single value to the calling function. The return type of the function is declared along with the function prototype.

Example of adding two numbers to demonstrate the return types.

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

int sum(int a, int b){
return (a+b);
  }
int main()
  {
```

```
int a, b, result;
cout<<"Enter the two numbers to be added: "; cin>>a>>b; result =

sum(a,b);

cout<<"\nSum of the two numbers : "<<result;
  }
```

Output:

```
Enter the two numbers to be added: 11 11 Sum of


the two numbers: 22
```

In the above example, we have a function sum that takes two integer parameters and returns an integer type. In the main function, we read two integers from the console input and pass it to the sum function. As the return type is an integer, we have a result variable on the LHS and RHS is a function call.

When a function is executed, the expression (a+b) returned by the function sum is assigned to the result variable. This shows how the return value of the function is used.

Void Functions

We have seen that the general syntax of function requires a return type to be defined. But if in case we have such a function that does not return any value, in that case, what do we specify as the return type? The answer is that we make use of valueless type "void" to indicate that the function does not return a value.

In such a case the function is called "void function" and its prototype will be like void

functionName(param1,param2,….param 3);

Note: It is considered as a good practice to include a statement "return;" at the end of the void function for clarity.

Default Arguments in C++

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn't provide a value for the argument. In case any value is passed, the default value is overridden.

1) The following is a simple C++ example to demonstrate the use of default arguments. Here, we don't have to write 3 sum functions; only one function works by using the default values for 3rd and 4th arguments.

```
// CPP Program to demonstrate Default Arguments #include
<iostream>
using namespace std;

// A function with default arguments,
// it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z = 0, int w = 0) //assigning default values to z,w as 0
  {
     return (x + y + z + w);
  }

// Driver Code int
main()
  {
     // Statement 1
     cout << sum(10, 15) << endl;

     // Statement 2
     cout << sum(10, 15, 25) << endl;

     // Statement 3
     cout << sum(10, 15, 25, 30) << endl;
     return 0;
  }
```
OUTPUT : 25 50 80

- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.
- Default arguments are overwritten when the calling function provides values for them. For example, calling the function sum(10, 15, 25, 30) overwrites the values of z and w to 25 and 30 respectively.
- When a function is called, the arguments are copied from the calling function to the called function in the order left to right. Therefore, sum(10, 15, 25) will assign 10, 15, and 25 to x, y, and z respectively, which means that only the default value of w is used.
- Once a default value is used for an argument in the function definition, all subsequent arguments to it must have a default value as well. It can also be stated that the default arguments are assigned from right to left. For example, the following function definition is invalid as the subsequent argument of the default variable z is not default.
  - // Invalid because z has default value, but w after it doesn't have a default value
  - int sum(int x, int y, int z = 0, int w).

Advantages of Default Arguments:

- Default arguments are useful when we want to increase the capabilities of an existing function as we can do it just by adding another default argument to the function.
- It helps in reducing the size of a program.
- It provides a simple and effective programming approach.
- Default arguments improve the consistency of a program.

Disadvantages of Default Arguments:

- It increases the execution time as the compiler needs to replace the omitted arguments by their default values in the function call.

C++ Recursion :

Recursion in C++ is a technique in which a function calls itself repeatedly until a given condition is satisfied. In other words, recursion is the process of solving a problem by breaking it down into smaller, simpler sub-problems.

Syntax Structure of Recursion

```
return_type recursive_func {
   ....
     // Base Condition
     // Recursive Case
```

```
    ....
}
```

Recursive Function

A function that calls itself is called a **recursive function**. When a recursive function is called, it executes a set of instructions and then calls itself to execute the same set of instructions with a smaller input. This process continues until a base case is reached, which is a condition that stops the recursion and returns a value.

Base Condition

The base condition is the condition that is used to terminate the recursion. The recursive function will keep calling itself till the base condition is satisfied.

Recursive Case

Recursive case is the way in which the recursive call is present in the function. Recursive case can contain multiple recursive calls, or different parameters such that at the end, the base condition is satisfied and the recursion is terminated.

Example of C++ Recursion

The following C++ program illustrates how to perform recursion.

```
// C++ Program to calculate the sum of first N natural
// numbers using recursion
#include <iostream>
using namespace std;

int nSum(int n)
{
    // base condition to terminate the recursion when N = 0
    if (n == 0) {
        return 0;
    }

    // recursive case / recursive call
    int res = n + nSum(n - 1);

    return res;
}

int main()
{
    int n = 5;

    // calling the function
    int sum = nSum(n);

    cout << "Sum = " << sum;
    return 0;
}
```

OUTPUT

```
Sum = 15
```

In the above example,

- **Recursive Function:** nSum() is the Recursive Function
- **Recursive Case:** The expression, **int res = n + nSum(n – 1)** is the Recursive Case.
- **Base Condition:** The base condition is **if (n == 0) { return 0;}**

Working of Recursion in C++

To understand how C recursion works, we will again refer to the example above and trace the flow of the program.

1. In the nSum() function, **Recursive Case** is

```
int res = n + nSum(n - 1);
```

2. In the example, n = 5, so as **nSum(5)'s** recursive case, we get

```
int res = 5 + nSum(4);
```

3. In **nSum(4)**, the recursion case and everything else will be the same, but n = 4. Let's evaluate the recursive case for n = 4,

```
int res = 4 + nSum(3);
```

4. Similarly, for **nSum(3), nSum(2) and nSum(1)**

```
int res = 3 + nSum(2);   // nSum(3) int
res = 2 + nSum(1);  // nSum(2) int res =
1 + nSum(0);    // nSum(1)
```

Let's not evaluate nSum(0) and furt er fo𝗋 now.

5. Now recall that the **return value** of the nSum() function in this same integer named **res**. So, instead of the function, we can put the value returned by these functions. As such, for nSum(5), we get

```
int res = 5 + 4 + nSum(3);
```

6. Similarly, putting return values of nSum() for every n, we get

```
int res = 5 + 4 + 3 + 2 + 1 + nSum(0);
```

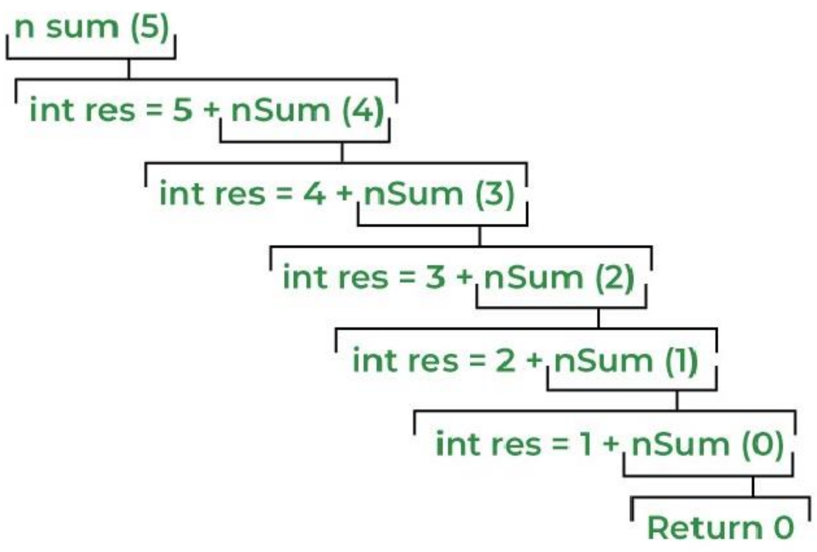
7. In nSum() function, the **base condition** is

```
if (n == 0) { return
    0;
}
```

which means that when nSum(0) will return 0. Putting this value in nSum(5)'s recursive case, we get

```
int res = 5 + 4 + 3 + 2 + 1 + 0;
        = 15
```

8. At this point, we can see that there are no function calls left in the recursive case. So the recursion will stop here and the final value returned by the function will be **15** which is the su   of the first 5 natural numbers.



Memory Management in C++ Recursion

Like all other functions, the recursive function’s data is stored in the stack memory in the form of a stack frame. This stack frame is deleted once the function returns some value. In recursion,

- The function call is made before returning the value, so the stack frame for the progressive recursive calls is stored on top of existing stack frames in the stack memory.
- When the topmost function copy returns some value, its stack frame is destroyed and the control comes to the function just before that particular copy after the point where the recursive call was made for the top copy.
- The compiler maintains an instruction pointer to track where to return after the function execution.

What is Stack Overflow?

Stack overflow is one of the most common errors associated with the recursion which occurs when a function calls itself too many times. As we know that each recursive call requires separate space in the limited stack memory. When there is a large number of recursive calls or recursion goes on infinite times, this stack memory may get exhausted and may not be able to store more data leading to programs’ termination.

Applications of Recursion

Recursion has many applications in computer science and programming. Here are some of the most common applications of recursion:

- **Solving:** Fibonacci sequences, Factorial Function, Reversing an array, Tower of Hanoi.
- **Backtracking:** It is a technique for solving problems by trying out different solutions and undoing them if they do not work. Recursive algorithms are often used in backtracking.
- **Searching and Sorting Algorithms:** Many searching and sorting algorithms, such as binary search and quicksort, use recursion to divide the problem into smaller sub-problems.
- **Tree and Graph Traversal:** Recursive algorithms are often used to traverse trees and graphs, such as depth-first search and breadth-first search.
- **Mathematical Computations:** Recursion is also used in many mathematical computations, such as the factorial function and the Fibonacci sequence.
- **Dynamic Programming:** It is a technique for solving optimization problems by breaking them down into smaller sub-problems. Recursive algorithms are often used in dynamic programming.

Overall, recursion is a powerful and versatile technique that can be used to solve a wide range of problems in programming and computer science.

Drawbacks of Recursion

- **Performance:** Recursive algorithms can be less efficient than iterative algorithms in some cases, particularly if the data structure is large or if the recursion goes too deep.
- **Memory usage:** Recursive algorithms can use a lot of memory, particularly if the recursion goes too deep or if the data structure is large. Each recursive call creates a new stack frame on the call stack, which can quickly add up to a significant amount of memory usage.
- **Code complexity:** Recursive algorithms can be more complex than iterative algorithms.
- **Debugging:** Recursive algorithms can be more difficult to debug than iterative algorithms, particularly if the recursion goes too deep or if the program is using multiple recursive calls.
- **Stack Overflow:** If the recursion goes too deep, it can cause a stack overflow error, which can crash the program.

Passing Parameters To Functions

We have already seen the concept of actual and formal parameters. We also know that actual parameters pass values to a function which is received by the format parameters. This is called the passing of parameters.

In C++, we have certain ways to pass parameters as discussed below.

Pass by Value

In the program to swap two integers that we discussed earlier, we have seen that we just read integers ‘a’ and ‘b’ in main and passed them to the swap function. This is the pass by value technique.

In pass by value technique of parameter passing, the copies of values of actual parameters are passed to the formal parameters. Due to this, the actual and formal parameters are stored at different memory locations. Thus, changes made to formal parameters inside the function do not reflect outside the function.

We can understand this better by once again visiting the swapping of two numbers.

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

  void swap(int a, int b) { //here a and b are formal parameters b = a
              + b;
              a = b - a;
              b = b - a;

cout<<"\nAfter swapping inside Swap:\n ";
cout<<"a = "<<a;
cout<<"\tb = "<<b; return;
  }
int main()
  {
int a,b;
cout<<"Enter the two numbers to be swapped: "; cin>>a>>b;

cout<<"a = "<<a;
cout<<"\tb = "<<b;
swap(a,b);
cout<<"\nAfter swapping inside Main:\n ";
cout<<"a = "<<a;
cout<<"\tb = "<<b;
   }
```

Output:

```
Enter the two numbers to be swapped: 3 2 a = 3
b = 2
After swapping inside Swap:
a = 2 b = 3
After swapping inside Main:
a = 3 b = 2
```

We have simply modified the earlier program to print the values of formal parameters & actual parameters before and after the function call.

As seen from the output, we pass values a=3 and b=2 initially. These are the actual parameters. Then after swapping inside the swap function, we see that the values are actually swapped and a=2 and b=3.

However, after the function call to swap, in the main function, the values of a and b are still 3 and 2 respectively. This is because the actual parameters passed to function where it has a copy of the variables. Hence although the formal parameters were exchanged in the swap function they were not reflected back.

Though Pass by value technique is the most basic and widely used one, because of the above limitation, we can only use it in the cases where we do not require the function to change values in calling the function.

Pass by Reference

Pass by reference is yet another technique used by C++ to pass parameters to functions. In this technique, instead of passing copies of actual parameters, we pass references to actual parameters.

Note: References are nothing but aliases of variables or in simple words, it is another name that is given to a variable. Hence a variable and its reference share same memory location. We will learn references in detail in our subsequent tutorial.
In pass by reference technique, we use these references of actual parameters and as a result, the changes made to formal parameters in the function are reflected back to the calling function.

We modify our swap function for our readers to understand the concept better.

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

  void swap(int &a, int &b){ int
     temp = a;
     a = b;
     b = temp;
   }

int main()
   {
     int a,b;
     cout<<"Enter the two numbers to be swapped: "; cin>>a>>b;

     cout<<"a = "<<a;
     cout<<"\tb = "<<b;
     swap(a,b);
     cout<<"\nAfter swapping inside Main:\n ";
     cout<<"a = "<<a;
     cout<<"\tb = "<<b;
   }
```

Output:

```
Enter the two numbers to be swapped: 25 50 a =
25 b = 50
After swapping inside Main: a =
50 b = 25
```

Note: The pass by reference technique shown in the above example. We can see that the actual parameters are passed as it is. But we append an '&' character to the formal parameters indicating that it's a reference that we are using for this particular parameter.

Hence the changes made to the formal parameters in the swap function reflect in the main function and we get the swapped values.

Default Parameters

In C++, we can provide default values for function parameters. In this case, when we invoke the function, we don't specify parameters. Instead, the function takes the default parameters that are provided in the prototype.

The following Example demonstrates the use of Default Parameters.

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

  int mathoperation(int a, int b = 3, int c = 2){

    return ((a*b)/c);
   }

int main()
   {
    int a,b,c;

    cout<<"Enter values for a,b and c: "; cin>>a>>b>>c;
    cout<<endl;
    cout<<"Call to mathoperation with 1 arg : "<<mathoperation(a);
    cout<<endl;
    cout<<"Call to mathoperation with 2 arg : "<<mathoperation(a,b);
    cout<<endl;
    cout<<"Call to mathoperation with 3 arg : "<<mathoperation(a,b,c);
    cout<<endl;
   }
```

Output:

```
Enter values for a,b and c: 10 4 6

Call to mathoperation with 1 arg: 15 Call
to mathoperation with 2 arg: 20 Call to
mathoperation with 3 arg: 6
```

As shown in the code example, we have a function 'mathoperation' that takes three parameters out of which we have provided default values for two parameters. Then in the main function, we call this function three times with a different argument list.

The first call is with only one argument. In this case, the other two arguments will have default values. The next call is with two arguments. In this case, the third argument will have a default value. The third call is with three arguments. In this case, as we have provided all the three arguments, default values will be ignored.

Note that while providing default parameters, we always start from the right-most parameter. Also, we cannot skip a parameter in between and provide a default value for the next parameter.

Now let us move onto a few special function related concepts that are important from a programmer's point of view.

Const Parameters

We can also pass constant parameters to functions using the 'const' keyword. When a parameter or reference is const, it cannot be changed inside the function.

Note that we cannot pass a const parameter to a non-const formal parameter. But we can pass const and non- const parameter to a const formal parameter.

Similarly, we can also have const return-type. In this case, also, the return type cannot be modified.

Let us see a code Example that uses const references.

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

  int addition(const int &a, const int &b){ return
     (a+b);
   }
int main()
   {
     int a,b;
     cout<<"Enter the two numbers to be swapped: "; cin>>a>>b; cout<<"a
     = "<<a;
     cout<<"\tb = "<<b;  int
     res = addition(a,b);
     cout<<"\nResult of addition: "<<res;
   }
```

Output:

```
Enter the two numbers to be swapped: 22 33 a = 2
b = 33
Result of addition: 55
```

In the above program, we have const formal parameters. Note that the actual parameters are ordinary non- const variables which we have successfully passed. As formal parameters are const, we cannot modify them inside the function. So we just perform the addition operation and return the value.

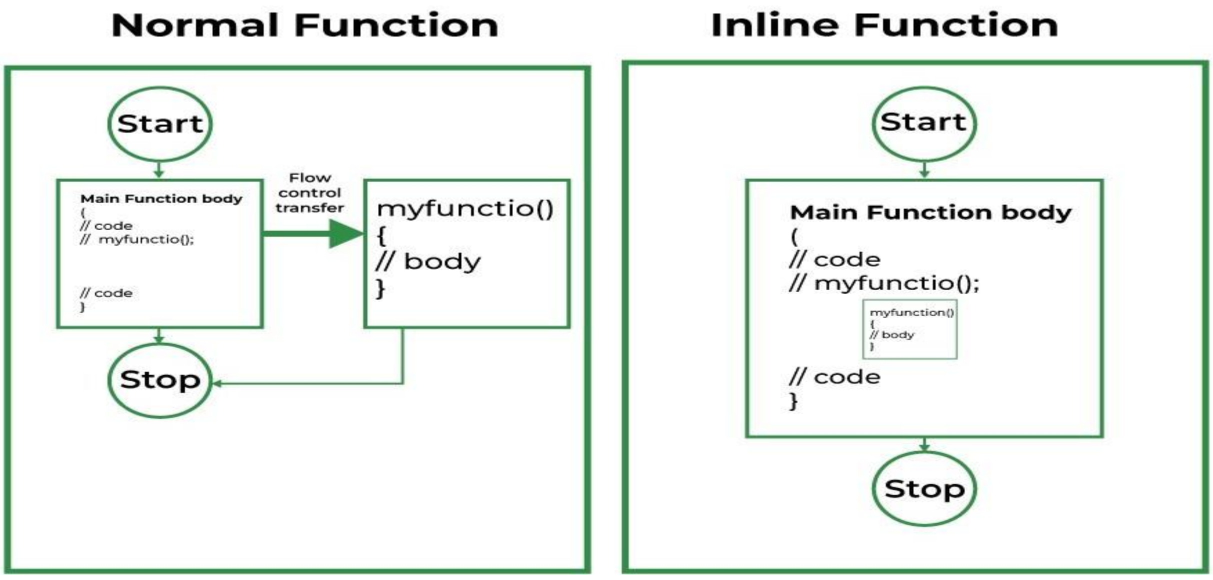
If we try to modify the values of a or b inside the function, then the compiler will issue an error.

Inline Functions

C++ provides inline functions to reduce the function call overhead. An inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call. This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

Syntax:

```
Inline return-type function-name(parameters)
  {
            //function code
  }
```



Remember, inlining is only a request to the compiler, not a command. The compiler can ignore the request for inlining. The compiler may not perform inlining in such circumstances as:

1. If a function contains a loop. (*for, while and do-while*)
2. If a function contains static variables.
3. If a function is recursive.
4. If a function return type is other than void, and the return statement doesn't exist in a function body.
5. If a function contains a switch or goto statement.

Why Inline Functions are Used?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack, and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register, and returns control to the calling function. This can become overhead if the execution time of the function is less than the switching time from the caller function to called function (callee).

For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly- used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because the execution time of a small function is less than the switching time.

Inline functions Advantages:

1. Function call overhead doesn't occur.
2. It also saves the overhead of push/pop variables on the stack when a function is called.
3. It also saves the overhead of a return call from a function.
4. When you inline a function, you may enable the compiler to perform context-specific optimization on the body of the function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of the calling context and the called context.
5. An inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function called preamble and return.

Inline function Disadvantages:

1. The added variables from the inlined function consume additional registers, After the in-lining function if the variable number which is going to use the register increases then they may create overhead on register variable resource utilization. This means that when the inline function body is substituted at the point of the function call, the total number of variables used by the function also gets inserted. So the number of registers going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause overhead on register utilization.
2. If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of the same code.
3. Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.
4. The inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because the compiler would be required to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.
5. Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.
6. Inline functions might cause thrashing because inlining might increase the size of the binary executable file. Thrashing in memory causes the performance of the computer to degrade. The following program demonstrates the use of the inline function.

Example:

```
#include <iostream>
using namespace std;
inline int cube(int s) { return s * s * s; } int
main()
  {
          cout << "The cube of 3 is: " << cube(3) << "\n"; return
          0;
  }
```

Output :

```
The cube of 3 is : 27
```

Function overloading

Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading. In Function Overloading "Function" name should be the same and the arguments should be different. If multiple functions having same name but parameters of the functions should be different is known as Function Overloading.
If we have to perform only one operation and having same name of the functions increases the readability of the program.
Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the function such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you to understand the behavior of the function because its name differs.

The parameters should follow any one or more than one of the following conditions for Function overloading:

- Parameters should have a different type

add(int a, int b)
add(double a, double b)

- Parameters should have a different number

add(int a, int b)
add(int a, int b, int c)

- Parameters should have a different sequence of parameters.

add(int a, double b)
add(double a, int b)

example:

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

void add(int a, int b)
  {
    cout << "sum = " << (a + b);
  }

void add(double a, double b)
  {
     cout << endl << "sum = " << (a + b);
  }

// Driver code int
main()
  {
     add(10, 2);
     add(5.3, 6.2);

     return 0;
  }
```

Output:

```
Sum=12  Sum=11.5
```

How does Function Overloading work?

- *Exact match*:- (Function name and Parameter)
- *If* a *not exact match is found*:–
  ->Char, Unsigned char, and short are promoted to an int.
  ->Float is promoted to double
- *If no match is found*:
  ->C++ tries to find a match through the standard conversion.

- *ELSE ERROR*

Math Library Functions .

C++ provides a large number of mathematical functions that can be used directly in the program. Being a subset of C language, C++ derives most of these mathematical functions from math.h header of C.
In C++, the mathematical functions are included in the header **<cmath>.**

Mathematical Functions In C++

Table Of C++ Mathematical Functions:

Given below is a list of the important mathematical functions in C++ along with their description, prototype, and example.

No	**Function**	**Prototype**	**Description**	**Example**
Trigonometric Functions				
1	cos	double cos (double x);	Returns cosine of angle x in radians.	cout<< cos (60.0 * PI / 180.0); (here PI = 3.142) **returns 0.540302
2	sin	double sin(double x);	Returns sine of angle x in radians.	cout<< sin (60.0 * PI / 180.0); (here PI = 3.142) **returns 0.841471
3	tan	double tan (double x);	Returns tangent of angle x in radians.	cout<< tan (45.0 * PI / 180.0); (here PI = 3.142) **returns 0.931596

4	acos	double acos (double x);	Returns arc cosine of angle x in radians. **Arc cosine is the inverse cosine of cos operation.	double param = 0.5; cout<< acos (param) * 180.0 / PI; (here PI = 3.142) **returns 62.8319

No	Function	Prototype	Description	Example

5	asin	double asin(double x);	Returns arc sine of angle x in radians.
			**Arc sine is the inverse sine of sin operation.
				double param = 0.5;
				cout<< asin (param) *
				180.0 / PI;
				(here PI = 3.142)
				**return 31.4159

6	atan	double atan (double x);	Returns arc tangent of angle x in radians. **Arc tangent is the inverse tangent of tan operation.	double param = 1.0; cout<< atan (param) * 180.0 / PI; (here PI = 3.142) **returns 47.1239
Power Functions				
7	pow	double pow (double base, double exponent);	Returns the base raised to power exponent.	cout<<"2^3 = "<< pow(2,3); **returns 8
8	sqrt	double sqrt(double x);	Returns square root of x.	cout<< sqrt(49); ** returns 7
Rounding and Remainder Functions				
9	ceil	double ceil (double x);	Returns smallest integer value that is not less than x; Rounds x upward.	cout<< ceil(3.8); **returns 4
10	floor	double floor (double x);	Returns larger integer value that is not greater than x; Rounds x downward.	cout<< floor(2.3); **returns 2
11	fmod	double fmod (double numer, double denom);	Returns floating-point remainder of numer/denom.	cout<< fmod(5.3,2); **returns 1.3

12	trunc	double trunc (double x); **also provides variations for float and long	Returns the nearest integral value not larger than x. Rounds x toward zero.	cout<< trunc(2.3); **returns 2

No	Function	Prototype	Description	Example
		double		
13	round	double round (double x); **also provides variations for float and long double	Returns integral value that is nearest to x.	cout<< round(4.6); **returns 5
14	remainder	double remainder (double numer, double denom); **also provides variations for float and long double	Returns floating point remainder of numer/denom rounded to nearest value.	cout<< remainder(18.5 ,4.2); **returns 1.7
Minimum, Maximum, Difference and Absolute Functions				
15	fmax	double fmax (double x, double y). **also provides variations for float and long double.	Returns larger value of the arguments x and y. If one number is NaN, other is returned.	cout<< fmax(100.0,1.0); **returns 100
16	fmin	double fmin (double x, double y); **also provides variations for float and long double.	Returns smaller value of the arguments x and y. If one number is NaN, other is returned.	cout<< fmin(100.0,1.0); **returns 1

17	fdim	double fdim (double x, double y); **also provides variations for float and long	Returns the positive difference between x and y. If x > y, returns x-y; otherwise returns zero.	cout<< fdim(2.0,1.0); **returns 1

No	Function	Prototype	Description	Example
		double.		
18	fabs	double fabs(double x);	Returns absolute value of x.	cout<< fabs(3.1416); **returns 3.1416
19	abs	double abs (double x); **also provides variations for float and long double.	Returns absolute value of x.	cout<< abs(3.1416); **returns 3.1416
Exponential and Logarithmic Functions				
20	exp	double exp (double x);	Returns exponential value of x i.e. e x.	cout<< exp(5.0); **returns 148.413
21	log	double log (double x);	Returns natural logarithm of x.(to the base e).	cout<< log(5); **returns 1.60944
22	log10	double log10 (double x);	Returns common logarithm of x (to the base 10).	cout<< log10(5); **returns 0.69897

C++ program that demonstrates all the functions discussed above.

```
#include <iostream>
#include <cmath> using
namespace std; int main ()
  {
    int PI = 3.142;
    cout<< "cos(60) = " << cos ( 60.0 * PI / 180.0 )<<endl; cout<<
    "sin(60) = " << sin ( 60.0 * PI / 180.0 )<<endl; cout<<
    "tan(45) = " << tan ( 45.0 * PI / 180.0 )<<endl; cout<<
    "acos(0.5) = " << acos (0.5) * 180.0 / PI<<endl; cout<<
    "asin(0.5) = " << asin (0.5) * 180.0 / PI<<endl; cout<<
    "atan(1.0) = " << atan (1.0) * 180.0 / PI<<endl; cout<< "2^3 =
    " << pow(2,3)<<endl;
    cout<< "sqrt(49) = " << sqrt(49)<<endl;
    cout<< "ceil(3.8) = " << ceil(3.8)<<endl;
```

```
cout<< "floor(2.3) = " << floor(2.3)<<endl;
cout<< "fmod(5.3,2) = " << fmod(5.3,2)<<endl; cout<<
"trunc(5.3,2) = " << trunc(2.3)<<endl; cout<<
"round(4.6) = " << round(4.6)<<endl;
```

```
    cout<< "remainder(18.5,4.2) = " << remainder(18.5 ,4.2)<<endl; cout<<
    "fmax(100.0,1.0) = " << fmax(100.0,1.0)<<endl;
    cout<< "fmin(100.0,1.0) = " << fmin(100.0,1.0)<<endl;
    cout<< "fdim(2.0,1.0) = " << fdim(2.0,1.0)<<endl; cout<<
    "fabs(3.1416) = " << fabs(3.1416)<<endl; cout<<
    "abs(3.1416) = " << abs(3.1416)<<endl;
    cout<< "log(5) = " << log(5)<<endl; cout<<
    "exp(5.0) = " << exp(5.0)<<endl; cout<<
    "log10(5) = " << log10(5)<<endl; return 0;
  }
```

Output:

```
cos(60) = 0.540302
sin(60) = 0.841471
tan(45) = 0.931596
acos(0.5) = 62.8319
asin(0.5) = 31.4159
atan(1.0) = 47.1239
2^3 = 8
sqrt(49) = 7
ceil(3.8) = 4
floor(2.3) = 2
fmod(5.3,2) = 1.3
trunc(5.3,2) = 2
round(4.6) = 5
remainder(18.5,4.2) = 1.7
fmax(100.0,1.0) = 100
fmin(100.0,1.0) = 1
fdim(2.0,1.0) = 1
fabs(3.1416) = 3.1416
abs(3.1416) = 3.1416
log(5) = 1.60944
exp(5.0) = 148.413
log10(5) = 0.69897
```

In the above program, we have executed the mathematical functions that we tabularized above along with their respective results.

Next, we will discuss some of the important mathematical functions used in C++.
Abs => Computes the absolute value of a given number.
Sqrt => Used to find the square root of the given number.
Pow => Returns the result by raisin base to the given exponent.
Fmax => Finds the maximum of two given numbers.
We will discuss each function in detail along with C++ examples. We will also get to know more about the mathematical constant M_PI that is often used in quantitative programs.

<u>C++ abs</u>

Function prototype: return_type abs (data_type x);
Function Parameters: x=> value whose absolute value is to be returned.

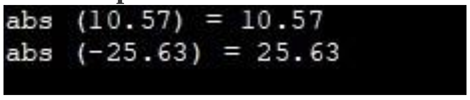
x can be of the following types:
double
float
long double

Return value: Returns the absolute value of x.
As parameters, the return value can also be of the following types:
double
float
long double

Description: Function abs is used to return the absolute value of the parameter passed to the function.
Example:

```
#include <iostream>
#include <cmath> using
namespace std; int main ()
  {
    cout << "abs (10.57) = " << abs (10.57) << '\n'; cout <<
    "abs (-25.63)  = " << abs (-25.63) << '\n'; return 0;
  }
```

Output:



```
abs (10.57) = 10.57
abs (-25.63) = 25.63
```

Here, we have used examples with a positive and negative number with the abs function for clarity purposes.

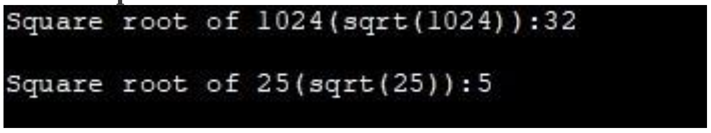
C++ sqrt
Function prototype: double sqrt (double x);
Function Parameters: x=>value whose square root is to be computed. If x is negative, domain_error occurs.

Return value: A double value indicating the square root of x. If x is negative, domain_error occurs.

Description: The sqrt function takes in the number as a parameter and computes their squares root. If the argument is negative, a domain error occurs. When domain error occurs, then the global variable errno is set **EDOM**.
Example:

```
#include <iostream>
#include <cmath> using
namespace std; int main ()
  {
    double param, result;
    param = 1024.0;
    result = sqrt (param);
    cout<<"Square root of "<<param<<"(sqrt("<<param<<")):"<<result<<endl; param = 25;
    result = sqrt (param);
    cout<<"Square root of "<<param<<"(sqrt("<<param<<")):"<<result<<endl;

    return 0;
  }
```

Output:



```
Square root of 1024(sqrt(1024)):32

Square root of 25(sqrt(25)):5
```

In the above program, we have computed the square root of 1024 and 25 using the sqrt function.

C++ pow

Function prototype: double pow (double base, double exponent).
Function Parameters: base=> base value.
Exponent=> exponent value

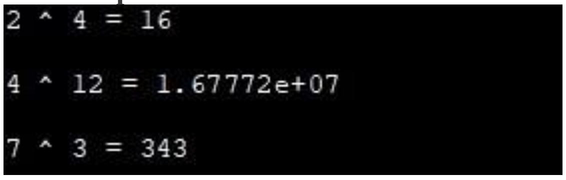
Return value: The value obtained after raising the base to the exponent.
Description: The function pow takes in two arguments i.e. base and exponent and then raises the base to the power of the exponent.
If the base if finite negative and exponent is negative but not an integer value then the domain error occurs. Certain implementations may cause domain error when both base and exponent are zero and if the base is zero and exponent is negative.

If the function result is too small or too large for the return type, then it may result in a range error.

Example:

```
#include <iostream>
#include <cmath> using
namespace std; int main ()
  {
    cout<< "2 ^ 4 = "<<pow (2.0, 4.0)<<endl;
    cout<< "4 ^ 12 = "<< pow (4, 12.0)<<endl;
    cout<< "7 ^ 3 = "<< pow (7, 3);
    return 0;
  }
```

Output:



```
2 ^ 4 = 16

4 ^ 12 = 1.67772e+07

7 ^ 3 = 343
```

The above program demonstrates the usage of the POW function in C++. We can see that it computes the value by raising a number to the specified power.

C++ max

Function prototype: double fmax (double x, double y);

Function Parameters: x, y=> two values to be compared to find the maximum.

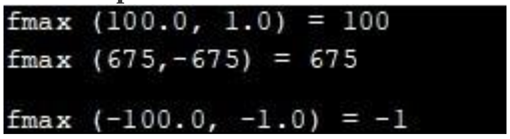
Return value: Returns the maximum value of the two parameters. If one of the parameters is Nan, the other value is returned.

Description: The function fmax takes in two numeric arguments and returns the maximum of the two values. Apart from the prototype mentioned above, this function also has overloads for other data types like float, long double, etc.

Example:

```
#include <iostream>
#include <cmath> using
namespace std; int main ()
  {
    cout <<"fmax (100.0, 1.0) = " << fmax(100.0,1.0)<<endl; cout <<
    "fmax (675, -675) = " << fmax(675.0, -675.0)<<endl; cout <<
    "fmax (-100.0, -1.0) = " << fmax(-100.0,-1.0)<<endl;

    return 0;
```

Output:



```
fmax (100.0, 1.0) = 100
fmax (675,-675) = 675

fmax (-100.0, -1.0) = -1
```

The above code shows the usage of the fmax function to find the maximum of two numbers. We see the cases where one of the numbers is negative, and both the numbers are negative.

Mathematical Constants In C++

The <cmath> header of C++ also includes several mathematical constants that can be used in mathematical and quantitative code.

To include mathematical constants in the program, we have to use a #define directive and specify a macro "_USE_MATH_DEFINES". This macro is to be added to the program before we include the <cmath> library.

This is done as shown below:

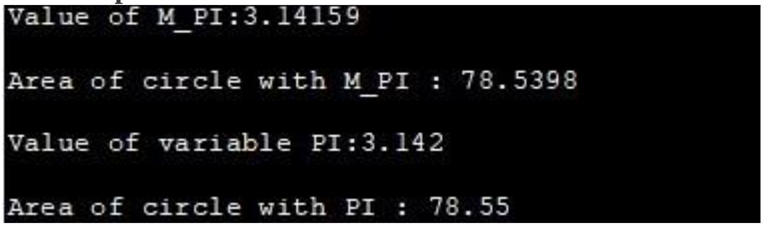
```
#define _USE_MATH_DEFINES
 #include <iostream>
 #include <cmath>
 ….C++ Code…..
```

One of the constants that we use frequently while writing mathematical and quantitative applications is PI. The following program shows the usage of predefined constant PI in the C++ program.

```
#define _USE_MATH_DEFINES #include


<cmath>
```

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

int main() {
    double area_circle, a_circle; int
    radius=5;
    double PI = 3.142;
    //using predefined PI constant  area_circle
    = M_PI * radius * radius; cout<<"Value of
    M_PI:"<<M_PI<<endl;
    cout << "Area of circle with M_PI : "<<area_circle << endl;
    //using variable PI
    a_circle = PI * radius * radius; cout<<"Value
    of variable PI:"<<PI<<endl;
    cout << "Area of circle with PI : "<<a_circle << endl;

    return 0;
  }
```

Output:



```
Value of M_PI:3.14159

Area of circle with M_PI : 78.5398

Value of variable PI:3.142

Area of circle with PI : 78.55
```

The above program demonstrates the mathematical constant M_PI available in <cmath>. We have also provided a local variable PI initialized to the value 3.142. The output shows the area of circle computed using M_PI and local PI variable using the same radius value.

UNIT 3

DERIVED DATA TYPES

Arrays:

An array in C or C++ is a collection of items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They are used to store similar types of elements as in the data type must be the same for all elements. They can be used to store the collection of primitive data types such as **int, float, double, char, etc** of any particular type. To add to it, an array in C or C++ can store derived data types such as the structures, pointers, etc.
There are two types of arrays:

- One Dimensional Array
- Multi Dimensional Array

One Dimensional Array: A one dimensional array is a collection of same data types. 1-D array is declared as:

```
data_type variable_name[size]
```

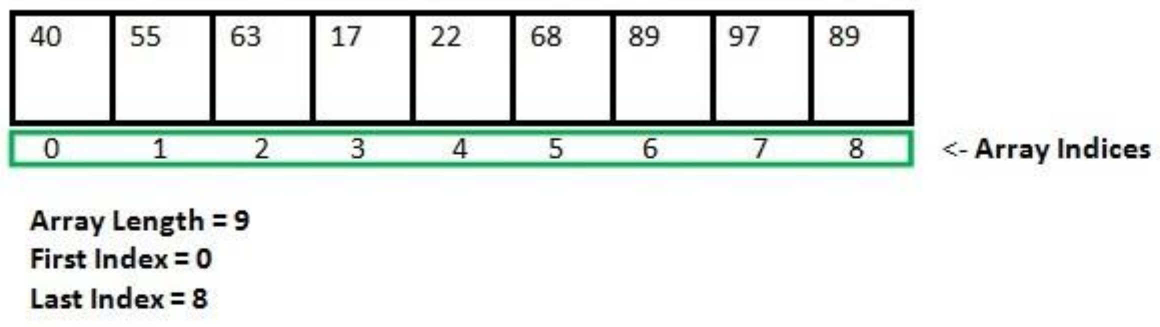
data_type is the type of array, like int, float, char, etc.

variable_name is the name of the array.

size is the length of the array which is fixed.

Note: The location of the array elements depends upon the data type we use. Below is the

illustration of the array:



Below is the program to illustrate the traversal of the array:

```
// C++ program to illustrate the traversal

// of the array

#include "iostream"
```

```
using namespace std;


// Function to illustrate traversal in arr[]

void traverseArray(int arr[], int N)

{

        // Iterate from [1, N-1] and print

        // the element at that index

        for (int i = 0; i < N; i++) {

                cout << arr[i] << ' ';

        }

}

// Driver Code

int main()

{

        // Given array

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

        // Size of the array

        int N = sizeof(arr) / sizeof(arr[0]);

        // Function call

        traverseArray(arr, N);

}
```

OUTPUT:

```
1 2 3 4
```

MultiDimensional Array: A multidimensional array is also known as array of arrays. Generally, we use a two-dimensional array. It is also known as the matrix. We use two indices to traverse the rows and columns of the 2D array. It is declared as:

```
data_type variable_name[N][M]
```

}

data_type is the type of array, like int, float, char, etc.

variable_name is the name of the array.

N is the number of rows.

M is the number of columns.

Below is the program to illustrate the traversal of the 2D array:

```
// C++ program to illustrate the traversal

// of the 2D array

#include "iostream"

using namespace std;

const int N = 2;

const int M = 2;

// Function to illustrate traversal in arr[][]

void traverse2DArray(int arr[][M], int N)

{

        // Iterate from [1, N-1] and print

        // the element at that index

        for (int i = 0; i < N; i++) {

                for (int j = 0; j < M; j++) {

                        cout << arr[i][j] << ' ';

                }

                cout << endl;

        }

}

// Driver Code
```

```
int main()
```

```
{       // Given array

        int arr[][M] = { { 1, 2 }, { 3, 4 } };

        // Function call

        traverse2DArray(arr, N);

        return 0;

} OUTPUT 1 2

    3 4
```

Strings

C++ string class internally uses character array to store character but all memory management, allocation, and null termination are handled by string class itself that is why it is easy to use. For example it is declared as:

```
char str[] = "C++STRINGS"
```

Below is the program to illustrate the traversal in the string:

```
// C++ program to illustrate the

//   traversal   of   string

#include      "iostream"

using namespace std;


// Function to illustrate traversal

// in string

void traverseString(char str[])

{

        int i = 0;


        // Iterate till we found '\0' while

        (str[i] != '\0') {
```

```
                        printf("%c ", str[i]);

                        i++;

            }

}


// Driver Code int

main()

{


            // Given string

            char str[] = "C++STRINGS";


            // Function call

            traverseString(str);


            return 0;

}

Output:

C + + S T R I N G S
```

The string data_type in C++ provides various functionality of string manipulation. They are:

1. **strcpy()**: It is used to copy characters from one string to another string.
2. **strcat()**: It is used to add the two given strings.
3. **strlen()**: It is used to find the length of the given string.
4. **strcmp()**: It is used to compare the two given string.

Below is the program to illustrate the above functions:

// C++ program to illustrate functions

// of string manipulation

```
#include "iostream"

#include "string.h"

using namespace std;


// Driver Code

int main()

{

        // Given two string

        char str1[100] = "HelloC++class";

        char str2[100] = "HelloC++";

        // To get the length of the string

        // use strlen() function

        int x = strlen(str1);

        cout << "Length of string " << str1

                << " is " << x << endl;

        cout << endl;

        // To compare the two string str1

        // and str2 use strcmp() function

        int result = strcmp(str1, str2);

        // If result is 0 then str1 and str2

        // are equals

        if (result == 0) {

                cout << "String " << str1

                        << " and String " << str2

                        << " are equal." << endl;
}
```

```
    else {
            cout << "String " << str1
            << " and String " << str2
            << " are not equal." << endl;
    }
    cout << endl;

    cout << "String str1 before: "
        << str1 << endl;

    // Use strcpy() to copy character
    // from one string to another
    strcpy(str1, str2);

    cout << "String str1 after: "
        << str1 << endl;

    cout << endl;

    return 0;
}
```

Output:

```
Length of string helloc++class is 13
String helloc++class and HelloC++ are not equal.
String str1 before: helloc++class
String str1 after: HelloC++
```

POINTERS

Introduction:

- **Pointers are a powerful concept in C++ and have the following advantages.**

  i. It is possible to write efficient programs.
  ii. Memory is utilized properly.
  iii. Dynamically allocate and de-allocate memory.

- **Memory Utilization of Pointer:**

- Memory is organized as an array of bytes. A byte is basic storage and accessible unit in memory.
- Each byte is identifiable by a unique number called address.
- We know that variables are declared before they are used in a program. Declaration of a variable tells the compiler to perform the following.
  - Allocate alocation in memory.The number of location depends on data type.
  - Establish relation between address of the location and the name of the variable.
- Consider the declaration, intnum;
- This declaration tells the compiler to reserve a location in memory. We know that size of int type is two bytes. So the location would be two bytes wide.

Address	Num
100	15
101	

- In the figure, num is the variable that stores the value 15 and address of num is 100. The address of a variable is also an unsigned integer number. It can be retrieved and stored in another variable.

- **Pointer:**

- ***A pointer is a variablethat holds amemory address ofanother variable*.**
- The pointer has the following advantages.
  - Pointers save memory space.
  - Dynamically allocate and de-allocate memory.
  - Easy to deal with hardware components.
  - Establishes communication between program and data.
  - Pointers are used for file handling.

    Pointers are used to create complex data structures such as linkedlist, stacks, queues trees and graphs.

- **Pointer Declaration:**

- Pointers are also variables and hence,they must be defined in a program like any other variable.
- The general syntax of pointer declaration is given below.

  Syntax: **Data_Type*Ptr_Variablename;**
- Where,
  - Data_Type is any valid data type supported by C++ or any user defined type.
  - Ptr_Variable name is the name of the pointer variable. The presence of **'*'** indicates that it is a **pointer variable**.
- Defining pointer variables:

int *iptr;	iptr is declared to be a pointer variable of int type.
float *fptr;	fptr is declared to be a pointer variable of float type.
char*cptr;	cptr is declared to be a pointer variable of character type.

➢ Pointer Initialization:

- Once we declare a pointer variable, we must make it to point to something.
- We can do this by assigning or initializing to the pointer the address of the variable you want topoint to as in: **iptr = &num**;
- The„**&'** is the **address operator** and it represents address of the variable.
- Example: A program to display the content of num and the address of the variable num using a pointer variable.

```
#include<iostream.h>

void main( )
{
    int num;                    //normal integer variable
    int *iptr;                  //Pointer declaration, pointing to integer data

    num = 574;                  //assign value to num
    iptr =&num                  //assign address of num to int pointer

    cout<<" Value of num is :"<<num<<endl;
    cout<<"Address of num is:"<<iptr<<endl;
}
```

➢ The address of operator(&):

- „&" is a unary operator that returns ***the memory address of its operand***.
- For example, if var is an integer variable, then &var is its address.

  We should read „&" operatoras"the address-of"which means & var will be read as"the address of var".

Ex:

```
int num=25; int *iptr;

iptr =&num;                              //The address of operator &
```

Pointer Operatoror Indirection Operator(*):

- The second operator is indirection operator„*‟,and it is the complement of„&‟.
- It is a unary operator that returns ***the value of the variable located*** at the address specified by its operand.
- Example:

```
                int num = 25;

                int *iptr;                              //Pointer Operator (Indirection Operator*) iptr

                = &num;
```

- Example:A program executes the two operations.

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

    void main( )
{

    Int   num;
    int*iptr;
    int val;

    num  =  300;
    iptr=&num;
    val = *iptr;

    cout<<"Value   of   num   is:"<<num<<endl;

    cout<<"  Value  of  pointer  :"<<iptr<<endl;

    cout<<" Value of val :"<<val<<endl;

}
```

OUTPUT:

```
Value of num is : 300

Value of pointer:0xbff64494 Value

of val : 300
```

Pointer Arithmetic:

- We can perform arithmetic operations on a pointer just as you can a numeric value.
- There are four arithmetic operators that can be used on pointers:
  - Increment ++
  - Decrement --
  - Addition +
  - Subtraction -

Example:

```
                in tnum,*iptr;
```

iptr→	1200	9
	1201	
iptr++→	1202	
	1203	

```
num = 9;
iptr=&num;
iptr++; cout<<iptr;
```

- The following operation can be performed on pointers.
    - We can add integer value to a pointer.
    - We can subtract an integer value from a pointer.
    - We can compare two pointers, if they point the elements of the same array.
    - We can subtract one pointer from another pointer if both point to the same array.
    - We can assign one pointer to another pointer provided both are of same type.
- The following operations cannot be performed on pointers.
    - Addition of two pointers.
    - Subtraction of one pointer from another pointer when they do not point to the same array.
    - Multiplication of two pointers.
    - Division of two pointers.
- A program to illustrate the pointer expression and pointer arithmetic.

```
#include<iostream.h>
    #include<conio.h> void
    main( )
{
    int a, b, x, y;int*ptr1,*ptr2
    ;
    a=30;
    b =6;
    ptr1 = &a
    ptr2 =&b;
    x=*ptr1 +*ptr2 – 6;
    y=6-*ptr1 / *ptr2 +30;

    cout<<"Addressofa="<<ptr1<<endl;
    cout<<"Addressofb="<<ptr2<<endl;
    cout<<"a = "<<a<<"b = "<<b<<endl;
    cout<<"x = "<<x<<"y = "<<y<<endl;

    *ptr1 =*ptr1 +70;
    *ptr2 =*ptr2 * 2;
     cout<<"a= "<<a<<"b= "<<b<<endl;
}
```

OUTPUT:

```
Addressofa=65524
Addressofb=65522  a
= 30            b = 6
x=30            y=6
a=100           b =12
```

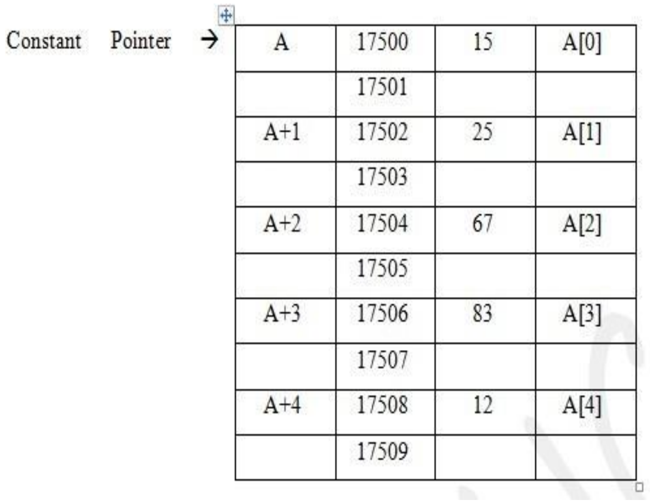
➢ **Pointers and Arrays:**

- There is a close relationship between array and pointers in C++.
- Consider the following program which prints the elementsof an array A.

```
#include<iostream.h>
     voidmain( )
{
                intA[5]={ 15, 25,67, 83, 12};
                for(int i = 0; i<5; i++)
                            cout<<A[i]<<"\t";
}
```

- Output of the above program is:15   25   67   83   12
- When we declare an array, its name is treated as a constant pointer to the first element of the array.
- This is also known as the **base address of the array**.
- In other words base address is the address of the first element in the array of the address of a[0].
- If we use constant pointer to print array elements.

```
#include<iostream.h>
     voidmain( )
{
                intA[5]={ 15, 25,67, 83, 12};
                cout<<*(A)              <<"\t";
                cout<<*(A+1)            <<"\t";
                cout<<*(A+2)            <<"\t";
                cout<<*(A+3)            <<"\t";
                cout<<*(A+4)            <<"\t";
}
```


Constant Pointer →

A	17500	15	A[0]
	17501		
A+1	17502	25	A[1]
	17503		
A+2	17504	67	A[2]
	17505		
A+3	17506	83	A[3]
	17507		
A+4	17508	12	A[4]
	17509		

- Outputof the above program is:1525

  678312

- Here the expression *(A+3) has exactly same effect as A[3]in the program.
- The difference between constant pointer and the pointer variable is that the constant pointer cannot be incremented or changed while the pointer to an array which carries the address of the first element of the array may be incremented.
- The following example shows the relationship between pointer and one dimensional array.

```
     #include<iostream.
     h>void main( )
```

{

```
    nt a[10],i,n;
    cout<<"Entertheinputforarray";
    cin>>n;
    cout<<"Enterarrayelements:";
    for(i=0; i<n; i++)
            cin>>*(a+i);
    cout<<Thegivenarrayelementsare:"; for(i=0; i<n;
    i++)
            cout<<"\t"<<*(a+i);g
    etch( );
}
```

```
OUTPUT:

Entertheinputforarray5 Enter
array elements
1       2       3       4       5
Thegiven arrayelementsare:
1       2       3       4       5
```

➢ Array of Pointers:

- There is an array of integers;array of float, similarly there can be an array of pointers.
- "An array of pointer means that it is a collection of address".
- The general form of array of pointers declaration is:

int*pint[5];

- The above statement declares an array of 5 pointers where each of the pointer to integer variable.
- Example: Program to illustrate the array of pointers of isolatedvariables.

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

    int *pint[5];
    inta=10, b =20, c=30,d=40, e=50;
    pint[0]=&a;
    pint[1]=&b;
    pint[2]=&c;
    pint[3]=&d;
    pint[4]=&e;

    for(inti=0;i<=4;i++)
            cout<<"Value"<<*pint[i]<<"storedat"<<pint[i]<<endl;
}
```

```
OUTPUT:

Value10storedat17500
Value20storedat17502
Value30storedat17504
Value40storedat17506
Value50storedat17508
```

➢ Pointers and strings:

- String is sequence of characters ends with null(„\0‟)character.
- C++ provides two methods of declaring and initializing a string.
- **Method1:**

```
charstr1[ ] =“HELLO”;
```

- Whena string is declared using an array,the compiler reserves one element longer than the Number of characters in the string to accommodate NULLcharacter.
- The string str1[] is 6 bytes longto initialize its first 5characters HELLO\0.
- **Method2:**

```
char*str2=“HELLO”;
```

- C++ treats string constants like other array and interrupts a string constant as a pointer to the first character of the string.
- This means that we can assign a string constanto a pointer that point to a char.
- Example:A program to illustrate the difference between strings as arrays and pointers.

```
#include<iostream.h>
    #include<conio.h> void
    main( )
{
    charstr1[]=“HELLO”; char
    *str2 = “HELLO”;
    cout<<str1<<endl;
    cout<<str2<<endl; str2++;
    cout<<str2<<endl;
}
```

```
OUTPUT:
HELLO
HELLO
ELLO
```

- **Pointers as Function Parameters:**

- A pointer can be a parameter.It works like a reference parameter to allow change to argument from within the function.

```
    voidswap(int*m,int *n)
{
        int temp;
        temp=*m;
        *m = *n;
        *n =temp;
}

    voidswap(&num1, &num2);
```

- **Pointers and Functions:**

- A function may be invoked in one of two ways :
  - Call by value
  - Call by reference

- The second method call by reference can be used in two ways :
  - By passing the references
  - By passing the pointers
- Reference is an alias name for a variable.
- ForExample:

```
int m = 23;
int&n=m; int
*p;
p =&m;
```

- Then the value of mi.e.23is printed in the following ways :

```
cout <<m;   // using variable name cout
<< n;   //usingreferencename  cout  <<
*p;   // using the pointer
```

✓ **<u>Invoking Function by Passing the References:</u>**

- When parameters are passed to the functions by reference, then the formal parameters become references (or aliases) to the actual parameters to the calling function.
- That means the called function does not create its own copy of original values, rather, it refers tothe original values by different names i.e. their references.
- For example the program of swapping two variables with reference method:

```
    #include<iostream.
    h>void main()
{
        voidswap(int&,int&); int a
        = 5, b = 6;
        cout<<"\nValueofa:"<<a<<"andb:"<<b; swap(a, b); cout<<"\nAfter
        swappingvalueofa:"<< a<< "and b:" <<b;
}

    voidswap(int&m,int&n)
{
        int temp;
        temp =m;
        m = n;
        n =temp;
}
```

OUTPUT:

```
Value of a: 5 and b : 6

After swapping value of a: 6 and b : 5
```

✓ **<u>Invoking Function by Passing the Pointers:</u>**

- When the pointers are passed to the function, the addresses of actual arguments in the calling function are copied into formal arguments of the called function.

- That means using the formal arguments (the addresses of original values) in the called function; we can make changing the actual arguments of the calling function.
- For example the program of swapping two variables with Pointers:
  #include<iostream.h>
- #include<iostream.h>

```
    void main()
{
        voidswap(int*m,int*n); int a
        = 5, b = 6;
        cout<<"\nValueofa:"<<a<<"andb:"<<b; swap(&a, &b); cout<<"\nAfter
        swappingvalueofa:"<< a<< "and b:" <<b;
}

    voidswap(int *m,int *n)
{
        int temp;
        temp=*m;
        *m = *n;
        *n =temp;
}
```

OUTPUT:

```
Valueofa: 5andb : 6 Afterswappingvalue of
a: 6 and b : 5
```

➢ **<u>Memory Allocation of pointers:</u>**

- Memory Allocation is donein two ways:
  - Static Allocation of memory
  - Dynamic allocation of memory.

✓ **<u>Static Allocation of Memory:</u>**

- Static memory allocation refers to the process of allocating memoryduring the compilation of the program i.e. before the program is executed.
- Example:

```
        int     a;          //Allocates 2bytes of memoryspace duringthe compilation.
```

✓ **<u>Dynamic Allocation of Memory:</u>**

- Dynamic memory allocation refers to the process of allocating memory during the execution ofthe program or at run time.
- Memory space allocated with this method is not fixed.

- C++ supports dynamic allocation and de-allocation of objects using **new** and **delete** operators.

- These operators allocate memory for objects from a pool called the **freestore**.
- The new operator calls the special function operator new and delete operators call the special function operator delete.

❖ **new operator:**

- We can allocate storage for a variable while program is running by using new operator.
- It is used to allocate memory without having to define variables and then make pointers point to them.
- The following code demonstrate show to allocate memory for different variables.
- To allocate memory type integer

```
int *pnumber;
pnumber=newint;
```

- The first line declares the pointer, pnumber. The second line then allocates memoryfor an integer and then makes pnumber point to this new memory.
- To allocate memory for array, double*dptr=newdouble[25];
- To allocates dynamic structure variables orobjects, student sp =new student;

❖ **Delete Operator:**

- The delete operator is used to destroy the variables space which has been created by using the new operator dynamically.
- Use delete operator to free dynamic memoryas : delete iptr;
- To free dynami carray memory: delete[]dptr;
- To free dynamic structure, delete structure;

✓ **<u>Difference between Static Memory Allocation and Dynamic Memory Allocation:</u>**

Sl no	**StaticMemoryAllocation**	**DynamicMemory Allocation**
1	Memory space is allocated before the Execution of program.	Memory space is allocated during the Execution of program.
2	Memory space allocated is fixed	Memory space allocated is not fixed
3	More memory space is required	Less memory space is required.
4	Memory allocation from stack area	Memory space form heap area.

✓ **Freestore (Heapmemory):**

- Free store is a pool of memory available to **allocated and de-allocated storage** for the objects during the **execution of the memory**.

- ✓ **MemoryLeak:**

- If the objects, that are allocated memory dynamically, are not deleted using delete, the memoryblock remains occupied even at the end of the program.
- Such memoryblocksare known as orphaned memoryblocks.
- These orphaned memory blocks when increases in number, bring adverse effect on the system. This situation is called ***memory leak***.

I/O STREAMS INTRODUCTION TO I/O STREAMS

Stream Concept

- C++ I/O operations use the stream concept.
- The stream refers to a series of bytes or data flow.
- Streams improve performance by managing the flow of data between the main memory
- and devices like printers, screens, and network connections.

Output Operation

- Bytes are transferred from the main-memory to an output device (e.g., printer, screen) during an output operation.

Input Operation

- Bytes flow from an input device (e.g., keyboard) to the main-memory during an input operation.

Header Files for I/O

- C++ provides predefined functions and declarations through header files to handle I/O tasks efficiently.

`<iostream>` Header File

- Essential for input/output operations in C++.
- Includes classes like `istream` (input stream) and `ostream` (output stream).
- Commonly used classes:

`cin`: Standard input stream.
`cout`: Standard output stream

Standard Output Stream (`cout`)

- An object of the `ostream` class, connected to the standard output device (typically a display screen).
- Used with the insertion operator (`<<`), to display output on the console.

Standard Input Stream (`cin`)

- An object of the `istream` class, connected to the standard input device (typically a keyboard).
- Used with the extraction operator (`>>`), to read input from the console.

Example

```
#include     <iostream>
using namespace std; int
main() {
int age;
cout << "Enter your age: "; cin
>> age;
cout << "Your age is: " << age << endl;
return 0;
  }
```

Output:

```
Enter your age: 22 Your
age is: 22
```

UNFORMATTED-I/O

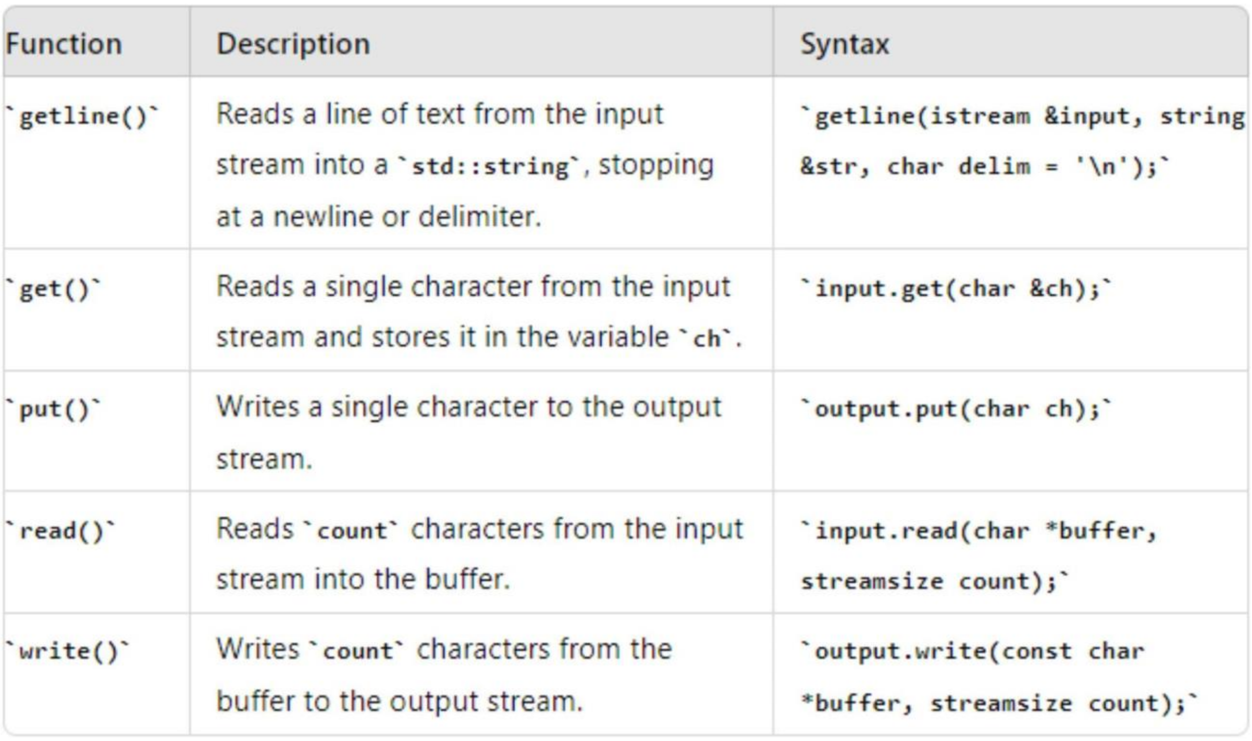
Definition: Input- and output-operations where data is read or written without applying any specific formatting-rules.
The data is transferred in its raw, unmodified form.

Purpose

- To perform simple, direct input- and output-operations without any concern for how the data is presented or formatted.
- Useful for efficient, low-level data handling where formatting is not necessary.

Usage

- Often used when handling binary-data or performing low-level file operations.
- Provides more control over how data is managed and transferred, without altering its format.
- Common unformatted-I/O functions are listed in below table:



Function	Description	Syntax
`getline()`	Reads a line of text from the input stream into a `std::string`, stopping at a newline or delimiter.	`getline(istream &input, string &str, char delim = '\n');`
`get()`	Reads a single character from the input stream and stores it in the variable `ch`.	`input.get(char &ch);`
`put()`	Writes a single character to the output stream.	`output.put(char ch);`
`read()`	Reads `count` characters from the input stream into the buffer.	`input.read(char *buffer, streamsize count);`
`write()`	Writes `count` characters from the buffer to the output stream.	`output.write(const char *buffer, streamsize count);`

getline(istream &input, string &str, char delim = '\n');

Description: This function reads a line-of-text from the input-stream and stores it in a string.
It reads until it encounters a newline-character (`'\n'`) or the end of the file, and then discards the newline-character.

Parameters:

`istream &input`: The input-stream from which the line is read (e.g., `cin`).
`string &str`: A reference to a `string` variable where the line-of-text will be stored.
`char delim` (optional): A delimiter-character that specifies where the input should stop. By default, it is `'\n'`.
• **Example Usage:** Demonstrating getline()

```
#include <iostream>
#include <string> using
namespace std; int
main() {
string line;
cout << "Enter a line-of-text: ";
getline(cin, line); // Reads a line-of-text from the user
cout << "You entered: " << line << endl; // Outputs the entered line return
0;
  }
```
Output:

```
Enter a line-of-text: Hello World You
entered: Hello World
```
get(char &ch)

- **Description:** This function reads a single character from the input-stream and stores it in the variable `ch`.
- **Parameters:**

`char &ch`: A reference to a `char` variable where read character will be stored

put(char ch)

- **Description:** This function writes a single character to the output stream.
- **Parameters:**

`char ch`: The character to be written to the output stream.

- **Example Usage:** Demonstrating get() and put()

```
  #include <iostream>
using namespace std; int
main() {
char c;
cout << "Enter a single character: ";
cin.get(c); // Reads a single character from the input cout
<< "You entered: ";
cout.put(c); // Writes the character to the output cout
<< endl;
return 0;
  }
```
Output:

```
Enter a single character: A You
entered: A
```
read(char *buffer, streamsize count);`

- **Description:** This function reads a block of characters from the input-stream and stores them in the array pointed to by `buffer`.
- **Parameters:**

`char *buffer`: A pointer to a character-array where the read characters will be stored.

`streamsize count`: The maximum number of characters to read.

write(const char *buffer, streamsize count)

- **Description:** This function writes a block of characters from the array pointed to by `buffer` to the output stream.

• **Parameters:**
`const char *buffer`: A pointer to a character-array containing the data to be written.
`streamsize count`: The number of characters to write.

• **Example Usage:** Demonstrating read() and write()

```
#include <iostream>
using namespace std;
int main() {
char data[20];
cout << "Enter up to 20 characters: ";
cin.read(data, 20); // Reads up to 20 characters into the 'data' array cout
<< "The data you entered: ";
cout.write(data, 20); // Writes 20 characters from 'data' to the output cout
<< endl;
return 0;
}
```

Output:

```
Enter up to 20 characters: gcwmaddur The
data you entered: gcwmaddur
```

FORMATTED-I/O

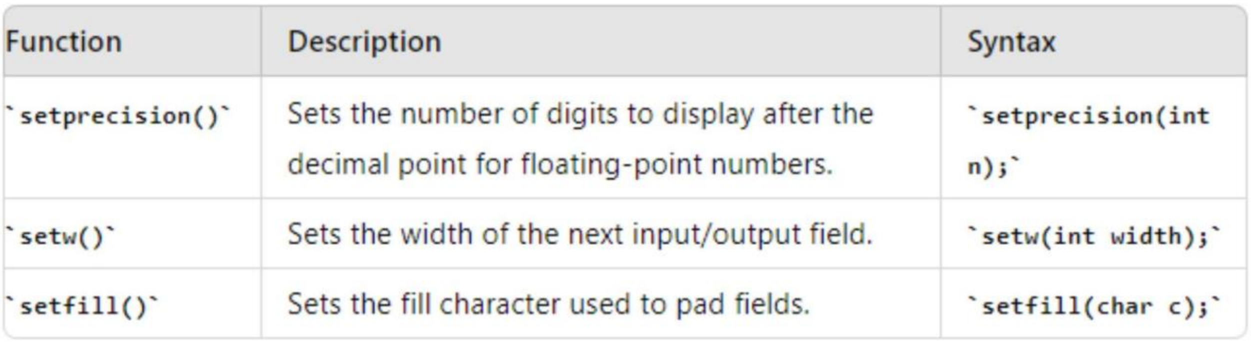
• **Definition:** Input- and output-operations where data is formatted according to specific rules or styles before being written to or read from a stream.

Purpose

- To present data in a readable and structured format.
- To ensure data is aligned, padded, or displayed in a particular way according to user requirements.

Usage

- Formatted-I/O functions are used with output streams (cout) to control how data is formatted when displayed.
- They can be applied to both numbers and text to ensure consistent and readable output.
- Common formatted-I/O functions are listed in below table:



Function	Description	Syntax
`setprecision()`	Sets the number of digits to display after the decimal point for floating-point numbers.	`setprecision(int n);`
`setw()`	Sets the width of the next input/output field.	`setw(int width);`
`setfill()`	Sets the fill character used to pad fields.	`setfill(char c);`

• **Description:** This function sets the number of digits to display after the decimal point for floating-point numbers in the output stream.

• **Parameters:**

`int n`: The number of digits to display

- **Example Usage:**

cout << setprecision(3) << 3.14159; // Outputs: 3.14

setw(int width)

- **Description:** This function sets the width of the next input/output field. If the data is shorter than the specified width, it will be padded with spaces or another character (set by `setfill()`).
- **Parameters:**

- `int width`: The width of the field for the next input/output operation.

- **Example Usage:**

cout << setw(10) << 123; // Outputs: " 123" (7 spaces before 123)

setfill(char c)

- **Description:** This function sets the fill character used to pad fields in the output stream. It is used in conjunction with `setw()` to pad the output with a specific character instead of spaces.
- **Parameters:**

`char c`: The character used to pad the output field.

- **Example Usage:**

cout << setfill('*') << setw(10) << 1234; // Outputs:"1234" (padded with *)

Example Program: Demonstrating Formatted-I/O

```
#include <iostream>
#include <iomanip> // For setprecision, setw, and setfill using
namespace std;
int main() {
// Setting precision to 3 digits after the decimal point
cout << setprecision(3) << 3.14159 << endl; // Outputs: 3.14
// Setting width to 10 and displaying the number 123
cout << setw(10) << 123 << endl; // O/Ps: " 123" (7 spaces before 123)
// Setting fill character to '*' and width to 10, then displaying the number 1234 cout
<< setfill('*') << setw(10) << 1234 << endl; // Outputs: "******1234" return 0;
}
```

Output:

```
3.14
123
******1234
```

Explanation:

- **Precision Example:** `setprecision(3)` formats the floating-point number `3.14159` to show only 3 digits after the decimal point, resulting in `3.14`.
- **Width Example:** `setw(10)` sets the width of the field to 10 characters. Since the integer `123` only takes up 3 characters, 7 spaces are added before it.
- **Padding with Fill Character Example:** setfill('*') << setw(10) << 1234 sets the width of the output field to 10 characters and uses the asterisk (*) as the padding character.

BUILT-IN CLASSES FOR I/O

Definition

• Built-in Classes for I/O handle input and output-operations.
• These classes simplify file and console I/O.

Features

• **Stream-Based I/O:** I/O-classes use streams to manage data flow.
This provides a consistent way to handle data from different sources like the console or files.

• **Overloaded Operators:** I/O-classes support overloaded operators.
The `<<` operator is used for output, and the `>>` operator is used for input. This makes it easy to format and handle data.

• Common built-in classes are listed in below table:
1) `iostream`
2) `istream`
3) `ostream`

1) `iostream`

• **Description:** The Base-class for input and output-stream classes.
• It provides functionalities for both input and output-operations.
• Derived-classes:
`istream`: For input-operations.
`ostream`: For output-operations.

2) `istream`

• **Description:** A class for input-stream operations.
• It is used for reading data from input-sources like the keyboard or files.
• Common Functions:
`>>` (extraction-operator) for reading formatted data.
`getline()` for reading lines of text.
`eof()` to check if the end of the input-stream is reached.

3) `ostream`

• **Description:** A class for output-stream operations.
• It is used for writing data to output-destinations like the console or files.
• Common Functions:
`<<` (insertion-operator) for writing formatted data.
`flush()` to flush the output buffer.
`endl` for inserting a newline and flushing the stream. **Example Program:** Demonstrating istream and ostream #include

```
<iostream>
int main() { int
number;
cout << "Enter an integer: "; // Using ostream to print to console cin >>
number; // Using istream to read from keyboard
cout << "You entered: " << number << endl; return 0;
  }
```

Output:

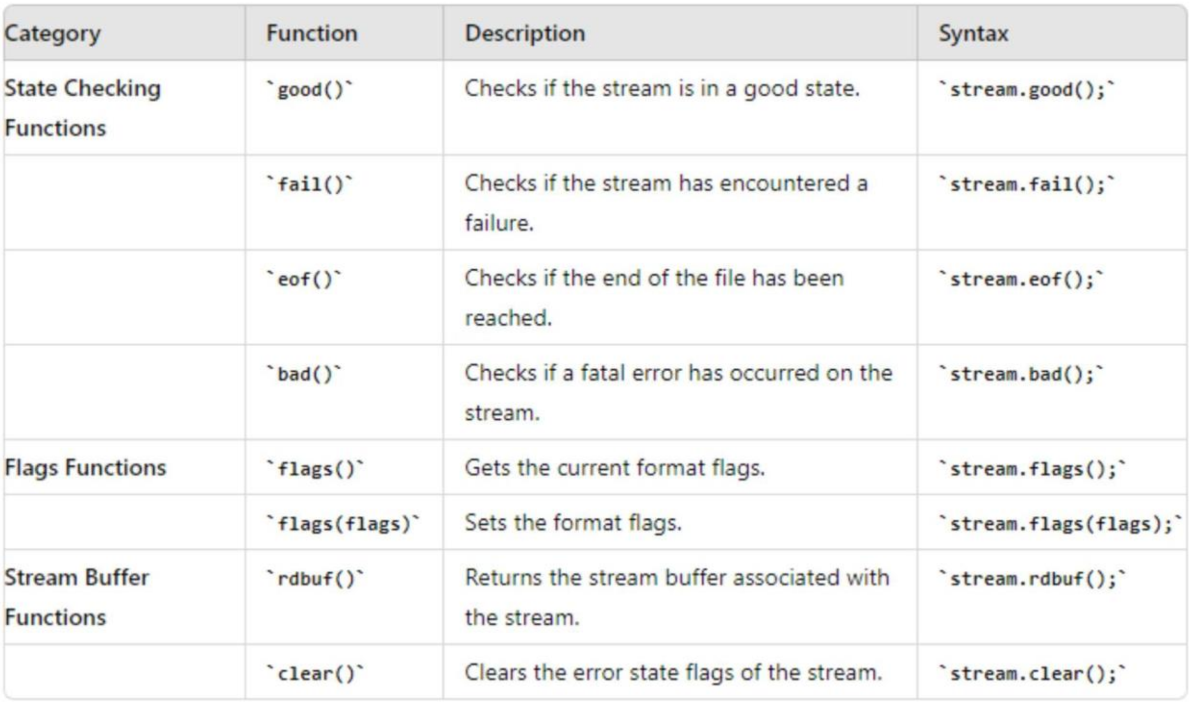
```
Enter an integer: 42
```

```
You entered: 42
```

`ios` CLASS FUNCTIONS AND FLAGS

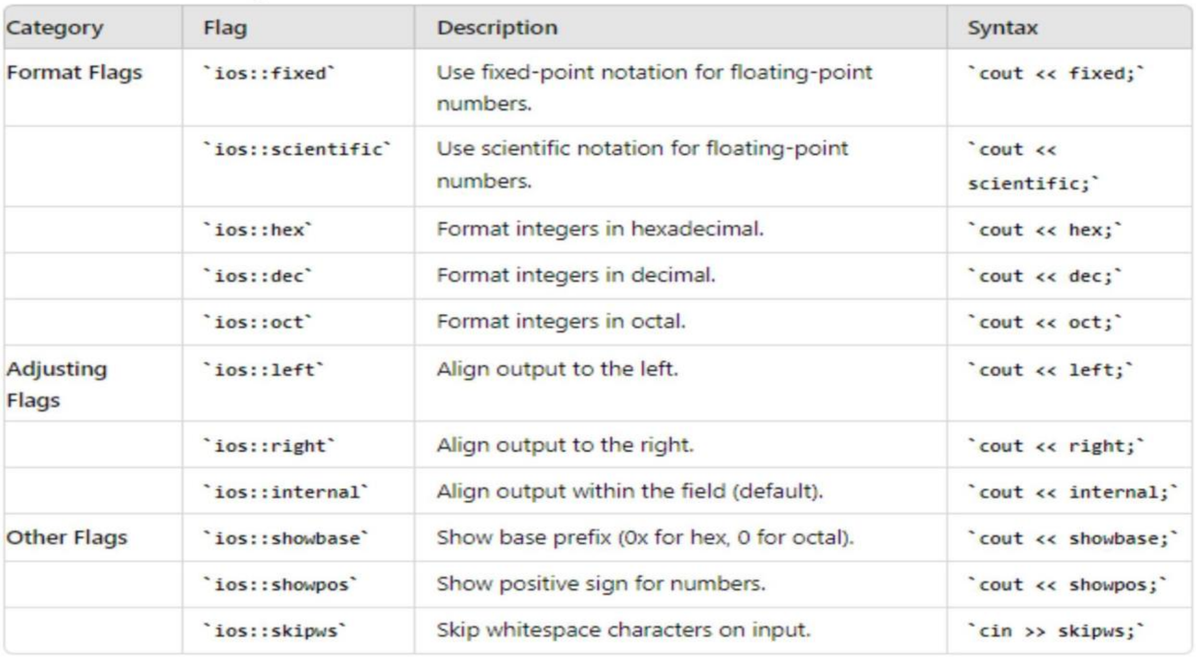
- **Definition:** The `ios` class is a Base-class for streams like `istream`, `ostream`, and `fstream`.
- It provides functions and flags to manage stream operations.

Table: ios Class Functions



Category	Function	Description	Syntax
State Checking Functions	`good()`	Checks if the stream is in a good state.	`stream.good();`
	`fail()`	Checks if the stream has encountered a failure.	`stream.fail();`
	`eof()`	Checks if the end of the file has been reached.	`stream.eof();`
	`bad()`	Checks if a fatal error has occurred on the stream.	`stream.bad();`
Flags Functions	`flags()`	Gets the current format flags.	`stream.flags();`
	`flags(flags)`	Sets the format flags.	`stream.flags(flags);`
Stream Buffer Functions	`rdbuf()`	Returns the stream buffer associated with the stream.	`stream.rdbuf();`
	`clear()`	Clears the error state flags of the stream.	`stream.clear();`

Table: ios Class Flags



Category	Flag	Description	Syntax
Format Flags	`ios::fixed`	Use fixed-point notation for floating-point numbers.	`cout << fixed;`
	`ios::scientific`	Use scientific notation for floating-point numbers.	`cout << scientific;`
	`ios::hex`	Format integers in hexadecimal.	`cout << hex;`
	`ios::dec`	Format integers in decimal.	`cout << dec;`
	`ios::oct`	Format integers in octal.	`cout << oct;`
Adjusting Flags	`ios::left`	Align output to the left.	`cout << left;`
	`ios::right`	Align output to the right.	`cout << right;`
	`ios::internal`	Align output within the field (default).	`cout << internal;`
Other Flags	`ios::showbase`	Show base prefix (0x for hex, 0 for octal).	`cout << showbase;`
	`ios::showpos`	Show positive sign for numbers.	`cout << showpos;`
	`ios::skipws`	Skip whitespace characters on input.	`cin >> skipws;`

Example Program: Demonstrating ios` Class Functions and Flags

```
#include <iostream>
#include <iomanip> // For formatting functions like setw, setprecision, etc. int
main() {
// Example of ios class functions and flags
// Stream state checking
cout << "Stream state checks:" << endl; if
(cout.good()) {
cout << "Stream is good." << endl;
  }
  if (cout.fail()) {
cout << "Stream has failed." << endl;
  }
if (cout.eof()) {
cout << "End of file has been reached." << endl;
  }
// Using flags
cout << "Hexadecimal and octal outputs:" << endl;
// Use hexadecimal formatting
cout << hex << "Hex: " << 255 << endl;
// Use additional flags
cout << "Additional flags:" << endl;
  // Show base prefix
  cout << showbase;
cout << "Hex with base: " << hex << 255 << endl;
// Show positive sign
cout << showpos;
cout << "Positive sign: " << 123 << endl; return 0;
  }
```

Output:

```
Stream state checks:
Stream is good.
Hexadecimal and octal outputs:
Hex: ff Additional
flags:
  Hex with base: 0xff
  Positive sign: +123
```

Explanation:

1) Stream State Checking:

• Checks the state of the stream with `good()`, `fail()`, and `eof()` methods.

2) Using Flags:

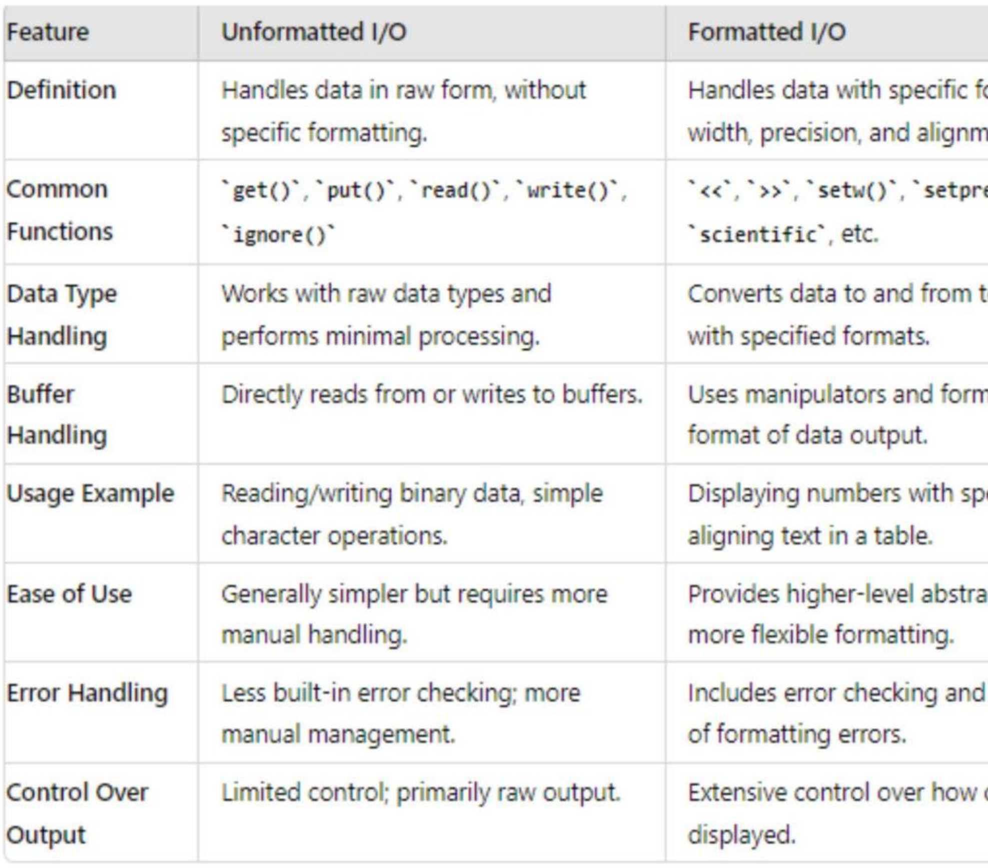
• `hex` formats numbers in hexadecimal.
• `oct` formats numbers in octal.
• `dec` resets the formatting to decimal.
• `showbase` displays the base prefix for hexadecimal (0x) and octal (0).

- `showpos` displays the positive sign for numbers.

3) Skipping Whitespace:

- Demonstrates `skipws`, though it's more relevant for input-streams. In the given output, it shows how spaces are managed (not impactful in this specific context).

UNFORMATTED-I/O VS. FORMATTED-I/O



Feature	Unformatted I/O	Formatted I/O
Definition	Handles data in raw form, without specific formatting.	Handles data with specific fo width, precision, and alignm
Common Functions	`get()`, `put()`, `read()`, `write()`, `ignore()`	`<<`, `>>`, `setw()`, `setpre `scientific`, etc.
Data Type Handling	Works with raw data types and performs minimal processing.	Converts data to and from t with specified formats.
Buffer Handling	Directly reads from or writes to buffers.	Uses manipulators and form format of data output.
Usage Example	Reading/writing binary data, simple character operations.	Displaying numbers with spe aligning text in a table.
Ease of Use	Generally simpler but requires more manual handling.	Provides higher-level abstra more flexible formatting.
Error Handling	Less built-in error checking; more manual management.	Includes error checking and of formatting errors.
Control Over Output	Limited control; primarily raw output.	Extensive control over how displayed.

UNIT 3
USER DEFINED DATA TYPES

Class and Objects

In C++, classes and objects are the basic building block that leads to Object-Oriented programming in C++.

What is a Class in C++?

A class is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range,* etc. So here, the Car is the class, and wheels, speed limits, and mileage are their properties.

- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables together, these data members and member functions define the properties and behaviour of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage,* etc, and member functions can be *applying brakes*, *increasing speed,* etc.

But we cannot use the class as it is. We first have to create an object of the class to use its features. An Object is an instance of a Class.

Note: When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

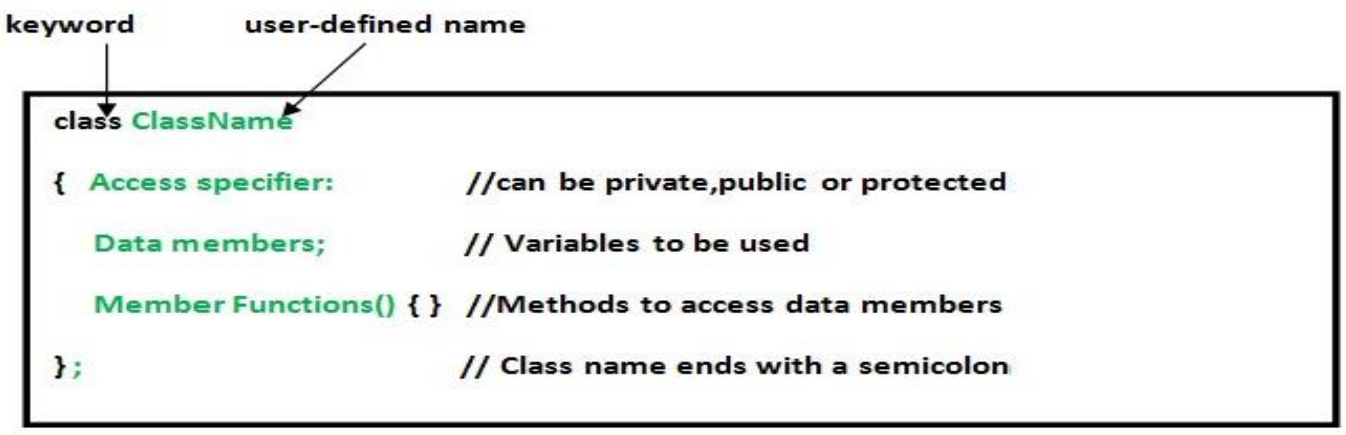
Defining Class in C++

A class is defined in C++ using the keyword class followed by the name of the class. The following is the syntax:

```
class ClassName {
    access_specifier:
    // Body of the class
};
```

Here, the access specifier defines the level of access to the class’s data members. Example

```
class ThisClass { public:
    int var;     // data member
    void print() {          // member method
        cout << "Hello";
     }
};
```



```
class ClassName
{  Access specifier:          //can be private,public or protected
   Data members;              // Variables to be used
   Member Functions() { }     //Methods to access data members
};                            // Class name ends with a semicolon
```

What is an Object in C++?

When a class is defined, only the specification for the object is defined; no memory or storage is allocated.
To use the data and access functions defined in the class, you need to create objects. Syntax
to Create an Object
We can create an object of the given class in the same way we declare the variables of any other inbuilt data type.

```
ClassName ObjectName;
```

Example

```
MyClass obj;
```

In the above statement, the object of MyClass with name obj is created.
Accessing Data Members and Member Functions
The data members and member functions of the class can be accessed using the dot('.') operator with the object. For example, if the name of the object is *obj* and you want to access the member function with the name *printName()* then you will have to write:

obj.printName()

Example of Class and Object in C++

The below program shows how to define a simple class and how to create an object of it.

```
// C++ program to illustrate how create a simple class and
// object
#include <iostream>
#include <string>

using namespace std;

// Define a class named 'Person'
class Person {
public:
    // Data members string
    name;
    int age;

    // Member function to introduce the person
    void introduce()
    {
        cout << "Hi, my name is " << name << " and I am "
             << age << " years old." << endl;
    }
};

int main()
{
    // Create an object of the Person class Person
    person1;

    // accessing data members
    person1.name = "Alice";
    person1.age = 30;

    // Call the introduce member method
    person1.introduce();

    return 0;
}
```

Output

```
Hi, my name is Alice and I am 30 years old.
```

Access Modifiers

In C++ classes, we can control the access to the members of the class using Access Specifiers. Also known as access modifier, they are the keywords that are specified in the class and all the members of the class under that access specifier will have particular access level.

In C++, there are 3 access specifiers that are as follows:

1. Public: Members declared as public can be accessed from outside the class.
2. Private: Members declared as private can only be accessed within the class itself.
3. Protected: Members declared as protected can be accessed within the class and by derived classes.

If we do not specify the access specifier, the private specifier is applied to every member by default.

Example of Access Specifiers

// C++ program to demonstrate accessing of data members

```
#include <iostream>
using namespace std;
class Geeks {  private:
    string geekname;
    // Access specifier
public:
    // Member Functions()
    void setName(string name) { geekname = name; }
    void printname() { cout << "Geekname is:" << geekname; }
};
int main()
{
    // Declare an object of class geeks Geeks
    obj1;
    // accessing data member
    // cannot do it like: obj1.geekname = "Abhi";
    obj1.setName("Abhi");
    // accessing member function
    obj1.printname();
    return 0;
}
```

Member Function in C++ Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

Till now, we have defined the member function inside the class, but we can also define the member function outside the class. To define a member function outside the class definition,

- We have to first declare the function prototype in the class definition.
- Then we have to use the scope resolution:: operator along with the class name and function name.

Example

```
// C++ program to demonstrate member function
// definition outside class
#include <iostream> using
namespace std; class Geeks
{
public:
    string geekname;
    int id;

    // printname is not defined inside class definition void
    printname();

    // printid is defined inside class definition  void
    printid() { cout << "Geek id is: " << id; }
};

// Definition of printname using scope resolution operator
// ::
void Geeks::printname()
{
    cout << "Geekname is: " << geekname;
}
int main()
```

```
{

    Geeks obj1;
    obj1.geekname = "xyz";
    obj1.id = 15;

    //  call  printname()
    obj1.printname();
    cout << endl;

    //  call  printid()
    obj1.printid();
    return 0;
}
```

Output

```
Geekname is: xyz Geek

id is: 15
```

Note that all the member functions defined inside the class definition are by default inline, but you can also make any non-class function inline by using the keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calls is reduced.

Note: Declaring a friend function is a way to give private access to a non-member function.

<u>The this pointer</u>

Each object declared has its own copy of data members. But note that there is only one copy of each class member function available for all the class objects. Each member function has a pointer which holds the address of the object itself and it is called the pointer.

<u>Points to be remember regarding this pointer</u>

- The this pointer is a built in pointer.
- This pointer points to the object being processed. That is, using this pointer any class member function can find out the address of the class object for which the member function is called.
- The this pointer is also used to access the data member of the object.
- The this pointer can be treated as just like an ordinary pointer to an object.
- Since, it is a pointer to arrow operator (->) is used as member access specifier.
- The this pointer is more useful in returning values from member functions and overloaded operators.

Write a program in C++ to illustrate the this pointer in accessing the data member of an object.

```
#include<conio.h>
#include<iostream.h>
class number
  {
   private:  int x;
   public:
            void setValue(int y )
            {
                  This-> x = y;        // this pointer
            }

           void  display( )
            {
                cout<<"\n\t x : "<<this->x <<endl; cout<<"\n\t
                address of the object :"<<this;
              }

       }; // end of the class

           void main()
               {
                     number  n1, n2; clrscr();
                     n1.setValue(30);
                     n2.setValue(50);
                     n1.display( );
                      n2.display( );
                getch( );
            }  // end of main function
Output              x = 30          Address of the object : 0x8fa9fff4 X
                     = 50           Address of the object ; 0x8fa9fff2
```

Friend functions

So far we have seen that only member functions can access private data of a class. This is the concept of encapsulation and data hiding. However, there are situations where such rigid discrimination leads to considerable inconvenience. There will be times when you want a function to have access to the private member of a class without that function actually being a member of that class. Towards this end, C++ supports friend function.

```
class className{
…………………………
friend returnType functionName(arguments);
…………………….
}
```

A friend is not a member of a class but still has access to its private elements. A friend is defined as a regular, non-member function. However, inside the class declaration for which it will be a friend, its prototype is also included, prefaced by the keyword friend. Friend functions are useful in the following situations.

- Operator overloading and creation of I/O operations
- There will be times when you want one function to have access to the private members of two or more different classes.

Remember the following points while using friend functions:

- The keyword friend is used to declare friend functions
- A friend function is not in the scope of the class to which it has been declared as friend.
- A friend function is called just like a normal C++ function.
- A friend function can be declared either in the public section of the class.
- Usually a friend function has the object as its arguments.
- A friend function can not access the class members directly. An object name followed by dot operator, followed by the individual data member is specified for valid access.

Write a program in C++ to find out the biggest of two numbers using friend function.

```
#include <iostream.h>
#include <conio.h> class
bigtwo
  {
    private:      int  a, b;
    public:        void input( )
                    {
                          cout<<"\n\t input the two numbers to read:"; cin>>a>>b;
                    }
           friend void biggest(bigtwo b);    // friend function declaration
```

```
        };
                void  biggest(bigtwo x)
                  {
                   if(x.a>x.b)
                         cout<<"\n\t a is big";
                  else
                         cout<<"\n\t b is big";
                  }
                    void main()
                       {
                            bigtwo b;
                            clrscr();
                            b.input();
                            biggest(b);
                       }
           Output:       Input the two numbers to read:  55  -89 a is
                                 big
```

Constructors

It would be simpler and more concise to initialize an object when it is first created. Java supports a special type of method, called a constructor, which enables an object to initialize itself. A constructor is a special member function whose task is to initialize the objects of its class, when it is created. Thus is also known as automatic initialization of objects.

Let us consider our RectAngle class again. We can now replace the InputData( ) method by a constructor method as shown below:

Example:

```
class rectangle
     {
          int length;
          int width;
        rectangle(int x, int y)
            {
             length = x;
                width  = y;
            }
                void rectArea( )
                   {
                      int area = length * width;
                   cout << “ area is :”<< area;
           }
   }// end of class


   void main()
```

```
{
  rectangle rect1(10 , 35);
  rect1. rectArea( ); getch();
}
```

```
Output:            area is :350
```

The constructor functions have some special characteristics.

- Constructor has the same name as the class itself.
- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Constructor can be overloaded.
- A constructor can not declared as virtual.
- The programmer has no direct control over the constructor functions.

TYPES OF CONSTRUCTORS

There are various types of constructors. They are as follows:

- Overloaded constructors
- Parameterized constructors
- Default Copy constructors

1. Overloaded constructor

A class may contain multiple constructors. The concept of multiple constructors in a class provides more flexibility. And, all these constructors are nothing but overloaded constructor.

Example: Write a program in C++ to compute the area of the room using overloaded constructor. #include

```
<iostream.h>
#include <conio.h>
class  Room
  {
   private:     float length;
                  float breadth;  public:
                Room (float x, float y)
                 {
                   length = x;
```

```
                    breadth = y;
                  }



                 Room (float x)
                  {
                   length = breadth = x;
                  }
                void room_area ( )
                  {
                    float area = length * breadth;
                   Cout << “\n\t The area of the room is :”<<area;
                   }
  }; // end of class
 void main()
  {
    Room  R1(20.5), R2(12.75, 15.0);
    clrscr();
    cout<< “\n\t THE AREA OF THE ROOM OBJEC T R1 IS  “;
    R1.area();
    cout<< “\n\t THE AREA OF THE ROOM OBJEC T R2 IS  “;
    R2.area();
   getch();
  } // end of main
```

```
     Output:                    THE AREA OF THE ROOM OBJEC T R1 IS
                              The area of the room is:  420.250000
                                 THE AREA OF THE ROOM OBJEC T R2 IS
                             The area of the room is: 191.250000
```

2. <u>Parameterized constructors</u>

It is possible to pass arguments to a constructor function. The constructor that can take arguments are called parameterized constructor. To allow this, simply add the appropriate parameters to the constructor function’s declaration and definition. Then, when you declare an object, specify the arguments.

Example: Write a program in C++ to compute the sum of natural numbers using concept of passing argument to the constructor.

```
#include <iostream.h>
#include <conio.h> class
SumNumber
  {
   private : int sum; public
   :
                SumNumber(int n )
```

```
                              {
                                   sum =0;
                                   for ( int =1; i<=n; i++)
                                       sum+=I;
                               }
                        void display( )
                         {
                           cout<<"\n\t THE SUM OF NATURAL NUMBERS IS :"<<sum;
                         }
               }; //end of class
void main( )
        {
          SumNumber  s(10);  // parameterized  constructor clrscr();
          s.display();
          getch( );
        }
output:       THE SUM OF NATURAL SUM IS : 55
```

3. <u>Default Copy Constructor</u>

One of the more important forms of an overload constructor is the copy constructor. A copy constructor is one which constructs another object by copying the members of a given object. This process is carried out by the compiler. Hence, the name default copy constructor. The process of initialization (actually copying) of one object from another object belong to the same class is performed on member by member basis. Therefore, we will call it as member wise initialization.

Example: Write a program in C++ to illustrate concept of copy constructor.

```
#include <iostream.h>
class number
  {
   private: int  x;
   public:
                      number( )  {  }        // default constructor
                      number(int n)   // constructor with argument
                        {
                          x = n;
                        }
                      number( number &obj)   //default copy constructor
                        {
                          x = obj.x;
                           cout<<"\n\t copy constructor is invoked ";
                        }
            void display( )
                {
                  cout<<"\n\t the value of x is :"<< x;
            }
    }; // end of main
          void main( )
            {
              clrscr( );
              number  n1(234);
              number n2(n1);
              n1.display( );
              n2.display( );
              getch( );
            }
```

Output:

```
Copy constructor is invoked The
value of x is: 345
```

Destructors

A destructor is just opposite to initialization. A destructor is a special member function which is also having the same name of the class but prefixed with the tilde (~) symbol. A destructor automatically frees up the memory resources used by the objects.

Characteristics of a destructor

- A destructor is invoked automatically by the compiler upon exit from the program and cleans up the memory that is no longer needed.
- A destructor does not return any value.
- A destructor can not be declared as static, const or volatile.
- A destructor does not accept arguments and there fore it can not be overloaded.
- A destructor must be declared in public section of the class.
- If the constructor uses the new expression to allocate memory, then the destructor should use the delete expression.

Example: Write a program in C++ to illustrate concept of destructor.

```
#include <iostream.h>
#include <conio.h> class
number
  {
   private: int  x;
   public:
                  number( )       // default constructor
                          {
                            cout<<"\n\t copy constructor is invoked "; x =
                             50;
                          }
```

```
                    ~ number( )      // destructor
                      {
                        cout<< “\n\t destructor is invoked “;
                      }
          void display( )
            {
              cout<<”\n\t the value of x is :” << x;
            }
      }; // end of main

          void main( )
            {
              clrscr( );
              number  N;                          output:     constructor invoked
              N.display( );                                          the value of x is : 50
              number  M;                                    constructor invoked
              getch( );                                              destructor is invoked
            }                                                        destructor is invoked
```

UNIT 4

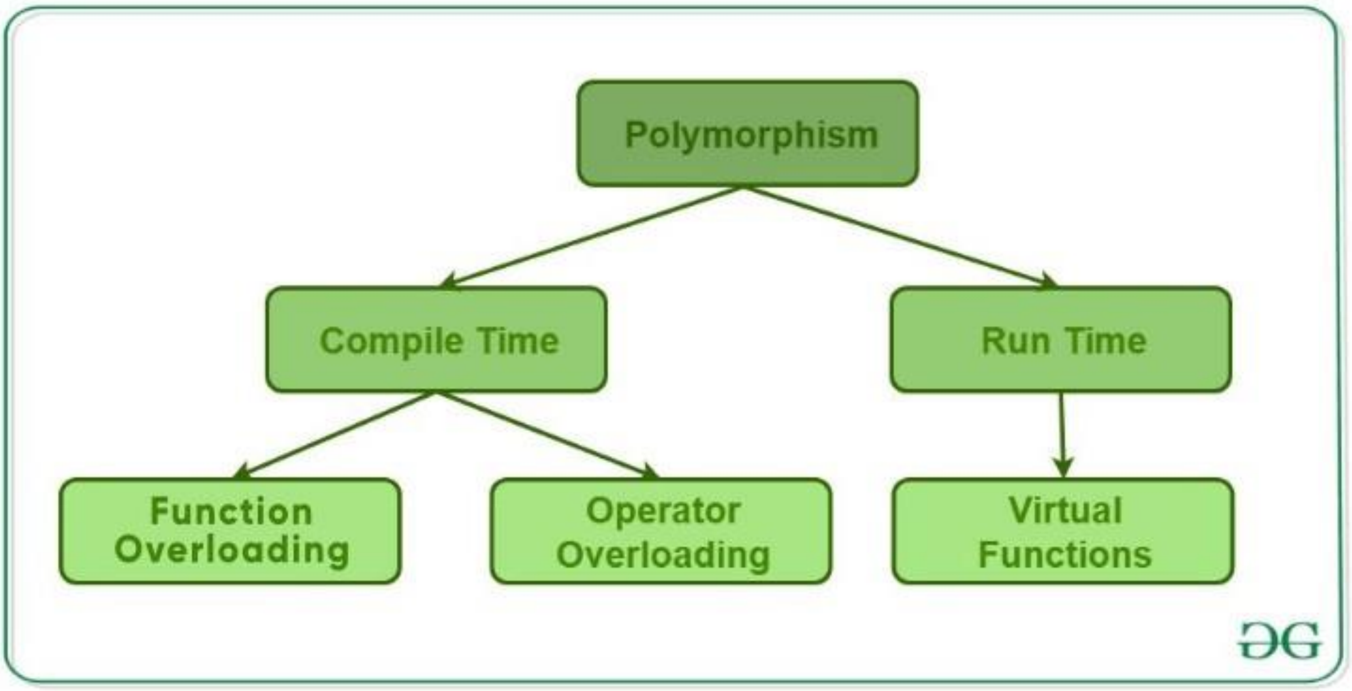
POLYMORPHISM AND INHERITENCE

POLYMORPHISM

The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism

- **Compile-time Polymorphism**
- **Runtime Polymorphism**



1. Compile-Time Polymorphism

This type of polymorphism is achieved by function overloading or operator overloading.

A. Function Overloading

When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded,** hence this is known as Function Overloading. Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments**. In simple terms, it is a feature of object-oriented programming providing many functions that have the same name but distinct parameters when numerous tasks are listed under one function name. There are certain Rules of Function Overloading that should be followed while overloading a function.

```
// C++ program to demonstrate
// function overloading or
// Compile-time Polymorphism #include
<iostream>
using namespace std; class
Geeks {
```

```
public:
    // Function with 1 int parameter void
    func(int x)
    {
        cout << "value of x is " << x << endl;
    }

    // Function with same name but
    // 1 double parameter
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }

    // Function with same name and
    // 2 int parameters void
    func(int x, int y)
    {
        cout << "value of x and y is " << x << ", " << y
             << endl;
    }
};

// Driver code int
main()
{
    Geeks obj1;

    // Function being called depends
    // on the parameters passed
    // func() is called with int value
    obj1.func(7);

    // func() is called with double value
    obj1.func(9.132);

    // func() is called with 2 int values
    obj1.func(85, 64);
    return 0;
}
```

Output

```
value of x is 7 value

of x is 9.132

value of x and y is 85, 64
```

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. For example, we can make use of the addition operator (+) for string class to concatenate two strings.

We know that the task of this operator is to add two operands. So a single operator ‘+’, when placed between integer operands, adds them and when placed between string operands, concatenates them.

```
// C++ program to demonstrate
// Operator Overloading or
// Compile-Time Polymorphism #include
<iostream>
using namespace std;

class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }

    // This is automatically called
    // when '+' is used with between
    // two Complex objects
    Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real; res.imag
        = imag + obj.imag; return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

// Driver code int
main()
{
    Complex c1(10, 5), c2(2, 4);

    // An example call to "operator+"
    Complex c3 = c1 + c2; c3.print();
}
```

Output

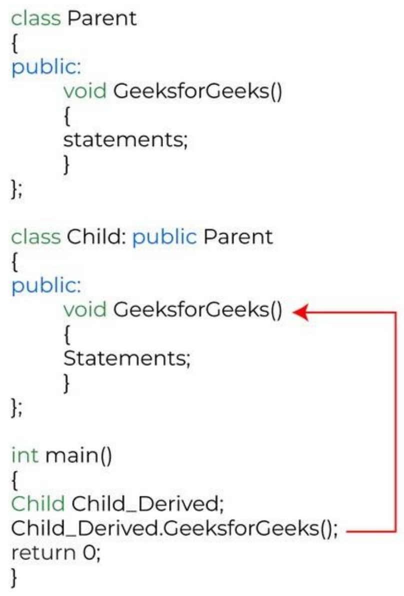
```
12 + i9
```

2. Runtime Polymorphism

This type of polymorphism is achieved by **Function Overriding**. Late binding and dynamic polymorphism are other names for runtime polymorphism. The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

A. Function Overriding

Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.



```
class Parent
{
public:
        void GeeksforGeeks()
        {
        statements;
        }
};

class Child: public Parent
{
public:
        void GeeksforGeeks()
        {
        Statements;
        }
};

int main()
{
Child Child_Derived;
Child_Derived.GeeksforGeeks();
return 0;
}
```

Function overriding Explanation

Runtime Polymorphism with Data Members

Runtime Polymorphism cannot be achieved by data members in C++. Let’s see an example where we are accessing the field by reference variable of parent class which refers to the instance of the derived class.

```
// C++ program for function overriding with data members #include
<bits/stdc++.h>
using namespace std;

//  base class declaration.
class Animal {
public:
    string color = "Black";
};

// inheriting Animal class. class
Dog : public Animal { public:
    string color = "Grey";
};

// Driver code int
main(void)
{
    Animal d = Dog(); // accessing the field by reference
                      // variable which refers to derived cout
    << d.color;
```

} **Output**

```
Black
```

B. Virtual Function

A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.

Some Key Points About Virtual Functions:

- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword “**virtual**” inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime

```
// C++ Program to demonstrate
// the Virtual Function
#include <iostream> using
namespace std;

// Declaring a Base class
class GFG_Base {

public:
    // virtual function
    virtual void display()
    {
        cout << "Called virtual Base Class function"
             << "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Base print function"
             << "\n\n";
    }
};

// Declaring a Child Class
class GFG_Child : public GFG_Base {

public:
    void display()
    {
        cout << "Called GFG_Child Display Function"
             << "\n\n";
    }
```

```
void print()
```

```
    {
        cout << "Called GFG_Child print Function"
             << "\n\n";
    }
};

int main()
{
    // Create a reference of class GFG_Base GFG_Base*
    base;

    GFG_Child child;

    base = &child;

    // This will call the virtual function
    base->display();

    // This will call the non-virtual function
    base->print();
}
```

```
Called GFG_Child Display Function


Called GFG_Base print function
```

Operator overloading

Operator overloading is the ability to tell the compiler how to perform a certain operation when its corresponding operator is used on one or more variables. Operator overloading is one of the most exciting features of C++. It can transform complex, obscure program listing into intuitively obvious ones. For example, if you need complex arithmetic, matrix algebra, logic signals or character strings in C++, you can use classes to represent these notations. **C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.**

Defining operators for such classes sometimes allows a programmer to provide a more conventional and convenient notations for manipulating objects that could be achieved using only the basic functional notation. By overloading operators, we can give additional meaning to operators like +, -, *, /, >, <, >=, =<,
++, -- etc., When an operator is overloaded, that operator loses none of its original meaning. Instead, it gains additional meaning relative to the class for which it is defined.

OPERATOR OVERLOADING FUNCTION

To overload an operator, you create an operator function. Most often an operator function is a member or a friend of the class for which it is defined. However, there is a slight difference between a member function and a friend operator function. A friend function will have one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. The general form of a member operator function is as follows:

```
return_type   operator # (argument-list)
 {
        ….     // operation to be performed
        ….
 }
```

WHERE,

return_type of an operator function often the class for which it is defined.

operator : is the keyword.

# : The operator being overloaded is substituted for # symbol. For example **+** operator.
Argument list : it is vary depending upon how the operator function is implemented and the

type of the operator being overloaded.

Rules for overloading operators:

- The precedence of the operator cannot be changed
- The number of operands that an operator takes cannot be altered. For example, you cannot overload the / (slash) operator so that it takes only one operand.
- The only operators that you cannot overload are . :: .* :? Preprocessors
- Only existing operators can be overloaded. New operators can not be created.
- The overloaded operator must have at least one operand that is of user-defined type.
- We can not change the meaning of an operator. That is, we can not redefine the (+) plus operator to subtract one value from the other.
- Overloaded operators follow the syntax rules of the original operators. That cannot be overridden.
- A friend function will have one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators.
- When using binary operators overloaded through a member function, the left-hand operators and must be an object of the relevant class.
- Binary arithmetic operators such as +, -,* and / must explicitly return a value. They must not attempt to change their own arguments.

Program: Write a C++ program to demonstrate unary (++&--) operator overloading.

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

```
{
private: int x;
```

```
                  public:   void read()
                                  {
                                   cout<<"\n\t input the values to read:";
                                   cin>>x;
                                  }
                          void write()
                          {
                                   cout<<"\n\t The given number  is:"<<x;
                          }
                void  operator ++( )
                        {
                          ++x;
                        }
                 void operator - - ( )
                       {
                       - -x;
                       }
                  };

           void main()
                  {
                       incre_decrement e;
                       clrscr();
                       e.read();
                       e.write();
                       ++e;
                       cout<<"\n\t After unary operator (++) overloading :" ;
                       - -e;
                       - -e;
                       cout<<"\n\t After unary operator (- -) overloading :" ;
                       e.write();
                       getch();
                  }
```

```
Output :            Input the values to read: 45
                           The given number  is: 45
                           After unary operator (++) overloading :
                           The given number  is: 46
                           After unary operator (- - ) overloading :
                           The given number  is: 44
```

Program: Write a C++ program to demonstrate binary arithmetic (+ - * /) operator overloading.

```
 #include<conio.h>
#include<iostream.h> class
complex
 {
     private: float  i, r;
         public:
                   void input()
```

{

```
                                    cout<<"\n\t input the value of real part:";
                                    cin>>r;
                                    cout<<"\n\t input the value of imaginary part:";
                                    cin>>i;
                                   }
                              void output()
                               {    cout<<"\n\n\t the complex no. is : ";
                                    cout<<"\t"<<r<<"\t+i"<<i;
                                }
                          complex  operator+ (complex c2)
                              {
                                   complex c3;
                                     c3.r=r+c2.r;
                                      c3.i=i+c2.i;   return(c3);
                              }
                        complex  operator- (complex c2)
                              {        complex c3;
                                       c3.r = r - c2.r;
                                       c3.i = i- c2.i;      return(c3);
                               }
                       complex  operator* (complex c2)
                             {     complex c3;
                                     c3.r=(r*c2.r-i*c2.i);
                                     c3.i=(r*c2.i+i*c2.r);       return(c3);
                             }
                       complex operator/ (complex c2)
                            {        complex c3;
                                       int d=(r*c2.r+i*c2.i);
                                      c3.r=(r*c2.r+i*c2.i)/(float) d; c3.i=(i*c2.r-
                                      r*c2.i)/(float) d;  return(c3);
                            }
                  }; // end of class definition
void main()
  {

  complex c1,c2,c3;
   clrscr();
     cout<<"\n\t INPUT & DISPLAY COMPLEX NO. OF OBJECT C1:\n"
                      c1.input();
                      c1.output();

     cout<<"\n\t INPUT & DISPLAY COMPLEX NO. OF OBJECT C2:\n"
                      c2.input();
                      c2.output();
     cout<<"\n\t The sum of 2 complex number is as follows:";
```

```
c3=c1+c2;
```

```
                c3.output();
   cout<<"\n\t The difference of 2 complex number is as follows:";
               c3=c1-c2;
               c3.output();
  cout<<"\n\t The multiplication of 2 complex no. is as follows:"; c3=c1*c2;
              c3.output();
  cout<<"\n\t The division of 2 complex number is as follows:"; c3=c1/c2;
               c3.output();
}
```

Output:

```
INPUT & DISPLAY COMPLEX NO. OF OBJECT C1:
          Input the value of real part:  6

          Input the value of imaginary part:  7 The

          complex no. is : 6+i7

          INPUT & DISPLAY COMPLEX NO. OF OBJECT C2:

          Input the value of real part:  8

          Input the value of imaginary part:  5 The

          complex no. is : 8+i5

          The sum of 2 complex number is as follows: The

          complex no. is : 14+i12

          The difference of 2 complex number is as follows: The

          complex no. is : -2+i2

          The multiplication of 2 complex no. is as follows: The

          complex no. is :  13+i91

          The division of 2 complex number is as follows: The

          complex no. is :  1+i0.313253
```

INHERITANCE

Inheritance is the most powerful feature of OOP's. In C++ it is implemented by creating the new classes from the existing class. Here, we can take the form of the existing class and then add code to it, without modifying the existing class i.e**., inheritance is the property by which we can derive a new class from an existing class. The new class is called as the derived class and the existing class is called as the base class.** The new class not only retains the properties of the base class from which it has been derived but also adds the properties of its own. The base class thus remains unchanged in the process. The inheritance allows subclasses to inherit all the variables and functions of their parent classes.

Advantages of inheritance

% **Code reusability:** It permits code reusability. Once a base class is written and debugged, it need not checked again, but nevertheless be adapted to work in different situations. Reusing code saves the time and money and increases a program's reliability. A programmer can use a class created by another person and without modifying it, derive other classes from it that are suited to particular situations.

% **Data abstraction:** The derived class can add data and function members to the class without affecting the base class behavior. This is called data abstraction.

% **Data protection:** The protection can be increased in the derived classes, but cannot be decreased, that is – a class item declared as protected in a class (base) cannot be made public later in the derived classes.

% **Making class library:** The creation of a class library makes the usage of a certain class easier to a new programmer, without having to bother about the source code. This also allows the programmer to adapt the class to various situations.

Base class and derived class

As already defined the existing class using which a new class is created is called as the base class. In other words the old class is known as **base class or super class or parent class** and the newly created class is called as the derived class or child class or sub class.

Defining a subclass

The general form of the derived class is given below:

```
class derived_class_name : scope_of_class base_class_name

{

        Statements of derived class

};
```

The scope_of_class is optional, when used may contain either private or public. The default scope_of_class is always private. **When the word private is used then the public members of the base class become the private members of the derived class, therefore the public members of the base class can be accessed by only the accessing functions of the derived class.**

When the word public is used then the public members of the base become the public members of the derived class, therefore the public members of the base class can be accessed by the data members and accessing functions of the derived class.

}

Visibility modifiers

We have seen that the variable and functions of a class are visible every where in the program. However, it may be necessary in some situations to restrict the access to certain variables and functions from outside the class. For example, we may not like the objects of a class directly alter the value of a variable or access functions. We can achieve this in C++ by applying visibility modifiers to the instance variables and functions. The visibility modifiers are also known as **access modifiers.**

C++ provides private, public and protected access modifiers. The data members of the protected section are accessible by the member functions within the existing class and any class immediately derived from it. However, any function which is creating outside the base class and the derived class cannot access the data members. The following table highlights the scope of all the modes of a class.

Basc class scope	Derived class scope	
	Public derivation	***Private derivation***
private	***not inherited***	***not inherited***
public	***Public***	***private***
protected	***Protected***	***protected***

When the protected member is derived in the public mode, it becomes the protected to the derived class also and therefore is accessible by the member functions of the derived class.

When the protected member is derived in the private mode, it becomes the private to the derived class also and therefore is not accessible for further inheritance. The keywords **public**, **protected** and **private** may appear in any order in the declaration of a class.

```
class sample
 {
        private:
                …       // optional
                …       // visible only to member functions
                …       // not visible outside the class
        protected :
                …       // visible to member functions
                …       // also visible to member functions of derived class
        public :
                …       // visible to all functions in the program
 }
```

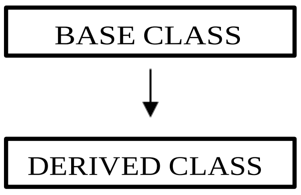
The following points highlights the concepts of inheritance

- O A derived class can inherit all the features of the base class, add its own feature and create a new class. The change made will not affect the base class.
- O The derived class can share the properties from only one class, more than one class or more than one level.
- O Protected data members behave like private data members for both the base class and the derived class.
- O Private members of the base class are not accessible in the derived class.
- O The programs can declare objects of both the base and derived classes. Both the independent of one another.

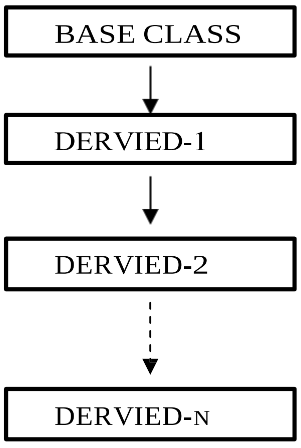
Inheritance can be classified in 5 forms:

- Single inheritance
- Multilevel inheritance
- Hierarchical inheritance
- Multiple inheritance
- Hybrid inheritance

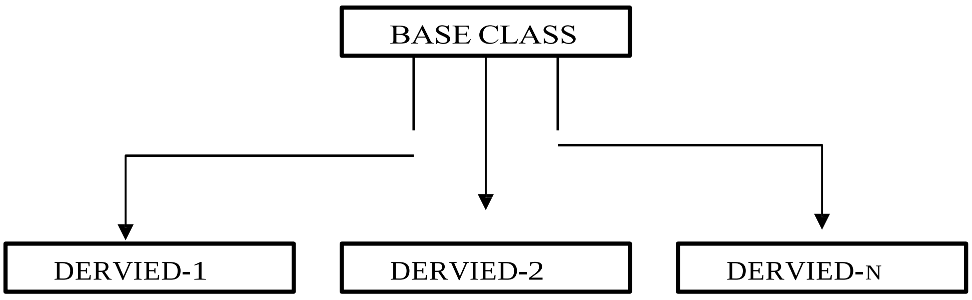
Single inheritance: A derived class with only one base class is called single inheritance. In other words, single inheritance is a mechanism of deriving a new class from the old class. There is a one-to-one relationship between the base class and the derived class.



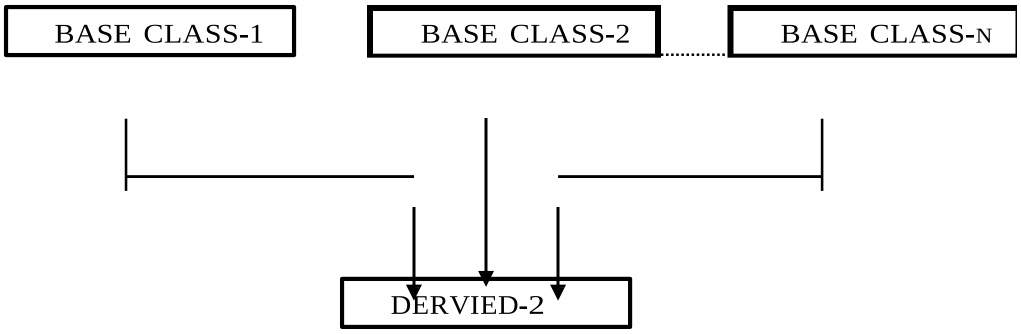
Multilevel inheritance: A class can be derived from another derive class which is known as multilevel inheritance. In other words, multilevel inheritance is an extension of derivation of in single inheritance level by level.



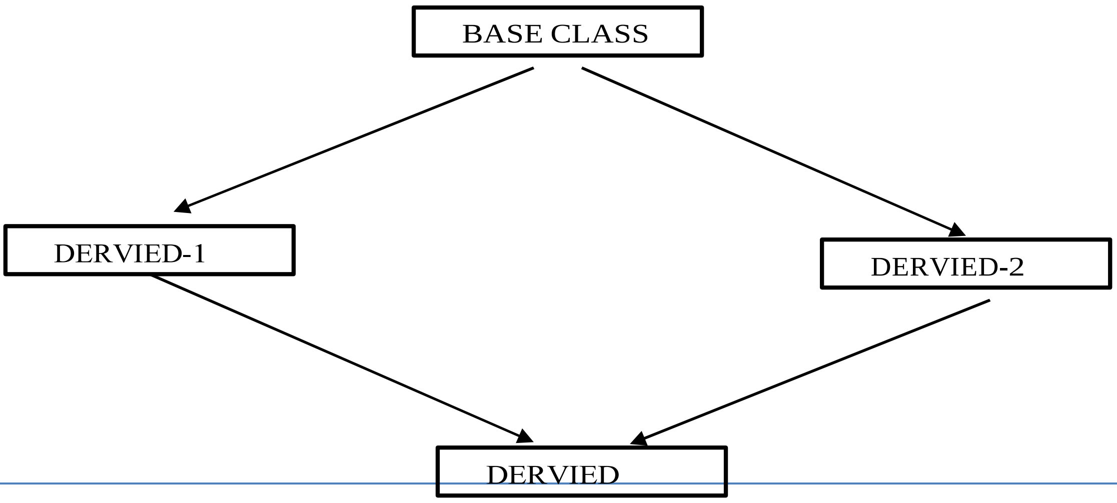
Hierarchical inheritance: When the properties of one class are inherited by more than one class which is known as Hierarchical inheritance. In other words, Hierarchical inheritance is a one in which create more than one derived class from a base class.



Multiple inheritance: A class can inherit properties from more than one class which is known as multiple inheritance. In other words, multiple inheritance is a mechanism where more than one base class is inherited into a single derived class.



Hybrid inheritance: The hybrid inheritance, as the name itself implies that, is a combination of one or more forms of inheritance. It may be combination of hierarchical and multi-level inheritance or it may be combination of hierarchical and multiple inheritance etc.,



PROGRAM: Write a program to demonstrate single inheritance.

```
#include<conio.h> #include<iostream.h>
  class room                      // BASE CLASS
    {
      protected: int  l, b;
      public:
                void input()
                  {
                     cout<<"\n\t input the length and breadth :";
                     cin>>l>>b;
                  }

                 void output()
                  {
                     cout<<"\n\tLENGTH :"<< l<<"\BREADTH:"<< b;
                  }
   };

   class bed_room: public room                    // DERIVED CLASS
       {
       private: int h;
       public:
                void input()
                  {
                        room::input();
                         cout<<"\n\t input the value of height :";
                          cin>>h;
                   }
                 void output()
                   {
                       room::output();
                       cout<<"\n\t HEIGHT :"<< h
                   }

            void compute()
              {
                    int volume=l*b*h;
                    cout<<"\n\t The volume of the bed-room is:"<<volume;
              }
    };

          void main()
             {
                bed_room d;
                clrscr();
                d.input();
                d.output();
                d.compute();
                getch();
            }
```

```
OUTPUT:         Input the length and breadth: 8     9
                      LENGTH : 8    BREADTH :  9

                      Input the value of height : 10

                      HEIGHT : 10

                      The volume of the bed-room is: 720
```

PROGRAM: Write a program to demonstrate hierarchical inheritance.

```
#include<conio.h> #include<iostream.h>
  class room                        // BASE CLASS
    {
      protected: int  l, b;
      public:
              void input()
                {
                   cout<<"\n\t Input the value LENGTH and BREADTH :";
                   cin>>l>>b;
                }
               void output()
                {
                   cout<<"\n\tLENGTH :"<< l<<"\BREADTH:"<< b;
                }
   };
   class bed_room: public room                    //DERIVED CLASS-1
         {
         private: int h;
         public:
              void input()
                {
                     room::input();
                       cout<<"\n\t input the value of HEIGHT:";
                        cin>>h;
                  }
                void output()
                  {
                    room::output();
                     cout<<h;
                    }
           void compute()
             {
                    int volume=l*b*h;
                    cout<<"\n\t The volume of the bed-room is:"<<volume;
             }
   };
  class bath_room: public room                    // DERIVED CLASS-2
        {
        private: int h;
        public:
                  void input()
                    {
                          room::input();
                            cout<<"\n\t input the value of HEIGHT:";
                             cin>>h;
                     }

                    void output()
                      {
                     room::output();
                      cout<<h;
                     }

           void compute()
             {
                    int volume=l*b*h;
                    cout<<"\n\t the volume of the bath-room is:"<<volume;
             }
   };
         void main()
            {
                bed_room    d;
                bath_room   b;
                clrscr();

                d.input();
                d.output();
                d.compute();

                b.input();
                b.output();
```

```
b.compute();
```

```
        getch();
    }
```

```
OUTPUT:           Input the length and breadth: 8      9
                         LENGTH : 8     BREADTH :  9
                         Input the value of height : 10
                         HEIGHT : 10
                         The volume of the bed-room is: 720
                         Input the length and breadth: 5      4
                         LENGTH : 5     BREADTH :  4
                         Input the value of height : 6
                         HEIGHT : 6
                         The volume of the bath-room is: 120
```

PROGRAM: Write a program to demonstrate hybrid inheritance.

```
#include<iostream.h>
#include<conio.h> class
student
   {

   private:
            char name[20]; int
            roll_no;
            int sem;

   public:
           void input()
             {
                 cout<<"\n\t Input the name of the student :"; cin>>name;
                 cout<<"\n\t Input the roll-number of a student :";
                 cin>>roll_no;
                 cout<<"\n\t Input which semester he studying :"; cin>>sem;
             }

          void output()
            {
            cout<<"\n\t The name is:"<<name;
            cout<<"\n\t The rollno is:"<<roll_no;
            cout<<"\n\t The semister is:"<<sem;
            }
   };
```

```
class  exam : public student
  {
  protected: int m1,m2,m3;
  public:
             void input()
               {
                   student::input();
                     cout<<"\n\t Input the marks of 3 subjects :";
                      cin>>m1>>m2>>m3;
               }

           void output()
            {
               student::output();
               cout<<"\n\t The marks of 3 subject is :"<<m1<<"\t"<<m2<<"\t"<<m3;
            }
  };

  class sports
        {
        protected:  int sports_marks;
        public:

          void input()
             {
                 cout<<"\n\t Input the value sports marks :";
                 cin>>sports_marks;
             }
            void output()
            {
            cout<<"\n\t The sports marks is :"<<sports_marks;
            }
        };
        class  award : public  exam, public sports
          {
              private:int total;
                          float avg;
           public:
                       void input()
                          {
                                 exam::input();
                                 sports::input();
                          }
                      void calculate()
```

{

```
                         total=m1+m2+m3+sports_marks; avg=total/4;
                      }
                    void output()
                     {
                        exam :: output();
                        sports :: output();
                        cout<<"\n\t The total of all subject marks is:"<<total;
                        cout<<"\n\t The average of all subject marks is :"<<avg;
                     }
              };
              void main()
               {
                 award a;
                  clrscr();
                  a.input();
                  a.calculate();
                  a.output();
                getch();
               }
```

```
Output          Input the name of the student : Barth Kumar
                             Input the roll-number of a student : S4465620
                             Input which semester he studying : Third Input
                             the marks of 3 subjects : 50  60  70 Input the
                             value sports marks : 80
                             The name is:  Barth Kumar
                             The rollno is: S4465620 The
                             semister is:  Third
                             The marks of 3 subject is : 50   60   70 The
                             sports marks is : 80
                             The total of all subject marks is: 260
                             The average of all subject marks : 65.000000
```

PROGRAM: Write a program to demonstrate multiple inheritance.

```
#include<conio.h> #include<iosteam.h>
   class father              // BASE CLASS-1
    {
    private:      char name[20];
                  char desg[20]; float
                  salary;
    public:
        void input()
          {
                  cout<<"\n\t Input the name and desg of father:"; cin>>name>>desg;
                  cout<<"\n\t Input the value of salary :"; cin>>salary;
          }

         void output()
           {
             cout<<"\n\t The name and the designation is:"<<name<<"\t"<<desg; cout<<"\n\t The
             salary is :"<<salary;
           }
      }; // end of base class FATHER

class mother                  // BASE CLASS-2
    {
    private:     char name[20];
                   char desg[20];
    public:

          void input()
           {
                  cout<<"\n\t Input the name and designation of mother :";
                  cin>>name>>desg;
           }

    void output()
      {
       cout<<"\n\t the name and the designation of mother is:"<<name<<"\t"<<desg;
      }
   };  //end of base class MOTHER

   class son :: public father, public mother          // DERIVED CLASS
    {
    private:  char name[20];
                  int age;
    public:
         void input()
          {
                  father :: input();
                  mother : : input();
                  cout<<"\n\t input the name and age of a son :"; cin>>name>>age;
          }
          void output()
           {
                father :: output();
                mother : : output();
               cout<<"\n\t The name and age of the son is:"<<name<<"\t"<<age;
           }
      };

          void main()
             {
               son   s;
               clrscr();
                a.input();
                a.output();
                getch();
             }
```

Output

```
Input the name and designation of father: Gopal Krishna   Lecturer

Input the value of salary : 15750.000000

The name and the designation is: Gopal Krishna Lecturer The

salary is : 15750.000000

Input the name and designation of mother : Radha Mani House Wife The

name and the designation of mother is: Radha Mani House Wife Input the

name and age of a son : Radha Krishna

The name and age of the son is: Radha Krishna
```

UNIT 4

FILE-HANDLING

INTRODUCTION TO FILE-HANDLING

• **Definition:** Files are named-locations on a storage-medium where data can be stored & retrieved by programs.
• Typically, the storage-medium is a disk.

FILE-HANDLING

• **Definition:** File-handling refers to the management and manipulation of files stored on a storage-device.

Purpose

• **Data Storage**: Enables programs to store large amounts of data permanently.
• **Data Retrieval**: Facilitates retrieving stored-data for processing or display.
• **Data Manipulation**: Supports operations such as updating existing data, appending new data, and deleting data.
• **File Management**: Includes file-operations such as opening, reading, writing, and closing files.

FILE STREAM-CLASS

• **Definition:** File-stream-classes are specialized classes used for performing input and output operations on files.
• File-stream-classes include
ifstream (Input File Stream)
ofstream (Output File Stream)
fstream (File Stream)

ifstream

• **Purpose:** Used to read data from files.
• **Functionality:** It allows opening a file and reading its contents, operating in input mode by default.
• **Example Usage:**

```
ifstream inFile("example.txt");
string line;
while (getline(inFile, line)) { cout
<< line << endl;
 }
inFile.close();
```

ofstream

• **Purpose:** Used to write data to files.
• **Functionality:** It allows creating a file or opening an existing file to write data to it, operating in output mode by default.
• **Example Usage:**

```
ofstream outFile("example.txt"); outFile <<
"Hello, World!" << endl; outFile.close();
```

fstream

• **Purpose:** Combines both input and output file-operations.

• **Functionality:** It allows both reading from and writing to files and can be used in input, output, or both modes.

• **Example Usage:**

```
fstream file("example.txt", ios::in | ios::out); string
line;
while (getline(file, line)) { cout
<< line << endl;
 }
file << "New line added." << endl;
file.close();
```

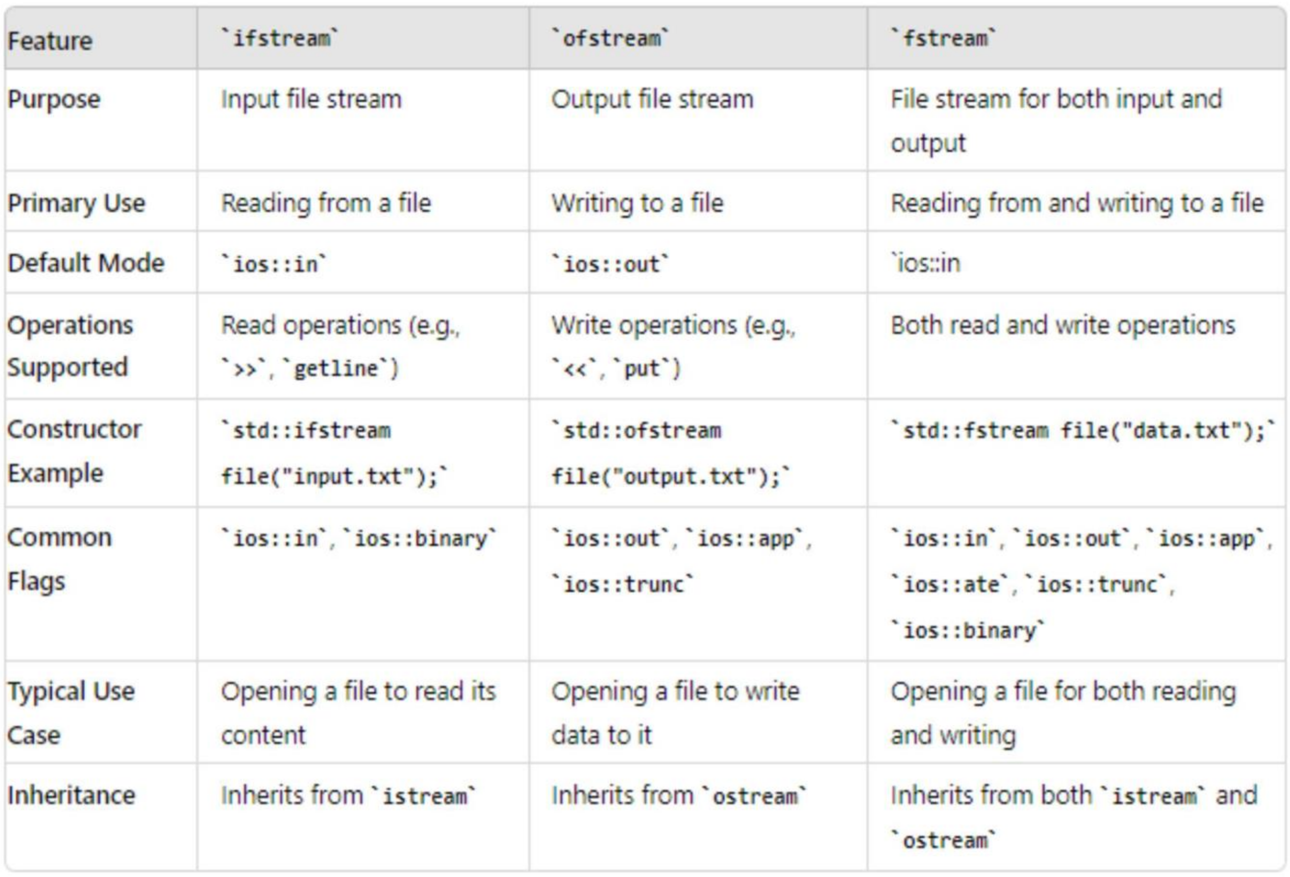
Example Program: Demonstrating ifstream and ofstream

```
 #include <iostream>
#include <fstream>
#include <string> using
namespace std; int main()
{
// Step 1: Write to the file using ofstream
ofstream outFile("example.txt");   outFile <<
"Hello, World!" << endl; outFile.close();
cout << "Data written to file successfully." << endl;
// Step 2: Read from the file using ifstream
 ifstream inFile("example.txt");
 string line;
while (getline(inFile, line)) {
cout << "Read from file: " << line << endl;
 }
inFile.close(); return
0;
 }
```

Output:

```
Data written to file successfully.
Read from file: Hello, World!
```

COMPARISON OF ifstream, ofstream AND fstream



Feature	`ifstream`	`ofstream`	`fstream`
Purpose	Input file stream	Output file stream	File stream for both input and output
Primary Use	Reading from a file	Writing to a file	Reading from and writing to a file
Default Mode	`ios::in`	`ios::out`	`ios::in
Operations Supported	Read operations (e.g., `>>`, `getline`)	Write operations (e.g., `<<`, `put`)	Both read and write operations
Constructor Example	`std::ifstream file("input.txt");`	`std::ofstream file("output.txt");`	`std::fstream file("data.txt");`
Common Flags	`ios::in`, `ios::binary`	`ios::out`, `ios::app`, `ios::trunc`	`ios::in`, `ios::out`, `ios::app`, `ios::ate`, `ios::trunc`, `ios::binary`
Typical Use Case	Opening a file to read its content	Opening a file to write data to it	Opening a file for both reading and writing
Inheritance	Inherits from `istream`	Inherits from `ostream`	Inherits from both `istream` and `ostream`

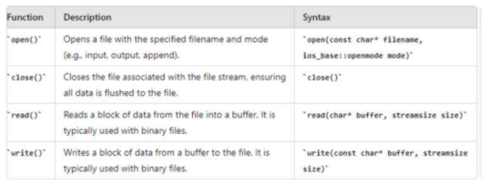
STANDARD CLASS-FUNCTIONS FOR FILE I/O

Definition

• These are functions provided by the file stream-classes to handle file-operations.
• The file stream-classes include `ifstream`, `ofstream`, `fstream`

Purpose

• To manage file input and output with built-in classes that handle file stream operations.
• To enable the reading from and writing to files in a structured and efficient manner.
• Common Standard Class-functions for File I/O are listed in below table:



Function	Description	Syntax
`open()`	Opens a file with the specified filename and mode (e.g., input, output, append).	`open(const char* filename, ios_base::openmode mode)`
`close()`	Closes the file associated with the file stream, ensuring all data is flushed to the file.	`close()`
`read()`	Reads a block of data from the file into a buffer. It is typically used with binary files.	`read(char* buffer, streamsize size)`
`write()`	Writes a block of data from a buffer to the file. It is typically used with binary files.	`write(const char* buffer, streamsize size)`

`open(const char* filename, ios_base::openmode mode)`

• **Description:** This function opens a file for reading, writing, or both.

• **Parameters:**

`const char* filename`: The name of the file to be opened.
`ios_base::openmode mode`: The mode in which the file should be opened e.g., `ios::in`, `ios::out`, `ios::app`, `ios::binary`

• **Example Usage**:

```
ofstream outFile;
outFile.open("example.txt", ios::out); // Opens the file "example.txt" for writing
```

`close()`

• **Description:** This function closes the file that was opened.
Closing a file ensures that all data is properly saved and frees system-resources.

• **Parameters:** None.

• **Example Usage**:

```
outFile.close(); // Closes the file after writing
```

`read(char* buffer, streamsize size)`

• **Description:** This function reads a block-of-data from the file into a buffer.
This function is typically used for reading binary-data.

• **Parameters:**

`char* buffer`: The memory-location where the read data will be stored.
`streamsize size`: The number of bytes to read from the file.

• **Example Usage**:

```
ifstream inFile("data.bin", ios::binary);
 inFile.read(data, sizeof(data)); // Reads 'sizeof(data)' bytes from the file into 'data'
```

`write(const char* buffer, streamsize size)`

• **Description:** This function writes a block-of-data from a buffer to a file.
This function is typically used for writing binary-data.

• **Parameters:**

`const char* buffer`: The memory-location containing the data to be written.
`streamsize size`: The no. of bytes to write to the file.

• **Example Usage**:

```
outFile.write(reinterpret_cast<const char*>(&number), sizeof(number));
// Writes 'sizeof(number)' bytes from 'number' to the file
```

Example Program: Demonstrating open(), close(), read() and write() #include <iostream>

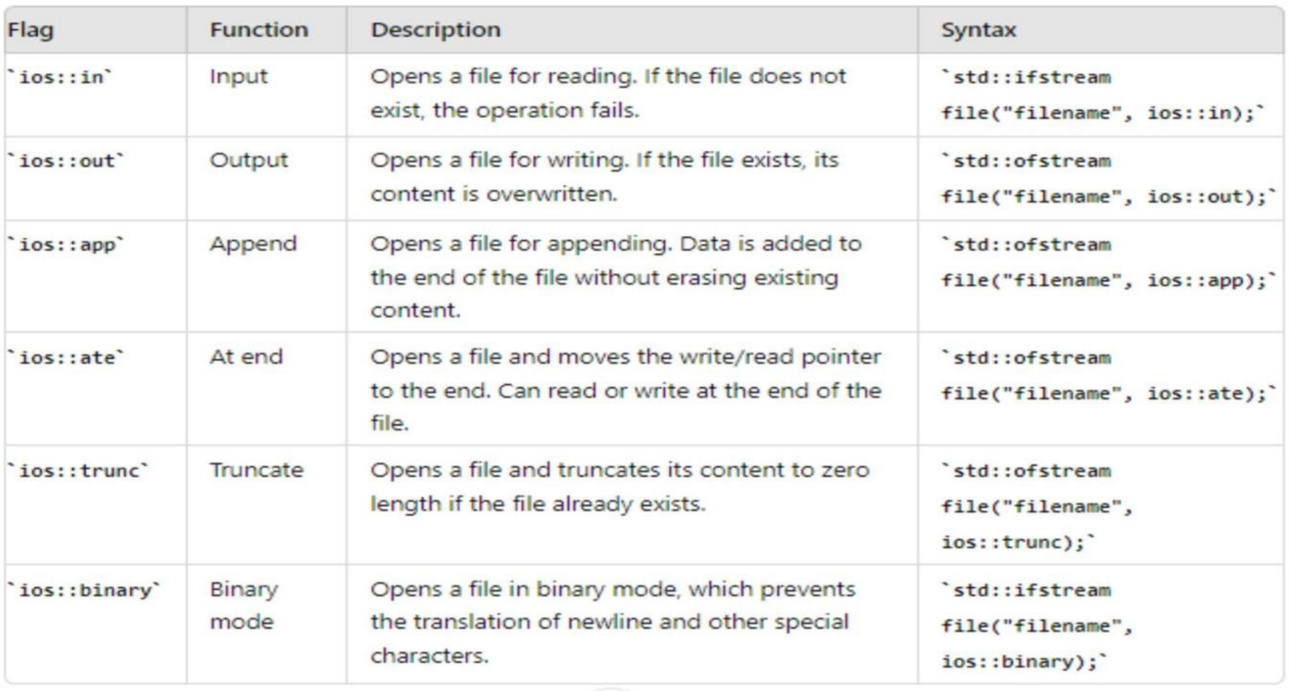
```
#include <fstream> // Include the fstream library for file I/O using
namespace std;
int main() {
// Part 1: Writing to a text file
ofstream outFile; // Create an ofstream object for writing to a file
outFile.open("example.txt", ios::out); // Opens the file "example.txt" for writing if
(outFile.is_open()) {
outFile << "Hello, this is a sample text file." << endl; // Write some text to the file outFile.close();
// Closes the file after writing
cout << "Text file written successfully." << endl;
} else {
cout << "Failed to open the file for writing." << endl;
 }
// Part 2: Reading from a binary file
ifstream inFile("data.bin",ios::binary);// Create ifstream object for reading from binary file
char data[100]; // Buffer to store the read data if
(inFile.is_open()) {
inFile.read(data, sizeof(data)); // Reads 'sizeof(data)' bytes from the file into 'data' inFile.close();
// Closes the file after reading
cout << "Binary file read successfully." << endl;
} else {
cout << "Failed to open the binary file for reading." << endl;
 }
// Part 3: Writing a number to a binary file
int number = 12345; // Example number to write to the file
outFile.open("data.bin", ios::binary | ios::out); // Opens file "data.bin" for binary writing if
(outFile.is_open()) {
outFile.write(reinterpret_cast<const char*>(&number), sizeof(number));
// Writes 'sizeof(number)' bytes from 'number' to the file
outFile.close(); // Closes the file after writing
cout << "Number written to binary file successfully." << endl;
} else {
cout << "Failed to open the binary file for writing." << endl;
 }
return 0;
 }
```

Output:

```
Text file written successfully.
Binary file read successfully.
Number written to binary file successfully.
```

FILE OPENING-MODES

- File opening-modes determine how a file is accessed and modified during fileoperations.
- They are specified when opening a file using file stream-classes (`ifstream`, `ofstream`, `fstream`).
- Common file opening-modes are listed in below table:



Flag	Function	Description	Syntax
`ios::in`	Input	Opens a file for reading. If the file does not exist, the operation fails.	`std::ifstream file("filename", ios::in);`
`ios::out`	Output	Opens a file for writing. If the file exists, its content is overwritten.	`std::ofstream file("filename", ios::out);`
`ios::app`	Append	Opens a file for appending. Data is added to the end of the file without erasing existing content.	`std::ofstream file("filename", ios::app);`
`ios::ate`	At end	Opens a file and moves the write/read pointer to the end. Can read or write at the end of the file.	`std::ofstream file("filename", ios::ate);`
`ios::trunc`	Truncate	Opens a file and truncates its content to zero length if the file already exists.	`std::ofstream file("filename", ios::trunc);`
`ios::binary`	Binary mode	Opens a file in binary mode, which prevents the translation of newline and other special characters.	`std::ifstream file("filename", ios::binary);`

TYPES OF FILES TEXT-FILES

- **Definition:** Text-files store data as readable-characters and -strings.
- **Features**

- Data is in plain-text, with lines separated by newline-characters.
- Can be edited with any text-editor.
- Used for configuration-files, logs, and simple data-storage.

- **Example Usage:**

```
#include <fstream>
using namespace std;
int main() {
ifstream inFile("example.txt");
// Read from file
inFile.close();
 }
```

BINARY-FILES

- **Definition:** Binary-files store data in a non-readable format, using bytes.
- **Features**

- Data is stored in binary-form, specific to the data-type.
- Cannot be easily edited with text-editors.
- Useful for storing images, executables, and complex data.

- **Example Usage:**

```
#include <fstream>
using namespace std;
int main() {
ofstream outFile("example.bin", ios::binary);
// Write to file
outFile.close();
 }
```