RESEARCH INNOVATION DISCOVERY

RID संस्था समस्या का समाधान

RUN BY TWKSAA WELFARE FOUNDATION

Reg.No: 048884
Foundation Day
05-09-2024

C++ Programming language

E-Book

Er. Rajesh Prasad(B.E, M.E)
Founder: TWF & RID Org

- **RID ORGANIZATION** यानि **Research, Innovation and Discovery** संस्था जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले **NEW (RID, PMS & TLR)** की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW (RID, PMS & TLR)** के माध्यम से किया जाये इसके लिए ही मैं राजेश प्रसाद **इस RID संस्था** की स्थपना किया हूँ |
- Research, Innovation & Discovery में रूचि रखने वाले आप सभी विधार्थियों, शिक्षकों एवं बुधीजिवियो से मैं आवाहनं करता हूँ की आप सभी **इस RID संस्था** से जुड़ें एवं अपने बुध्दि, विवेक एवं प्रतिभा से दुनियां को कुछ नई **(RID, PMS & TLR)** की खोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें |

"C++ प्रोग्रामिंग लैंग्वेज के इस ई-पुस्तक में आप C++ से जुड़ी सभी बुनियादी अवधारणाएँ सीखेंगे। । मुझे आशा है कि इस ई-पुस्तक को पढ़ने के बाद आपके ज्ञान में वृद्धि होगी और आपको कंप्यूटर विज्ञान के बारे में और अधिक जानने में रुचि होगी"

"In this E-Book of C++ Programming language you will learn all the basic concepts related to C++. I hope after reading this E-Book your knowledge will be improve and you will get more interest to know more thing about computer Science".

RID हमें क्यों करना चाहिए

(Research)

अनुसंधान हमें क्यों करना चाहिए ?

Why should we do research?

1. नई ज्ञान की प्राप्ति (Acquisition of new knowledge)
2. समस्याओं का समाधान(To Solving problems)
3. सामाजिक प्रगति (To Social progress)
4. विकास को बढ़ावा देने(To promote development)
5. तकनीकी और व्यापार में उन्नति(To advances in technology & business)
6. देश विज्ञान और प्रौद्योगिकी के विकास(To develop the country's science & technology)

(Innovation)

नवीनीकरण हमें क्यों करना चाहिए ?

Why should we do Innovation?

1. प्रगति के लिए(To progress)
2. परिवर्तन के लिए(For change)
3. उत्पादन में सुधार(To Improvement in production)
4. समाज को लाभ(To Benefit to society)
5. प्रतिस्पर्धा में अग्रणी (To be ahead of competition)
6. देश विज्ञान और प्रौद्योगिकी के विकास। (To develop the country's science & technology)

(Discovery)

खोज हमें क्यों करना चाहिए?

Why should we do Discovery?

1. नए ज्ञान की प्राप्ति(Acquisition of new knowledge)
2. अविष्कारों की खोज(To Discovery of inventions)
3. समस्याओं का समाधान(To Solving problems)
4. ज्ञान के विकास में योगदान(Contribution to development of knowledge)
5. समाज के उन्नति के लिए (for progress of society)
6. देश विज्ञान और तकनीक के विकास (To develop the country's science & technology)

New Technology पर **Research** करने के लिए सम्पर्क करें: **ridorg.in@gmail.com Mob.No: 9202707903**

❖ Research(अनुसंधान):

- अनुसंधान एक प्रणालीकरण कार्य होता है जिसमें विशेष विषय या विषय की नई ज्ञान एवं समझ को प्राप्त करने के लिए सिद्धांतिक जांच और अध्ययन किया जाता है। इसकी प्रक्रिया में डेटा का संग्रह और विश्लेषण, निष्कर्ष निकालना और विशेष क्षेत्र में मौजूदा ज्ञान में योगदान किया जाता है। अनुसंधान के माध्यम से विज्ञान, प्रोधोगिकी, चिकित्सा, सामाजिक विज्ञान, मानविकी, और अन्य क्षेत्रों में विकास किया जाता है। अनुसंधान की प्रक्रिया में अनुसंधान प्रश्न या कल्पनाएँ तैयार की जाती हैं, एक अनुसंधान योजना डिज़ाइन की जाती है, डेटा का संग्रह किया जाता है, विश्लेषण किया जाता है, निष्कर्ष निकाला जाता है और परिणामों को उचित दर्शाने के लिए समाप्ति तक पहुंचाया जाता है।

❖ Innovation(नवीनीकरण): -

- Innovation एक विशेषता या नई विचारधारा की उत्पत्ति या नवीनीकरण है। यह नए और आधुनिक विचारों, तकनीकों, उत्पादों, प्रक्रियाओं, सेवाओं या संगठनात्मक ढंगों का सृजन करने की प्रक्रिया है जिससे समस्याओं का समाधान, प्रतिस्पर्धा में अग्रणी होने, और उपयोगकर्ताओं के अनुकूलता में सुधार किया जा सकता है।

❖ Discovery (आविष्कार):

- Discovery का अर्थ होता है "खोज" या "आविष्कार"। यह एक विशेषता है जो किसी नए ज्ञान, अविष्कार, या तत्व की खोज करने की प्रक्रिया को संदर्भित करता है। खोज विज्ञान, इतिहास, भूगोल, तकनीक, या किसी अन्य क्षेत्र में हो सकती है। इस प्रक्रिया में, व्यक्ति या समूह नए और अज्ञात ज्ञान को खोजकर समझने का प्रयास करते हैं और इससे मानव सभ्यता और विज्ञान-तकनीकी के विकास में योगदान देते हैं।

नोट : अनुसंधान विशेषता या विषय पर नई ज्ञान के प्राप्ति के लिए सिस्टमैटिक अध्ययन है, जबकि आविष्कार नए और अज्ञात ज्ञान की खोज है।

सुविचार:

1.	**समस्याओं का समाधान करने का उतम मार्ग हैं |** → शिक्षा ,RID, प्रतिभा, सहयोग, एकता एवं समाजिक-कार्य
2.	**एक इंसान के लिए जरूरी हैं |** → रोटी, कपड़ा, मकान, शिक्षा, रोजगार, इज्जत और सम्मान
3.	**एक देश के लिए जरूरी हैं |** - → संस्कृति-सभ्यता, भाषा, एकता, आजादी, संविधान एवं अखंडता
4.	**सफलता पाने के लिए होना चाहिए |** → लक्ष्य, त्याग, इच्छा-शक्ति, प्रतिबद्धता, प्रतिभा, एवं सतता
5.	**मरने के बाद इंसान छोडकर जाता हैं |** → शरीर, अन-धन, घर-परिवार, नाम, कर्म एवं विचार
6.	**मरने के बाद इंसान को इस धरती पर याद किया जाता हैं उनके**
	➔ नाम, काम, दान, विचार, सेवा-समपर्ण एवं कर्मो से...

आशीर्वाद (बड़े भैया जी)

Mr. RAMASHANKAR KUMAR

मार्गदर्शन एवं सहयोग

Mr. GAUTAM KUMAR

... सोच है जिनकी नई कुछ कर दिखाने की, खोज है रीड संस्था को उन सभी इंसानों की...

"अगर आप भी **Research, Innovation and Discovery** के क्षेत्र में रूचि रखतें हैं एवं अपनी प्रतिभा से दुनियां को कुछ नया देना चाहतें एवं अपनी समस्या का समाधान **RID** के माध्यम से करना चाहतें हैं तो **RID ORGANIZATION (रीड संस्था)** से जरुर जुड़ें" || धन्यवाद || **Er. Rajesh Prasad (B.E, M.E)**

Index

What is C++?

- C++ is a high-level, general-purpose, case-sensitive, free, versatile, modular, procedural and object-oriented programming language.
- C++ is an extension of the C programming language.
- It was created by Bjarne Stroustrup in the early 1980s at Bell Labs.
- C++ is known for its efficiency, flexibility, and versatility.
- C++ supports both procedural and object-oriented programming paradigms, allowing developers to write code using functions and data structures (procedural) as well as classes and objects (object-oriented).

❖ Use of C++ Programming language:

- There are following use of C++ programming language.
  1) System Software Development
  2) Game Development
  3) Application Software
  4) Embedded Systems
  5) High-Performance Computing (HPC)
  6) Graphics and Multimedia
  7) Financial and Trading Systems
  8) Database Systems
  9) Networking and Communication
  10) Artificial Intelligence and Machine Learning

History of C++:

Bjarne Stroustrup

- **1979-1983:** C++ was developed by Bjarne Stroustrup at Bell Labs as an extension of the C programming language. Stroustrup's aim was to add features for object-oriented programming (OOP) while retaining C's efficiency and flexibility.
- **1985:** The first edition of "The C++ Programming Language" was published, introducing the language to a wider audience and solidifying its syntax and features.
- **1989: C++** 2.0 was released, adding multiple inheritance, abstract classes, and static member functions among other features. This version of C++ laid the groundwork for the object-oriented paradigm that became central to the language.
- **1998:** The ISO/IEC standardized C++ as an international standard known as C++98. This formalized the language and established a baseline for future revisions.
- **2011:** C++11, the next major revision of the language, introduced significant new features such as lambda expressions, auto keyword, and smart pointers, enhancing productivity and safety in C++ programming.
- **2014:** C++14 was released, building upon the foundation laid by C++11 and introducing additional improvements and features, albeit in a more incremental manner.

- **2017:** C++17 was finalized, bringing further enhancements and additions to the language, including constexpr if, structured bindings, and filesystem library.
- **2020:** C++20 was released, introducing concepts, coroutines, ranges, and modules among other features, further modernizing and refining the language.

History of Programming language		
Language	**Year**	**Developed By**
Algol	**1960**	**International Group**
BCPL	**1967**	**Martin Richard**
B	**1970**	**Ken Thompson**
C	**1972**	**Dennis Ritchie**
K & R C	**1978**	**Kernighan & Dennis Ritchie**
C++	**1980**	**Bjarne Stroustrup**

❖ Features of C++ Programming language:

- There are following Features of C++ Programming language.
  1. Simple
  2. Machine Independent or Portable
  3. Structured programming language
  4. Rich Library
  5. Object-Oriented
  6. Compiler based
  7. Reusability
  8. Errors are easily detected
  9. Power and Flexibility
  10. Strongly typed language
  11. Modeling Real-World Problems
  12. Memory Management
  13. Quicker Compilation
  14. Pointers
  15. Recursion

1. **Simple:** C++ offers a straightforward syntax and easy-to-understand structure, making it accessible for beginners.
2. **Machine Independent or Portable:** Programs written in C++ can be compiled and executed on different platforms without modification, enhancing portability.
3. **Structured Programming Language:** C++ supports structured programming techniques, allowing for clear and organized code.
4. **Rich Library:** C++ comes with a comprehensive standard library (STL) that provides ready-to-use functions and data structures for various tasks, speeding up development.
5. **Object-Oriented:** C++ supports object-oriented programming (OOP) principles, including classes, objects, inheritance, and polymorphism, enabling modular and reusable code.

6. **Compiler-based:** C++ requires a compiler to translate human-readable code into machine-executable instructions, ensuring efficient execution.
7. **Reusability:** C++ facilitates code reuse through features like classes, functions, and templates, minimizing redundancy and improving maintainability.
8. **Errors are easily detected:** C++ offers compile-time and runtime error checking, helping developers identify and fix issues early in the development process.
9. **Power and Flexibility:** C++ provides low-level control over hardware resources, allowing for high-performance and flexible programming.
10. **Strongly Typed Language:** C++ enforces strict data type checking at compile time, reducing the risk of runtime errors and enhancing program reliability.
11. **Modular:** C++ allows developers to model real-world entities and their interactions through object-oriented design, facilitating the creation of complex systems.
12. **Memory Management:** C++ offers manual memory management through features like pointers and dynamic memory allocation, giving developers control over memory usage and optimization.
13. **Quicker Compilation:** C++ compilers typically produce fast executable code, resulting in quicker compilation times and improved development efficiency.
14. **Pointers:** C++ supports pointers, which are variables that store memory addresses, enabling direct manipulation of memory and efficient data access.
15. **Recursion:** C++ allows functions to call themselves recursively, making it possible to solve problems more elegantly and efficiently.

❖ C++ installation step by step process:

1. Choose a Compiler:

- Before installing C++, you need to decide which compiler you want to use. Popular choices include GCC (GNU Compiler Collection), Clang, and Visual C++ for Windows.

2. Windows Installation:

- For Windows, you can use Visual Studio, which includes the necessary tools for C++ development. You can download Visual Studio from the official Microsoft website.
- Follow the installation instructions provided by the Visual Studio installer. Make sure to select the C++ workload during installation.

3. macOS Installation:

- macOS typically comes with Clang, which serves as the default compiler for C++. You can use Xcode, Apple's integrated development environment (IDE), for C++ development.
- Install Xcode from the Mac App Store. Once installed, open Xcode and follow the prompts to install the command line tools, which include Clang.

4. Linux Installation:

- Most Linux distributions come with GCC installed by default. However, you can install it manually if necessary. Open a terminal and use your package manager to install GCC. For example, on Ubuntu, you can run: sudo apt-get install build-essential.
- Alternatively, you can install Clang by running: sudo apt-get install clang.

5. Verify Installation:

- Once the installation is complete, you can verify that the compiler is installed correctly by opening a command prompt or terminal and typing gcc --version or clang --version, depending on the compiler you installed. This should display the version information for the compiler.

6. IDE Installation (Optional):

- While you can write and compile C++ code using just a text editor and the command line, using an Integrated Development Environment (IDE) can enhance your development experience. Popular C++ IDEs include Visual Studio (Windows), Xcode (macOS), and **Code::Blocks (cross-platform).**
- Download and install your preferred IDE from the respective official website or package manager.

7. Begin Coding:

- Once everything is set up, you can start writing and compiling C++ code. Open your chosen IDE or text editor, create a new C++ source file (typically with a .cpp extension), write your code, and then compile and run it using the commands provided by your compiler or IDE.

❖step-by-step guides for downloading and installing Code::Blocks, Turbo C++, and Visual Studio Code

❖ Code::Blocks:

1. Download Code::Blocks:

- Go to the official Code::Blocks website: **http://www.codeblocks.org/**
- Navigate to the "Downloads" section.
- Choose the appropriate version for your operating system (Windows, macOS, Linux) and click on the download link.

2. Install Code::Blocks:

- Once the download is complete, run the installer file.
- Follow the on-screen instructions to install Code::Blocks. You can choose the default settings or customize them according to your preferences.
- Complete the installation process.

3. Verify Installation:

- After installation, launch Code::Blocks from the Start menu (Windows) or Applications folder (macOS/Linux). Create a new project or open an existing one to ensure that Code::Blocks is installed correctly.

❖ Turbo C++:

- Turbo C++ is an older IDE primarily used for DOS and Windows XP. However, an updated version called "Embarcadero Dev-C++" is available, which offers a modern development environment.

1. Download Dev-C++:

- Go to the Embarcadero Dev-C++ website: **https://sourceforge.net/projects/orwelldevcpp/**
- Click on the "Download" button to download the installer for Dev-C++.

2. **Install Dev-C++:**
   - Run the downloaded installer file.
   - Follow the installation wizard's instructions to install Dev-C++. You can choose the default settings or customize them as needed.
   - Complete the installation process.
3. **Verify Installation:**
   - Once installed, launch Dev-C++ from the Start menu (Windows).
   - Create a new project or open an existing one to ensure that Dev-C++ is installed correctly.

❖ Visual Studio Code (VS Code):

1. **Download VS Code:**
   - Go to the official Visual Studio Code website: https://code.visualstudio.com/
   - Click on the "Download" button to download the installer for your operating system (Windows, macOS, Linux).
2. **Install VS Code:**
   - Run the downloaded installer file.
   - Follow the on-screen instructions to install Visual Studio Code. You can choose the default settings or customize them according to your preferences.
   - Complete the installation process.
3. **Install C++ Extension:**
   - Open Visual Studio Code after installation.
   - Go to the Extensions view by clicking on the square icon on the left sidebar or pressing Ctrl+Shift+X.
   - Search for "C++" in the Extensions Marketplace.
   - Install the "C/C++" extension provided by Microsoft.
4. **Verify Installation:**
   - Once installed, you can create a new C++ project or open an existing one in Visual Studio Code.
   - Write some C++ code and verify that syntax highlighting and code suggestions are working correctly.

C vs C++

1. Paradigm:

- **C:** C is a procedural programming language, focusing on functions and structured programming.
- **C++:** C++ is a multi-paradigm language, supporting procedural, object-oriented, and generic programming paradigms.

2. Object-Oriented Programming (OOP) Support:

- **C:** C lacks native support for OOP concepts like classes, objects, inheritance, and polymorphism.
- **C++:** C++ was designed to extend C with OOP features, allowing developers to use classes, objects, inheritance, polymorphism, encapsulation, and abstraction.

3. Standard Libraries:

- **C:** Standard C library includes functions for fundamental operations like input/output, string manipulation, memory allocation, etc.
- **C++:** C++ Standard Library builds upon the C Standard Library and includes additional features like containers, algorithms, iterators, streams, and other utilities.

4. Memory Management:

- **C:** In C, memory management is done manually using functions like malloc() and free() for dynamic memory allocation and deallocation.
- **C++:** C++ supports both manual memory management like C and also provides automatic memory management through features like constructors, destructors, and smart pointers (e.g., unique_ptr, shared_ptr).

5. Compatibility:

- **C:** C code can often be used within C++ programs through linkage, but C++ code cannot be directly compiled by a C compiler.
- **C++:** C++ is not entirely backward compatible with C due to the introduction of new keywords and features.

6. Exception Handling:

- **C:** C does not have built-in support for exception handling.
- **C++:** C++ supports exception handling through try-catch blocks, allowing programmers to handle runtime errors gracefully.

7. Namespace Support:

- **C:** C does not support namespaces.
- **C++:** C++ introduces namespaces, providing a way to organize code into logical groups and prevent naming conflicts.

8. Operator Overloading:

- **C:** Operators in C have fixed meanings and cannot be overloaded.
- **C++:** C++ allows operators to be overloaded, enabling custom definitions for operations on user-defined types.

9. Templates:

- **C:** C does not support templates.
- **C++:** C++ supports templates, allowing generic programming and creating functions and classes that work with any data type.

C++ vs Java

1. Language Paradigm:

- **C++** is a multi-paradigm language supporting procedural, object-oriented, and generic programming.
- **Java** is primarily an object-oriented programming language with some support for procedural programming.

2. Memory Management:

- **C++** gives the programmer control over memory management through features like pointers, manual memory allocation, and deallocation (using new and delete operators).
- **Java** has automatic memory management through garbage collection, meaning the programmer does not directly deal with memory allocation and deallocation.

3. Platform Dependency:

- **C++** is platform-dependent, meaning the same code might not run on different platforms without modification.
- **Java** is designed to be platform-independent, running on any platform with a Java Virtual Machine (JVM), which interprets Java bytecode.

4. Compilation vs. Interpretation:

- **C++** code is compiled directly into machine code by a compiler, producing an executable file that runs directly on the target machine's hardware.
- **Java** code is compiled into bytecode by a compiler, and then the bytecode is executed by the JVM, making Java code generally slower than C++ but more portable.

5. Syntax and Semantics:

- **C++** syntax is similar to C, with features like pointers, manual memory management, and operator overloading.
- **Java** syntax is influenced by C and C++ but with significant differences, such as automatic memory management (garbage collection), absence of pointers, and explicit support for multi-threading.

6. Error Handling:

- **C++** primarily relies on exceptions for error handling, though error codes and return values are also used.
- **Java** uses exceptions extensively for error handling and has a comprehensive exception hierarchy.

7. Standard Libraries:

- **C++** Standard Library provides a rich set of libraries for various functionalities, including input/output operations, containers, algorithms, and concurrency.
- **Java** Standard Library (Java API) includes packages for networking, GUI development (Swing, JavaFX), database access (JDBC), and more, providing a comprehensive set of tools for application development.

8. Multiple Inheritance and Interfaces:

- **C++** supports multiple inheritance, allowing a class to inherit from multiple base classes.
- **Java** does not support multiple inheritance for classes but allows interface implementation, enabling multiple inheritance of types.

9. Pointers:

- **C++** allows direct manipulation of memory through pointers.
- **Java** does not have explicit pointer support and instead uses references, which are similar but more restricted than pointers.

10. Runtime Environment:

- **C++** programs run directly on the host machine's hardware without any virtual machine.
- **Java** programs run on the Java Virtual Machine (JVM), which adds a layer of abstraction between the code and the hardware.

C++ vs Python

1. Type System:

- C++ is statically typed, meaning variable types are determined at compile time and cannot change during runtime without explicit casting.
- Python is dynamically typed, allowing variables to hold values of any type, and types are resolved at runtime.

2. Syntax:

- C++ syntax is more verbose and requires explicit declaration of data types, curly braces for block structure, and semicolons to terminate statements.
- Python syntax is more concise and uses indentation to denote blocks of code. It does not require explicit declaration of data types, and statements are terminated by line breaks.

3. Memory Management:

- C++ requires manual memory management using features like pointers, new and delete operators for dynamic memory allocation and deallocation.
- Python has automatic memory management through garbage collection, relieving the programmer from explicit memory management tasks.

4. Performance:

- C++ typically offers better performance than Python due to its statically typed nature, compiled execution, and direct access to system resources.
- Python is generally slower than C++ as it is interpreted at runtime and has dynamic typing, which incurs additional runtime overhead.

5. Programming Paradigm:

- C++ supports procedural, object-oriented, and generic programming paradigms.
- Python supports procedural, object-oriented, functional, and imperative programming paradigms, offering more flexibility in coding styles.

6. Libraries and Ecosystem:

- C++ has a standard library (STL) providing core functionalities like data structures, algorithms, input/output operations, and concurrency support. Additionally, there are numerous third-party libraries available for various purposes.
- Python has a vast ecosystem of libraries and frameworks for almost every conceivable task, including web development (Django, Flask), scientific computing (NumPy, SciPy), data analysis (Pandas), machine learning (TensorFlow, PyTorch), and more.

7. Development Speed and Ease:

- C++ programs typically require more lines of code and effort to develop due to its lower-level nature and manual memory management.
- Python programs can be developed more quickly and with less code due to its high-level syntax, dynamic typing, and built-in data structures and functionalities.

8. Platform Independence:

- C++ code needs to be recompiled for each target platform, and platform-specific code may be required for system-level operations.
- Python code is platform-independent, running on any platform with a Python interpreter without modification.

9. Community and Support:

- Both C++ and Python have large and active communities providing support, documentation, tutorials, and open-source contributions.
- Python's community is often regarded as more beginner-friendly and accessible due to its simplicity and readability.

❖ Example of C++ Program:

Program-1:

```
#include <iostream>                    // Include the iostream header for input/output operations
using namespace std;                   // Use the std namespace to access cout
int main()                             // The main function should return an int
{
  cout << "Welcome to T3 SKILLS CENTER." << endl;    // Print the message and end the line
  cout << "Welcome to C++ Programming.\n" ;          // \n used for new line
  cout << "Join ROJGAR RID MISSION." << endl;
  return 0;                                         // Return 0 to indicate successful execution
}
```

Output-2:

```
Welcome to T3 SKILLS CENTER.
Welcome to C++ Programming.
Join ROJGAR RID MISSION.
```

Example-1 Explanation

1. **#include <iostream>:** This line includes the iostream header file, which contains declarations for input/output streams such as cin (for input) and cout (for output).
2. **using namespace std;:** This line tells the compiler to use the std namespace, which contains the standard C++ library components. This allows you to use cout and endl without prefixing them with **std::**
3. **int main():** This is the main function where program execution begins. It returns an integer, typically 0 to indicate successful execution.
4. **{}:** These curly braces denote the beginning and end of the main function's body.
5. **cout << "Welcome to T3 SKILLS CENTER." << endl;:** This line uses the cout object (which is part of the standard namespace std) to output the string "Welcome to T3 SKILLS CENTER." to the console. The << endl part inserts a newline character after the string, so the next output will appear on a new line.
6. **cout << "Welcome to C++ Programming.\n";:** Similar to the previous line, this prints the string "Welcome to C++ Programming." to the console. The \n is an escape sequence representing a newline character, which achieves the same effect as endl.
7. **return 0;:** This line exits the main function and returns 0 to the operating system, indicating successful execution. In C++, returning 0 from the main function typically indicates that the program terminated without errors.

❖ Namespace:

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

• key points about namespaces:

- **Avoiding Name Collisions:** Namespaces help to avoid naming conflicts between different parts of a program. For example, if two libraries define a function with the same name, putting each function in its own namespace prevents conflicts when using both libraries.
- **Encapsulation:** Namespaces encapsulate the identifiers within them, allowing you to control their visibility and access. Items within a namespace can be made public or private, just like in classes.
- **Usage:** You can define namespaces using the namespace keyword, followed by the namespace name and the items you want to include in that namespace. You can then access these items using the scope resolution operator ::.

Example:

```
using namespace std;
```

❖ Return Type of main():

- In C++, the main() function serves as the entry point for the program where execution begins.
- The main() function can have two valid return types.

1) **int:** This is the most commonly used return type for main(). It indicates the status of program execution to the operating system or the environment that invoked the program. By convention, a return value of 0 typically indicates successful execution, while a non-zero value indicates an error or abnormal termination. The specific meaning of non-zero return values can be defined by the programmer and may vary depending on the context.

Example:

```
int main() {

  // Program logic

  return 0; // Indicates successful execution

}
```

2) **void:** Although less common, main() can also have a return type of void, indicating that the function does not return a value. In this case, the operating system assumes that the program terminated successfully regardless of whether an explicit return statement is present or not.

Example:

```
void main() {
  // Program logic
}
```

COMMENTS

- comments are used to document code, provide explanations, and make the code more understandable for other developers.
- There are two main types of comments in C++.
  1) **Single-line comments:** These comments begin with two forward slashes // and continue until the end of the line. They are typically used for short explanations or comments on a single line of code.

Example:

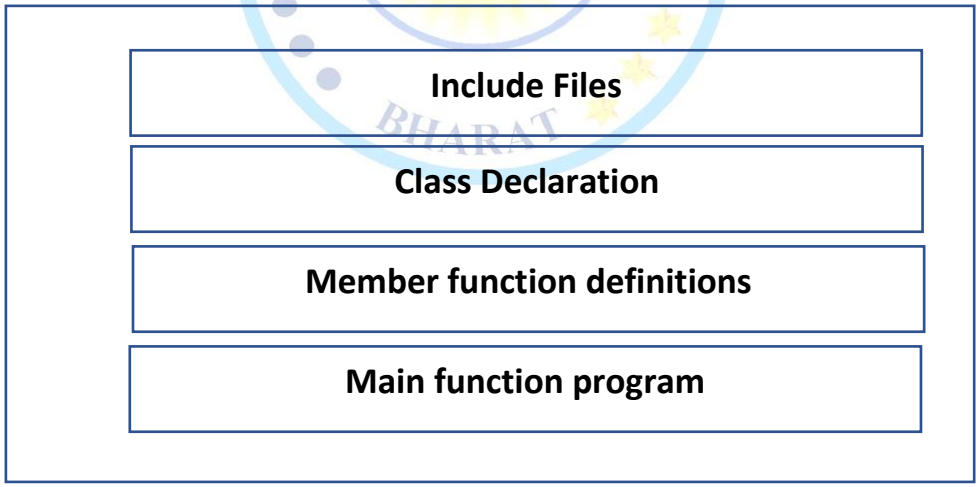
```
// This is a single-line comment
int x = 10; // Assigning the value 10 to variable
```

  2) **Multi-line comments:** These comments begin with /* and end with */. They can span multiple lines and are often used for longer explanations or comments that cover several lines of code.

Example:

```
/* This is a
   multi-line comment
   spanning multiple lines */
int y = 39; /* Assigning the value 20 to variable y */
```

Structure of a C++ Program



1. **Include Files:** #include <header_file> or #include "header_file"
2. **Class Declaration:** class ClassName { /* class members */ };
3. **Member Function Definitions:** return_type ClassName::member_function() { /* function body */ }
4. **Main Function:** int main() { /* main function body */ }

C++ BASIC INPUT/OUTPUT

- C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

1) If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as output operation.
2) If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as input operation.

❖ input output (I/O) Library Header:

1. **<iostream>:** This header file provides functionalities for standard input/output operations. It includes definitions for the **cin and cout** objects, which are used for reading input from the standard input stream (usually the keyboard) and writing output to the standard output stream (usually the console).

#include <iostream>

2. **<iomanip>:** This header file provides facilities for manipulating input/output formatting, such as setting the width of output fields, specifying the precision of floating-point numbers, and controlling the display format.

#include <iomanip>

3. **<fstream>:** This header file provides facilities for file input/output operations. It includes definitions for classes like ifstream, ofstream, and fstream, which are used for reading from and writing to files.

#include <fstream>

4. **<sstream>:** This header file provides facilities for string input/output operations. It includes definitions for classes like istringstream, ostringstream, and stringstream, which are used for reading from and writing to strings as if they were input/output streams.

#include <sstream>

5. **<cstdio> (or <stdio.h>):** Although primarily used in C programming, <cstdio> provides similar functionalities for input/output operations as <iostream>, but with a C-style interface. It includes definitions for functions like printf, scanf, fopen, fclose, etc.

#include <cstdio>

6. **<cstdlib> (or <stdlib.h>):** This header file provides facilities for general utilities, including input/output operations. It includes definitions for functions like atoi, atof, exit, system, etc.

#include <cstdlib>

❖ Standard output stream (cout):

- The standard output stream, cout, is used in C++ for printing output to the console. It's part of the <iostream> header and is commonly used in conjunction with the insertion operator <<.

Example-1:

```
#include <iostream>
int main() {
   // Using cout to display a message
   std::cout << "Hello World! RID MISSION" << std::endl;
   return 0;
}
```

Output-1:

```
Hello World! RID MISSION
```

Example-2:

```
#include <iostream>
using namespace std;
int main( ) {
   char ary[] = "Welcome to T3 SKILLS CENTER";
   cout << "Value of ary is: " << ary << endl;
}
```

Output-2:

```
Welcome to T3 SKILLS CENTER
```

Example-3:

```
#include <iostream>
using namespace std;
int main() {
   cout << "ROJGAR" << endl; // Display text with newline
   cout << "RID" << endl;
   cout << "MISSION!" << endl;
   return 0;
}
```

Output-3:

```
ROJGAR
RID
MISSION!
```

Example-1 Explanation:

- **#include <iostream>:** This line includes the <iostream> header, which provides definitions for cout and other input/output stream objects.
- **int main() { ... }:** This is the main function where program execution begins.
- **std::cout:** cout is the standard output stream object provided by the <iostream> library. The std:: prefix is used to specify that cout belongs to the std namespace.
- **"Hello, World! RID MISSION":** This is the message that we want to display. It's enclosed in double quotes to indicate that it's a string literal.
- **<<:** This is the insertion operator, used to insert the string into the cout stream.
- **std::endl:** endl is a manipulator that inserts a newline character into the output stream and flushes the buffer. It's used to end the current line of output.
- **return 0;:** This statement indicates that the program has executed successfully and returns 0 to the operating system.

Example-2 Explanation:

- **#include <iostream>:** This line includes the <iostream> header, which provides the necessary declarations for input/output operations.
- **using namespace std;:** This line brings the entire std namespace into scope, allowing us to use standard library functions and objects without needing to prefix them with std::.
- **int main() { ... }:** This is the main function where program execution begins.
- **char ary[] = "Welcome to T3 SKILLS CENTER";:** This line declares a character array named ary and initializes it with the string "Welcome to T3 SKILLS CENTER". The array size is automatically determined based on the length of the string.
- **cout << "Value of ary is: "** << ary << endl;: This line prints message "Value of ary is: " followed by the content of the ary array. Since ary is a character array, cout treats it as a C-style string and prints all characters until it encounters the null terminator ('\0').
- **endl:** This is a manipulator that inserts a newline character into the output stream and flushes the buffer, ensuring that the output is displayed immediately.

Example-3 Explanation:

- **#include <iostream>:** This line includes the <iostream> header, which provides the necessary declarations for input/output operations.
- **using namespace std;:** This line brings the entire std namespace into scope, allowing us to use standard library functions and objects without needing to prefix them with std::.
- **int main() { ... }:** This is the main function where program execution begins.
- **cout << "ROJGAR" << endl**;: This line uses the cout object from the <iostream> library to output the string "ROJGAR" to the console. The << operator is used to insert the string into the cout stream. The endl manipulator is then used to insert a newline character, ending the current line of output.
- **cout << "RID" << endl;:** Similarly, this line outputs the string "RID" to the console followed by a newline character.
- **return 0;:** This statement indicates that the program has executed successfully and returns 0 to the operating system.

❖ Standard input stream (cin):

- In C++, the standard input stream cin is an object of the istream class, which is defined in the <iostream> header file. It is used to read input from the standard input device, typically the keyboard.

Example-1:

```
#include <iostream>
using namespace std;
int main() {
   int number;
   cout << "Enter a number: ";                    // Prompt the user to enter a number
   cin >> number;                                 // Read the input from the user using cin
   cout << "You entered: " << number << endl;     // Display the number entered by the user
   return 0;  }
```

Output-1:

```
Enter a number: 39
You entered: 39
```

Example-2:

```
#include <iostream>
using namespace std;
int main( ) {
  int age;
   cout << "Enter your age: ";
   cin >> age;
   cout << "Your age is: " << age << endl;
}
```

Output-2:

```
Enter your age: 27
Your age is: 27
```

Example-3:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
   string name;                          // Declare a string variable to store the user's name
   cout << "Enter your name: ";          // Prompt the user to enter their name
   getline(cin, name);                   // Use getline() to read a line of text including spaces
   cout << "Hello, " << name << "!" << endl;
   return 0;}
```

Output-3:

```
Enter your name: Sangam Kumar
Hello, Sangam Kumar!
```

Example-1 Explanation:

- **#include <iostream>:** This line includes the <iostream> header file, which provides input/output (I/O) operations functionality in C++.
- **using namespace std;:** This line declares that we are using the std namespace. Namespaces are used to organize code into logical groups and prevent naming conflicts. The std namespace contains all the standard C++ library functions and objects.
- **int main() { ... }:** This defines the main function, which serves as the entry point of the program. Execution of a C++ program begins with the main() function.
- **int number;:** This declares an integer variable named number. This variable will be used to store the number entered by the user.
- **cout << "Enter a number:** ";: This line uses the cout object from the std namespace to display the text "Enter a number: " on the console.
- **cin >> number;:** This line uses the cin object from the std namespace to read input from the user. cin is used for input operations, and >> is the extraction operator, which is used to extract data from the input stream. The entered number is stored in the number variable.
- **cout << "You entered: " << number << endl;:** This line uses cout to display the text "You entered: " followed by the value stored in the number variable.
- **return 0;:** This statement indicates that the main() function has completed execution and returns an integer value of 0 to the operating system, indicating successful program termination.

Example-2 Explanation:

- **#include <iostream>:** Includes the input/output stream library, allowing input and output operations.
- **using namespace std;:** Declares the usage of the std namespace, which contains the standard C++ library functions.
- **int main() { ... }:** Defines the main function, the starting point of the program.
- **int age;:** Declares an integer variable named age to store the user's age.
- **cout << "Enter your age: ";:** Displays the prompt "Enter your age: " on the console using the cout object.
- **cin >> age;:** Reads the user input from the console and stores it in the age variable using the cin object.
- **cout << "Your age is: " << age << endl;:** Displays the message "Your age is: " followed by the value stored in the age variable on the console. endl is used to insert a newline character.

Example-3 Explanation:

- **#include <iostream>:** Includes the input/output stream library, enabling input and output operations.
- **#include <string>:** Includes the string library, allowing the use of string variables and functions.
- **using namespace std;:** Declares the usage of the std namespace, which contains standard C++ library functions.
- **int main() { ... }:** Defines the main function, the starting point of the program.
- **string name;:** Declares a string variable named name to store the user's name.
- **cout << "Enter your name: ";:** Displays the prompt "Enter your name: " on the console using the cout object.
- **getline(cin, name);:** Reads a line of text from the console, including spaces, and stores it in the name variable using the getline() function.
- **cout << "Hello, " << name << "!" << endl;:** Displays a greeting message "Hello, " followed by the value stored in the name variable, and an exclamation mark on the console. endl is used to insert a newline character.

❖Standard end line (endl):

- The endl is a predefined object of ostream class. It is used to insert a new line character and flushes the stream.

Example:

```
#include <iostream>
using namespace std;
int main( ) {
cout << "C++ Tutorial";
cout << " t3skillscenter"<<endl;
cout << "End of line"<<endl;
}
```

Output:

```
C++ Tutorial t3skillscenter
End of line
```

TOKENS

- token is the smallest individual unit of a program that the compiler can recognize and process. Tokens are the building blocks of C++ code and are categorized into different types based on their functions and roles within the program.

Types of tokens:

1. **Keywords:** Keywords are reserved words that have special meanings in the C++ language. Examples include int, if, else, while, class, return, etc.
2. **Identifiers:** Identifiers are user-defined names used to identify variables, functions, classes, and other entities within a program.
3. **Literals:** Literals are constant values used directly in the code. Examples include integer literals (10, -5, 0xFF), floating-point literals (3.14, 2.5e-3), character literals ('a', '\n'), and string literals ("Hello", "C++").
4. **Operators:** Operators are symbols used to perform operations on operands. C++ supports various types of operators, such as arithmetic operators (+, -, *, /), relational operators (==, !=, <, >), logical operators (&&, ||, !), assignment operators (=, +=, -=, etc.), and more.
5. **Punctuation:** Punctuation symbols are used to separate tokens or specify structure in the code. Examples include braces {}, parentheses (), square brackets [], commas ,, semicolons ;, periods ., etc.
6. **Comments:** Comments are used to provide explanations or notes within the code and are ignored by the compiler during compilation. In C++, comments can be single-line (//) or multi-line (/* */).
7. **Preprocessor Directives:** Preprocessor directives are instructions to the preprocessor, which is a program that processes the source code before compilation. Examples include #include, #define, #ifdef, #ifndef, etc.

KEYWORDS IN C++

- keyword is a reserved word that has a predefined meaning and is part of the syntax of the language. Keywords cannot be used as identifiers because they are already reserved for specific purposes within the language.
- Keywords are an essential part of the language's grammar and are used to define the structure and behavior of programs.

Example of Keyword:

- **auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while**

1) **auto:** Specifies automatic type deduction for variables.
2) **break:** Terminates the loop or switch statement and transfers control to the statement immediately following the loop or switch.
3) **case:** Marks a branch in a switch statement.
4) **char:** Declares a character type.
5) **const:** Declares an object as constant or a member function as not modifying the object.
6) **continue:** Skips the rest of the loop body and continues with the next iteration.
7) **default:** Specifies the default case in a switch statement.
8) **do:** Initiates a do-while loop.
9) **double:** Declares a double-precision floating-point type.
10) **else:** alternative statement or statements to execute if condition in if statement evaluates to false.
11) **enum:** Declares an enumeration type.
12) **extern:** Declares a variable or function as defined elsewhere.
13) **float:** Declares a floating-point type.
14) **for:** Initiates a for loop.
15) **goto:** Transfers control to a labeled statement.
16) **if:** Initiates an if statement.
17) **int:** Declares an integer type.
18) **long:** Declares a long integer type.
19) **register:** Specifies a variable as having register storage.
20) **return:** Exits a function and optionally returns a value.
21) **short:** Declares a short integer type.
22) **signed:** Declares a signed type.
23) **sizeof:** Returns the size of a data type or variable.
24) **static:** Specifies that a variable or function has static storage duration.
25) **struct:** Declares a structure type.
26) **switch:** Initiates a switch statement.
27) **typedef:** Creates a new name for an existing type.
28) **union:** Declares a union type.
29) **unsigned:** Declares an unsigned type.
30) **void:** Specifies that a function returns no value or that a pointer does not point to any data type.
31) **volatile:** Indicates that a variable may be changed unexpectedly.
32) **while:** Initiates a while loop.

KEYWORD SYMBOLS WITH NAMES

`	back quote	[	open square-bracket
~	tilde	]	close square-bracket
!	exclamation point	{	open curly-bracket/brace
@	at-sign	}	close curly-bracket/brace
#	pound-sign, number-sign, hash-tag	\	back-slash
$	dollar sign	|	pipe, bar
%	percent	;	semi colon
^	carat	:	colon
&	and-sign, ampersand	'	apostrophe, single-quote
*	asterisks	"	double-quote
(	open-parentheses	,	comma
)	close-parentheses	<	less-than, open angle-bracket
-	dash, minus-sign	.	period, dot, full-stop
_	underscore	>	greater-than, close angle-bracket
=	equals	/	slash
+	plus	?	question-mark

NUMBER SYSTEM

1) Binary Number:

- Binary number system, is a base-2 numeral system that uses two symbols: **0 and 1.**
- It's the foundation of digital technology and computing.
- Binary is fundamental in modern computing because digital devices, such as computers and microcontrollers, use binary to process and store data.
- All data, including text, images, sound, and videos, is ultimately represented in binary form within these devices.
- it directly corresponds to the on and off states of electronic switches and represents information using two distinct states.
- (0, 1) $( )_2$ {Base or Radix}

  Example: 0, 1, 01, 10, 1110, 10101011, 111001110101 etc.

2) Decimal Number:

- The decimal number system, also known as the base-10 number system, is a positional numeral system that uses ten symbols (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) to represent numbers.
- It's the most common number system used by humans in everyday life.
- The decimal system is essential for everyday arithmetic, commerce, science, and a wide range of applications.
- (0,1…..9) $( )_{10}$ {Base or radix}

  Example: 0,1,2,4,5,6,7,8,9,10,11,12,13,14, 15……

3) Octal Number:

- The octal number system, also known as base-8, is a positional numeral system that uses eight symbols (0, 1, 2, 3, 4, 5, 6, and 7) to represent numbers.
- The octal system was more commonly used in computing systems that were based on multiples of 3 (as opposed to the binary system's base-2).
- However, octal has largely been replaced by hexadecimal (base-16) in modern computing due to its compatibility with binary and its more compact representation.
- **(0,1,2…….7) $( )_{8\ \{Base\ or\ radix\}}$**

Example: 0,1,2,3,4,5,6,7,10, 11,…17,20,21……..27,30,31……

4) Hexadecimal Number:

- Hexadecimal number system, often referred to as "hex" or base-16, is a positional numeral system that uses sixteen symbols: 0-9 for values 0 to 9, and A-F (or a-f) for values 10 to 15.
- The hexadecimal system is widely used in computing and digital systems as a concise way to represent binary data and memory addresses.
- Hexadecimal is often used in programming and computer science because it provides a more compact representation of binary data, and it's easier to work with when converting between binary and other number systems.
- (0 …….9, A,B,C,D,E,F) $( )_{16}$ {Base or Radix}

Example: 0, 1, 2, 3, 4, 5, 6, 7,8, 9, A, B, C, D, E, F, 1a,1b,1c …….. etc

NUMBER SYSTEM CONVERSION

- Number system conversion is the process of converting a value from one numeral system (base) to another.

❖ Decimal Number to Binary Number Conversion

- **Question-1: Convert $(27)_{10}$, $(137)_{10}$ and $(66)_{10}$ Decimal into binary number**

1st Method

2	27
2	13 ----- 1
2	6 ------1
2	3 ------ 0
	1 ----- 1

Ans.11011

2	137
2	68 ------ 1
2	34 ------ 0
2	17 ------ 0
2	8 -------- 1
2	4---------- 0
2	2---------- 0
	1---------- 0

Ans.10001001

2	66
2	33 ------ 0
2	16 ------ 1
2	8 ------ 0
2	4 ------ 0
2	2 ------ 0
	1 ------ 0

Ans.1000010

2st Method

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1
		(27>16	27=	16	+8		+2	+1
				1	1	0	1	1

Ans: $(27)_{10} = (11011)_2$

❖ Decimal Number to Octal Number Conversion:

- **Question-2: Convert $(27)_{10}$, $(294)_{10}$ and $(137)_{10}$ Decimal into octal number**

8	27
	3 ----- 3

Ans. $(33)_8$

8	294
8	36 ----- 6
	4 ----- 4

Ans. $(446)_8$

8	137
8	17 ----- 1
	2 ----- 1

Ans. $(211)_8$

❖ Decimal Number to Hexadecimal Number Conversion:

- **Question-3: Convert $(27)_{10}$, $(294)_{10}$ and $(137)_{10}$ Decimal into Hexadecimal number**

16	27
	1 ----- B

Ans. $(1B)_{16}$

16	294
16	18 ----- 6
	1 ----- 2

Ans. $(126)_{16}$

16	137
	8 ----- 9

Ans. $(89)_{16}$

❖ Binary to Decimal:

1. **$(100010010)_2 = (\ )_{10}$**
   - Solution: $1 \times 2^8 + 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
     $256+0+0+0+16+0+0+2+0=274 \quad (274)_{10}$ Ans.
2. **$(10010)_2 = (\ )_{10}$**
   - Solution: $1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
     $16+0+0+2+0=18 \quad (18)_{10}$ Ans.
3. **$(1110100)_2 = (\ )_{10}$**
   - Solution: $1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
     $64+32+16+0+4+0+0= 116 \quad (116)_{10}$ Ans.

❖ Octal to Decimal:

1. **$(422)_8 = (\ )_{10}$**
   - Solution: $4 \times 8^2 + 2 \times 8^1 + 2 \times 8^0$
     $4 \times 64 + 2 \times 8 + 2 \times 1$
     $256 + 16 + 2= 274 \quad (274)_{10}$ Ans.
2. **$(4211)_8 = (\ )_{10}$**
   - Solution: $4 \times 8^3 + 2 \times 8^2 + 1 \times 8^1 + 1 \times 8^0$
     $4X 512 + 2 \times 64 + 1 \times 8 +1$
     $2048 + 128 + 8 + 1= 2185 \quad (2185)_{10}$ Ans.
3. **$(333)_8 = (\ )_{10}$**
   - Solution: $3 \times 8^2 + 3 \times 8^1 + 3 \times 8^0$
     $3 \times 64 + 3 \times 8 + 3 \times 1$
     $192 + 24 + 3= 219 \quad (219)_{10}$ Ans.

❖ Hexadecimal to decimal:

1. **$(422)_{16} = (\ )_{10}$**
   - Solution: $4 \times 16^2 + 2 \times 16^1 + 2 \times 16^0$
     $4 \times 256 + 2 \times 16 + 2 \times 1$
     $1024 +32 + 2= 1058 \quad (1058)_{10}$ Ans.
2. **$(4211)_{16} = (\ )_{10}$**
   - Solution: $4 \times 16^3 + 2 \times 16^2 + 1 \times 16^1 + 1 \times 16^0$
     $4 \times 4096 + 2 \times 256 + 1 \times 16 +1$
     $16384+ 512 + 16+1= 16913 \quad (16913)_{10}$ Ans.
3. **$(333)_{16} = (\ )_{10}$**
   - Solution: $3 \times 16^2 + 3 \times 16^1 + 3 \times 16^0$
     $3 \times 256 + 3 \times 16+ 3 \times 1$
     $768 + 48 + 3= 819 \quad (819)_{10}$ Ans.
4. **$(333AB)_{16}$** $= (3 \times 16^4) + (3 \times 16^3) + (3 \times 16^2) + (10 \times 16^1) + (11 \times 16^0) = (209835)_{10}$
5. $(DCF39)_{16} = (13 \times 16^4) + (12 \times 16^3) + (15 \times 16^2) + (3 \times 16^1) + (9 \times 16^0) = (905017)_{10}$
6. $(AB23F34)_{16} = (10 \times 16^6) + (11 \times 16^5) + (2 \times 16^4) + (3 \times 16^3) + (15 \times 16^2) + (3 \times 16^1) + (4 \times 16^0) = (179453748)_{10}$

IDENTIFIERS AND CONSTANTS

- identifiers and constants are fundamental concepts used to name entities and represent fixed values within a program.

1. Identifiers:

- Identifiers are user-defined names used to identify variables, functions, classes, and other entities within a program.

❖ Rules for identifiers:

1. Must begin with a letter (uppercase or lowercase) or an underscore `_`.
2. Can be followed by letters, digits, or underscores.
3. Cannot contain spaces or special characters (except underscores).
4. Case sensitivity: C++ is case-sensitive, so `myVariable` and `MyVariable` are considered different identifiers.

Examples of valid identifiers:

- Counter
- _value,
- myFunction,
- ClassName,
- MAX_SIZE, etc.
- _39abt

Examples of invalid identifiers in C++:

- 123abc: → Begins with a digit.
- my variable: → Contains a space.
- my-variable: → Contains a hyphen.
- my*variable: → Contains an asterisk.
- my.variable: → Contains a period.
- if: → Reserved keyword.

- **Constants:**
  - Constants are fixed values that do not change during the execution of a program.
  - Constants can be classified into two main types:
    1. literal constants and
    2. symbolic constants.

1. Literal Constants:

- Literal constants are constant values used directly in the code.

Examples: :Integer literals: `10`, `-5`, `0xFF` (hexadecimal), `057` (octal). Floating-point literals: `3.14`, `2.5e-3`. Character literals: `'A'`, `'5'`, `'\n'` (newline character). String literals: `"Hello"`, `"C++"`.

2. Symbolic Constants:

- Symbolic constants are identifiers that represent fixed values.
- They are declared using the `const` keyword or by using preprocessor `#define` directives.

Example using const: const int MAX_SIZE = 30939;, const double PI = 3.14;

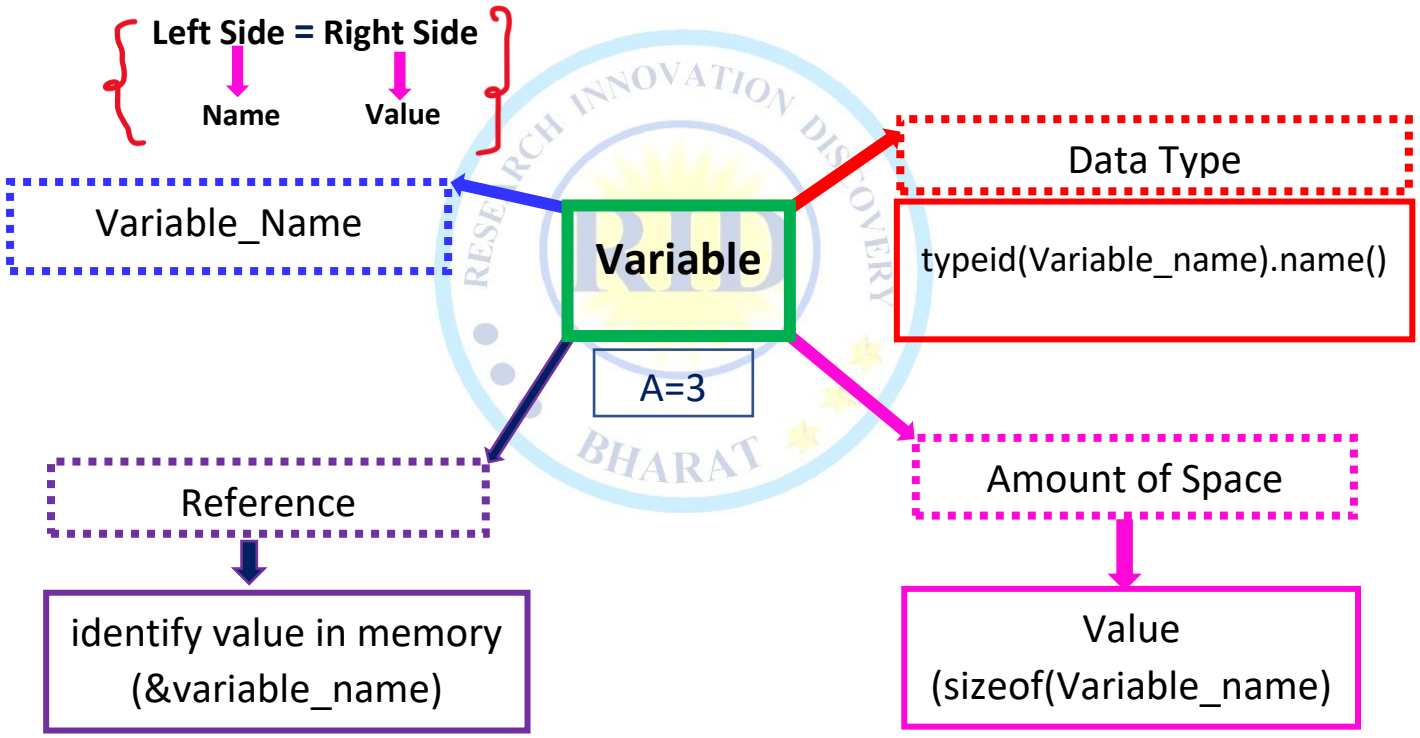
VARIABLE

- variable is a named storage location in the computer's memory where you can store a value that can be manipulated or accessed within a program. It acts as a placeholder for data that can change during the execution of the program.

Variables in C++ have the following characteristics:

1. **Name:** A variable has a unique identifier, known as its name.
2. **Data Type:** Every variable has a data type that specifies what kind of data it can hold, such as integers, floating-point numbers, characters, etc.
3. **Value:** A variable holds a value, which can be assigned during declaration or later in the program's execution. The value can change during the program's execution.
4. **Reference & Scope:** it help for system to identify value in memory The scope of a variable defines where in the program it can be accessed. Variables can be local to a function or block, or they can be global, accessible throughout the entire program.

Syntax:



Example:

```
#include <iostream>
int main() {                                    // Declaration of a variable named 'age' of type int
   int age;                                     // Assigning a value to the variable 'age'
   age = 60;                                    // Printing the value of 'age'
   std::cout << "Age: " << age << std::endl;
  std::cout << "Memory address of variable x: " << &age << std::endl;
   return 0;  }
```

Output:

Where age is a variable of type **int**, which holds the value **60.** The value is assigned using the assignment operator =. Finally, the value of age is 60 printed to the console. Memory address of variable age: 0x7ffee1757b24

Rules for defining variables:

1. **Valid Characters:** (both uppercase and lowercase), digits, and underscores (_).
2. **Variable names:** must begin with a letter (uppercase or lowercase) or an underscore (_).
3. **Case Sensitivity:** meaning uppercase and lowercase letters are considered distinct.
4. **Keywords:** Variable names cannot be the same as C++ keywords (reserved words)
5. **No Spaces:** Variable names cannot contain spaces.
6. **Length Limitation:** There is no strict limit on the length of variable names,T he C++ standard requires support for at least 63 characters in an identifier.
7. **Naming Convention:** it's a good practice to follow a consistent naming convention for variables. Common conventions include camelCase, PascalCase, or snake_case.
8. **Meaningful Names:** Choose descriptive and meaningful names for variables that indicate their purpose or the data they hold. This enhances code readability and maintainability.
9. **Avoid Non-ASCII Characters:** While some compilers might support non-ASCII characters in variable names, it's generally a good idea to stick to ASCII characters for portability and compatibility reasons.

Valid Variable Names:

1) age
2) firstName
3) _temperature
4) numberOfStudents
5) totalMarks
6) MAX_SIZE
7) is_valid
8) userInput
9) numCars
10) average_score

Invalid Variable Names:

1) 2ndPlace (starts with a digit)
2) my variable (contains a space)
3) break (a C++ keyword)
4) #count (contains special character)
5) total$ (contains special character)
6) bool (a C++ keyword)
7) float (a C++ keyword)
8) long long (contains spaces)
9) my-variable (contains a hyphen)
10) class (a C++ keyword)

Example: For find the size or amount of variable.

```
#include <iostream>
#include <string>
int main() {
   int intValue = 42;
   char charValue = 'A';
   std::string stringValue = "Hello, world!";
   float floatValue = 3.14;
   std::cout << "Size of intValue: " << sizeof(intValue) << " bytes" << std::endl;
   std::cout << "Size of charValue: " << sizeof(charValue) << " bytes" << std::endl;
   std::cout << "Size of stringValue: " << sizeof(stringValue) << " bytes" << std::endl;
   std::cout << "Size of floatValue: " << sizeof(floatValue) << " bytes" << std::endl;
   return 0; }
```

Output:

```
Size of intValue: 4 bytes
Size of charValue: 1 bytes
Size of stringValue: 32 bytes
Size of floatValue: 4 bytes
```

Example: For find the data types of variables.

```
#include <iostream>
#include <typeinfo>
#include <string>
int main() {
   int intValue = 42;
   char charValue = 'A';
   std::string stringValue = "Hello, world!";
   float floatValue = 3.14;
   std::cout << "Data type of intValue: " << typeid(intValue).name() << std::endl;
   std::cout << "Data type of charValue: " << typeid(charValue).name() << std::endl;
   std::cout << "Data type of stringValue: " << typeid(stringValue).name() << std::endl;
   std::cout << "Data type of floatValue: " << typeid(floatValue).name() << std::endl;
```

Output:

```
Data type of intValue: int
Data type of charValue: char
Data type of stringValue: class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >
Data type of floatValue: float
```

Example: For find the Memory address of variables.

```
#include <iostream>
#include <string>
int main() {
   int intValue = 42;
   char charValue = 'A';
   std::string stringValue = "Hello, world!";
   float floatValue = 3.14;
   std::cout << "Value of intValue: " << intValue << ", Memory address: " << &intValue << std::endl;
   std::cout << "Value of charValue: " << charValue << ", Memory address: " << (void *)&charValue << std::endl;
   std::cout << "Value of stringValue: " << stringValue << ", Memory address: " << &stringValue << std::endl;
   std::cout << "Value of floatValue: " << floatValue << ", Memory address: " << &floatValue << std::endl;
   return 0; }
```

Output:

```
Value of intValue: 42, Memory address: 0x7ffee2a4a9dc
Value of charValue: A, Memory address: 0x7ffee2a4a9db
Value of stringValue: Hello, world!, Memory address: 0x7ffee2a4a9e0
Value of floatValue: 3.14, Memory address: 0x7ffee2a4a9d8
```

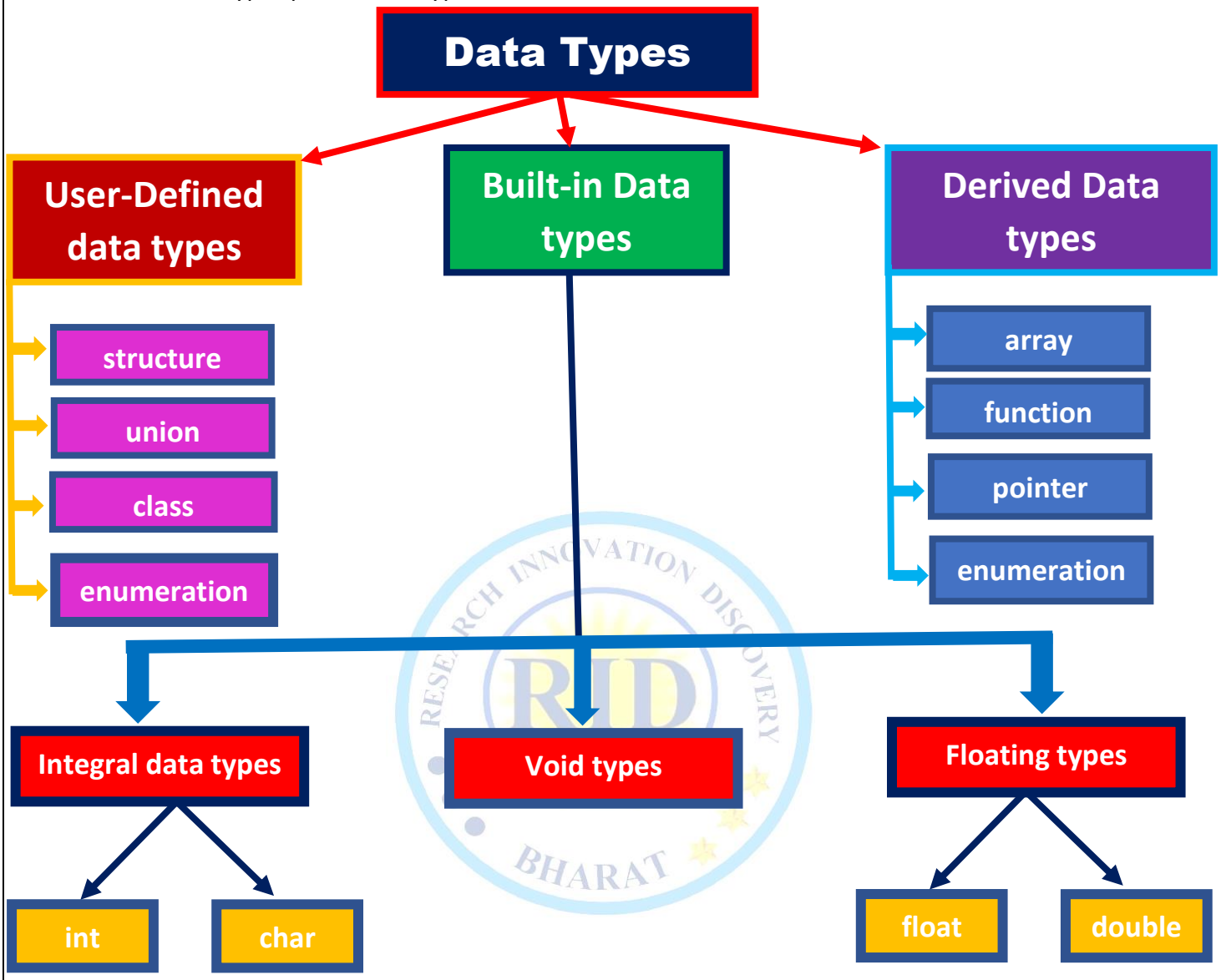
Uses of C++ Variables:

- C++ variables serve various purposes in programming, facilitating data storage, manipulation, and organization. Here are some common uses of C++ variables

Data Storage, Data Manipulation, Input and Output, Control Structures, Function Parameters and Return Values, Data Storage in Arrays and Collections, Error Handling,Memory Management,Object-Oriented Programming,Global Configuration and Constants:

DATA TYPES

- A data type specifies the type of data that a variable store.



- There are 3 types of data types in C++ language.
  1. Basic Data Type or Built-in data types
  2. Derived Data Type
  3. User Defined Data Type

1. Basic Data Type:

- Basic data types are fundamental data types provided by the C++ language. They represent the basic building blocks for storing data.
  - **a) int:** Used to store integer values.
  - **b) float:** Used to store single-precision floating-point numbers.
  - **c) double:** Used to store double-precision floating-point numbers.
  - **d) char:** Used to store single characters.
  - **e) bool:** Used to store boolean values (true or false).

Example:

```
int myInteger = 10;
float myFloat = 3.14f;
double myDouble = 3.14159265359;
char myChar = 'A';
bool myBoolean = true;
```

2. Derived Data Type:

- Derived data types are constructed from basic data types.
  - a) **Arrays:** Collection of elements of the same data type.
  - b) **Pointers:** Variables that hold memory addresses.
  - c) **References:** Aliases for existing variables.
  - d) **Functions:** Blocks of code that can be called.

Example:

```
int myArray[5]; // Array of 5 integers
int* myPointer; // Pointer to an integer
int& myReference = myInteger; // Reference to myInteger
void myFunction(); // Example function declaration
```

3. User Defined Data Type:

- User-defined data types are created by the programmer to suit specific needs. They include:
  - a) **Structures:** Groups of variables under one name.
  - b) **Classes:** Similar to structures but with member functions.
  - c) **Unions:** Similar to structures but share the same memory location.

Example usage (structures):

```
struct Point {
   int x;
   int y;
};
Point p;
p.x = 5;
p.y = 10;
```

Example usage (classes):

```
class Car {
public:
   int speed;
   void accelerate(int increment) {
      speed += increment;
   } };
Car myCar;
myCar.speed = 0;
myCar.accelerate(10);
```

Example usage (unions):

```
union MyUnion {
   int intValue;
   float floatValue;
};
MyUnion u;
u.intValue = 5;
std::cout << u.floatValue; // May produce garbage value due to type mismatch
```

❖ Size and Range of C++ Basic Data types.

Data Types	Memory Size	Range
1. char	1 byte	-128 to 127
2. signed char	1 byte	-128 to 127
3. unsigned char	1 byte	0 to 127
4. short	2 byte	-32,768 to 32,767
5. signed short	2 byte	-32,768 to 32,767
6. unsigned short	2 byte	0 to 32,767
7. int	2 byte	-32,768 to 32,767
8. signed int	2 byte	-32,768 to 32,767
9. unsigned int	2 byte	0 to 32,767
10. short int	2 byte	-32,768 to 32,767
11. signed short int	2 byte	-32,768 to 32,767
12. unsigned short int	2 byte	0 to 32,767
13. long int	4 byte	
14. signed long int	4 byte	
15. unsigned long int	4 byte	
16. float	4 byte	
17. double	8 byte	
18. long double	10 byte	

Program:

```
// C++ program to Demonstrate the sizes of data types
#include <iostream>
#include <limits.h>
using namespace std;
int main()
{
cout << "Size of char : " << sizeof(char) << " byte"<< endl;
cout << "char minimum value: " << CHAR_MIN << endl;
cout << "char maximum value: " << CHAR_MAX << endl;
cout << "Size of int : " << sizeof(int) << " bytes"<< endl;
cout << "Size of short int : " << sizeof(short int)<< " bytes" << endl;
cout << "Size of long int : " << sizeof(long int)<< " bytes" << endl;
cout << "Size of signed long int : "<< sizeof(signed long int) << " bytes" << endl;
```

```
cout << "Size of unsigned long int : "<< sizeof(unsigned long int) << " bytes" << endl;
cout << "Size of float : " << sizeof(float) << " bytes"<< endl;
cout << "Size of double : " << sizeof(double)<< " bytes" << endl;
cout << "Size of wchar_t : " << sizeof(wchar_t)<< " bytes" << endl;
return 0; }
```

Output:

```
Size of char : 1 byte
char minimum value: -128
char maximum value: 127
Size of int : 4 bytes
Size of short int : 2 bytes
Size of long int : 8 bytes
Size of signed long int : 8 bytes
Size of unsigned long int : 8 bytes
Size of float : 4 bytes
Size of double : 8 bytes
Size of wchar_t : 4 bytes
```

OPERATORS

- operators are symbols used to perform operations on operands. They enable you to manipulate data and perform various computations in your program.
- C++ provides a wide range of operators, which can be categorized into several groups based on their functionality.

1. Arithmetic Operators:

- Arithmetic operators are used to perform basic arithmetic operations on numerical operands.
  1. **Addition (+)** : Adds two operands.
  2. **Subtraction (-)** : Subtracts the second operand from the first.
  3. **Multiplication (*)** : Multiplies two operands.
  4. **Division (/)** : Divides the first operand by the second.
  5. **Modulus (%)** : Returns the remainder of the division operation.

2. Relational Operators:

- Relational operators are used to compare two values and determine the relationship between them.
  1. **Equal to (==)** : Checks if two operands are equal.
  2. **Not equal to (!=)** : Checks if two operands are not equal.
  3. **Greater than (>)** : Checks if the first operand is greater than the second.
  4. **Less than (<)** : Checks if the first operand is less than the second.
  5. **Greater than or equal to (>=)** : Checks if the first operand is greater than or equal to the second.
  6. **Less than or equal to (<=):** : Checks if the first operand is less than or equal to the second.

3. Logical Operators:

- Logical operators are used to perform logical operations on boolean operands.
  1. **Logical AND (&&)** : Returns true if both operands are true.
  2. **Logical OR (||)** : Returns true if at least one of the operands is true.
  3. **Logical NOT (!)** : Returns the opposite of the operand's value.

4. Assignment Operators:

- Assignment operators are used to assign values to variables.
  1) **Assignment (=)** : Assigns the value of the right operand to the left operand.
  2) **Addition assignment (+=)** : Adds value of right operand to left operand and assigns result to left operand.
  3) **Subtraction assignment (-=)** : Subtracts value of right operand from left operand and assigns result to left operand.
  4) **Multiplication assignment (*=)** : Multiplies value of left operand by right operand and assigns result to left operand.
  5) **Division assignment (/=)** : Divides the value of left operand by right operand and assigns result to left operand.
  6) **Modulus assignment (%=)** : Calculates modulus of left operand with right operand and assigns result to left operand.

5. Increment and Decrement Operators:

- Increment and decrement operators are used to increase or decrease the value of a variable by one.
  1. **Increment (++)** : Increases the value of the operand by one.
  2. **Decrement (--)** : Decreases the value of the operand by one.

6. Bitwise Operators:

- Bitwise operators are used to perform operations on individual bits of numerical operands.
  1. **Bitwise AND (&)** : Performs bitwise AND operation.
  2. **Bitwise OR (|)** : Performs bitwise OR operation.
  3. **Bitwise XOR (^)** : Performs bitwise exclusive OR operation.
  4. **Bitwise NOT (~)** : Performs bitwise complement operation.
  5. **Left shift (<<)** : Shifts bits of left operand to left by number of positions specified by right operand.
  6. **Right shift (>>)** : Shifts the bits of left operand to right by number of positions specified by right operand.

7. Other Operators:

- C++ also provides other operators for specific purposes:
  1. **Conditional Operator (?:)** : Also known as the ternary operator, it evaluates a condition and returns one of two possible values depending on whether the condition is true or false.
  2. **Comma Operator (,)** : Evaluates multiple expressions separated by commas and returns the value of the last expression.
  3. **Member Access Operators (. and ->):** Used to access members of objects and pointers to objects.
  4. **Sizeof Operator (sizeof)** : Returns the size of a variable or data type in bytes.
  5. **Pointer Operators (* and &)** : Used to declare and manipulate pointers.
  6. **Type Cast Operator** : Used to convert one data type into another.

1. Arithmetic Operators:

Example:

```
int a = 10, b = 3;
int sum = a + b;                // sum = 13
int difference = a - b;         // difference = 7
int product = a * b;            // product = 30
int quotient = a / b;           // quotient = 3
int remainder = a % b;          // remainder = 1
```

Program:

```
#include <iostream>
using namespace std;
int main() {
   int a = 10, b = 3;
   // Arithmetic Operators
   int sum = a + b;             // Addition
   int difference = a - b;      // Subtraction
   int product = a * b;         // Multiplication
   int quotient = a / b;        // Division
   int remainder = a % b;       // Modulus
   // Output
   cout << "Arithmetic Operators:" << endl;
   cout << "Sum: " << sum << endl;
   cout << "Difference: " << difference << endl;
   cout << "Product: " << product << endl;
   cout << "Quotient: " << quotient << endl;
   cout << "Remainder: " << remainder << endl;
   return 0;
}
```

Output:

```
Arithmetic Operators:
Sum: 13
Difference: 7
Product: 30
Quotient: 3
Remainder: 1
```

2. Relational Operators:

Example:

```
int x = 5, y = 10;
bool isEqual = (x == y);              // isEqual = false
bool isNotEqual = (x != y);           // isNotEqual = true
bool isGreaterThan = (x > y);         // isGreaterThan = false
bool isLessThan = (x < y);            // isLessThan = true
bool isGreaterOrEqual = (x >= y);     // isGreaterOrEqual = false
bool isLessOrEqual = (x <= y);        // isLessOrEqual = true
```

Program:

```
#include <iostream>
using namespace std;
int main() {
   int x = 5, y = 10;
   // Relational Operators
   bool isEqual = (x == y);              // Equal to
   bool isNotEqual = (x != y);           // Not equal to
   bool isGreaterThan = (x > y);         // Greater than
   bool isLessThan = (x < y);            // Less than
   bool isGreaterOrEqual = (x >= y);     // Greater than or equal to
   bool isLessOrEqual = (x <= y);        // Less than or equal to
   // Output
   cout << "Relational Operators:" << endl;
   cout << "isEqual: " << isEqual << endl;
   cout << "isNotEqual: " << isNotEqual << endl;
   cout << "isGreaterThan: " << isGreaterThan << endl;
   cout << "isLessThan: " << isLessThan << endl;
   cout << "isGreaterOrEqual: " << isGreaterOrEqual << endl;
   cout << "isLessOrEqual: " << isLessOrEqual << endl;
   return 0;
}
```

Output:

```
Relational Operators:
isEqual: 0
isNotEqual: 1
isGreaterThan: 0
isLessThan: 1
isGreaterOrEqual: 0
isLessOrEqual: 1
```

3. Logical Operators:

Example:

```
bool a = true, b = false;
bool resultAnd = (a && b);            // resultAnd = false
bool resultOr = (a || b);             // resultOr = true
bool resultNotA = !a;                 // resultNotA = false
```

Program:

```
#include <iostream>
using namespace std;
int main() {
   bool a = true, b = false;
   // Logical Operators
```

```
    bool resultAnd = (a && b);          // Logical AND
    bool resultOr = (a || b);           // Logical OR
    bool resultNotA = !a;               // Logical NOT
    // Output
    cout << "Logical Operators:" << endl;
    cout << "resultAnd: " << resultAnd << endl;
    cout << "resultOr: " << resultOr << endl;
    cout << "resultNotA: " << resultNotA << endl;
    return 0;
}
```

Output:

```
Logical Operators:
resultAnd: 0
resultOr: 1
resultNotA: 0
```

4. Assignment Operator:

Example:

```
int x = 5;
x += 3;         // equivalent to x = x + 3; x becomes 8
x -= 2;         // equivalent to x = x - 2; x becomes 6
x *= 4;         // equivalent to x = x * 4; x becomes 24
x /= 3;         // equivalent to x = x / 3; x becomes 8
x %= 5;         // equivalent to x = x % 5; x becomes 3
```

Program:

```
#include <iostream>
using namespace std;
int main() {
    int x = 5;
    // Assignment Operators
    x += 3;         // Addition assignment
    x -= 2;         // Subtraction assignment
    x *= 4;         // Multiplication assignment
    x /= 3;         // Division assignment
    x %= 5;         // Modulus assignment
    // Output
    cout << "Assignment Operator:" << endl;
    cout << "x: " << x << endl;
    return 0;
}
```

Output:

```
Assignment Operator:
x: 3
```

5. Increment and Decrement Operators:

Program:

```
#include <iostream>
using namespace std;
int main() {
   int num = 5;
   // Increment operator (++num)
   cout << "Initial value of num: " << num << endl;
   cout << "Incrementing num using prefix increment operator: " << ++num << endl;
   cout << "Value of num after prefix increment: " << num << endl;
   cout << "\nResetting num to 5..." << endl;
   num = 5; // Reset num to 5
   cout << "Initial value of num: " << num << endl;
   cout << "Decrementing num using prefix decrement operator: " << --num << endl;
   cout << "Value of num after prefix decrement: " << num << endl;
   // Increment and decrement operators can also be used postfix
   cout << "\nResetting num to 5..." << endl;
   num = 5; // Reset num to 5
   cout << "Initial value of num: " << num << endl;
   cout << "Incrementing num using postfix increment operator: " << num++ << endl;
   cout << "Value of num after postfix increment: " << num << endl;
   cout << "\nResetting num to 6..." << endl;
   num = 6; // Reset num to 6
   cout << "Initial value of num: " << num << endl;
   cout << "Decrementing num using postfix decrement operator: " << num-- << endl;
   cout << "Value of num after postfix decrement: " << num << endl;
   return 0;
}
```

Output:

```
Initial value of num: 5
Incrementing num using prefix increment operator: 6
Value of num after prefix increment: 6
Resetting num to 5...
Initial value of num: 5
Decrementing num using prefix decrement operator: 4
Value of num after prefix decrement: 4
Resetting num to 5...
Initial value of num: 5
Incrementing num using postfix increment operator: 5
Value of num after postfix increment: 6
Resetting num to 6...
Initial value of num: 6
Decrementing num using postfix decrement operator: 6
Value of num after postfix decrement:
```

6. Bitwise Operators:

Example:

```
int a = 5, b = 3;
int bitwiseAND = a & b;          // bitwiseAND = 1
int bitwiseOR = a | b;           // bitwiseOR = 7
int bitwiseXOR = a ^ b;          // bitwiseXOR = 6
int bitwiseNOT = ~a;             // bitwiseNOT = -6
int leftShift = a << 1;          // leftShift = 10
int rightShift = a >> 1;         // rightShift = 2
```

Program:

```
#include <iostream>
using namespace std;
int main() {
   int a = 5, b = 3;
   // Bitwise Operators
   int bitwiseAND = a & b;       // Bitwise AND
   int bitwiseOR = a | b;        // Bitwise OR
   int bitwiseXOR = a ^ b;       // Bitwise XOR
   int bitwiseNOT = ~a;          // Bitwise NOT
   int leftShift = a << 1;       // Left shift
   int rightShift = a >> 1;      // Right shift
   // Output
   cout << "Bitwise Operators:" << endl;
   cout << "bitwiseAND: " << bitwiseAND << endl;
   cout << "bitwiseOR: " << bitwiseOR << endl;
   cout << "bitwiseXOR: " << bitwiseXOR << endl;
   cout << "bitwiseNOT: " << bitwiseNOT << endl;
   cout << "leftShift: " << leftShift << endl;
   cout << "rightShift: " << rightShift << endl;
   return 0;
}
```

Output:

```
Bitwise Operators:
bitwiseAND: 1
bitwiseOR: 7
bitwiseXOR: 6
bitwiseNOT: -6
leftShift: 10
rightShift: 2
rightShift: 2
```

7. Other Operator:

❖ Conditional Operator (?:)

Program:

```
#include <iostream>
using namespace std;
int main() {
   int x = 10, y = 5;
   int max = (x > y) ? x : y;
   cout << "Maximum value between " << x << " and " << y << " is: " << max << endl;
   return 0;
}
```

Output:

```
Maximum value between 10 and 5 is: 10
```

❖ Comma Operator (,)

Program:

```
#include <iostream>
using namespace std;
int main() {
   int a = 1, b = 2, c = 3;
   int result = (a++, b++, c++);
   cout << "Value of result after using comma operator: " << result << endl;
   return 0;
}
```

Output: Value of result after using comma operator: 3

❖ Member Access Operators (. and ->):

Program:

```
#include <iostream>
using namespace std;
struct Example {
   int value;
};
int main() {
   Example obj;
   obj.value = 100;
   Example *ptr = &obj;
   cout << "Value of obj using dot operator: " << obj.value << endl;
   cout << "Value of obj using arrow operator: " << ptr->value << endl;
   return 0;
}
```

Output:

```
Value of obj using dot operator: 100
Value of obj using arrow operator: 100
```

❖ Sizeof Operator (sizeof):

Program:

```
#include <iostream>
using namespace std;
int main() {
   cout << "Size of int data type: " << sizeof(int) << " bytes" << endl;
   return 0;
}
```

Output:

Size of int data type: 4 bytes

❖ Pointer Operators (* and &):

Program:

```
#include <iostream>
using namespace std;
int main() {
   int num = 10;
   int *ptrNum = &num;
   cout << "Value of num: " << num << endl;
   cout << "Address of num: " << &num << endl;
   cout << "Value stored at the address pointed by ptrNum: " << *ptrNum << endl;
   return 0;
}
```

Output:

```
Value of num: 10
Address of num: 0x7ffe04a56efc
Value stored at the address pointed by ptrNum: 10
```

❖ Type Cast Operator:

Program:

```
#include <iostream>
using namespace std;
int main() {
   double doubleNum = 3.14;
   int intNum = static_cast<int>(doubleNum);
   cout << "Value of doubleNum after type casting to int: " << intNum << endl;
   return 0;
}
```

Output:

Value of doubleNum after type casting to int: 3

❖ Unary Operator:

Example:

```
int x = 5;
int unaryPlus = +x;     // unaryPlus = 5
int unaryMinus = -x;    // unaryMinus = -5
x++;    // Increment
x--;    // Decrement
bool notValue = !true;  // notValue = false
int bitwiseNOT = ~x;    // bitwiseNOT = -6
```

Program:

```
#include <iostream>
using namespace std;
int main() {
   int x = 5;
   // Unary Operators
   x++;    // Increment
   x--;    // Decrement
   // Output
   cout << "Unary Operator:" << endl;
   cout << "x: " << x << endl;
   return 0;
}
```

Output:

```
Unary Operator:
x: 5
```

❖ Ternary or Conditional Operator:

Program:

```
#include <iostream>
using namespace std;
int main() {
   int x = 5, y = 10;
   // Ternary or Conditional Operator
   int result = (x > y) ? x : y;
   // Output
   cout << "Ternary or Conditional Operator:" << endl;
   cout << "result: " << result << endl;
   return 0;
}
```

Output:

```
Ternary or Conditional Operator:
result: 10
```

OPERATOR PRECEDENCE

1)	::	Scope resolution
2)	(type)	Type casting
3)	++ --	Increment and decrement (postfix)
4)	sizeof	Size of
5)	new delete	Dynamic memory allocation and deallocation
6)	+ - ! ~	Unary operators
7)	* / %	Multiplication, division, modulus
8)	+ -	Addition and subtraction
9)	<< >>	Bitwise shift left and right
10)	< <= > >=	Relational operators
11)	== !=	Equality and inequality
12)	&	Bitwise AND
13)	^	Bitwise XOR
14)	|	Bitwise OR
15)	&&	Logical AND
16)	||	Logical OR
17)	?:	Ternary conditional
18)	= += -= *= /= %= &=	Assignment operators
19)	^= |= <<= >>=	Bitwise assignment operators
20)	,	Comma operator

Top

Bottom

Right to Left

Program-1:

```
#include <iostream>
using namespace std;
int main() {
   // Example with all operators
   int a = 5, b = 10, c = 2;
   int result;
   // Scope resolution (::)
   int x = 20;
   int y = ::x;
   cout << "Value of y using scope resolution: " << y << endl;
   // Type casting ((type))
   double num1 = 10.5;
   int num2 = static_cast<int>(num1);
   cout << "Converted integer value: " << num2 << endl;
   // Increment and decrement (++ --)
   int num = 5;
   cout << "Original value of num: " << num << endl;
   cout << "Postfix increment: " << num++ << endl;
```

```
    cout << "After postfix increment: " << num << endl;
    // Sizeof (sizeof)
    cout << "Size of int data type: " << sizeof(int) << " bytes" << endl;
    // Dynamic memory allocation and deallocation (new delete)
    int *ptr = new int(10);
    cout << "Value stored in dynamically allocated memory: " << *ptr << endl;
    delete ptr;
    // Unary operators (+ - ! ~)
    int value = 5;
    cout << "Unary plus: " << +value << endl;
    cout << "Unary minus: " << -value << endl;
    bool flag = true;
    cout << "Logical NOT: " << !flag << endl;
    cout << "Bitwise NOT: " << ~value << endl;
    // Multiplication, division, modulus (* / %)
    result = a * b / c % 3;
    cout << "Result of multiplication, division, modulus: " << result << endl;
    return 0;
}
```

Output:

```
Value of y using scope resolution: 20
Converted integer value: 10
Original value of num: 5
Postfix increment: 5
After postfix increment: 6
Size of int data type: 4 bytes
Value stored in dynamically allocated memory: 10
Unary plus: 5
Unary minus: -5
Logical NOT: 0
Bitwise NOT: -6
Result of multiplication, division, modulus: 1
```

Program-2:

```
#include <iostream>
using namespace std;
int main()
{
   int x;       // variable declaration.
   x=(3/2) + 2;  // constant expression
   cout<<"Value of x is : "<<x;  // displaying the value of x.
   return 0;
}
```

Output:

```
Value of x is : 3
```

Program-3:

```
#include <iostream>
using namespace std;
int main()
{
 int a=4;    // variable declaration
 int b=5;    // variable declaration
 int x=3;    // variable declaration
 int y=6;   // variable declaration
 cout<<((a+b)>=(x+y));   // relational expression
 return 0;
}
```

Output: 1

Program:

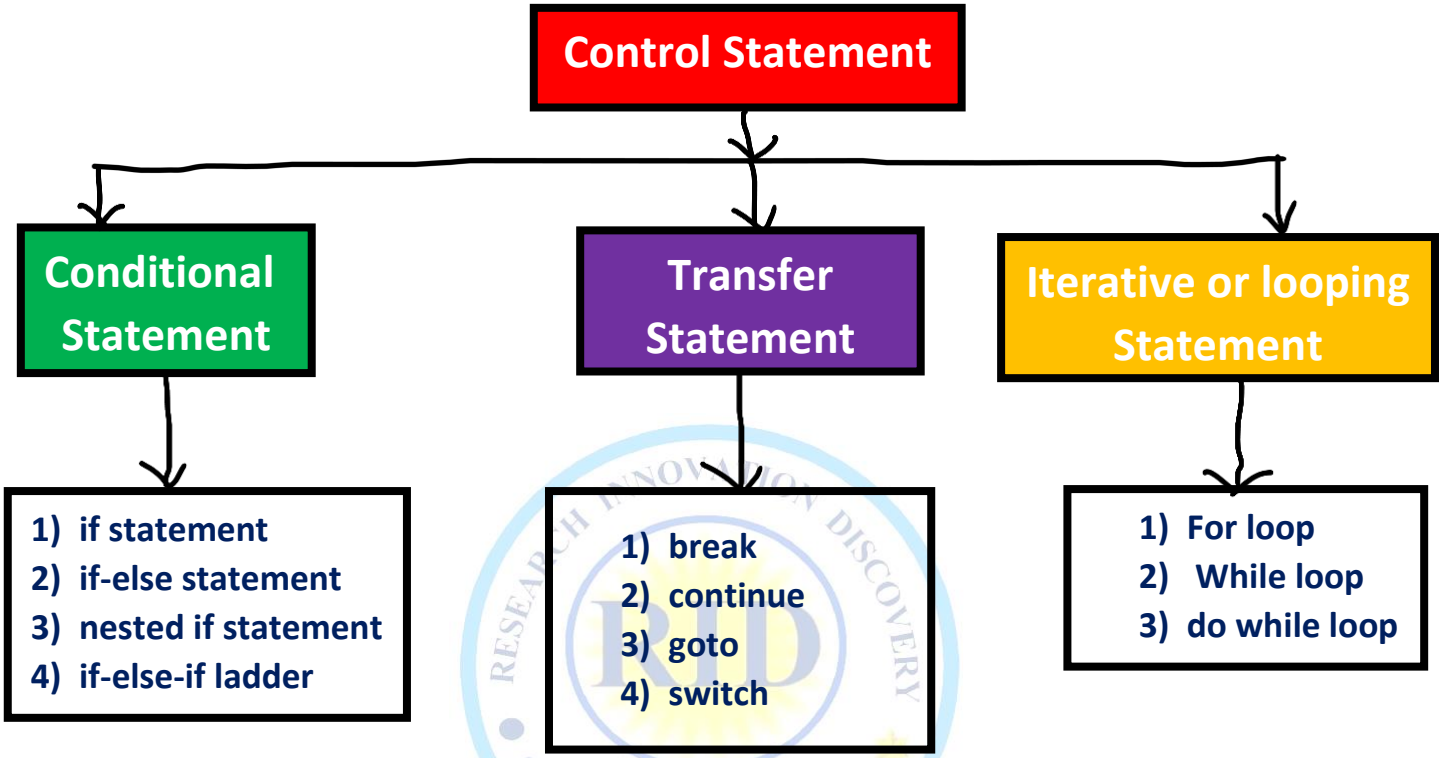
```
#include <iostream>
using namespace std;
int main()
{
 int a;  // variable declaration
 int b;  // variable declaration
 a=10+(b=90);  // embedded assignment expression
 std::cout <<"Values of 'a' is " <<a<< std::endl;
 return 0;
}
```

Output:

```
Values of 'a' is 100
```

CONTROL STATEMENT

- Control Statement or Flow control describe the order in which statements will be executed at runtime.
- To change the programing follow based on the condition we will used control statement.
- This is used for Decision making.



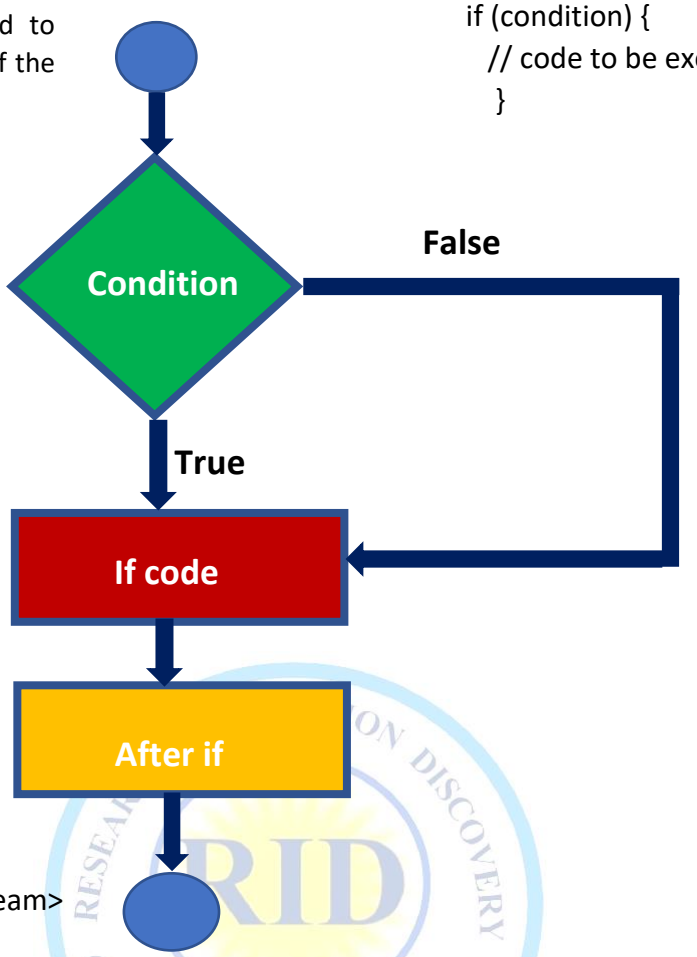
1. Conditional Statement:

- A conditional statement in C++ is a programming construct that allows executing different blocks of code based on the evaluation of a condition.
- There are following conditional statement in C++
    1. if statement
    2. if-else statement
    3. nested if statement
    4. if-else-if ladder

1) if statement:

- The if statement is used to execute a block of code if the condition is true.

Flow Chart:



Syntax:

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

Example-1:

```
#include <iostream>
int main() {
   int A = 10;
   if (A > 5) {
      std::cout << "A is greater than 6" << std::endl;
   }
   return 0;
}
```

Output:

A is greater than 6

Example-2:

```
#include <iostream>
using namespace std;
int main () {
  int num = 10;
        if (num % 2 == 0)
        {
           cout<<"It is even number";
        }
  return 0;
}
```

Output:

It is even number

Example-3

```
#include <iostream>
using namespace std;
int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;  // Input from user
    if (num > 0) {  // Checking condition
       cout << "The number is positive." << endl;
   }
   return 0;
}
```

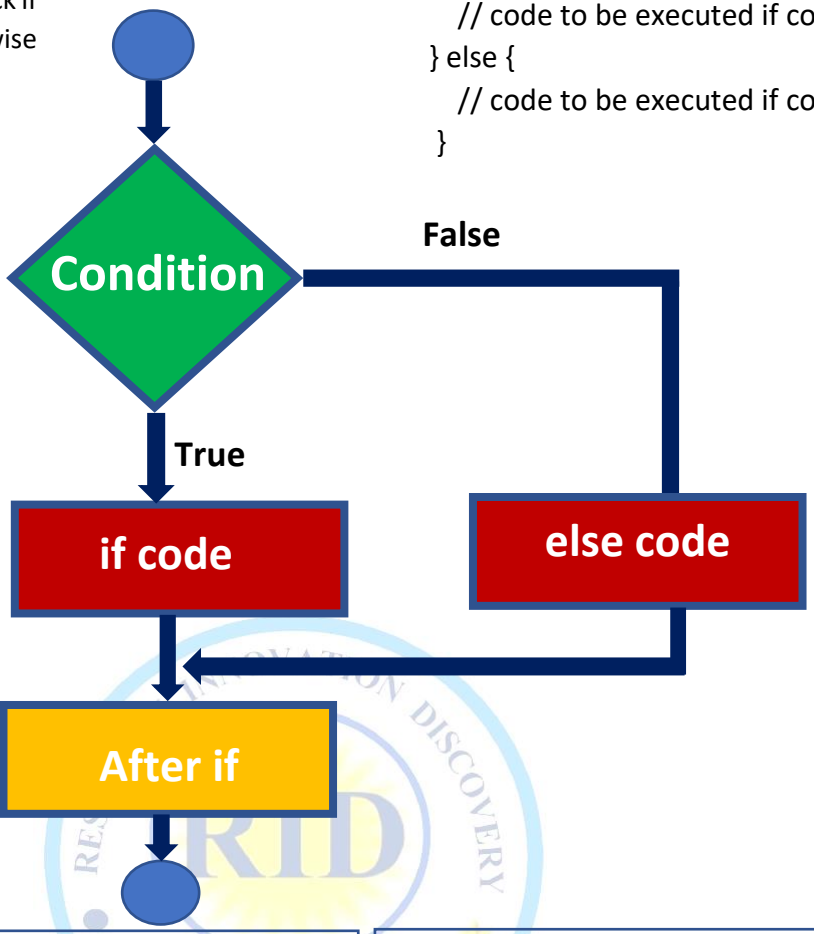
Output:

Enter a number: 5
The number is positive.

2) if else statement:

- if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed.

Flow Chart:



Syntax:

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

Example-1: Check a number is positive or negative

Program:

```
#include <iostream>
using namespace std;
int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;    // Input from user
   if (num > 0) { // Checking if the number is
positive or negative
      cout << "The number is positive." << endl;
   } else {
cout << "The number is negative or zero." << endl;
   }
   return 0; }
```

Output:

```
Enter a number: 6
The number is positive.
Enter a number: -3
The number is negative or zero.
```

Example-2: Check a number is odd or even

Program:

```
#include <iostream>
using namespace std;
int main () {
  int num = 13;
        if (num % 2 == 0)
        {
           cout<<"It is even number";
        }
        else
        {
           cout<<"It is odd number";
        }
  return 0;
      }
```

Output:

```
It is odd number
```

Example-3: Check a number is even or odd (user)

```
#include <iostream>
using namespace std;
int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;
   if (num % 2 == 0) {
      cout << "The number is even." << endl;
   } else {
      cout << "The number is odd." << endl;
   }
   return 0;
}
```

Output :

```
Enter a number: 6
The number is even.
Enter a number: 7
The number is odd.
```

Example 4: Check a number is divisible by 5 or not

```
#include <iostream>
using namespace std;
int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;
   if (num % 5 == 0) {
      cout << "The number is divisible by 5." << endl;
   } else {
      cout << "The number is not divisible by 5." << endl;
   }
   return 0;
}
```

Output:

```
Enter a number: 10
The number is divisible by 5.
Enter a number: 7
The number is not divisible by 5.
```

Example 5: Checka number is between 1 and 100 (inclusive)

```
#include <iostream>
using namespace std;
int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;
   if (num >= 1 && num <= 100) {
      cout << "The number is between 1 and 100."
<< endl;
   } else {
      cout << "The number is not between 1 and
100." << endl;
   }
   return 0;
}
```

Output:

```
Enter a number: 50
The number is between 1 and 100.
Enter a number: 150
The number is not between 1 and 100.
```

Example 6: Check a number is a multiple of both 3 and 5

```
#include <iostream>
using namespace std;
int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;
   if (num % 3 == 0 && num % 5 == 0) {
      cout << "The number is a multiple of both 3 and
5." << endl;
   } else {
      cout << "The number is not a multiple of both 3
and 5." << endl;
   }
   return 0;
}
```

Output:

```
Enter a number: 15
The number is a multiple of both 3 and 5.
Enter a number: 9
The number is not a multiple of both 3 and
```

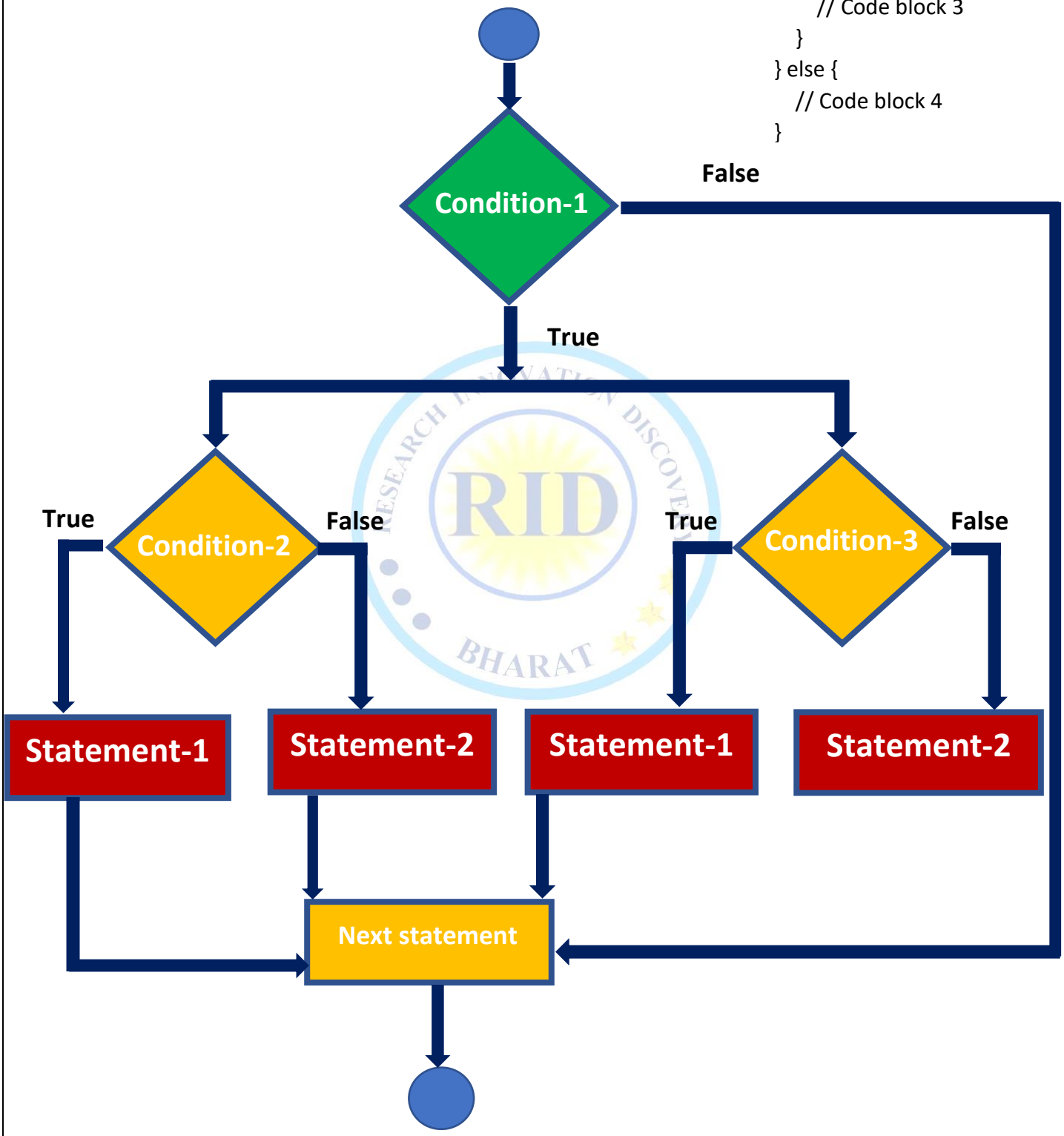
3) nested if else statement:

- A nested if statement in C++ is a conditional construct where an if statement is placed inside another if statement or inside an if-else statement.

Syntax:

```
if (condition1) {
   // Code block 1
   if (condition2) {
      // Code block 2
   } else {
      // Code block 3
   }
} else {
   // Code block 4
}
```

Flow Chart:



Example 1: Determining eligibility for voting.

```
#include <iostream>
using namespace std;
int main() {
   int age;
   cout << "Enter your age: ";
   cin >> age;
   if (age >= 18) { cout << "You are eligible to vote." << endl;}  else
{
      if (age >= 16) {
         cout << "You are eligible to vote in some countries
with parental consent." << endl;
      } else {
     cout << "You are not eligible to vote yet." << endl;
      }}
   return 0;}
```

Output:

```
Enter your age: 20
You are eligible to vote.
```

Example 2: Checking the eligibility for a discount based on purchase amount

```
#include <iostream>
using namespace std;
int main() {
   double purchaseAmount;
   cout << "Enter the purchase amount: $";
   cin >> purchaseAmount;
   if (purchaseAmount >= 1000) {
      if (purchaseAmount >= 2000) {
         cout << "You are eligible for a 10%
discount." << endl;} else {
         cout << "You are eligible for a 5%
discount." << endl;  }
   } else {
      cout << "You are not eligible for any
discount." << endl;}
   return 0; }
```

Output:

```
Enter the purchase amount: $1500
You are eligible for a 5% discount.
```

Example 3: Determining the quadrant of a point on the coordinate plane

```
#include <iostream>
using namespace std;
int main() {
   int x, y;
   cout << "Enter the x-coordinate: ";
   cin >> x;
   cout << "Enter the y-coordinate: ";
   cin >> y;
   if (x > 0) {
      if (y > 0) { cout << "The point is in Quadrant I." << endl;
      } else if (y < 0) { cout << "The point is in Quadrant IV." << endl;
      } else {
         cout << "The point is on the positive x-axis." << endl;}
   } else if (x < 0) {
      if (y > 0) {cout << "The point is in Quadrant II." << endl;
      } else if (y < 0) {
         cout << "The point is in Quadrant III." << endl;
      } else {
         cout << "The point is on the negative x-axis." << endl;}
   } else {
      if (y != 0) {cout << "The point is on the y-axis." << endl;
      } else {
         cout << "The point is at the origin." << endl; } }
   return 0;}
```

Output:

```
Enter the x-coordinate: 3
Enter the y-coordinate: 4
The point is in Quadrant I.
```

Example 4: Checking if a number is divisible by 5, 7, or neither

```
#include <iostream>
using namespace std;
int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;
   if (num % 5 == 0) {
cout << "The number is divisible by 5." << endl;
   } else if (num % 7 == 0) {
   cout << "The number is divisible by
   7." << endl;
   } else {
      cout << "The number is not
divisible by 5 or 7." << endl;
   }
   return 0;
}
```

Output:

```
Enter a number: 15
The number is divisible by 5.
```

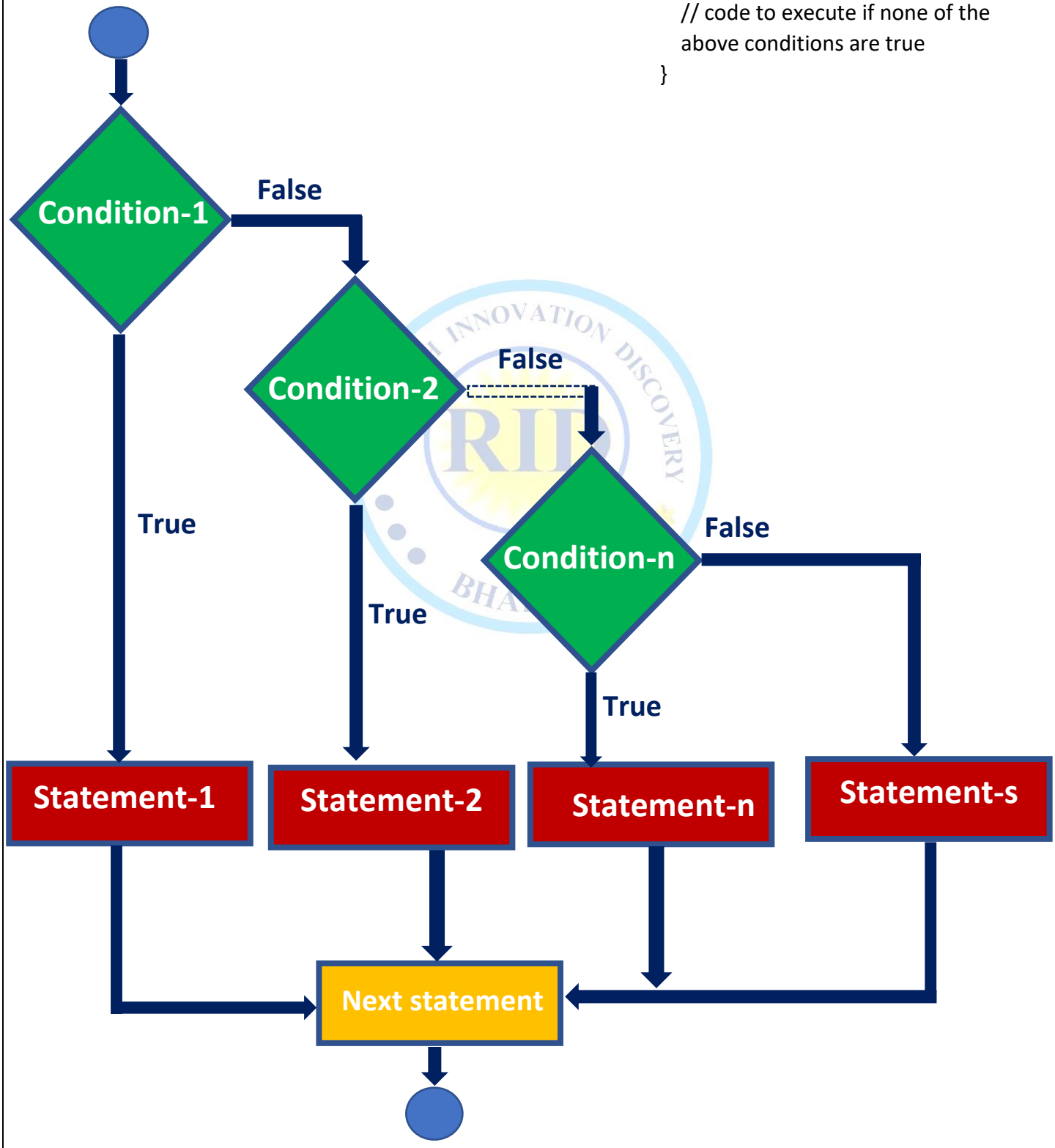
4) if else if ladder statement:

- The if-else-if ladder statement is a series of if-else statements where each 'if' condition is checked sequentially until a true condition is found, or the final else block is executed if none of the conditions are true.

Syntax:

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

Flow Chart:



Example 1: Checking if a number is positive, negative, or zero

```
#include <iostream>
using namespace std;
int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;
   if (num > 0) {
      cout << "The number is positive." << endl;
   } else if (num < 0) {
      cout << "The number is negative." << endl;
   } else {
      cout << "The number is zero." << endl;
   }
   return 0;
}
```

Output:

```
Enter a number: 5
The number is positive.
Enter a number: -3
The number is negative.
```

Example 2: Checking the grade based on marks

```
#include <iostream>
using namespace std;
int main() {
   int marks;
   cout << "Enter your marks: ";
   cin >> marks;
   if (marks >= 90) {
      cout << "Grade: A" << endl;
   } else if (marks >= 80) {
      cout << "Grade: B" << endl;
   } else if (marks >= 70) {
      cout << "Grade: C" << endl;
   } else if (marks >= 60) {
      cout << "Grade: D" << endl;
   } else {
      cout << "Grade: F" << endl;
   }
   return 0; }
```

Output:

```
Enter your marks: 95
Grade: A
```

Example 3: Determining the season based on month number

```
#include <iostream>
using namespace std;
int main() {
   int month;
   cout << "Enter the month number (1-12): ";
   cin >> month;
   if (month >= 3 && month <= 5) {
      cout << "Spring" << endl;
   } else if (month >= 6 && month <= 8) {
      cout << "Summer" << endl;
   } else if (month >= 9 && month <= 11) {
      cout << "Autumn" << endl;
   } else {
      cout << "Winter" << endl;
   }
   return 0;
}
```

Output:

```
Enter the month number (1-12): 7
Summer The number is negative.
```

Example 4: Checking if a number is divisible by 2, 3, or neither

```
#include <iostream>
using namespace std;
int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;
   if (num % 2 == 0) {
cout << "The number is divisible by 2." << endl;
   } else if (num % 3 == 0) {
   cout << "The number is divisible by
   3." << endl;
   } else {
      cout << "The number is not
divisible by 2 or 3." << endl;
   }
   return 0;
}
```

Output:

```
Enter a number: 6
The number is divisible by 2.
```

Example 5: Finding the largest of three numbers

```
#include <iostream>
using namespace std;
int main() {
   int num1, num2, num3;
   cout << "Enter three numbers: ";
   cin >> num1 >> num2 >> num3;
   if (num1 >= num2 && num1 >= num3) {
      cout << "The largest number is: " << num1 <<
endl;
   } else if (num2 >= num1 && num2 >= num3) {
      cout << "The largest number is: " << num2 <<
endl;
   } else {
      cout << "The largest number is: " << num3 <<
endl;
   }
   return 0;
}
```

Output:

```
Enter three numbers: 10 20 15
The largest number is: 20
```

Example 6: Determining the time of the day.

```
#include <iostream>
using namespace std;
int main() {
   int hour;
   cout << "Enter the hour (in 24-hour
format): ";
   cin >> hour;
   if (hour >= 0 && hour < 12) {
      cout << "Good morning!" << endl;
   } else if (hour >= 12 && hour < 18) {
      cout << "Good afternoon!" << endl;
   } else if (hour >= 18 && hour < 24) {
      cout << "Good evening!" << endl;
   } else {
      cout << "Invalid hour" << endl;
   }
   return 0;
}
```

Output:

```
Enter a number: 6
The number is divisible by 2.
```

Example-7: Checking the type of triangle based on sides.

```
#include <iostream>
using namespace std;
int main() {
   int side1, side2, side3;
   cout << "Enter three sides of the triangle: ";
   cin >> side1 >> side2 >> side3;
   if (side1 == side2 && side2 == side3) {
      cout << "Equilateral triangle" << endl;
   } else if (side1 == side2 || side2 == side3 || side1 == side3) {
      cout << "Isosceles triangle" << endl;
   } else {
      cout << "Scalene triangle" << endl;
   }
   return 0;
}
```

Output:

```
Enter three sides of the triangle: 5 5 5
Equilateral triangle
Enter three sides of the triangle: 3 4 4
Isosceles triangle
```

Example-8: Sorting three numbers in ascending order.

Program:

```
#include <iostream>
using namespace std;
int main() {
   int num1, num2, num3;
   cout << "Enter three numbers: ";
   cin >> num1 >> num2 >> num3;
   if (num1 <= num2) {
      if (num2 <= num3) {
         cout << "Sorted numbers: " << num1 << " " << num2 << " " << num3 << endl;
      } else {
         if (num1 <= num3) {
            cout << "Sorted numbers: " << num1 << " " << num3 << " " << num2 << endl;
         } else {
            cout << "Sorted numbers: " << num3 << " " << num1 << " " << num2 << endl;
         }
      }
   } else {
      if (num1 <= num3) {
         cout << "Sorted numbers: " << num2 << " " << num1 << " " << num3 << endl;
      } else {
         if (num2 <= num3) {
            cout << "Sorted numbers: " << num2 << " " << num3 << " " << num1 << endl;
         } else {
            cout << "Sorted numbers: " << num3 << " " << num2 << " " << num1 << endl;
         }
      }
   }
   return 0;
}
```

Output:

```
Enter three numbers: 5 2 7
Sorted numbers: 2 5 7
```

❖ iterative or looping statement:

- An iterative or looping statement in C++ is a programming construct that allows the execution of a block of code repeatedly based on a specified condition.
- There are three types of iterative or looping statement in C++.
    1. **for loop**
    2. **while loop**
    3. **do-while loop**

1. For Loop:

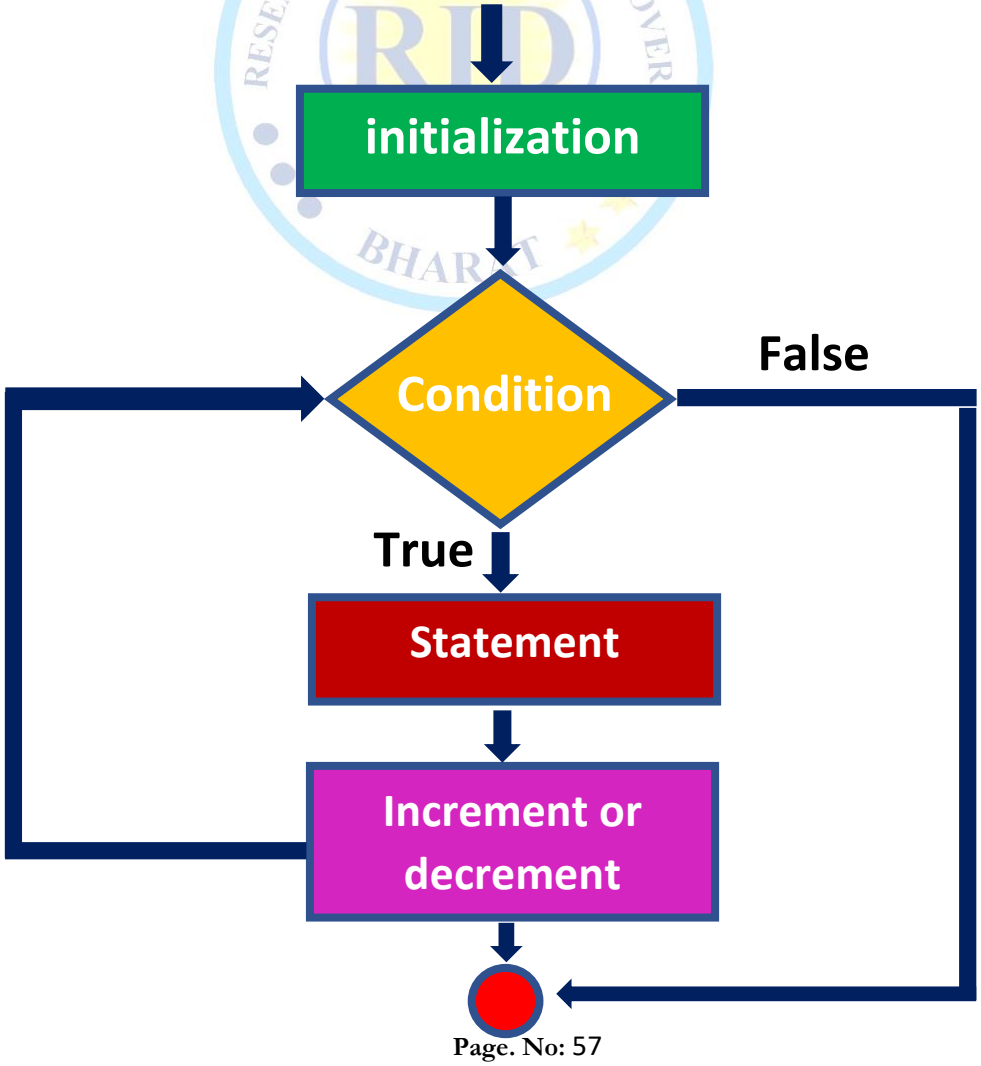
- for loop is a control flow statement that allows you to execute a block of code repeatedly based on a specified condition.
- It consists of three parts.
    1. Initialization:
    2. Condition:
    3. Increment/Decrement:

Syntax:

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

- **initialization:** It is used to initialize loop control variable.
- **condition:** It is expression that is evaluated before each iteration of loop. If it evaluates to true, loop continues; otherwise, it terminates.
- **increment/decrement:** It updates loop control variable after each iteration.

Flowchart:



Example-1: Printing numbers from 1 to 10

```
#include <iostream>
using namespace std;
int main() {
    for(int i=1;i<=10;i++){
      cout<<i <<" , ";
     }
  }
```

Output:

1,2,3,4,5,6,7,8,9

Example-2: Printing numbers in reverse order from 10 to 1.

```
#include <iostream>
using namespace std;
int main() {
   for (int i = 10; i >= 1; --i) {
      cout << i << " ";
   }
   return 0;
}
```

Output:

10 9 8 7 6 5 4 3 2 1

Example-3: Printing squares of numbers from 1 to 5.

```
#include <iostream>
using namespace std;
int main() {
   for (int i = 1; i <= 5; ++i) {
      cout << i * i << " ";
   }
   return 0;
}
```

Example-4: Computing the factorial of a number.

```
#include <iostream>
using namespace std;
int main() {
   int n = 5;
   int factorial = 1;
   for (int i = 1; i <= n; ++i) {
      factorial *= i;
   }
   cout << "Factorial of " << n << " is " << factorial << endl;
   return 0;
}
```

Output:

Factorial of 5 is 120

Example-5: Printing Fibonacci series

```
#include <iostream>
using namespace std;
int main() {
   int n = 10;
   int prev = 0, curr = 1, next;
   cout << "Fibonacci Series: ";
   for (int i = 1; i <= n; ++i) {
      cout << prev << " ";
      next = prev + curr;
      prev = curr;
      curr = next;
   }
   return 0; }
```

Output: Fibonacci Series: 0 1 1 2 3 5 8 13 21 34

Example-6: Printing ASCII characters from 'A' to 'Z'.

```
#include <iostream>
using namespace std;
int main() {
   cout << "ASCII Characters: ";
   for (char c = 'A'; c <= 'Z'; ++c) {
      cout << c << " ";
   }
   return 0;
}
```

Output: 10 9 8 7 6 5 4 3 2 1

Example-7: print the prime numbers between 1 and 20.

```
#include <iostream>
using namespace std;
int main() {
   cout << "Prime numbers between 1 and 20: ";
   for (int num = 2; num <= 20; ++num) {
      bool isPrime = true;
      for (int i = 2; i <= num / 2; ++i) {
         if (num % i == 0) {
            isPrime = false;
            break;
         } }
      if (isPrime) {
         cout << num << " ";
      }}
   return 0;
}
```

Output:

Prime numbers between 1 and 20: 2 3 5 7 11 13 17 19

Types of for loop:

- There are following types of for loop.
  1. Nested for loop
  2. Infinite for loop

❖ Nested for loop.

- A nested for loop in C++ is a control flow statement where one for loop is placed inside another for loop, allowing iteration over multiple dimensions or creating complex loop structures.

Syntax:

```
for (initialization; condition; update) {
   for (initialization; condition; update) {
      // Inner loop code
   }
   // Outer loop code
}
```

Example-1: Printing a square pattern

```
#include <iostream>
int main() {
   for (int i = 0; i < 5; ++i) {
      for (int j = 0; j < 5; ++j) {
         std::cout << "* ";
      }
      std::cout << std::endl;
   }
   return 0;
}
```

Output:

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Example-2: Printing a right-angled triangle pattern

```
#include <iostream>
int main() {
   for (int i = 0; i < 5; ++i) {
      for (int j = 0; j <= i; ++j) {
         std::cout << "* ";
      }
      std::cout << std::endl;
   }
   return 0; }
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

Example-3: Printing a hollow square pattern.

```
#include <iostream>
int main() {
   int size = 5;
   for (int i = 0; i < size; ++i) {
      for (int j = 0; j < size; ++j) {
         if (i == 0 || i == size - 1 || j == 0 || j == size - 1)
            std::cout << "* ";
         else
            std::cout << "  ";
      }
      std::cout << std::endl;
   }
   return 0;
}
```

Output:

```
* * * * *
*       *
*       *
*       *
* * * * *
```

Example-4: Printing a mirrored right-angled triangle pattern

```
#include <iostream>
int main() {
   for (int i = 0; i < 5; ++i) {
      for (int j = 0; j < 5 - i; ++j) {
         std::cout << "* ";
      }
      std::cout << std::endl;
   }
   return 0;
}
```

Output:

```
* * * * *
* * * *
* * *
* *
*
```

Example-5: Printing a half pyramid of numbers.

```
#include <iostream>
int main() {
   for (int i = 1; i <= 5; ++i) {
      for (int j = 1; j <= i; ++j) {
         std::cout << j << " ";
      }
      std::cout << std::endl;
   }
   return 0;
}
```

Example-3: Printing a hollow right-angled triangle pattern.

```
#include <iostream>
int main() {
   for (int i = 0; i < 5; ++i) {
      for (int j = 0; j <= i; ++j) {
         if (j == 0 || i == 4 || i == j)
            std::cout << "* ";
         else
            std::cout << "  ";
      }
      std::cout << std::endl;
   }
   return 0;
}
```

Output:

```
*
* *
*  *
*    *
* * * * *
```

Example-4: Printing a reversed hollow right-angled triangle pattern:

```
#include <iostream>
int main() {
   for (int i = 0; i < 5; ++i) {
      for (int j = 0; j < 5 - i; ++j) {
         if (j == 0 || i == 4 || i == j)
            std::cout << "* ";
         else
            std::cout << "  ";
      }
      std::cout << std::endl;
   }
   return 0;
}
```

Output:

```
* * * * *
*   *
*  *
* *
*
```

Example-5: Printing a pyramid pattern.

```
#include <iostream>
int main() {
   for (int i = 1; i <= 5; ++i) {
      for (int j = 1; j <= 5 - i; ++j)
         std::cout << " ";
      for (int k = 1; k <= 2 * i - 1; ++k)
         std::cout << "*";
      std::cout << std::endl;
   }
   return 0;
}
```

Output:

```
    *
   ***
  *****
 *******
*********
```

❖ Infinite for loop:

- An infinite loop is a loop construct that continually executes its body without ever halting, typically because its termination condition is never met.

Syntax:

```
for (;;) {
   // Loop body
}
```

Example:

```
#include <iostream>
int main() {
   // Infinite for loop
   std::cout << "Infinite loop using for:" << std::endl;
   for (;;) {
      std::cout << i << " ";
      ++i;
      if (i > 20)
         break; // Breaking the loop after printing 20 numbers
   }
   std::cout << std::endl;
   return 0;
}
```

Output:

```
Infinite loop using for:
11 12 13 14 15 16 17 18 19 20
```

Example-1: Print multiplication table of a number

```
#include <iostream>
int main() {
   int number = 7;
   for (int i = 1; i <= 10; ++i) {
      std::cout << number << " * "
<< i << " = " << (number * i) <<
std::endl;
   }
   return 0;
}
```

Output:

```
7 * 1 = 7
……..
7 * 9 = 63
7 * 10 = 70
```

Example-2: Print characters of a string.

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   for (char c : str) {
      std::cout << c << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output:

```
H e l l o
```

Example-3: Print even numbers using continue statement.

```
#include <iostream>
int main() {
   for (int i = 1; i <= 10; ++i) {
      if (i % 2 != 0)
         continue;
      std::cout << i << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output:

```
2 4 6 8 10
```

Example-4: Find the sum of digits of a number.

```
#include <iostream>
int main() {
   int number = 12345;
   int sum = 0;
   for (int n = number; n > 0; n /= 10) {
      sum += n % 10;
   }
   std::cout << "Sum of digits of " << number << ": " << sum << std::endl;
   return 0;
}
```

Output: Sum of digits of 12345: 15

Example-5: Check if a number is prime or notZ'.

```
#include <iostream>
int main() {
   int number = 17;
   bool isPrime = true;
   for (int i = 2; i <= number / 2; ++i) {
      if (number % i == 0) {
         isPrime = false;
         break;
      } }
   if (isPrime)
      std::cout << number << " is prime." << std::endl;
   else
      std::cout << number << " is not prime." << std::endl;
   return 0;
}
```

Output: 17 is prime.

Example-6: Calculate the power of a number (e.g., 2^5).

```
#include <iostream>
int main() {
   int base = 2;
   int exponent = 5;
   int result = 1;
   for (int i = 1; i <= exponent; ++i) {
      result *= base;
   }
   std::cout << base << " raised to the power of " << exponent << " is
" << result << std::endl;
   return 0;
}
```

Output:

```
2 raised to the power of 5 is 32
```

Example-7: Calculate the (GCD) of two numbers

```
#include <iostream>
int main() {
   int num1 = 12, num2 = 18;
   while (num1 != num2) {
      if (num1 > num2)
         num1 -= num2;
      else
         num2 -= num1;
   }
   std::cout << "GCD: " << num1 << std::endl;
   return 0;
}
```

Output: GCD: 6

Example-8: Check if a number is palindrome or not.

```
#include <iostream>
int main() {
   int number = 12321;
   int originalNumber = number;
   int reversed = 0;
   while (number != 0) {
      int digit = number % 10;
      reversed = reversed * 10 + digit;
      number /= 10;
   }
   if (originalNumber == reversed)
      std::cout << originalNumber << " is a palindrome."
<< std::endl;
   else
      std::cout << originalNumber << " is not a
palindrome." << std::endl;
   return 0;
}
```

Output: 12321 is a palindrome.

Example-9: Generate a random number b/w 1 and 100.

```
#include <iostream>
#include <cstdlib> // For rand() and srand()
#include <ctime>   // For time()
int main() {
   srand(time(nullptr));
   int randomNumber = rand() % 100 + 1;
   std::cout << "Random number: " << randomNumber <<
std::endl;
   return 0;
}
```

Output: Random number: (any random number between 1 and 100)

Example-10: Count the number of vowels in a string.

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello World";
   int countVowels = 0;
   for (char c : str) {
      if (c == 'a' || c == 'e' || c == 'i' || c == 'o' ||
c == 'u' ||
         c == 'A' || c == 'E' || c == 'I' || c == 'O' ||
c == 'U') {
         countVowels++;
      } }
   std::cout << "Number of vowels: " <<
countVowels << std::endl;
   return 0;
}
```

Output: Number of vowels: 3

Example-6: Check if a number is Armstrong number or not.

```
#include <iostream>
#include <cmath>
int main() {
   int number = 153;
   int originalNumber = number;
   int numDigits = 0;
   int sum = 0;
   for (int temp = number; temp != 0; temp /= 10) {
      ++numDigits;
   }
   for (int temp = number; temp != 0; temp /= 10) {
      int digit = temp % 10;
      sum += pow(digit, numDigits);
   }
   if (sum == originalNumber) {
      std::cout << originalNumber << " is an
Armstrong number." << std::endl;
   } else {
      std::cout << originalNumber << " is not an
Armstrong number." << std::endl;
   }
   return 0;
}
```

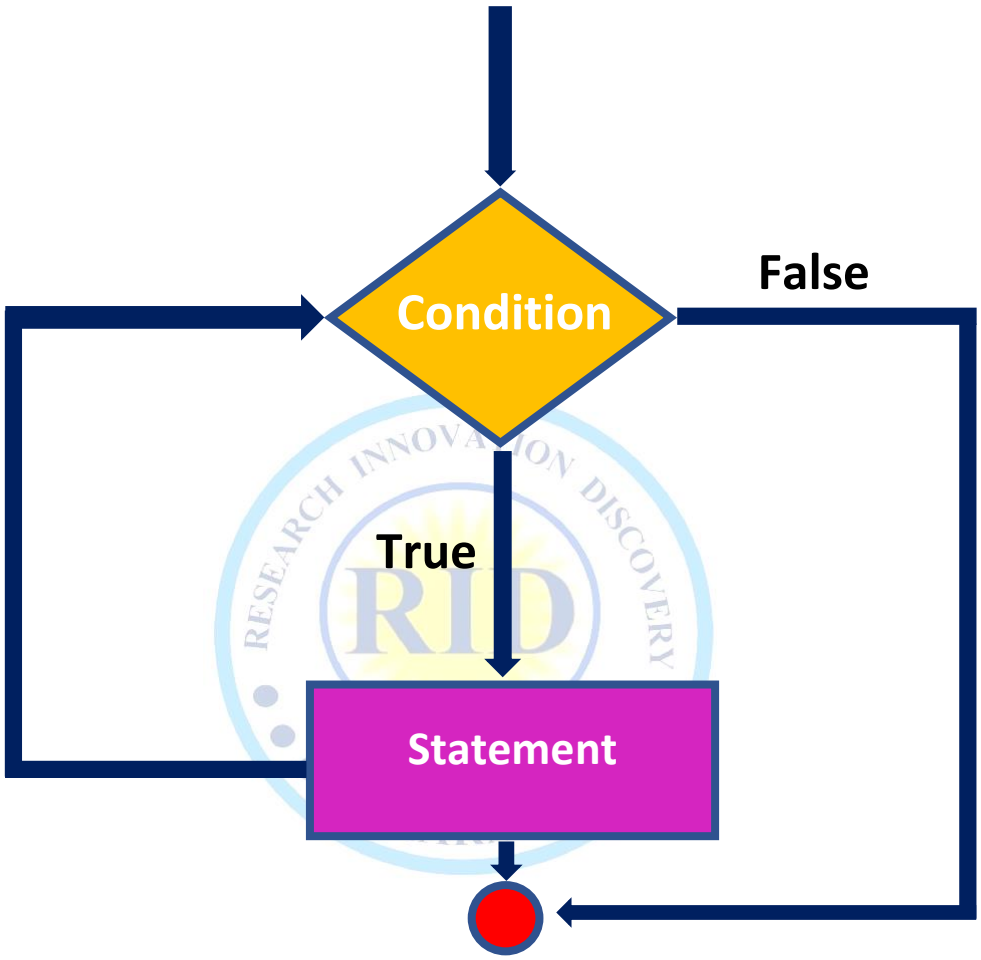
Output:

153 is an Armstrong number.

❖ While loop:

- A while loop in C++ is a control flow statement that allows a block of code to be executed repeatedly as long as a specified condition evaluates to true. It continues execution until the condition becomes false.

Flowchart:



Example-1: Print numbers from 1 to 5.

```
#include <iostream>
int main() {
   int i = 1;
   while (i <= 5) {
      std::cout << i << " ";
      ++i;
   }
   std::cout << std::endl;
   return 0;
}
```

Output:

1 2 3 4 5

Example-2: Print even numbers from 2 to 10.

```
#include <iostream>
int main() {
   int num = 2;
   while (num <= 10) {
      std::cout << num << " ";
      num += 2;
   }
   std::cout << std::endl;
   return 0;
}
```

Output: 2 4 6 8 10

Example-3: Find the factorial of a number.

```
#include <iostream>
int main() {
   int number = 5;
   int factorial = 1;
   int i = 1;
   while (i <= number) {
      factorial *= i;
      ++i;
   }
   std::cout << "Factorial of " << number << ": " <<
factorial << std::endl;
   return 0;
}
```

Output: Factorial of 5: 120.

Example-4: Print Fibonacci series up to 10 terms.

```
#include <iostream>
int main() {
   int n1 = 0, n2 = 1, nextTerm = 0;
   int i = 1;
   while (i <= 10) {
      std::cout << n1 << " ";
      nextTerm = n1 + n2;
      n1 = n2;
      n2 = nextTerm;
      ++i;
   }
   std::cout << std::endl;
   return 0;
}
}
```

Output:

0 1 1 2 3 5 8 13 21 34

Example-10: Check if a number is prime or not.

```
#include <iostream>
int main() {
   int number = 13     ;
   int i = 2;
   bool isPrime = true;
   while (i <= number / 2) {
      if (number % i == 0) {
         isPrime = false;
         break;
      }
      ++i;
   }
   if (isPrime)
      std::cout << number << " is prime." <<
std::endl;
   else
      std::cout << number << " is not prime." <<
std::endl;
   return 0;
}
```

Output: 13 is prime.

Example-6: Calculate the sum of digits of a number.

```
#include <iostream>
int main() {
   int number = 12345;
   int sum = 0;
   int temp = number;
   while (temp != 0) {
      sum += temp % 10;
      temp /= 10;
   }
   std::cout << "Sum of digits of " << number <<
": " << sum << std::endl;
   return 0;
}
```

Output:

Sum of digits of 12345: 15.

❖ Types of for loop:

- There are following types of for loop.
  1. Nested while loop
  2. Infinite while loop

❖ Nested for loop.

- A nested while loop in C++ is a loop inside another loop. It allows you to execute a block of code repeatedly within another block of code. Nested while loops are useful for iterating through two-dimensional arrays.
- **Syntax:**

```
while (condition1) {
   // Code to be executed
   while (condition2) {
      // Code to be executed
   }
}
```

Example-1: Print a pattern of stars in a right-angled triangle form.

```
#include <iostream>
int main() {
   int rows = 5;
   int i = 1;
   while (i <= rows) {
      int j = 1;
      while (j <= i) {
         std::cout << "* ";
         ++j;
      }
      std::cout << std::endl;
      ++i;
   }
   return 0;
}
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

Example-3: Print the multiplication table of numbers from 1 to 5.

```
#include <iostream>
int main() {
   int i = 2;
   while (i <= 6) {
      int j = 1;
      while (j <= 10) {
         std::cout << i << " * " << j << " = " << (i * j) <<
std::endl;
         ++j;
      }
      std::cout << std::endl;
      ++i;
   }
   return 0;
}
```

Output:

```
2 * 1 = 2
.....
2 * 9 = 18
2 * 10 = 20
.....
.....
```

❖ Infinite while loop:

- An infinite while loop is a loop that runs indefinitely, executing its body repeatedly without terminating.

Syntax:

```
while (true) {
   // Loop body
}
```

Example:

```
#include <iostream>
int main() {
   // Infinite while loop
   std::cout << "Infinite while loop:" << std::endl;
   int i = 1;
   while (true) {
      std::cout << i << " ";
      ++i;
      if (i > 10)
         break; // Breaking the loop after printing 10 numbers
   }
   std::cout << std::endl;
   return 0;
}
```

Output:

```
Infinite while loop:
1 2 3 4 5 6 7 8 9 10
```

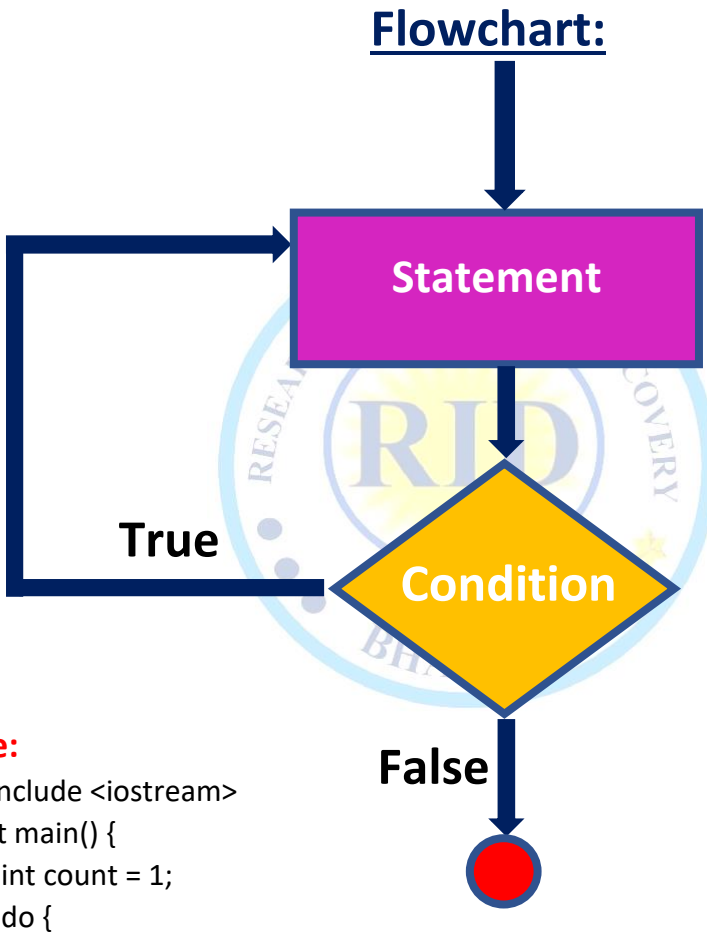
❖ Do-While loop:

- A Do-While loop in C++ is a type of loop control statement that executes a block of code repeatedly until a specified condition evaluates to false.
- The key difference between a Do-While loop and a While loop is that the Do-While loop always executes the block of code at least once, even if the condition is initially false.

Syntax:

```
do {
  // block of code to be executed
} while (condition);
```

Flowchart:



Example:

```
#include <iostream>
int main() {
  int count = 1;
  do {
    std::cout << "Count: " << count << std::endl;
    count++;
  } while (count <= 5);
  return 0;
}
```

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

❖ Types of do while loop:

- There are following types of for loop.
    1. Nested do while loop
    2. Infinite do while loop

❖ Nested do while loop:

- A nested Do-While loop in C++ is a loop inside another loop. This structure allows for more complex looping patterns where one loop is contained within the body of another loop.

Syntax:

```
do {
   // outer loop block of code
   do {
      // inner loop block of code
   } while (condition_inner);
} while (condition_outer);
```

Example:

```
#include <iostream>
int main() {
   int outerCount = 1;
   do {
      int innerCount = 1;
      do {
         std::cout << "Outer Count: " << outerCount << ", Inner Count: " << innerCount
<< std::endl;
         innerCount++;
      } while (innerCount <= 3);
      outerCount++;
   } while (outerCount <= 2);
   return 0;
}
```

Output:

```
Outer Count: 1, Inner Count: 1
Outer Count: 1, Inner Count: 2
Outer Count: 1, Inner Count: 3
Outer Count: 2, Inner Count: 1
Outer Count: 2, Inner Count: 2
Outer Count: 2, Inner Count: 3
```

❖infinite Do-While loop:

- An infinite Do-While loop in C++ is a loop that continues to execute indefinitely, without an explicit condition for termination. It relies on other mechanisms within the loop to control its execution, such as break or return statements.

Syntax:

```
do {
   // block of code to be executed indefinitely
} while (true);
```

Example-1:

```
#include <iostream>
int main() {
   int count = 1;
   do {
      std::cout << "Count: " << count << std::endl;
      count++;
      if (count > 5) // Adding a condition to break the loop after count exceeds 5
         break;
   } while (true);
   return 0;
}
```

Output: Count: 1
Count: 2
Count: 3
Count: 4
Count: 5

Example-2:

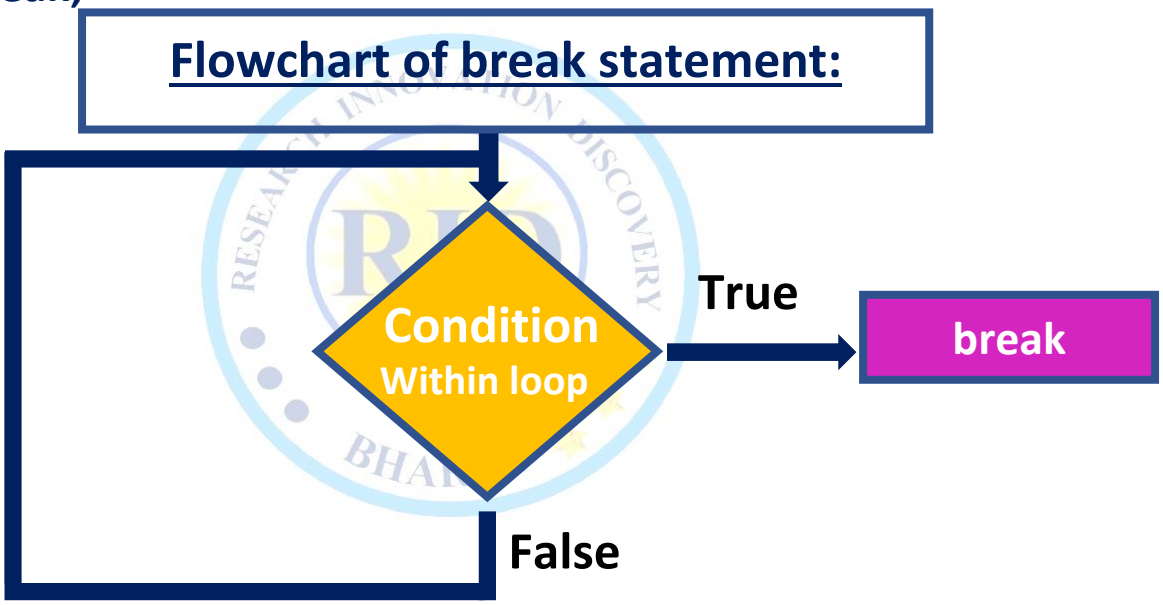
```
#include <iostream>
int main() {
   int number;
   do {
      std::cout << "Enter a number (-1 to exit): ";
      std::cin >> number;
      if (number == -1)
         break;
      std::cout << "You entered: " << number << std::endl;
   } while (true);
   std::cout << "Loop terminated." << std::endl;
   return 0;
}
```

Output: Enter a number (-1 to exit): 5
You entered: 5
Enter a number (-1 to exit): 10
You entered: 10
Enter a number (-1 to exit): -1
Loop terminated.

❖ Transfer statement:

- transfer statements are used to transfer control from one part of the program to another. They allow you to change the sequence of execution of your program based on certain conditions or requirements.
- There are several types of transfer statements in C++.
  1. **break**
  2. **continue**
  3. **goto**
  4. **switch**
  5. **throw**
- The break statement in C++ is used to exit or terminate the innermost loop statement in which it appears. It allows you to prematurely break out of the loop statement and resume execution from the next statement after the loop.

- **Break;**



Example-1: Exiting a loop based on a condition

```
#include <iostream>
int main() {
   for (int i = 1; i <= 10; ++i) {
      std::cout << i << " ";
      if (i == 5) {
         std::cout << "\nBreaking out of the loop.\n";
         break;
      } }
   return 0;
}
```

Output:

```
1 2 3 4 5
Breaking out of the loop.
```

Example-2: Exiting a loop when encountering a specific value.

```
#include <iostream>
int main() {
   int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   for (int i = 0; i < 10; ++i) {
      std::cout << numbers[i] << " ";
      if (numbers[i] == 5) {
         std::cout << "\nEncountered 5. Exiting the loop.\n";
         break;
      }
   }
   return 0;
}
```

Output:

```
1 2 3 4 5
Encountered 5. Exiting the loop.
```

Example-3: Breaking out of a nested loop

```
#include <iostream>
int main() {
   for (int i = 1; i <= 3; ++i) {
      std::cout << "Outer loop iteration: " << i << std::endl;
      for (int j = 1; j <= 3; ++j) {
         std::cout << "Inner loop iteration: " << j << std::endl;
         if (i + j == 4) {
            std::cout << "Sum of indices is 4. Breaking out of both loops.\n";
            break; // Exiting the inner loop
         }
      } }
   return 0;
}
```

Output:

```
Outer loop iteration: 1
Inner loop iteration: 1
Inner loop iteration: 2
Inner loop iteration: 3
Outer loop iteration: 2
Inner loop iteration: 1
Sum of indices is 4. Breaking out of both loops..
```

❖ Continue:

- The continue statement is used in C++ to skip the remaining code in the current iteration of a loop and proceed to the next iteration. It is typically used within loops to bypass certain iterations based on specific conditions.

Syntax:

```
continue;
```

Example-1:

```
#include <iostream>
int main() {
   for (int i = 1; i <= 5; ++i) {
      if (i == 3)
         continue;
      std::cout << i << " ";
   }
   return 0;
}
```

Output:

```
1 2 4 5
```

Example-2:

```
#include <iostream>
int main() {
   int sum = 0;
   for (int i = 1; i <= 10; ++i) {
      if (i % 2 == 0)
         continue;
      sum += i;
   }
   std::cout << "Sum of odd numbers from 1 to 10: " << sum << std::endl;
   return 0;
}
```

Output:

```
Sum of odd numbers from 1 to 10: 25
```

Example-3:

```
#include <iostream>
int main() {
   int count = 0;
   for (int i = 1; i <= 100; ++i) {
      if (i % 7 != 0)
         continue;
      ++count;
   }
   std::cout << "Count of numbers divisible by
7 from 1 to 100: " << count << std::endl;
   return 0; }
```

Output:

Count of numbers divisible by 7 from 1 to 100: 14

Example-4:

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

Output:

1 2 4 5

❖ goto statement:

- The goto statement in C++ is a jump statement that allows transferring the control of the program to a specified labeled statement within the same function or block. While goto can be useful in certain situations, its use is generally discouraged due to its potential to make code less readable and harder to maintain.
- The goto statement allows transferring the control of the program to a specified labeled statement within the same function or block.

Syntax:

```
goto label;
label:
// Statement
```

Example-1:

```
#include <iostream>
int main() {
   int num = 1;
   start:
   std::cout << "Number: " << num << std::endl;
   num++;
   if (num <= 5)
      goto start;
   return 0;
}
```

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Example-2:

```
#include <iostream>
int main() {
   int i = 1;
   int j = 1;
   loop1:
   std::cout << "Outer loop: " << i << std::endl;
   i++;
   if (i <= 3)
      goto loop2;
   else
      goto end;
   loop2:
   std::cout << "Inner loop: " << j << std::endl;
   j++;
   if (j <= 2)
      goto loop2;
   else
      goto loop1;
   end:
   std::cout << "End of program." << std::endl;
   return 0;
}
```

Output:

```
Outer loop: 1
Inner loop: 1
Inner loop: 2
Outer loop: 2
Inner loop: 1
Inner loop: 2
Outer loop: 3
Inner loop: 1
Inner loop: 2
End of program.
```

Example-3:

```
#include <iostream>
int main() {
   int num;
   std::cout << "Enter a number greater than 10: ";
   std::cin >> num;
   if (num <= 10)
      goto error;
   std::cout << "Number entered is: " << num << std::endl;
   return 0;
   error:
   std::cout << "Error: Number entered is not greater than 10." << std::endl;
   return 1;
}
```

Output:

```
Enter a number greater than 10: 15
Number entered is: 15
```

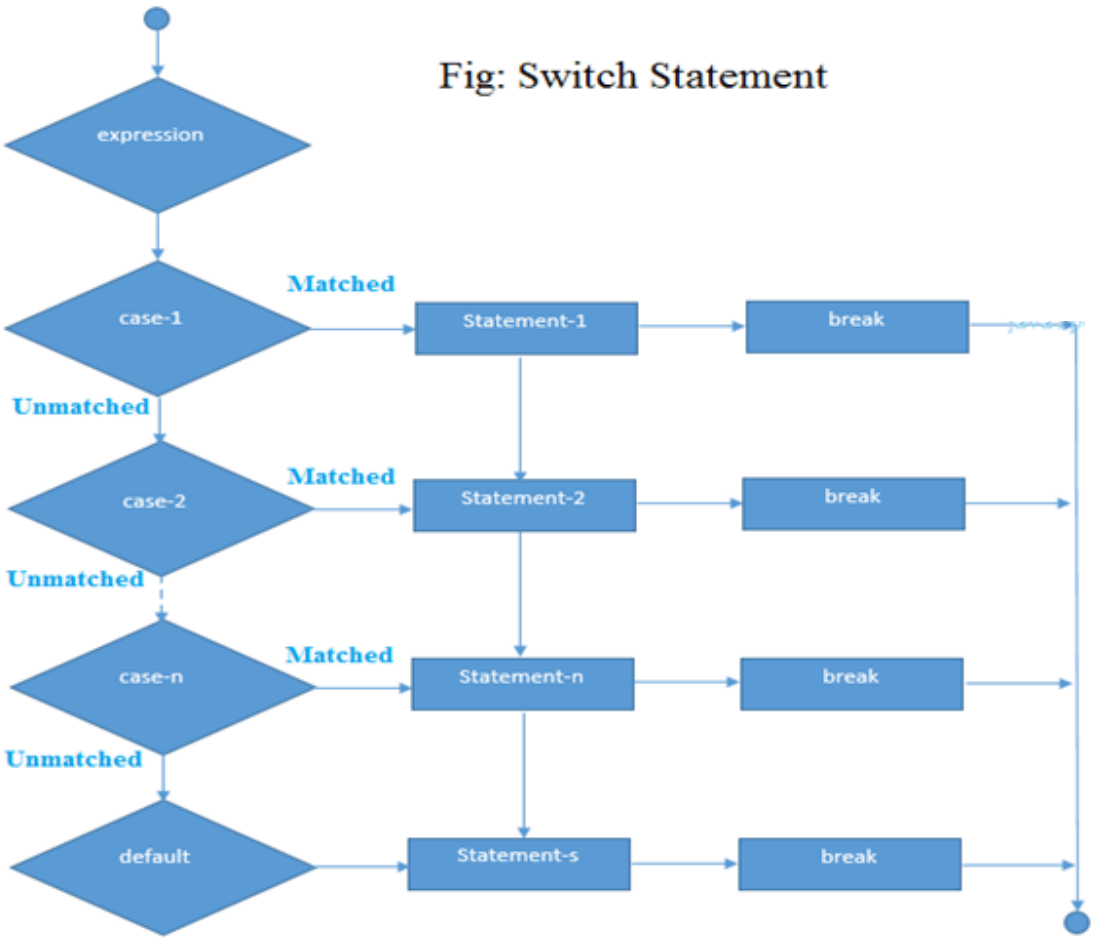
❖ Switch Statement.

- The switch statement in C++ provides a way to perform different actions based on the value of a variable or an expression. It evaluates the value of an expression and compares it with various case labels. When a match is found, the corresponding block of code is executed.
- The switch statement is a control flow statement used to select one of many code blocks to be executed based on the value of an expression.

Syntax:

```
switch (expression) {
  case value1:
    // Statements to be executed if expression equals value1
    break;
  case value2:
    // Statements to be executed if expression equals value2
    break;
  .
  .
  .
  default:
    // Statements to be executed if expression doesn't match any case
}
```

Fig: Switch Statement



Example-1:

```
#include <iostream>
int main() {
   int choice;
   std::cout << "Enter a number (1-3): ";
   std::cin >> choice;
   switch (choice) {
      case 1:
         std::cout << "You chose option 1." << std::endl;
         break;
      case 2:
         std::cout << "You chose option 2." << std::endl;
         break;
      case 3:
         std::cout << "You chose option 3." << std::endl;
         break;
      default:
         std::cout << "Invalid choice." << std::endl;
   }
   return 0;
}
```

Output:

```
Enter a number (1-3): 2
You chose option 2.
```

Example-2:

```
#include <iostream>
int main() {
   char grade;
   std::cout << "Enter your grade (A, B, C, D, or F): ";
   std::cin >> grade;
   switch (grade) {
      case 'A':
      case 'a':
         std::cout << "Excellent!" << std::endl;
         break;
      case 'B':
      case 'b':
         std::cout << "Good!" << std::endl;
         break;
      case 'C':
      case 'c':
         std::cout << "Satisfactory." << std::endl;
         break;
```

```
        case 'D':
        case 'd':
            std::cout << "Needs improvement." << std::endl;
            break;
        case 'F':
        case 'f':
            std::cout << "Failed." << std::endl;
            break;
        default:
            std::cout << "Invalid grade." << std::endl;
    }
    return 0;
}
```

Output:

```
Enter your grade (A, B, C, D, or F): B
Good!
```

Example-3:

```
#include <iostream>
int main() {
    int day;
    std::cout << "Enter a day number (1-7): ";
    std::cin >> day;
    switch (day) {
        case 1:
            std::cout << "Monday" << std::endl;
            break;
        case 2:
            std::cout << "Tuesday" << std::endl;
            break;
        case 3:
            std::cout << "Wednesday" << std::endl;
            break;
        case 4:
            std::cout << "Thursday" << std::endl;
            break;
        case 5:
            std::cout << "Friday" << std::endl;
            break;
        case 6:
            std::cout << "Saturday" << std::endl;
            break;
        case 7:
            std::cout << "Sunday" << std::endl;
            break;
        default:
            std::cout << "Invalid day number." << std::endl;
    }
    return 0;
}
```

Output:

```
Enter a day number (1-7): 4
Thursday
```

Function

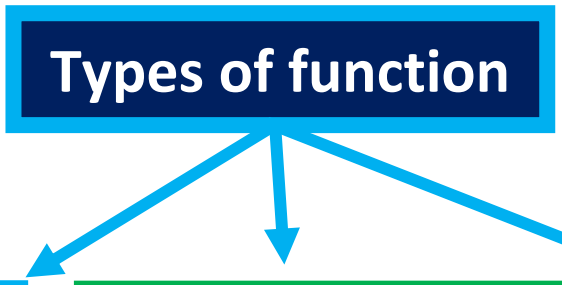
- A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function.
- It encapsulates a sequence of statements that can be executed repeatedly from various parts of a program.
- Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

❖ Advantage of function:

- There are following advantage of function.

1) **Modularity:** Functions allow you to break down a large program into smaller, manageable units. Each function can focus on a specific task, promoting code reusability and maintainability.
2) **Abstraction:** Functions hide the implementation details of a particular task, providing a clear interface for other parts of the program to interact with. This abstraction simplifies the complexity of the code, making it easier to work with.
3) **Code Reusability:** Once a function is defined, it can be called from different parts of the program whenever needed, reducing redundancy and promoting efficient code reuse.
4) **Encapsulation:** Functions can encapsulate related operations and data, allowing for better organization and separation of concerns within a program. This helps in managing complexity and avoiding conflicts between different parts of the code.
5) **Parameter Passing:** Functions can accept parameters, allowing them to work with different input values. This parameterization enhances the flexibility and versatility of functions, making them adaptable to various scenarios.
6) **Return Values:** Functions can return values, enabling them to produce results that can be used by other parts of the program. This facilitates communication between different components and enhances the overall functionality of the program.
7) **Code Readability and Maintainability:** By dividing the code into smaller, self-contained functions, C++ programs become easier to understand, debug, and modify. Functions with well-defined purposes and meaningful names contribute to the readability and maintainability of the codebase.
8) **Library Functions:** C++ provides a rich set of built-in functions and libraries for common tasks, such as mathematical calculations, string manipulation, file handling, and more. These pre-defined functions save development time and effort by providing ready-made solutions to frequently encountered problems.

❖ Types of function:



Built-in Function

❖ **Input/Output Functions:**
- cin
- cout
- getline()
- scanf()
- printf()
- gets()
- puts()

❖ **Mathematical Functions:**
- sqrt()
- abs()
- sin()
- cos()
- tan()
- pow()
- floor()
- ceil()

❖ **Character Functions:**
- tolower()
- toupper()
- isalpha()
- isdigit()
- isspace()

❖ **String Functions:**
- strlen()
- strcpy()
- strcat()
- strcmp()
- strchr()
- strstr()
- toupper()
- tolower()

User-defined Function

1. Function with no arguments and no return value
2. Function with no arguments and a return value

Inline Functions

❖ **Memory Management Functions:**
- malloc()
- free()
- calloc()
- realloc()

❖ **Time and Date Functions:**
- time()
- ctime()
- asctime()
- localtime()
- mktime()
- strftime()

❖ **File Handling Functions:**
- fopen()
- fclose()
- fread()
- fwrite()
- fseek()
- fprintf()
- fscanf()

❖ **Conversion Functions:**
- atoi()
- atof()
- itoa()

- There are following types of function in c++.
  1. Built-in Functions
  2. User-defined Functions:
  3. Inline Functions:

1. Built-in Function:

- Built-in functions are predefined functions that are part of the C++ programming language or its standard libraries.
- A built-in function in C++ refers to a function that is provided by the C++ language itself.
- These functions provide essential functionalities, such as mathematical calculations, string manipulation, input/output operations, memory management, time and date handling, and file handling.

❖ Advantage of Built-in Function:

1. **Efficiency:** Built-in functions are optimized for fast execution, often using efficient algorithms and compiler optimizations.
2. **Ease of Use:** Built-in functions provide ready-made solutions for common tasks, reducing the need for custom code and simplifying programming.
3. **Standardization:** Built-in functions are part of the C++ standard library, ensuring consistency and compatibility across different platforms and compilers.
4. **Reliability:** Built-in functions are rigorously tested and validated, ensuring correctness and dependability in various scenarios.
5. **Code Readability:** Built-in functions use descriptive names and well-defined interfaces, improving code readability and making programs easier to understand and maintain.
6. **Portability:** Built-in functions abstract away platform-specific details, making code portable and enabling it to run on different systems without modification.
7. **Productivity:** Built-in functions save time and effort by providing pre-built functionality, allowing developers to focus on solving higher-level problems and increasing overall productivity.
8. **Scalability:** Built-in functions can handle a wide range of inputs and scenarios, making them suitable for use in both small-scale and large-scale projects.

❖ Example of Built-in Function:

1. Input/Output Functions:

- **cin:** Standard input stream used for reading input from the user.
- **cout:** Standard output stream used for displaying output to the console.
- **getline():** Reads a line of text from input.
- **scanf():** Reads formatted input from the standard input stream.
- **printf():** Prints formatted output to the standard output stream.
- **gets():** Reads a string from standard input.
- **puts():** Prints a string to standard output.

2. **Mathematical Functions:**
   - **sqrt():** Calculates the square root of a number.
   - **abs():** Computes the absolute value of a number.
   - **sin(), cos(), tan():** Compute trigonometric functions.
   - **pow():** Raises a number to a power.
   - **floor(), ceil():** Rounds a floating-point number down or up to the nearest integer.
3. **Character Functions:**
   - **tolower(), toupper():** Convert characters to lowercase or uppercase.
   - **isalpha(), isdigit(), isspace():** Check if a character is alphabetic, numeric, or whitespace.
4. **String Functions:**
   - **strlen():** Returns the length of a string.
   - **strcpy(), strcat():** Copy or concatenate strings.
   - **strcmp():** Compare two strings.
   - **strchr(), strstr():** Search for characters or substrings in strings.
   - **toupper(), tolower():** Convert characters to uppercase or lowercase.
5. **Memory Management Functions:**
   - **malloc(), free():** Allocate and deallocate memory dynamically.
   - **calloc():** Allocate memory for an array and initialize it with zeros.
   - **realloc():** Resize a previously allocated block of memory.
6. **Time and Date Functions:**
   - **time():** Get the current system time.
   - **ctime(), asctime():** Convert a time value to a string.
   - **localtime():** Convert a time value to a local time structure.
   - **mktime():** Convert a local time structure to a time value.
   - **strftime():** Format a time value as a string.
7. **File Handling Functions:**
   - **fopen(), fclose():** Open and close files.
   - **fread(), fwrite():** Read from and write to files.
   - **fseek():** Move the file pointer to a specific position.
   - **fprintf(), fscanf():** Read and write formatted data to and from files.
8. **Conversion Functions:**
   - **atoi(), atof():** Convert strings to integers or floating-point numbers.
   - **itoa():** Convert integers to strings.
   - **atol():** Convert strings to long integers.

1. Input/Output Functions Example:

Proram:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
   int num;
   cout << "Enter a number: ";
   cin >> num;
   cout << "You entered: " << num << endl;
   // Input using getline
   string name;
   cout << "Enter your name: ";
   getline(cin, name);
   cout << "Hello, " << name << "!" << endl;
   // Output using cout
   cout << "Output using cout" << endl;
   // Output using printf (not recommended in C++, included for demonstration)
   int age = 25;
   printf("My age is %d\n", age);
   // Input using scanf (not recommended in C++, included for demonstration)
   int number;
   printf("Enter a number: ");
   scanf("%d", &number);
   printf("You entered: %d\n", number);
   // Output using puts (not commonly used for formatted output)
   puts("Output using puts");
   return 0;
}
```

Output:

```
Enter a number: 42
You entered: 42
Enter your name: John Doe
Hello, John Doe!
Output using cout
My age is 25
Enter a number: 100
You entered: 100
Output using puts
```

2. Mathematical Functions Example:

Program:

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
   // Using sqrt() function to calculate square root
   double x = 25.0;
   cout << "Square root of " << x << " is: " << sqrt(x) << endl;
   // Using abs() function to calculate absolute value
   int y = -10;
   cout << "Absolute value of " << y << " is: " << abs(y) << endl;
   // Using sin(), cos(), and tan() functions
   double angle = 45.0;
   cout << "Sine of " << angle << " degrees is: " << sin(angle * M_PI / 180.0) << endl;
   cout << "Cosine of " << angle << " degrees is: " << cos(angle * M_PI / 180.0) << endl;
   cout << "Tangent of " << angle << " degrees is: " << tan(angle * M_PI / 180.0) << endl;
   // Using pow() function to calculate power
   double base = 2.0;
   double exponent = 3.0;
   cout << base << " raised to the power of " << exponent << " is: " << pow(base,
exponent) << endl;
   // Using floor() and ceil() functions
   double number = 5.8;
   cout << "Floor of " << number << " is: " << floor(number) << endl;
   cout << "Ceiling of " << number << " is: " << ceil(number) << endl;
   return 0;
}
```

Output:

```
Square root of 25 is: 5
Absolute value of -10 is: 10
Sine of 45 degrees is: 0.707107
Cosine of 45 degrees is: 0.707107
Tangent of 45 degrees is: 1
2 raised to the power of 3 is: 8
Floor of 5.8 is: 5
Ceiling of 5.8 is: 6
```

3. Character Functions:

Program:

```
#include <iostream>
#include <cctype>
using namespace std;
int main() {
   // Using tolower() function to convert characters to lowercase
   char ch1 = 'A';
   char ch2 = 'b';
   cout << "Lowercase of " << ch1 << " is: " << static_cast<char>(tolower(ch1)) << endl;
   cout << "Lowercase of " << ch2 << " is: " << static_cast<char>(tolower(ch2)) << endl;
   // Using toupper() function to convert characters to uppercase
   char ch3 = 'x';
   char ch4 = 'Y';
   cout << "Uppercase of " << ch3 << " is: " << static_cast<char>(toupper(ch3)) << endl;
   cout << "Uppercase of " << ch4 << " is: " << static_cast<char>(toupper(ch4)) << endl;
   // Using isalpha() function to check if a character is alphabetic
   char ch5 = 'A';
   char ch6 = '7';
   cout << ch5 << " is alphabetic: " << isalpha(ch5) << endl;
   cout << ch6 << " is alphabetic: " << isalpha(ch6) << endl;
   // Using isdigit() function to check if a character is a digit
   char ch7 = '5';
   char ch8 = 'x';
   cout << ch7 << " is a digit: " << isdigit(ch7) << endl;
   cout << ch8 << " is a digit: " << isdigit(ch8) << endl;
   // Using isspace() function to check if a character is whitespace
   char ch9 = ' ';
   char ch10 = 'A';
   cout << ch9 << " is whitespace: " << isspace(ch9) << endl;
   cout << ch10 << " is whitespace: " << isspace(ch10) << endl;
   return 0;
}
```

Output:

```
Lowercase of A is: a
Lowercase of b is: b
Uppercase of x is: X
Uppercase of Y is: Y
A is alphabetic: 1024
7 is alphabetic: 0
5 is a digit: 1024
x is a digit: 0
 is whitespace: 1024
A is whitespace: 0
```

4. String function Example:

Program:

```
#include <iostream>
#include <cstring>
using namespace std;
int main() {
   // Using strlen() to calculate the length of a string
   char str1[] = "Hello";
   cout << "Length of \"" << str1 << "\" is: " << strlen(str1) << endl;
   // Using strcpy() to copy a string
   char str2[20];
   strcpy(str2, str1);
   cout << "Copied string: " << str2 << endl;
   // Using strcat() to concatenate strings
   char str3[20] = " world";
   strcat(str1, str3);
   cout << "Concatenated string: " << str1 << endl;
   // Using strcmp() to compare strings
   char str4[] = "hello";
   char str5[] = "HELLO";
   cout << "Comparison result: " << strcmp(str4, str5) << endl;
   // Using strchr() to find the first occurrence of a character in a string
   char* ptr = strchr(str1, 'o');
   cout << "First occurrence of 'o' in \"" << str1 << "\" is at position: " << (ptr - str1) + 1 << endl;
   char* subStr = strstr(str1, "world");
   cout << "Substring \"world\" found in \"" << str1 << "\" at position: " << (subStr - str1) + 1 << endl;
   // Using toupper() to convert string to uppercase
   char str6[] = "hello";
   for (int i = 0; i < strlen(str6); ++i) {
      str6[i] = toupper(str6[i]);
   }
   cout << "Uppercase string: " << str6 << endl;   // Using tolower() to convert string to lowercase
   char str7[] = "WORLD";
   for (int i = 0; i < strlen(str7); ++i) {
      str7[i] = tolower(str7[i]);
   }
   cout << "Lowercase string: " << str7 << endl;
   return 0;
}
```

Output:

```
Length of "Hello" is: 5
Copied string: Hello
Concatenated string: Hello world
Comparison result: 32
First occurrence of 'o' in "Hello world" is at position: 5
Substring "world" found in "Hello world" at position: 6
Uppercase string: HELLO
Lowercase string: world
```

5. Memory Management Functions Example:

Program:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
   // Using malloc() to allocate memory for an array of integers
   int* ptr = (int*)malloc(5 * sizeof(int));
   if (ptr == nullptr) {
      cout << "Memory allocation failed!" << endl;
      return 1;
   }
   // Using calloc() to allocate and initialize memory for an array of integers
   int* ptr2 = (int*)calloc(5, sizeof(int));
   if (ptr2 == nullptr) {
      cout << "Memory allocation failed!" << endl;
      return 1;
   }
   // Using realloc() to reallocate memory for an array of integers
   ptr = (int*)realloc(ptr, 10 * sizeof(int));
   if (ptr == nullptr) {
      cout << "Memory reallocation failed!" << endl;
      return 1;
   }
   // Using free() to deallocate memory
   free(ptr);
   free(ptr2);
   cout << "Memory allocation and deallocation successful." << endl;
   return 0;
}
```

Output:

Memory allocation and deallocation successful.

6. Time and Date Function:

Program:

```
#include <iostream>
#include <ctime>
using namespace std;
int main() {
   // Using time() to get the current system time
   time_t now = time(nullptr);
   cout << "Current system time: " << now << endl;
```

```
    // Using ctime() to convert time_t value to string representation
    cout << "String representation of current system time: " << ctime(&now);
    // Using asctime() to convert tm structure to string representation
    struct tm* timeinfo = localtime(&now);
    cout << "String representation of local time: " << asctime(timeinfo);
    // Using localtime() to convert time_t value to tm structure
    cout << "Local time: " << timeinfo->tm_hour << ":" << timeinfo->tm_min << ":" <<
timeinfo->tm_sec << endl;
    // Using mktime() to convert tm structure to time_t value
    timeinfo->tm_year = 121; // Year since 1900
    timeinfo->tm_mon = 0;    // Month (0-11)
    timeinfo->tm_mday = 1;   // Day of the month (1-31)
    time_t newYear = mktime(timeinfo);
    cout << "New Year's Day 2021: " << ctime(&newYear);
    // Using strftime() to format time and date
    char buffer[80];
    strftime(buffer, 80, "Today is %A, %B %d, %Y.", timeinfo);
    cout << buffer << endl;
    return 0;
}
```

Output:

```
Current system time: <current time in seconds>
String representation of current system time: <current date and time>
String representation of local time: <current date and time>
Local time: <current hour>:<current minute>:<current second>
New Year's Day 2021: Thu Jan  1 00:00:00 2021
Today is <day of the week>, <month> <day of the month>, <year>.
```

7. File Handling Function Example:

Program:

```
#include <iostream>
#include <cstdio>
using namespace std;
int main() {
   // Open a file for writing
   FILE* file = fopen("example.txt", "w");
   if (file == nullptr) {
      cout << "Error opening file for writing!" << endl;
      return 1;
   }
   // Write data to the file using fwrite()
   const char* data = "Hello, world!";
   fwrite(data, sizeof(char), strlen(data), file);
   fclose(file);
```

```
    file = fopen("example.txt", "r");    // Open the file for reading
    if (file == nullptr) {
        cout << "Error opening file for reading!" << endl;
        return 1;
    }
    char buffer[100];  // Read data from the file using fread()
    size_t bytesRead = fread(buffer, sizeof(char), sizeof(buffer), file);
    if (bytesRead > 0) {
        cout << "Data read from file: " << buffer << endl;
    } else {
        cout << "Error reading from file!" << endl;
    }
    fclose(file);      // Close the file
    return 0;
}
```

Output:

Data read from file: Hello, world!

8. Conversion Functions Example:

Program:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
    // Using atoi() to convert string to integer
    const char* str1 = "123";
    int num1 = atoi(str1);
    cout << "String \"" << str1 << "\" converted to integer: " << num1 << endl;
    // Using atof() to convert string to double
    const char* str2 = "3.14";
    double num2 = atof(str2);
    cout << "String \"" << str2 << "\" converted to double: " << num2 << endl;
    // Using itoa() to convert integer to string (non-standard function)
    int num3 = 456;
    char buffer[10];
    itoa(num3, buffer, 10);
    cout << "Integer " << num3 << " converted to string: " << buffer << endl;
    return 0;
}
```

Output:

String "123" converted to integer: 123
String "3.14" converted to double: 3.14
Integer 456 converted to string: 456

2. User-defined function.

- A user-defined function in C++ is a function created by the programmer to perform a specific task.
- These functions can have varying numbers of arguments and may or may not return a value
- There are four types of user-defined function
    1. Function with no arguments and no return value
    2. Function with no arguments and a return value
    3. Function with arguments and no return value
    4. Function with arguments and with return value

❖ Function with no arguments and no return value.

- A function in C++ with no arguments and no return value is defined using the following

syntax:

```
void functionName() {
   // Function body
   // Statements to be executed
}
```

Explanation:

- **void:** Specifies that the function does not return any value.
- **functionName:** Name of the function, which can be any valid identifier.
- **():** Parentheses indicating that the function takes no arguments.
- **{}:** Encloses the function body, which contains the statements to be executed when the function is called.

Example-1:

```
#include <iostream>
using namespace std;
void greet() {
   cout << "Hello, welcome to the program!" << endl;
}
int main() {
   greet(); // Function call
   return 0;
}
```

Output: Hello, welcome to the program!

Example-2: Printing a simple message.

```
#include <iostream>
void printMessage() {
   std::cout << "This is a function with no arguments and no return value" << std::endl;
}
int main() {
   printMessage();
   return 0;
}
```

Output: This is a function with no arguments and no return value.

Example-3: Displaying a pattern.

```
#include <iostream>
void displayPattern() {
   std::cout << "* * * * *" << std::endl;
   std::cout << "* * * * *" << std::endl;
   std::cout << "* * * * *" << std::endl;
}
int main() {
   displayPattern();
   return 0;
}
```

Output:

```
* * * * *
* * * * *
* * * * *
```

Example-4: Clearing the screen

```
#include <iostream>
#include <cstdlib> // for system function
void clearScreen() {
   system("cls");
}
int main() {
   std::cout << "This will be cleared." << std::endl;
   clearScreen();
   std::cout << "Cleared!" << std::endl;
   return 0;
}
```

Output:

(The screen will be cleared)

Example-5: Playing a beep sound.

```
#include <iostream>
#include <Windows.h> // for Beep function
void playBeep() {
   Beep(1000, 500); // Beep at 1000 Hz for 500 milliseconds
}
int main() {
   std::cout << "Playing a beep sound..." << std::endl;
   playBeep();
   std::cout << "Beep played!" << std::endl;
   return 0;
}
```

Output: (A beep sound will be heard)

Example-6: Pausing execution.

```
#include <iostream>
#include <chrono> // for std::this_thread and std::chrono_literals
```

```
#include <thread> // for std::this_thread::sleep_for
void pauseExecution() {
   std::this_thread::sleep_for(std::chrono::seconds(2)); // Pause for 2 seconds
}
int main() {
   std::cout << "Execution will pause for 2 seconds..." << std::endl;
   pauseExecution();
   std::cout << "Paused execution completed!" << std::endl;
   return 0;
}
```

Output:

(Execution will pause for 2 seconds)

Example-7: Displaying the current date and time

```
#include <iostream>
#include <ctime> // for std::time and std::localtime
#include <iomanip> // for std::put_time
void displayDateTime() {
   std::time_t now = std::time(nullptr);
   std::tm *localTime = std::localtime(&now);
   std::cout << "Current date and time: ";
   std::cout << std::put_time(localTime, "%Y-%m-%d %H:%M:%S") << std::endl;
}
int main() {
   displayDateTime();
   return 0;
}
```

Output: (Current date and time will be displayed)

❖ Function with no arguments and a return value.

- A function no arguments and a return value is defined using the following **syntax.**

Syntax:

```
returnType functionName() {
   // Function body
   // Statements to be executed
   return value; // Return statement
}
```

Explanation:

- **returnType:** Specifies the type of data that the function will return.
- **functionName:** Name of the function, which can be any valid identifier.
- **():** Parentheses indicating that the function takes no arguments.
- **{}:** Encloses the function body, which contains the statements to be executed when the function is called.
- **return value;:** Return statement, where value is the data to be returned.

Example-1: Generating a random number.

```
#include <iostream>
#include <cstdlib> // for rand() and srand()
#include <ctime> // for time()
int generateRandomNumber() {
   srand(time(nullptr)); // Seed for random number generator
   return rand() % 100; // Return a random number between 0 and 99
}
int main() {
   std::cout << "Random Number: " << generateRandomNumber() << std::endl;
   return 0;
}
```

Output: Random Number: (random number)

Example-2: Getting the current year.

```
#include <iostream>
#include <ctime> // for std::time and std::localtime
int getCurrentYear() {
   std::time_t now = std::time(nullptr);
   std::tm *localTime = std::localtime(&now);
   return localTime->tm_year + 1900; // Adding 1900 to get the current year
}
int main() {
   std::cout << "Current Year: " << getCurrentYear() << std::endl;
   return 0;
}
```

Output: Current Year: (current year)

Example-3: Getting the length of a string.

```
#include <iostream>
#include <string>
int getStringLength() {
   std::string str = "Hello, World!";
   return str.length();
}
int main() {
   std::cout << "Length of the string: " << getStringLength() << std::endl;
   return 0;
}
```

Output: Length of the string: 13

Example-4: Getting the ASCII value of a character.

```
#include <iostream>
int getAsciiValue() {
   char ch = 'A';
   return int(ch); // Implicit conversion to ASCII value
}
int main() {
```

```
    std::cout << "ASCII value of 'A': " << getAsciiValue() << std::endl;
    return 0;
}
```

Output: ASCII value of 'A': 65

Example-5: Checking if a number is even or odd.

```
#include <iostream>
bool isEven() {
    int num = 7;
    return num % 2 == 0;
}
int main() {
    if (isEven()) {
        std::cout << "Number is even." << std::endl;
    } else {
        std::cout << "Number is odd." << std::endl;
    }
    return 0;
}
```

Output: Number is odd.

Example- 6: Getting the current hour.

```
#include <iostream>
#include <ctime> // for std::time and std::localtime
int getCurrentHour() {
    std::time_t now = std::time(nullptr);
    std::tm *localTime = std::localtime(&now);
    return localTime->tm_hour;
}
int main() {
    std::cout << "Current Hour: " << getCurrentHour() << std::endl;
    return 0;
}
```

Output: Current Hour: (current hour)

Example-7:

```
#include <iostream>
using namespace std;
int getRandomNumber() {
    return rand(); // Returns a random number
}
int main() {
    cout << "Random number: " << getRandomNumber() << endl; // Function call
    return 0;
}
```

❖ Function with arguments and no return value.

- A function with arguments and no return value is a function that takes one or more arguments as input parameters but does not return any value.

Syntax:

```
void functionName(type1 arg1, type2 arg2, ..., typen argn) {
   // Function body
   // Statements to be executed
}
```

Syntax Explanation:

void: Specifies that the function does not return any value.
functionName: Name of the function, which can be any valid identifier.
type1 arg1, type2 arg2, ..., typen argn: Input parameters of the function along with their data types.
{}: Encloses the function body, which contains the statements to be executed when the function is called.

Example-1:

```
#include <iostream>
 using namespace std;
 void add(int a, int b) {
    cout << "Sum: " << a + b << endl;
 }
 int main() {
    add(5, 7); // Function call
    return 0;
 }
```

Output: Sum: 12

Example-2: Displaying a rectangle of stars.

```
#include <iostream>
void drawRectangle(int rows, int cols) {
   for (int i = 0; i < rows; ++i) {
      for (int j = 0; j < cols; ++j) {
         std::cout << "* ";
      }
      std::cout << std::endl;
   }
}
int main() {
   drawRectangle(4, 6);
   return 0;
}
```

Output:

```
* * * * * *
* * * * * *
* * * * * *
* * * * * *
```

Example-3: Printing a message multiple times.

```
#include <iostream>
void printMessageMultipleTimes(std::string message, int times) {
   for (int i = 0; i < times; ++i) {
      std::cout << message << std::endl;
   }
}
int main() {
   printMessageMultipleTimes("Hello!", 3);
   return 0;
}
```

Output:

```
Hello!
Hello!
Hello!
```

Example-4: Finding the maximum of two numbers.

```
#include <iostream>
void findMax(int a, int b) {
   if (a > b)
      std::cout << "Maximum number is: " << a << std::endl;
   else
      std::cout << "Maximum number is: " << b << std::endl;
}
int main() {
   findMax(10, 5);
   return 0;
}
```

Output:

```
Maximum number is: 10
```

Example-5: Generating Fibonacci series.

```
#include <iostream>
void generateFibonacci(int n) {
   int a = 0, b = 1, c;
   std::cout << "Fibonacci series up to " << n << " terms:" << std::endl;
   for (int i = 0; i < n; ++i) {
      std::cout << a << " ";
      c = a + b;
      a = b;
      b = c;
   }
   std::cout << std::endl;
}
int main() {
   generateFibonacci(8);
   return 0;
}
```

Output: Fibonacci series up to 8 terms:

0 1 1 2 3 5 8 13

Example-6: Checking whether a number is prime.

```
#include <iostream>
void checkPrime(int num) {
   bool isPrime = true;
   if (num <= 1) {
      isPrime = false;
   } else {
      for (int i = 2; i <= num / 2; ++i) {
         if (num % i == 0) {
            isPrime = false;
            break;
         }
      }
   }
   if (isPrime)
      std::cout << num << " is a prime number" << std::endl;
   else
      std::cout << num << " is not a prime number" << std::endl;
}
int main() {
   checkPrime(13);
   return 0;
}
```

Output:

13 is a prime number

❖Function with arguments and with return value:

- A function with arguments and a return value allows you to pass data to the function and receive a result back from it.

Syntax:

```
returnType functionName(argumentType1 arg1, argumentType2 arg2, ...) {
   // Function body
   // Perform operations using arguments
   return result; // Return value of returnType
}
```

Explanation:

- **returnType:** Specifies the type of the value that the function will return.
- **functionName:** Name of the function.
- argumentType1 arg1, argumentType2 arg2, ...: List of arguments along with their data types that the function expects.
- **return result:** The result that the function computes and returns.

Example-1: Adding two integers.

```
#include <iostream>
int add(int a, int b) {
   return a + b;
}
int main() {
   int result = add(5, 3);
   std::cout << "Result of addition: " << result << std::endl;
   return 0;
}
```

Output: Result of addition: 8

Example-2: Calculating the area of a rectangle.

```
#include <iostream>
float calculateRectangleArea(float length, float width) {
   return length * width;
}
int main() {
   float area = calculateRectangleArea(5.0, 3.0);
   std::cout << "Area of the rectangle: " << area << std::endl;
   return 0;
}
```

Output: Area of the rectangle: 15

Example-3: Finding the maximum of two numbers.

```
#include <iostream>
int max(int a, int b) {
   return (a > b) ? a : b;
}
int main() {
   int maximum = max(5, 8);
   std::cout << "Maximum of the two numbers: " << maximum << std::endl;
   return 0;
}
```

Output: Maximum of the two numbers: 8

Example-4: Concatenating two strings.

```
#include <iostream>
#include <string>
std::string concatenateStrings(std::string str1, std::string str2) {
   return str1 + str2;
}
int main() {
   std::string result = concatenateStrings("Hello, ", "world!");
   std::cout << "Concatenated string: " << result << std::endl;
   return 0;
}
```

Output: Concatenated string: Hello, world!

Example-5: Checking if a number is even.

```
#include <iostream>
bool isEven(int num) {
   return num % 2 == 0;
}
int main() {
   int number = 6;
   if (isEven(number)) {
      std::cout << number << " is even." << std::endl;
   } else {
      std::cout << number << " is odd." << std::endl;
   }
   return 0;
}
```

Output: 6 is even.

Example-6: **Calculating the factorial of a number.**

```
#include <iostream>
int factorial(int n) {
   if (n == 0 || n == 1)
      return 1;
   else
      return n * factorial(n - 1);
}
int main() {
   int num = 5;
   std::cout << "Factorial of " << num << " is: " << factorial(num) << std::endl;
   return 0;
}
```

Output: Factorial of 5 is: 120.

Example-7.

```
#include <iostream>
 using namespace std;
 int multiply(int x, int y) {
    return x * y;
 }
 int main() {
    cout << "Product: " << multiply(3, 4) << endl; // Function call
    return 0;
 }
```

Output: Product: 12

❖ inline function.

- An inline function is a function that is expanded in place at each point it is called, rather than being called like a regular function. This can improve performance by reducing the overhead of function calls.

Syntax:

```
inline returnType functionName(arguments) {
   // Function body
}
```

Explanation:

- inline: Keyword used to specify that the function should be expanded inline at each call site.
- returnType: Specifies the type of the value that the function will return.
- functionName: Name of the inline function.
- arguments: List of arguments along with their data types that the function expects.
- Function body: Contains the code to be executed when the function is called.

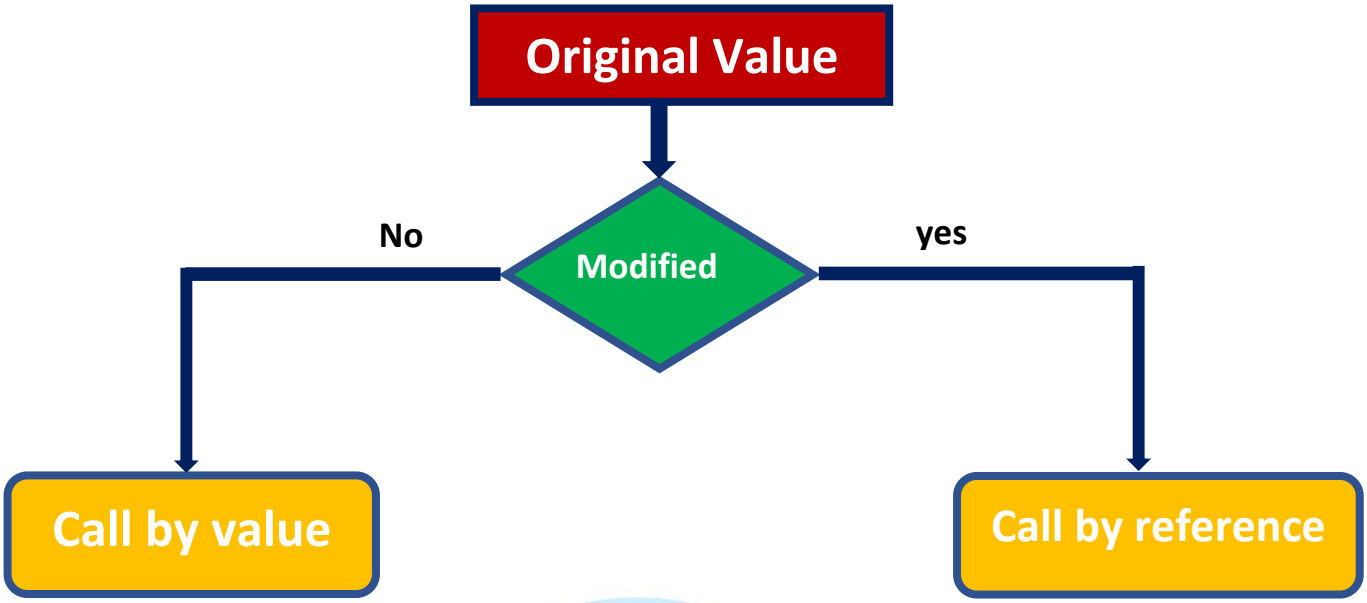
Example:

```
#include <iostream>
// Inline function definition
inline int add(int a, int b) {
   return a + b;
}
int main() {
   int result = add(5, 3); // Function call
   std::cout << "Result of addition: " << result << std::endl;
   return 0;
}
```

Output: Result of addition: 8

Call by value and call by reference

- when you pass arguments to a function, you can do so either by value or by reference.



❖ Call by Value:

- In call by value, original value is not modified.
- In call by value, a copy of the actual parameter is passed to the function. Any modifications made to the parameters inside the function do not affect the original values outside the function.

Program:

```
#include <iostream>
void increment(int x) {
   x++; // Increment the value of x
   std::cout << "Inside the function: " << x << std::endl;
}
int main() {
   int num = 5;
   std::cout << "Before function call: " << num << std::endl;
   // Function call
   increment(num); // Passing num by value
   std::cout << "After function call: " << num << std::endl;
   return 0;
}
```

Output:

```
Before function call: 5
Inside the function: 6
After function call: 5
```

- As you can see, the value of `num` remains unchanged outside the function despite being modified inside the function.

❖ Call by Reference.

- n call by reference, original value is modified because we pass reference (address)
- In call by reference, the actual memory address of the variable is passed to the function. Any modifications made to the parameters inside the function affect the original values outside the function.

Program:

```
#include <iostream>
void increment(int &x) {
   x++; // Increment the value at the memory address of x
   std::cout << "Inside the function: " << x << std::endl;
}
int main() {
   int num = 5;
   std::cout << "Before function call: " << num << std::endl;
   increment(num); // Passing num by reference
   std::cout << "After function call: " << num << std::endl;
   return 0;
}
```

Output:

```
Before function call: 5
Inside the function: 6
After function call: 6
```

- Here, the value of `num` changes outside the function because it's passed by reference, and modifications made inside the function affect the original variable.

Difference between call by value and call by reference

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

❖ Recursion Function.

- Recursion in C++ refers to the technique where a function calls itself to solve smaller instances of the same problem until it reaches a base case where the solution can be directly determined.
- Recursion can be a powerful and elegant way to solve certain types of problems, particularly those that exhibit a self-similar structure or can be broken down into smaller, simpler subproblems.

Syntax:

```
recursionfunction(){
recursionfunction(); //calling self function
}
```

Example: **calculate factorial.**

```
#include <iostream>
// Recursive function to calculate factorial
unsigned long long factorial(int n) {
   // Base case: factorial of 0 is 1
   if (n == 0)
      return 1;
   // Recursive case: n * factorial(n-1)
   else
      return n * factorial(n - 1);
}
int main() {
   int num = 5;
   std::cout << "Factorial of " << num << " is " << factorial(num) << std::endl;
   return 0;
}
```

Output:

Factorial of 5 is 120

- We can understand the above program of recursive method call by the figure given below

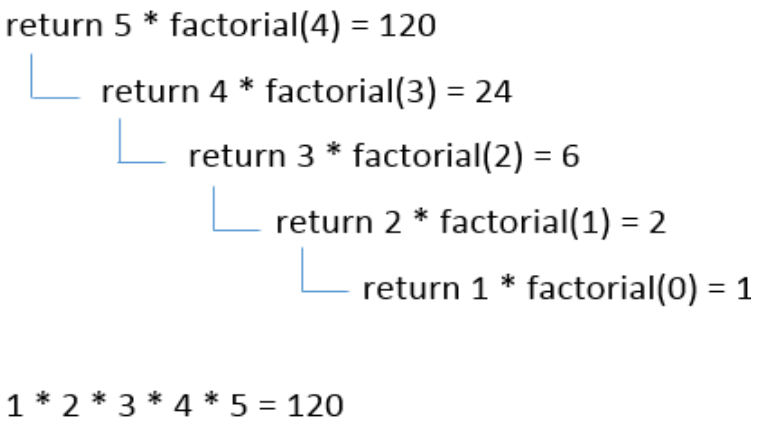


Fig: Recursion

❖ Storage Classes.

- Storage class is used to define the lifetime and visibility of a variable and/or function within a C++ program.
- Lifetime refers to the period during which the variable remains active and visibility refers to the module of a program in which the variable is accessible.
- There are five types of storage classes, which can be used in a C++ program
    1. Automatic
    2. Register
    3. Static
    4. External
    5. Mutable

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero

1. **Automatic Storage Class:** Variables declared inside a function without any storage class specifier are by default automatic variables. They are created when the block in which they are declared is entered and destroyed when the block is exited.

Syntax:

```
void function() {
   int x; // Automatic variable
}
```

Example:

```
#include <iostream>
void function() {
   int x = 5; // Automatic variable
   std::cout << "Value of x inside function: " << x << std::endl;
}
int main() {
   function();
   // Error: x is not accessible outside the function
   // std::cout << "Value of x outside function: " << x << std::endl;
   return 0;
}
```

Output: Value of x inside function: 5

2. **Register Storage Class:** This storage class is rarely used in modern C++. It suggests to the compiler that the variable will be heavily used and may be stored in a CPU register for faster access.

Syntax: register int x;

Example:

```
#include <iostream>
int main() {
   register int x = 10; // Register variable
   std::cout << "Value of x: " << x << std::endl;
   return 0;
}
```

Output: (Output may vary depending on the compiler and optimization settings)

Value of x: 10

3. **Static Storage Class:** Static variables have a lifetime throughout the program's execution and retain their values between function calls. They are initialized only once.

Syntax: static int x;

Example:

```
#include <iostream>
 void function() {
   static int count = 0; // Static variable
   count++;
   std::cout << "Function call count: " << count << std::endl;
}
int main() {
   function(); // Function call 1
   function(); // Function call 2
   function(); // Function call 3
   return 0;
}
```

Output:

```
Function call count: 1
Function call count: 2
Function call count: 3
```

4. **External Storage Class:** This storage class is used to declare variables that can be accessed by any function or file in a program. It's often used when variables need to be shared across multiple files.

Syntax: extern int x;

Example (using it across multiple files):

File: `file1.cpp`

```
#include <iostream>
extern int x;
void function() {
   std::cout << "Value of x in file1: " << x << std::endl;
}
```

File: `file2.cpp`

```
#include <iostream>
int x = 10; // Definition of x
int main() {
   function(); // Accessing x from file1.cpp
   return 0;
}
```

Output: Value of x in file1: 10

ARRAY

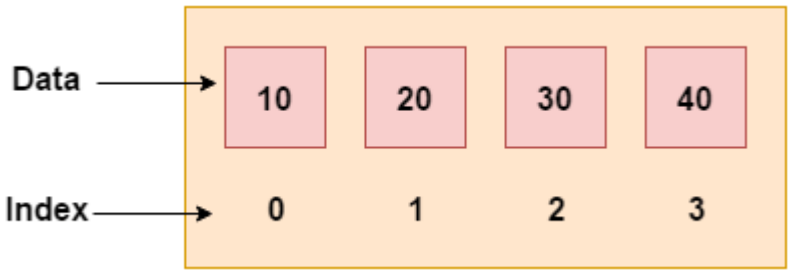
- An array in C++ is a data structure that stores a fixed-size sequential collection of elements of the same data type.
- These elements are stored in contiguous memory locations, and each element can be accessed using its index.

❖ Advantages:

1. **Random Access:** Arrays allow direct access to any element by using its index, making retrieval of elements efficient.
2. **Memory Efficiency:** Arrays store elements in contiguous memory, resulting in efficient memory usage.
3. **Performance:** Accessing elements of an array is typically faster compared to other data structures like linked lists due to the direct access capability.
4. **Simplicity:** Arrays are simple to understand and use, especially for storing a fixed number of elements.

❖ Disadvantages:

1. **Fixed Size:** Arrays have a fixed size determined at compile time, which makes it challenging to resize them dynamically.
2. **Static Memory Allocation:** The memory for an array is allocated at compile time, and the size must be known beforehand. This can lead to wasted memory if the array size is larger than required.
3. **Contiguous Memory Requirement:** Since arrays require contiguous memory, it may not always be possible to allocate a large enough block of memory, especially in systems with memory fragmentation.
4. **No Built-in Bounds Checking**: C++ arrays do not perform bounds checking, so accessing elements beyond the array bounds can lead to undefined behavior, such as accessing uninitialized memory or overwriting other variables.



- In C++ std::array is a container that encapsulates fixed size arrays. In C++, array index starts from 0. We can store only fixed set of elements in C++ array.

❖ Why do we need arrays?

➢ Arrays in C++ serve several important purposes, making them a fundamental data structure in programming. Here's why we need arrays in C++:

1. **Data Storage:** Arrays provide a way to store multiple elements of the same data type in a single variable. This allows for efficient management of related data, such as a list of numbers or a collection of objects.
2. **Random Access:** Arrays allow direct access to any element using its index. This means you can quickly retrieve or modify individual elements based on their position in the array without having to traverse the entire collection.
3. **Memory Efficiency:** Arrays store elements in contiguous memory locations, which results in efficient memory usage. This contiguous memory allocation makes it easy to calculate the memory address of each element based on its index.
4. **Performance:** Accessing elements of an array is typically faster than accessing elements in other data structures like linked lists because arrays offer constant-time access to any element. This makes arrays well-suited for scenarios where fast access to data is crucial.
5. **Simplicity:** Arrays are simple to understand and use, especially for storing a fixed number of elements. They provide a straightforward way to organize and manipulate data, making them an essential tool for programmers, especially those new to programming.
6. **Versatility:** Arrays can be used to implement various data structures and algorithms, such as stacks, queues, matrices, and sorting algorithms. They serve as building blocks for more complex data structures and algorithms, enabling the development of efficient and scalable solutions to a wide range of problems.

❖ Types of Array.

- There are 2 types of arrays in C++ programming:
  1. Single Dimensional Array
  2. Multidimensional Array

1. Single Dimensional Array:

- A Single Dimensional Array in C++ is a basic type of array that stores elements in a linear sequence. It is a collection of elements of the same data type arranged in a single row or column.
- Each element in the array is identified by its index, starting from zero for the first element and incrementing by one for each subsequent element. Here's a brief overview:

❖ Characteristics of Single Dimensional Array:

1. **Linear Structure:** Elements are stored sequentially in a single row or column.
2. **Fixed Size:** The size of a single-dimensional array is fixed and specified during declaration. It cannot be changed dynamically during program execution.
3. **Direct Access:** Elements can be accessed directly by using their index, allowing for efficient random access.

Syntax:

```
dataType arrayName[arraySize];
```

Example-1:

```
#include <iostream>
int main() {
   int numbers[5];
   numbers[0] = 10;
   numbers[1] = 20;
   numbers[2] = 30;
   numbers[3] = 40;
   numbers[4] = 50;
   std::cout << "Array Elements:" << std::endl;
   for (int i = 0; i < 5; ++i) {
      std::cout << "Element at index "
      << i << ": " << numbers[i] << std::endl;
   }
   return 0;
}
```

Output:

```
Array Elements:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50
```

Example-2: Sum of Array Elements.

```
#include <iostream>
int main() {
   int numbers[] = {10, 20, 30, 40, 50};
   int sum = 0;
   for (int i = 0; i < 5; ++i) {
      sum += numbers[i];
   }
   std::cout << "Sum of array elements: " <<
sum << std::endl;
   return 0;
}
```

Output: Sum of array elements: 150

Example 3: Finding Maximum Element.

```
#include <iostream>
int main() {
  int numbers[] = {10, 20, 30, 40, 50};
   int maxElement = numbers[0];
   int maxIndex = 0;
   for (int i = 1; i < 5; ++i) {
      if (numbers[i] > maxElement) {
         maxElement = numbers[i];
         maxIndex = i;
      } }
   std::cout << "Maximum element: " << maxElement << " at
index " << maxIndex << std::endl;
   return 0;
}
```

Output: Maximum element: 50 at index 4

Example 4: Average of Array Elements.

```
#include <iostream>
int main() {
   int numbers[] = {10, 20, 30, 40, 50};
   int sum = 0;
   for (int i = 0; i < 5; ++i) {
      sum += numbers[i];
   }
   double average =
static_cast<double>(sum) / 5;
   std::cout << "Average of array elements: "
<< average << std::endl;
   return 0;
}
```

Output: Average of array elements: 30

Example 5: Reverse Array Elements.

```
#include <iostream>
int main() {
   int numbers[] = {10, 20, 30, 40, 50};
   for (int i = 0, j = 4; i < j; ++i, --j) {
      int temp = numbers[i];
      numbers[i] = numbers[j];
      numbers[j] = temp; }
   std::cout << "Reversed array elements:" << std::endl;
   for (int i = 0; i < 5; ++i) {
      std::cout << numbers[i] << " ";}
   std::cout << std::endl;
   return 0; }
```

Output:

```
Reversed array elements:
50 40 30 20 10
```

Example 6: Counting Even Numbers.

```
#include <iostream>
int main() {
   int numbers[] = {10, 21, 30, 45, 50, 66, 72};
   int evenCount = 0;
   for (int i = 0; i < 7; ++i) {
      if (numbers[i] % 2 == 0) {
         evenCount++;
      }
   }
   std::cout << "Number of even elements: " << evenCount << std::endl;
   return 0;
}
```

Output: Number of even elements: 5

Example 7: Finding Element by Value.

```
#include <iostream>
int main() {
   int numbers[] = {10, 20, 30, 40, 50, 60};
   int target = 40;
   int index = -1;
   for (int i = 0; i < 6; ++i) {
      if (numbers[i] == target) {
         index = i;
         break;
      }
   }
   if (index != -1) {
      std::cout << "Element " << target << " found at index " << index << std::endl;
   } else {
      std::cout << "Element " << target << " not found in the array" << std::endl;
   }
   return 0;
}
```

Output:

Element 40 found at index 3

2. Two-dimensional array:

- two-dimensional array is a data structure that stores elements in a tabular format with rows and columns. It can be visualized as a grid, where each cell contains an element of the same data type. Unlike a one-dimensional array, which represents a linear sequence of elements, a two-dimensional array arranges elements in rows and columns.

❖ Characteristics of Two-Dimensional Array:

1. **Tabular Structure:** Elements are organized in rows and columns, forming a grid-like structure.
2. **Fixed Size:** The size of a two-dimensional array is fixed and specified during declaration. It consists of both the number of rows and the number of columns.
3. **Direct Access:** Elements can be accessed directly using their row and column indices.
4. **Rectangular Shape:** All rows in a two-dimensional array have the same number of columns, ensuring a rectangular shape.

Syntax:

```
dataType arrayName[rows][columns];
```

Example 1: Printing a 2D Array

```
#include <iostream>
int main() {
   int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   std::cout << "2D Array Elements:" << std::endl;
   for (int i = 0; i < 3; ++i) {
      for (int j = 0; j < 3; ++j) {
         std::cout << matrix[i][j] << " ";
      }
      std::cout << std::endl;
   }
   return 0;
}
```

Output

```
2D Array Elements:
1 2 3
4 5 6
7 8 9
```

Example 2: Sum of All Elements

```
#include <iostream>
int main() {
   int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   int sum = 0;
   for (int i = 0; i < 3; ++i) {
      for (int j = 0; j < 3; ++j) {
         sum += matrix[i][j];
      }
```

```
    }
    std::cout << "Sum of all elements: " << sum << std::endl;
    return 0;
}
```

Output

Sum of all elements: 45

Example 3: Transposing a Matrix

```
#include <iostream>
int main() {
    int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int transposed[3][3];
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            transposed[j][i] = matrix[i][j];
        }
    }
    std::cout << "Transposed Matrix:" << std::endl;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            std::cout << transposed[i][j] << " ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

Output:

```
Transposed Matrix:
1 4 7
2 5 8
3 6 9
```

Example 4: Finding Maximum Element

```
#include <iostream>
int main() {
    int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int maxElement = matrix[0][0];
    int maxRow = 0, maxCol = 0;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if (matrix[i][j] > maxElement) {
                maxElement = matrix[i][j];
                maxRow = i;
                maxCol = j;
            }
        }
```

```
    }
    std::cout << "Maximum element: " << maxElement << " at row " << maxRow << "
and column " << maxCol << std::endl;
    return 0;
}
```

Output

Maximum element: 9 at row 2 and column 2

Example 5: Diagonal Sum of a Matrix

```
#include <iostream>
int main() {
    int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int diagonalSum = 0;
    for (int i = 0; i < 3; ++i) {
        diagonalSum += matrix[i][i];
    }
    std::cout << "Sum of diagonal elements: " << diagonalSum << std::endl;
    return 0;
}
```

Output: Sum of diagonal elements: 15

Example 6: Matrix Multiplication

```
#include <iostream>
int main() {
    int matrix1[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int matrix2[3][2] = {{1, 2}, {3, 4}, {5, 6}};
    int result[2][2] = {{0, 0}, {0, 0}};
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            for (int k = 0; k < 3; ++k) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            } } }
    std::cout << "Resulting Matrix:" << std::endl;
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            std::cout << result[i][j] << " ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

Output: Resulting Matrix:

```
22 28
49 64
```

❖ Operation in array:

- various operations that can be performed on arrays in C

1. Declaration and Initialization:

- **Definition:** Declaration refers to the process of specifying the type and name of an array, while initialization assigns initial values to its elements.

Syntax:

```
dataType arrayName[arraySize] = {value1, value2, ..., valueN};
```

Example

```
#include <iostream>
int main() {
   int numbers[5] = {100, 200, 300, 400, 500};
   std::cout << "Array Elements:" << std::endl;
   for (int i = 0; i < 5; ++i) {
      std::cout << numbers[i] << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output

```
Array Elements:
100 200 300 400 500
```

2. Accessing Array Elements:

- **Definition:** Accessing array elements involves retrieving the value stored at a specific index within the array.

Syntax: arrayName[index]

Example

```
#include <iostream>
int main() {
   int numbers[5] = {10, 20, 30, 40, 50};
   std::cout << "Element at index 2: " << numbers[2] << std::endl;
   std::cout << "Element at index 4: " << numbers[4] << std::endl;
   return 0;
}
```

Output

```
Element at index 2: 30
Element at index 4: 50
```

3. Modifying Array Elements:

- **Definition:** Modifying array elements involves changing the value stored at a specific index within the array.

Syntax: arrayName[index] = newValue;

Example

```
#include <iostream>
```

```
int main() {
   int numbers[5] = {10, 20, 30, 40, 50};
   numbers[2] = 35;
   std::cout << "Modified element at index 2: " << numbers[2] << std::endl;
   return 0;
}
```

Output: Modified element at index 2: 35

4. Traversing an Array:

- Definition:Traversing an array involves iterating over all its elements to perform a specific operation on each element.

Syntax

```
for (int i = 0; i < arraySize; ++i) {
   // Perform operation on array element at index i
}
```

Example:

```
#include <iostream>
int main() {
   int numbers[5] = {10, 20, 30, 40, 50};
   std::cout << "Array Elements:" << std::endl;
   for (int i = 0; i < 5; ++i) {
      std::cout << numbers[i] << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output:

```
Array Elements:
10 20 30 40 50
```

5. Finding the Size of an Array:

- Definition: Finding the size of an array involves determining the number of elements it contains.

Syntax:

```
sizeof(arrayName) / sizeof(arrayName[0])
```

Example:

```
#include <iostream>
int main() {
   int numbers[5] = {10, 20, 30, 40, 50};
   int size = sizeof(numbers) / sizeof(numbers[0]);
   std::cout << "Size of the array: " << size << std::endl;
   return 0;
}
```

Output: Size of the array: 5

6. Searching for an Element:

- Definition: Searching for an element involves finding the index of the first occurrence of a specific value within the array.

Syntax:

```
for (int i = 0; i < arraySize; ++i) {
   if (arrayName[i] == searchValue) {
      // Element found at index i
      break;
   }
}
```

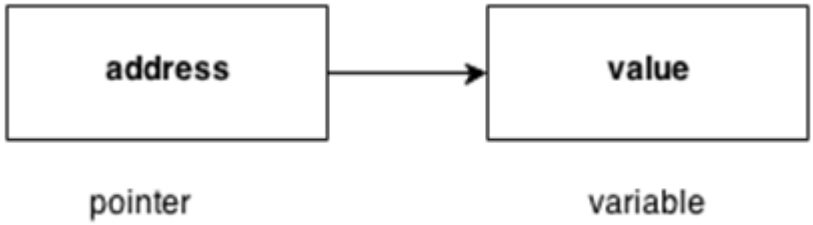
Example:

```
#include <iostream>
int main() {
   int numbers[5] = {10, 20, 30, 40, 50};
   int searchValue = 30;
   int index = -1;
   for (int i = 0; i < 5; ++i) {
      if (numbers[i] == searchValue) {
         index = i;
         break;
      }
   }
   if (index != -1) {
      std::cout << "Element " << searchValue << " found at index " << index << std::endl;
   } else {
      std::cout << "Element " << searchValue << " not found in the array" << std::endl;
   }
   return 0;
}
```

Output: Element 30 found at index 2

POINTER

- A pointer in C++ is a variable that stores the memory address of another variable. In simpler terms, a pointer "points" to a memory location where some data is stored.



Syntax:

type *pointer_name;

- **type:** It is the data type of the variable whose address the pointer will store.
- *: It is the dereference operator used to declare a pointer.
- **pointer_name:** It is the name of the pointer variable.

❖ Advantages of pointers.

1. **Dynamic Memory Allocation:** Pointers allow dynamic allocation and deallocation of memory, providing flexibility in managing memory resources efficiently.
2. **Data Structures:** Pointers facilitate the implementation of complex data structures like linked lists, trees, and graphs, optimizing memory usage and enhancing performance.
3. **Passing by Reference:** Pointers enable passing parameters by reference, allowing functions to modify original data directly, reducing the overhead of copying large data structures.
4. **Low-Level Manipulation:** Pointers provide direct access to memory addresses, enabling low-level manipulation of data and efficient handling of system resources.
5. **Arrays and Strings:** Pointers simplify array and string operations by allowing efficient traversal and manipulation, enhancing code readability and performance.
6. **Polymorphism and Dynamic Binding:** Pointers to base class objects enable polymorphism and dynamic binding, facilitating runtime polymorphism and enhancing code flexibility.
7. **Resource Management:** Pointers are essential for managing resources like file handles, network connections, and memory buffers, enabling efficient resource utilization and cleanup.
8. **Function Pointers:** Pointers to functions allow dynamic function calls and callbacks, enabling advanced programming techniques like event handling and callback mechanisms.

Example:

```
int *ptr; // Declares a pointer to an integer
```

❖ Using Pointers:

1. **Initializing Pointers:** Pointers should be initialized before they are used. They can be initialized with the address of another variable.

```
int x = 10;
int *ptr = &x; // Initializes ptr with the address of x
```

2. **Accessing Value Through Pointers:** You can access the value stored at the memory location pointed to by a pointer using the dereference operator (`*`).

```
int x = 10;
int *ptr = &x;
cout << *ptr; // Prints the value of x (which is 10)
```

3. **Modifying Value Through Pointers:** You can modify the value stored at the memory location pointed to by a pointer.

```
int x = 10;
int *ptr = &x;
*ptr = 20; // Changes the value of x to 20
cout << x; // Prints 20
```

4. **Pointer Arithmetic:** Pointer arithmetic allows you to perform arithmetic operations on pointers.

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr; // Points to the first element of the array
// Accessing elements using pointer arithmetic
cout << *(ptr + 2); // Prints the third element of the array (3)
```

5. **Null Pointers:** Pointers can also be assigned a special value `nullptr` which indicates that they are not pointing to any valid memory location.

```
int *ptr = nullptr; // Assigning a null pointer
```

6. Pointer to Pointer (Double Pointer): In C++, you can have pointers to pointers. These are called double pointers.

```
int x = 10;
int *ptr = &x;
int **ptr_ptr = &ptr; // Pointer to pointer
```

❖ Usage of pointer:

- There are many usages of pointers in C++ language.

1) Dynamic memory allocation

➢ In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

➢ Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

❖ Symbols used in pointer:

Symbol	Name	Description
1. & (ampersand sign)	Address operator	Determine the address of a variable.
2. ∗ (asterisk sign)	Indirection operator	Access the value of an address.

Example 1: Pointer Initialization and Dereferencing.

```
#include <iostream>
int main() {
   int x = 10;
   int *ptr = &x; // Pointer initialization with the address of x
   std::cout << "Value of x: " << x << std::endl;
   std::cout << "Value stored at the memory location pointed by ptr: " << *ptr <<
std::endl;
   return 0;
}
```

Output:

```
Value of x: 10
Value stored at the memory location pointed by ptr: 10
```

Example 2: Modifying Value Through Pointers.

```
#include <iostream>
int main() {
   int x = 10;
   int *ptr = &x; // Pointer initialization with the address of x
   *ptr = 20; // Modifying the value of x through ptr
   std::cout << "New value of x: " << x << std::endl;
   return 0;
}
```

Output:

```
New value of x: 20
```

Example 3: Pointer Arithmetic.

```
#include <iostream>
int main() {
   int arr[5] = {1, 2, 3, 4, 5};
   int *ptr = arr; // Pointer pointing to the first element of the array
   std::cout << "Value of the third element using pointer arithmetic: " << *(ptr + 2) <<
std::endl;
   return 0;
}
```

Output:

```
Value of the third element using pointer arithmetic: 3
```

SIZEOF() OPERATOR

➢ The sizeof() operator in C++ is used to determine the size, in bytes, of a data type or a variable. It is a compile-time operator that returns the number of bytes required to store an object of the specified type or the size of the specified variable. Here's how it works:

Syntax: sizeof (type)
sizeof (expression)

- **type:** Specifies the data type whose size needs to be determined.
- **expression:** Represents the variable or expression whose size needs to be determined.

Example 1: Size of Data Types.

```
#include <iostream>
int main() {
   std::cout << "Size of int: " << sizeof(int) << " bytes" << std::endl;
   std::cout << "Size of char: " << sizeof(char) << " byte" << std::endl;
   std::cout << "Size of float: " << sizeof(float) << " bytes" << std::endl;
   std::cout << "Size of double: " << sizeof(double) << " bytes" << std::endl;
   return 0;
}
```

Output:

```
Size of int: 4 bytes
Size of char: 1 byte
Size of float: 4 bytes
Size of double: 8 bytes
```

Example 2: Size of Variables.

```
#include <iostream>
int main() {
   int arr[5];
   double value;
   std::cout << "Size of arr: " << sizeof(arr) << " bytes" << std::endl;
   std::cout << "Size of value: " << sizeof(value) << " bytes" << std::endl;
   return 0;
}
```

Output:

```
Size of arr: 20 bytes   // (5 * sizeof(int))
Size of value: 8 bytes // (sizeof(double))
```

❖ Important Points:

- The sizeof() operator returns the size of the operand in bytes.
- The result of sizeof() is of type size_t, which is an unsigned integer type.
- The size returned by sizeof() may vary depending on the compiler and the system architecture.
- Overall, sizeof() is a useful operator in C++ for determining the size of data types, variables, and structures, which is crucial for memory allocation and manipulation.

ARRAY OF POINTERS

- An array of pointers in C++ is an array where each element is a pointer to another data type. This concept allows you to create an array that can hold pointers to different objects or data types. Here's a detailed explanation.

Syntax:

```
data_type *array_name[size];
```

- **data_type:** It is the type of data that the pointers in the array will point to.
- **array_name:** It is the name of the array of pointers.
- **size:** It specifies the number of elements in the array.

Example:

```
#include <iostream>
int main() {
   int *ptrArray[5]; // Array of 5 integer pointers
   // Assigning values to each pointer element
   for (int i = 0; i < 5; ++i) {
      ptrArray[i] = new int(i * 10);
   }  // Accessing and printing values using the pointers
   for (int i = 0; i < 5; ++i) {
      std::cout << "Value at index " << i << ": " << *ptrArray[i] << std::endl;
   } // Freeing dynamically allocated memory
   for (int i = 0; i < 5; ++i) {
      delete ptrArray[i];
   }
   return 0;
}
```

Output:

```
Value at index 0: 0
Value at index 1: 10
Value at index 2: 20
Value at index 3: 30
Value at index 4: 40
```

❖ Use Cases:

1. **Array of Strings:** An array of pointers is commonly used to create arrays of strings where each element points to a different string.
2. **Array of Objects:** In object-oriented programming, an array of pointers can hold pointers to objects of different classes, allowing polymorphic behavior.
3. **Dynamic Arrays:** Arrays of pointers are useful for creating dynamic arrays where the size of each element can vary.
4. **Array of Function Pointers:** An array of pointers can hold pointers to functions, enabling the creation of function tables and callbacks.

VOID POINTER

- void pointer, also known as a generic pointer, is a pointer that does not have any data type associated with it. It can hold the address of any type of data, but it cannot be dereferenced directly because the compiler doesn't know the size or type of data.

Syntax:

```
void *ptr;
```

- void: Represents the absence of a data type. A `void` pointer can point to any type of data.
- ptr: Name of the `void` pointer variable.

Example:

```
#include <iostream>
int main() {
   int x = 10;
   float y = 3.14;
   char ch = 'A';
   void *ptr;
   ptr = &x; // Address of an integer variable
   std::cout << "Value of x through void pointer: " << *(static_cast<int*>(ptr)) << std::endl;
   ptr = &y; // Address of a float variable
   std::cout << "Value of y through void pointer: " << *(static_cast<float*>(ptr)) << std::endl;
   ptr = &ch; // Address of a char variable
   std::cout << "Value of ch through void pointer: " << *(static_cast<char*>(ptr)) << std::endl;
   return 0;
}
```

Output: Value of x through void pointer: 10
Value of y through void pointer: 3.14
Value of ch through void pointer: A

❖ Use Cases:

1. void pointers are used in memory allocation functions like malloc() and free() to allocate and deallocate memory for data of unknown types.
2. **Generic Functions:** void pointers are used in functions that need to accept arguments of different data types.
3. **Function Pointers:** void pointers can be used to hold addresses of functions of different signatures, enabling polymorphism and dynamic function calls.
4. **Data Structures:** void pointers can be used in implementing generic data structures like linked lists and trees where the data type of elements can vary.

Important Points:

- Dereferencing a void pointer directly is illegal because the compiler doesn't know the size or type of data it points to.
- Proper casting is required before dereferencing a `void` pointer to ensure type safety.
- Care should be taken when using `void` pointers to avoid type mismatch errors and undefined behavior.

REFERENCES

- a reference is an alias or an alternative name for an existing variable. It provides a way to access and manipulate the original variable using a different name. Unlike pointers, once a reference is initialized, it cannot be changed to refer to another variable.

Syntax:

```
data_type& reference_variable = original_variable;
```

- data_type: Represents the type of data that the reference refers to.
- reference_variable: Name of the reference variable.
- original_variable: Name of the original variable.

Example:

```
#include <iostream>
int main() {
   int x = 10;
   int& ref = x; // Reference variable ref refers to the original variable x
   std::cout << "Value of x: " << x << std::endl;
   std::cout << "Value of ref: " << ref << std::endl;
   ref = 20;
   std::cout << "New value of x: " << x << std::endl;
   std::cout << "New value of ref: " << ref << std::endl;
   return 0;
}
```

Output:
```
Value of x: 10
Value of ref: 10
New value of x: 20
New value of ref: 20
```

Use Cases:

1. **Passing Parameters to Functions:** References are commonly used to pass variables to functions by reference, allowing functions to modify the original variables directly.
2. **Avoiding Pointer Syntax:** References provide a safer and more intuitive alternative to pointers, especially when dealing with functions and complex data structures.
3. **Function Return Values:** Functions can return references to local variables or dynamically allocated memory, allowing the caller to manipulate the returned value directly.
4. **Operator Overloading:** References are often used in operator overloading to provide a more natural syntax for user-defined types.

Important Points:

- References must be initialized when they are declared and cannot be changed to refer to another variable.
- References cannot be null and must always refer to a valid object.
- References provide a safer alternative to pointers for passing parameters to functions and managing memory.

REFERENCE VS POINTER

- references and pointers are both used to manipulate data indirectly, but they have different characteristics and usage patterns.

❖ References.

1. Initialization:

- References must be initialized when they are declared and cannot be changed to refer to another object.
- They provide an alternative name or alias to an existing variable.

2. Syntax:

- References use the `&` operator in their declaration and initialization.
- They provide a cleaner syntax compared to pointers, especially in function parameters.

3. Null Values:

- References cannot be null and must always refer to a valid object.
- They guarantee that the object being referred to exists.

4. Dereferencing:

- References are automatically dereferenced by the compiler, so there is no need to use the dereference operator (`*`).

5. Reassignment:

- Once initialized, references cannot be reassigned to refer to another object.
- They provide a level of safety by preventing accidental reassignment.

6. Function Parameters:

- References are commonly used in function parameters to pass variables by reference, allowing the function to modify the original variables directly.

❖ Pointers:

1. Initialization:

- Pointers can be declared without initialization or initialized to null.
- They hold the memory address of another object.

2. Syntax:

- Pointers use the `*` operator to declare and dereference them.
- They provide more flexibility in terms of memory management and manipulation.

3. Null Values:

- Pointers can have null values, indicating that they are not pointing to any valid object.
- They allow for more dynamic memory management, but require explicit null checks to avoid runtime errors.

4. Dereferencing:

- Pointers require explicit dereferencing using the `*` operator to access the value stored at the memory address they point to.

5. Reassignment:

- Pointers can be reassigned to point to different objects during their lifetime.
- They provide flexibility but require careful management to avoid dangling pointers and memory leaks.

6. Function Parameters:

- Pointers can be used in function parameters to pass variables by reference or by value, providing more control over memory usage and data manipulation.

❖ Which to Use:

References:

- ➢ Use references when you want a cleaner syntax, especially for passing parameters to functions by reference.
- ➢ Use them when you are sure that the object being referred to exists throughout its lifetime.

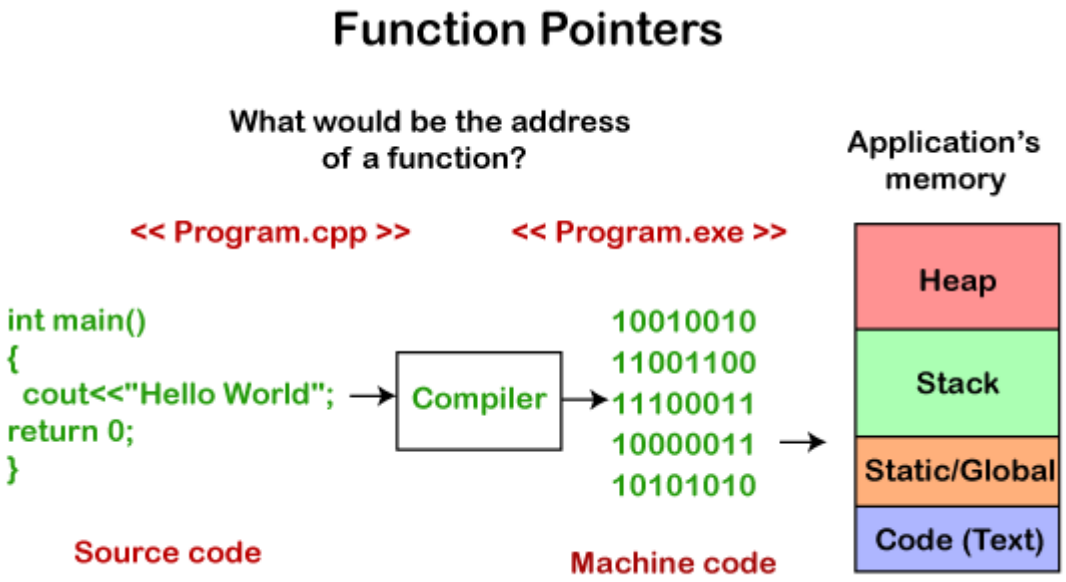
Pointers:

- ➢ Use pointers when you need more flexibility in memory management or when dealing with dynamic data structures like arrays or linked lists.
- ➢ Use them when you need to represent the absence of an object using null values.

FUNCTION POINTER

- As we know that pointers are used to point some variables; similarly, the function pointer is a pointer used to point functions.
- It is basically used to store the address of a function. We can call the function by using the function pointer, or we can also pass the pointer to another function as a parameter.

➢ What is the address of a function?



Syntax:

return_type (*pointer_name)(parameter_types);

- **return_type:** Represents the return type of the function that the pointer can point to.
- **pointer_name:** Name of the function pointer variable.
- **parameter_types:** Represents the types of parameters accepted by the function that the pointer can point to.

1. Return Type:

- Specifies the type of data that the function returns upon execution.
- It can be any valid data type, including fundamental types (int, float, char), user-defined types (struct, class), or even void.

2. Pointer Name:

- It is the name of the function pointer variable.
- It follows the same naming conventions as any other variable in C++.

3. Parameter Types:

- Represents the types of parameters accepted by the function that the pointer can point to.
- If the function takes no parameters, the parameter_types part can be left empty or specified as void.

Example:

```
#include <iostream>
int add(int a, int b) {
   return a + b;
}
int main() {
   int (*ptr)(int, int);
   ptr = add;
   int result = ptr(3, 4);
   std::cout << "Result: " << result << std::endl;
   return 0;
}
```

Output:

Result: 7

Example-2:

```
#include <iostream>
int add(int a, int b) {
   return a + b;
}
int subtract(int a, int b) {
   return a - b;
}
int main() {
   int (*operation)(int, int);
   operation = add;
   int result_add = operation(5, 3);
   std::cout << "Result of addition: " << result_add << std::endl;
   operation = subtract;
   int result_subtract = operation(5, 3);
   std::cout << "Result of subtraction: " << result_subtract << std::endl;
   return 0;
}
```

Output:

Result of addition: 8
Result of subtraction: 2

MEMORY MANAGEMENT

- Memory management in C++ refers to the process of allocating and deallocating memory for variables, objects, and data structures during program execution.
- It involves efficiently utilizing the available memory resources to store and manipulate data, as well as ensuring that memory is properly released when it is no longer needed. Memory management is crucial in C++ for several reasons.
  1. **Dynamic Memory Allocation:** C++ allows for dynamic memory allocation, where memory is allocated at runtime using operators like new and malloc(). Proper memory management ensures that memory is allocated and released appropriately to avoid memory leaks and optimize memory usage.
  2. **Data Structures:** Many data structures, such as arrays, linked lists, trees, and graphs, require memory allocation and deallocation to store and manage data efficiently. Memory management is essential for creating, manipulating, and destroying these data structures dynamically.
  3. **Object-Oriented Programming:** In C++, objects are created and destroyed dynamically using constructors and destructors. Memory management ensures that memory is allocated for objects when they are created and deallocated when they are destroyed, preventing memory leaks and resource wastage.
  4. **Efficient Resource Utilization:** Proper memory management improves the efficiency of resource utilization by releasing memory that is no longer needed. It helps prevent memory fragmentation and ensures that memory is available for other parts of the program.

❖ Memory Management Operators in C++.

1. new and delete Operators:

- **new:** Dynamically allocates memory for an object or an array and returns a pointer to the allocated memory.
- **delete:** Deallocates memory that was previously allocated using the new operator.

Example:

```
#include <iostream>
int main() {
   // Dynamic memory allocation for a single integer
   int *ptr = new int(10);
   // Output the value stored at the memory location pointed by ptr
   std::cout << "Value: " << *ptr << std::endl;
   // Deallocate the dynamically allocated memory
   delete ptr;
   return 0;
}
```

Output: Value: 10

.

2. malloc() and free() Functions:

- **malloc():** Allocates a block of memory of the specified size in bytes and returns a pointer to the allocated memory.
- **free():** Deallocates the memory block that was previously allocated using the malloc() function.

Example:

```
#include <iostream>
#include <cstdlib>
int main() {
   int *ptr = (int*)malloc(sizeof(int));
   *ptr = 20;
   std::cout << "Value: " << *ptr << std::endl;
   free(ptr);
   return 0;
}
```

Output: Value: 20

Example:

```
#include <iostream>
using namespace std;
int main() {
   int size;  // variable declaration
   cout << "Enter the size of the array : ";
   cin >> size; // Input the size of the array
   int *arr = new int[size];
   cout << "\nEnter the elements : ";
   for (int i = 0; i < size; i++) {
      cin >> arr[i];
   }
   cout << "\nThe elements that you have entered are: ";
   for (int i = 0; i < size; i++) {
      cout << arr[i] << ",";
   }
   delete[] arr;
   return 0;
}
```

Output:

```
Enter the size of the array : 3
Enter the elements : 1 2 3
The elements that you have entered are: 1,2,3,
The elements that you have entered are: 1,2,3,
```

MALLOC() VS NEW

- Both malloc() and new are used for dynamic memory allocation, but they have some key differences in usage and behavior.

❖ malloc():

1. **Function:**
   - malloc() is a function from the C standard library (declared in <cstdlib>) used for dynamic memory allocation.
2. **Return Type:**
   - It returns a void pointer (void*) to the allocated memory block.

3.Usage:

- Syntax: void* malloc(size_t size);
- It allocates a block of memory of the specified size in bytes.
- Memory allocated using malloc() is uninitialized; it does not call constructors for non-trivial types.

4.Failure Handling:

- If malloc() fails to allocate memory, it returns a null pointer (nullptr).

5.Deallocation:

- Memory allocated using malloc() should be deallocated using the free() function (free(pointer)).

❖ new:

1. **Operator:**
   - new is an operator in C++ used for dynamic memory allocation and object construction.
2. **Return Type:**
   - It returns a typed pointer to the allocated memory block.
3. **Usage:**
   - Syntax: new data_type; or new data_type[size];
   - It allocates memory for a single object of the specified type or an array of objects.
   - Memory allocated using new is initialized by calling constructors for non-trivial types.
4. **Failure Handling:**
   - If new fails to allocate memory, it throws a std::bad_alloc exception.
   - To handle allocation failure, use std::nothrow with new (new (std::nothrow) data_type;) to return a null pointer on failure.
5. **Deallocation:**
   - Memory allocated using new should be deallocated using the delete operator for a single object (delete pointer;) or the delete[] operator for arrays (delete[] pointer;).

Comparison:

- **malloc()** is a function from the C standard library, while new is an operator in C++.
- **malloc()** returns a void pointer, while new returns a typed pointer.
- Memory allocated using new initializes objects by calling their constructors, while memory allocated using malloc() remains uninitialized.
- **new** automatically calls constructors and delete automatically calls destructors, ensuring proper initialization and cleanup, respectively.
- **new** throws an exception on failure, while malloc() returns a null pointer.
- **malloc()** is commonly used in C programs, while new is preferred in C++ for its type safety and support for constructors/destructors.
- In modern C++ programming, new and delete are generally preferred over malloc() and free() due to their better type safety, exception handling, and support for constructors/destructors.

Example:

```
#include <iostream>
#include <cstdlib> // for malloc and free
int main() {
   int* ptr_malloc = (int*)malloc(sizeof(int));
   if (ptr_malloc == nullptr) {
      std::cerr << "Memory allocation failed using malloc!" << std::endl;
      return 1;
   }
   *ptr_malloc = 10;
   int* ptr_new = new int(20);
   std::cout << "Value allocated using malloc: " << *ptr_malloc << std::endl;
   std::cout << "Value allocated using new: " << *ptr_new << std::endl;
   free(ptr_malloc);
   delete ptr_new;
   return 0;
}
```

Output:

```
Value allocated using malloc: 10
Value allocated using new: 20
```

FREE VS DELETE

- In C++, both `free()` and `delete` are used for deallocating memory, but they have some key differences in usage and behavior.

❖ free():

1.Function:

- free() is a function from the C standard library (declared in <cstdlib>) used for deallocating memory allocated using malloc(), calloc(), or realloc().

2. Usage:

- Syntax: void free(void* ptr);
- It deallocates the memory block pointed to by the ptr pointer.

3. Behavior:

- free() does not call destructors for C++ objects. It simply releases the memory allocated by `malloc()`, `calloc()`, or `realloc()`.

4. **Memory Type:**

- It can be used to deallocate memory allocated by both C and C++ memory allocation functions (malloc(), calloc(), realloc()).

❖ delete:

1. Operator:

- delete is an operator in C++ used for deallocating memory allocated using new.

2. Usage:

- Syntax: delete ptr; or delete[] ptr;
- It deallocates the memory block pointed to by the `ptr` pointer.
- delete[]` is used for deallocating memory allocated for arrays, while `delete` is used for single objects.

3. Behavior:

- delete calls the destructor of the object pointed to by `ptr` before deallocating the memory.
- It is used for deallocating memory allocated by `new`.

4. Memory Type:

- It is used specifically for deallocating memory allocated using `new`.

❖ Comparison:

- free() is a function from the C standard library, while `delete` is an operator in C++.
- free() does not call destructors for C++ objects, while `delete` calls destructors before deallocating memory.
- free() can be used to deallocate memory allocated by both C and C++ memory allocation functions, while `delete` is specifically for memory allocated by `new`.

Example:

```
#include <iostream>
#include <cstdlib>
```

```
int main() {
   int* ptr_malloc = (int*)malloc(sizeof(int));
   *ptr_malloc = 10;
   int* ptr_new = new int(20);
   free(ptr_malloc);  // Using free for memory allocated by malloc
   delete ptr_new;    // Using delete for memory allocated by new
   return 0;
}
```

STRING

- string is an object of std::string class that represents sequence of characters.
- A string variable contains a collection of characters surrounded by double quotes.
- Strings are used for storing text.

Example

- Create a variable of type string and assign it a value:

```
string greeting = "skills";
```

Example-1:

```
#include <iostream>
using namespace std;
int main( ) {
   string s1 = "skills";
      char ch[] = { 'C', '+', '+'};
      string s2 = string(ch);
      cout<<s1<<endl;
      cout<<s2<<endl;
}
```

Output:

```
skills
C++
```

Example-2: String Comparison:

```
#include <iostream>
#include <string>
int main() {
   std::string str1 = "apple";
   std::string str2 = "banana";
   if (str1 == str2) {
      std::cout << "Strings are equal" << std::endl;
   } else if (str1 < str2) {
      std::cout << "str1 comes before str2" << std::endl;
   } else {
      std::cout << "str2 comes before str1" << std::endl;
   }
   return 0;
}
```

Output: str1 comes before str2

Example-3: String Copy:

```
#include <iostream>
#include <string>
int main() {
   std::string originalStr = "Hello";
   std::string copiedStr = originalStr;
   std::cout << "Copied string: " << copiedStr <<
std::endl;
   return 0;
}
```

Output: Copied string: Hello

Example-4: String Concatenation Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str1 = "Hello";
   std::string str2 = "World";
   std::string concatStr = str1 + " " + str2;
   std::cout << "Concatenated string: " << concatStr <<
std::endl;
   return 0;
}
```

Output: Concatenated string: Hello World

Example-5: String Length Example:

```
#include <iostream>
#include <string>
int main() {
   std::string longStr = "This is a long string";
   std::cout << "Length of the string: " <<
longStr.length() << std::endl;
   return 0;
}
```

Output: Length of the string: 20

String Function

Function	Description
1. int compare(const string& str)	It is used to compare two string objects.
2. int length()	It is used to find the length of the string.
3. void swap(string& str)	It is used to swap the values of two string objects.
4. string substr(int pos,int n)	It creates a new string object of n characters.
5. int size()	It returns the length of the string in terms of bytes.
6. void resize(int n)	It is used to resize the length of the string up to n characters.
7. string& replace(int pos,int len,string& str)	It replaces portion of the string that begins at character position pos and spans len characters.
8. string& append(const string& str)	It adds new characters at the end of another string object.
9. char& at(int pos)	It is used to access an individual character at specified position pos.
10. int find(string& str,int pos,int n)	It is used to find the string specified in the parameter.
11. int find_first_of(string& str,int pos,int n)	It is used to find the first occurrence of the specified sequence.
12. int find_first_not_of(string& str,int pos,int n	It is used to search the string for the first character that does not match.
13. int find_last_of(string& str,int pos,int n)	It is used to search the string for the last character of specified sequence.
14. int find_last_not_of(string& str,int pos)	It searches for the last character that does not match with the specified sequence.
15. string& insert()	It inserts a new character before the character indicated by the position pos.
16. int max_size()	It finds the maximum length of the string.
17. void push_back(char ch)	It adds a new character ch at the end of the string.
18. void pop_back()	It removes a last character of the string.
19. string& assign()	It assigns new value to the string.
20. int copy(string& str)	It copies the contents of string into another.
21. char& back()	It returns the reference of last character.
22. Iterator begin()	It returns the reference of first character.
23. int capacity()	It returns the allocated space for the string.
24. const_iterator cbegin()	It points to the first element of the string.
25. const_iterator cend()	It points to the last element of the string.
26. void clear()	It removes all the elements from the string.
27. const_reverse_iterator crbegin()	It points to the last character of the string.
28. const_char* data()	It copies the characters of string into an array.
29. bool empty()	It checks whether the string is empty or not.
30. string& erase()	It removes the characters as specified.
31. char& front()	It returns a reference of the first character.
32. string& operator+=()	It appends a new character at the end of the string.
33. string& operator=()	It assigns a new value to the string.
34. char operator[](pos)	It retrieves a character at specified position pos.
35. int rfind()	It searches for the last occurrence of the string.
36. iterator end()	It references the last character of the string.
37. reverse_iterator rend()	It points to the first character of the string.
38. void shrink_to_fit()	It reduces the capacity and makes it equal to the size of the string.
39. char* c_str()	It returns pointer to an array that contains null terminated sequence of characters.
40. const_reverse_iterator crend()	It references the first character of the string.
41. reverse_iterator rbegin()	It reference the last character of the string.
42. void reserve(inr len)	It requests a change in capacity.
43. allocator_type get_allocator();	It returns the allocated object associated with the string.

1. int compare(const string& str):

- **Description:** Compares the string with another string.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str1 = "apple";
    std::string str2 = "banana";
    int result = str1.compare(str2);
    if (result == 0) {
        std::cout << "Strings are equal" << std::endl;
    } else if (result < 0) {
        std::cout << "str1 comes before str2" <<
std::endl;
    } else {
        std::cout << "str2 comes before str1" <<
std::endl;
    }
    return 0;
}
```

Output: str1 comes before str2

2. int length():

- **Description:** Returns the length of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello";
    int len = str.length();
    std::cout << "Length of the string: " << len <<
std::endl;
    return 0;
}
```

Output: Length of the string: 5

3. void swap(string& str):

Description: Swaps the content of the string with another string.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str1 = "Hello";
    std::string str2 = "World";
    str1.swap(str2);
    std::cout << "str1: " << str1 << std::endl;
    std::cout << "str2: " << str2 << std::endl;
    return 0;
}
```

Output:

str1: World
str2: Hello

4. string substr(int pos, int n):

Description: Returns a substring of length `n` starting from position `pos`.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, World";
    std::string subStr = str.substr(7, 5);
    std::cout << "Substring: " << subStr <<
std::endl;
    return 0;
}
```

Output: Substring: World

5. int size():

Description: Returns the size of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello";
    int size = str.size();
    std::cout << "Size of the string: " << size <<
std::endl;
    return 0;
}
```

Output: Size of the string: 5

6. void resize(int n):

Description: Resizes the string to have `n` characters.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello";
    str.resize(8);
    std::cout << "Resized string: " << str <<
std::endl;
    return 0;
}
```

Output: Resized string: Hello\0\0\0

7. string& replace(int pos, int len, string& str):

Description: Replaces a substring of length `len` starting from position `pos` with another string `str`.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, World";
    std::string replacement = "Universe";
    str.replace(7, 5, replacement);
    std::cout << "Modified string: " << str <<
std::endl;
    return 0;
}
```

Output: Modified string: Hello, Universe

8. string& append(const string& str):

Description: Appends another string `str` at the end of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello";
    str.append(", World");
    std::cout << "Appended string: " << str <<
std::endl;
    return 0;
}
```

Output: Appended string: Hello, World

9. char& at(int pos):

Description: Returns the reference to the character at position `pos`.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello";
    char& ch = str.at(1);
    ch = 'E'; // Modify the character at position 1
std::cout << "Modified string: " << str << std::endl;
    return 0;
}
```

Output: Modified string: HEllo

11. int find_first_of(string& str, int pos, int n):

it is used first occurrence of any character in the given string `str` within the specified range `[pos, pos+n)`. Returns the position of the first occurrence of any character in `str` within the range.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, world!";
    int pos = str.find_first_of("aeiou", 0, 5);
    std::cout << "First vowel found at position: " << pos <<
std::endl;
    return 0;
}
```

Output: First vowel found at position: 1

10. int find(string& str, int pos, int n):

Description: Searches for the first occurrence of the substring `str` within the string, starting from position `pos` and considering up to `n` characters.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, World";
    int pos = str.find("World", 0, 12);
    std::cout << "Position of substring: " << pos
<< std::endl;
    return 0;
}
```

Output: Position of substring: 7

6. void resize(int n):

Description: Resizes the string to have `n` characters.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello";
    str.resize(8);
    std::cout << "Resized string: " << str <<
std::endl;
    return 0;
}
```

Output: Resized string: Hello\0\0\0

7. string& replace(int pos, int len, string& str):

Description: Replaces a substring of length `len` starting from position `pos` with another string `str`.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, World";
    std::string replacement = "Universe";
    str.replace(7, 5, replacement);
    std::cout << "Modified string: " << str <<
std::endl;
    return 0;
}
```

Output: Modified string: Hello, Universe

8. string& append(const string& str):

Description: Appends another string `str` at the end of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello";
    str.append(", World");
    std::cout << "Appended string: " << str <<
std::endl;
    return 0;
}
```

Output: Appended string: Hello, World

9. char& at(int pos):

Description: Returns the reference to the character at position `pos`.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello";
    char& ch = str.at(1);
    ch = 'E'; // Modify the character at position 1
std::cout << "Modified string: " << str << std::endl;
    return 0;
}
```

Output: Modified string: HEllo

11. int find_first_of(string& str, int pos, int n):

it is used first occurrence of any character in the given string `str` within the specified range `[pos, pos+n)`. Returns the position of the first occurrence of any character in `str` within the range.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, world!";
    int pos = str.find_first_of("aeiou", 0, 5);
    std::cout << "First vowel found at position: " << pos <<
std::endl;
    return 0;
}
```

Output: First vowel found at position: 1

10. int find(string& str, int pos, int n):

Description: Searches for the first occurrence of the substring `str` within the string, starting from position `pos` and considering up to `n` characters.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, World";
    int pos = str.find("World", 0, 12);
    std::cout << "Position of substring: " << pos
<< std::endl;
    return 0;
}
```

Output: Position of substring: 7

12. int find_first_not_of(string& str, int pos, int n):

This function searches the string for the first character that does not match any character in the given string `str` within the specified range `[pos, pos+n)`. Returns the position of the first character that does not match any character in `str` within the range.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, world!";
    int pos = str.find_first_not_of("Helo, ", 0, 5);
    std::cout << "First character not found at position: "
<< pos << std::endl;
    return 0;
}
```

Output: First character not found at position: 5

13. int find_last_of(string& str, int pos, int n):

This function searches the string for the last occurrence of any character in the given string `str` within the specified range `[0, pos+n)`. Returns the position of the last occurrence of any character in `str` within the range.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, world!";
    int pos = str.find_last_of("l", str.length()-1, 3);
    std::cout << "Last 'l' found at position: " << pos <<
std::endl;
    return 0;
}
```

Output: Last 'l' found at position: 9

14. int find_last_not_of(string& str, int pos):

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, world!";
    int pos = str.find_last_not_of("dlro", str.length()-1);
    std::cout << "Last character not found at position: " <<
pos << std::endl;
    return 0;
}
```

Output: Last character not found at position: 10

15. string& insert():

This function inserts characters into the string at a specified position. Returns a reference to the modified string.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "world!";
    str.insert(0, "Hello, ");
    std::cout << "Modified string: " << str <<
std::endl;
    return 0;
}
```

Output: Modified string: Hello, world!

16. int max_size():

This function returns the maximum number of characters the string can hold.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, world!";
    std::cout << "Maximum size of the string: " <<
str.max_size() << std::endl;
    return 0;
}
```

Output (may vary): Maximum size of the string: 9223372036854775807

17. void push_back(char ch):

This function appends a character to the end of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello";
    str.push_back('!');
    std::cout << "Modified string: " << str << std::endl;
    return 0;
}
```

Output: Modified string: Hello!

12. int find_first_not_of(string& str, int pos, int n):

This function searches the string for the first character that does not match any character in the given string `str` within the specified range `[pos, pos+n)`. Returns the position of the first character that does not match any character in `str` within the range.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, world!";
    int pos = str.find_first_not_of("Helo, ", 0, 5);
    std::cout << "First character not found at position: "
<< pos << std::endl;
    return 0;
}
```

Output: First character not found at position: 5

13. int find_last_of(string& str, int pos, int n):

This function searches the string for the last occurrence of any character in the given string `str` within the specified range `[0, pos+n)`. Returns the position of the last occurrence of any character in `str` within the range.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, world!";
    int pos = str.find_last_of("l", str.length()-1, 3);
    std::cout << "Last 'l' found at position: " << pos <<
std::endl;
    return 0;
}
```

Output: Last 'l' found at position: 9

14. int find_last_not_of(string& str, int pos):

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = “Hello, world!";
    int pos = str.find_last_not_of("dlro", str.length()-1);
    std::cout << "Last character not found at position: " <<
pos << std::endl;
    return 0;
}
```

Output: Last character not found at position: 10

15. string& insert():

This function inserts characters into the string at a specified position. Returns a reference to the modified string.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "world!";
    str.insert(0, "Hello, ");
    std::cout << "Modified string: " << str <<
std::endl;
    return 0;
}
```

Output: Modified string: Hello, world!

16. int max_size():

This function returns the maximum number of characters the string can hold.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello, world!";
    std::cout << "Maximum size of the string: " <<
str.max_size() << std::endl;
    return 0;
}
```

Output (may vary): Maximum size of the string: 9223372036854775807

17. void push_back(char ch):

This function appends a character to the end of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
    std::string str = "Hello";
    str.push_back('!');
    std::cout << "Modified string: " << str << std::endl;
    return 0;
}
```

Output: Modified string: Hello!

18. void pop_back():

- This function removes the last character from the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello!";
   str.pop_back();
   std::cout << "Modified string: " << str <<
std::endl;
   return 0;
}
```

Output: Modified string: Hello

19. string& assign():

This function assigns new contents to the string. Returns a reference to the modified string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "initial";
   str.assign("assigned");
   std::cout << "Modified string: " << str <<
std::endl;
   return 0;
}
```

Output: Modified string: assigned

20. int copy(string& str): This function copies characters from the string into a character array. Returns the number of characters copied.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   char buffer[10];
   int length = str.copy(buffer, 5);
   buffer[length] = '\0'; // Null-terminate the copied
characters
   std::cout << "Copied string: " << buffer << std::endl;
   return 0;
}
```

Output: Copied string: Hello

21. char& back(): Returns a reference to the last character in the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   char& lastChar = str.back();
   std::cout << "Last character: " <<
lastChar << std::endl;
   return 0;
}
```

Output:

Last character: o

22. Iterator begin(): Returns an iterator pointing to the first character of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   std::string::iterator it = str.begin();
   std::cout << "First character: " << *it
<< std::endl;
   return 0;
}
```

Output: First character: H

23. int capacity(): Returns the allocated storage capacity of the string.

Example:

```
include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   int cap = str.capacity();
   std::cout << "Capacity: " << cap <<
std::endl;
   return 0;
}
```

Output: capacity: 15

24. const_iterator cbegin(): Returns a constant iterator pointing to the first character of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   std::string::const_iterator it =
str.cbegin();
std::cout << "First character: " << *it << std::endl;
   return 0;
}
```

Output: First character: H

25. const_iterator cend(): Returns a constant iterator pointing to one past the last character of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   std::string::const_iterator it = str.cend();
std::cout << "Last character: " << *(--it) << std::endl;
   return 0;
}
```

Output: Last character: o

26. void clear(): Clears the contents of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
std::cout << "Before clearing: " << str << std::endl;
   str.clear();
std::cout << "After clearing: " << str << std::endl;
   return 0;
}
```

Output: Before clearing: Hello
After clearing:

27. const_reverse_iterator crbegin(): Returns a constant reverse iterator pointing to the last character of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
std::string::const_reverse_iterator it = str.crbegin();
std::cout << "Last character: " << *it << std::endl;
   return 0;
}
```

Output: Last character: o

28. const_char data()`: Returns a pointer to an array that contains the same sequence of characters as the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   const char* charPtr = str.data();
std::cout << "Characters: " << charPtr << std::endl;
   return 0;
}
```

Output: Characters: Hello

29. bool empty(): Checks if the string is empty (i.e., has no characters).

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   if (str.empty()) {
std::cout << "String is empty" << std::endl;
   } else {
std::cout << "String is not empty" << std::endl;
   }
   return 0;
}
```

Output: String is not empty

30. string& erase(): Erases part of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello, World!";
std::cout << "Before erase: " << str << std::endl;
   str.erase(5, 7); // Erase 7 characters
starting from index 5
std::cout << "After erase: " << str << std::endl;
   return 0;
}
```

Output: Before erase: Hello, World!
After erase: Hello!

31. char& front(): Returns a reference to the first character of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   char& firstChar = str.front();
   std::cout << "First character: " << firstChar
<< std::endl;
   return 0;
}
```

Output: First character: H

32. string& operator+=(const string& str): Appends the contents of `str` to the end of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   str += ", world!";
   std::cout << "Concatenated string: " <<
str << std::endl;
   return 0;
}
```

Output: Concatenated string: Hello, world!

33. string& operator=(const string& str): Assigns the value of `str` to the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str1 = "Hello";
   std::string str2 = "Goodbye";
   str1 = str2;
std::cout << "str1 after assignment: " << str1 << std::endl;
   return 0;}
```

Output: str1 after assignment: Goodbye

34. char operator[](size_type pos): Returns a reference to the character at position `pos` in the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   char character = str[1];
   std::cout << "Character at position 1: "
<< character << std::endl;
   return 0;
}
```

Output:

Character at position 1: e

35. int rfind(const string& str): Finds the last occurrence of `str` within the string.

```
#include <iostream>
#include <string>
int main() {
   std::string str = "hello world hello";
   int lastIndex = str.rfind("hello");
   std::cout << "Last occurrence of 'hello' is at
index: " << lastIndex << std::endl;
   return 0;
}
```

Output:

Last occurrence of 'hello' is at index: 12

36. iterator end(): Returns an iterator referring to the past-the-end character of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   std::string::iterator it = str.end();
std::cout << "Iterator points to: " << *it << std::endl;
   return 0;
}
```

Output: Iterator points to: (Note: Printing `*it` here won't print a visible character as it points to the past-the-end character.)

37. reverse_iterator rend(): Returns a reverse iterator referring to the theoretical element right before the first character in the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   std::string::reverse_iterator it = str.rend();
 std::cout << "Reverse iterator points to: " << *it <<
std::endl;
   return 0;
}
```

Output: Reverse iterator points to:

38. void shrink_to_fit(): Requests the string to reduce its capacity to fit its size.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   std::cout << "Capacity before shrinking:
" << str.capacity() << std::endl;
   str.shrink_to_fit();
   std::cout << "Capacity after shrinking: "
<< str.capacity() << std::endl;
   return 0;
}
```

Output: Capacity before shrinking: 15
Capacity after shrinking: 5

39. const char* c_str() const noexcept: Returns a pointer to an array that contains a null-terminated sequence of characters (i.e., a C-string representation of the string).

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   const char* cString = str.c_str();
std::cout << "C-style string: " << cString << std::endl;
   return 0;  }
```

Output: C-style string: Hello

40. const_reverse_iterator crend() const noexcept: Returns a constant reverse iterator referring to the theoretical element right before the first character in the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   std::string::const_reverse_iterator it =
str.crend();
   std::cout << "Constant reverse iterator
points to: " << *it << std::endl;
   return 0;
}
```

Output: Constant reverse iterator points to:

41. reverse_iterator rbegin() noexcept: Returns a reverse iterator pointing to the last character of the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   std::string::reverse_iterator it = str.rbegin();
   std::cout << "Reverse iterator points to: "
<< *it << std::endl;
   return 0;
}
```

Output: Reverse iterator points to: o

42. void reserve(size_type len): Requests that the string capacity be enough to contain at least `len` characters.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   std::cout << "Capacity before reserving: " << str.capacity() << std::endl;
   str.reserve(20);
   std::cout << "Capacity after reserving: " << str.capacity() << std::endl;
   return 0;
}
```

Output:

```
Capacity before reserving: 15
Capacity after reserving: 20
```

43.allocator_type get_allocator() const noexcept`: Returns the allocator associated with the string.

Example:

```
#include <iostream>
#include <string>
int main() {
   std::string str = "Hello";
   std::allocator<char> alloc = str.get_allocator();
   std::cout << "Allocator's maximum size: " << alloc.max_size() << std::endl;
   return 0;
}
```

Output:

```
Allocator's maximum size: 9223372036854775807
(Note: This function doesn't have a direct output to demonstrate, but it returns an
allocator type associated with the string.)
```

EXCEPTION HANDLING

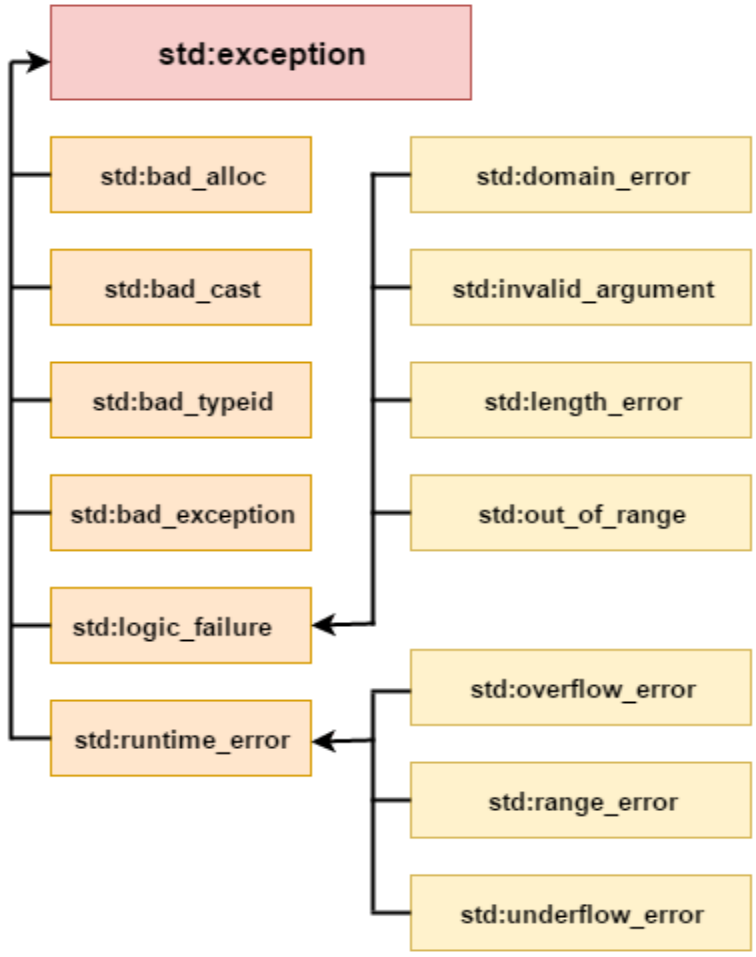
- Exception handling in C++ is a mechanism that allows you to handle exceptional conditions or errors that occur during the execution of a program in a structured and controlled manner.
- Exception Handling in C++ is a process to handle runtime errors.

❖ Advantage of Exception Handling:

1. **Robustness:** Prevents unexpected program termination by gracefully handling errors.
2. **Separation of Concerns:** Separates error detection from error handling, enhancing modularity.
3. **Cleanup:** Enables proper resource cleanup, preventing resource leaks.
4. **Readability:** Leads to cleaner, more readable code by separating error handling from main logic.
5. **Flexibility:** Offers flexible error-handling strategies tailored to specific application needs.
6. **Debugging:** Facilitates debugging by providing detailed error information.

❖ Exception Classes:

- in C++ standard exceptions are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:



Exception	Description
1. std::exception	:It is an exception and parent class of all standard C++ exceptions.
2. std::logic_failure	:It is an exception that can be detected by reading a code.
3. std::runtime_error	:It is an exception that cannot be detected by reading a code.
4. std::bad_exception	:It is used to handle the unexpected exceptions in a c++ program.
5. std::bad_cast	:This exception is generally be thrown by dynamic_cast.
6. std::bad_typeid	:This exception is generally be thrown by typeid.
7. std::bad_alloc	:This exception is generally be thrown by new.

❖ Exception Handling Keywords:

➢ we use 3 keywords to perform exception handling:

- try
- catch, and
- throw

1. **try:** Marks a block of code where exceptions might be thrown.
2. **throw:** Explicitly raises an exception within the try block.
3. **catch:** Catches and handles exceptions thrown within the try block.

❖ C++ try/catch:

- In C++ programming, exception handling is performed using try/catch statement. The C++ try block is used to place the code that may occur exception. The catch block is used to handle the exception.

Example-1 :

```
#include <iostream>
using namespace std;
float division(int x, int y) {
  return (x/y);
}
int main () {
  int i = 50;
  int j = 0;
  float k = 0;
    k = division(i, j);
    cout << k << endl;
  return 0;
}
```

Output:

Floating point exception (core dumped)

Example-2:

```
 include <iostream>
using namespace std;
float division(int x, int y) {
  if( y == 0 ) {
    throw "Attempted to divide by zero!";
  }
  return (x/y);
}
int main () {
  int i = 25;
  int j = 0;
  float k = 0;
  try {
    k = division(i, j);
    cout << k << endl;
  }catch (const char* e) {
    cerr << e << endl;
  }
  return 0;
}
```

Output:

Attempted to divide by zero!

❖ C++ User-Defined Exceptions:

- The new exception can be defined by overriding and inheriting exception class functionality.

Example:

```
#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception{
   public:
      const char * what() const throw()
      {
         return "Attempted to divide by zero!\n";
      } };
int main()
{
   try
   {
      int x, y;
      cout << "Enter the two numbers : \n";
      cin >> x >> y;
      if (y == 0)
      {
         MyException z;
         throw z;
      }
      else
      {
         cout << "x / y = " << x/y << endl;
      }  }
   catch(exception& e)
   {
      cout << e.what();
   } }
```

Output: Enter the two numbers :

```
10
2
x / y = 5
```

Example 1: Using try and catch to handle a division by zero error.

```
#include <iostream>
int main() {
   try {
      int x = 10;
      int y = 0;
```

```
        if (y == 0)
            throw "Division by zero!";
        std::cout << "Result: " << x / y << std::endl;
    } catch (const char* error) {
        std::cerr << "Error: " << error << std::endl;
    }
    return 0; }
```

Output: Error: Division by zero!

Example 2: Using try, throw, and catch to handle a custom exception.

```
#include <iostream>
#include <string>
class MyException {
private:
    std::string message;
public:
    MyException(const std::string& msg) : message(msg) {}
    const std::string& what() const {
        return message;
    } };
int main() {
    try {
        throw MyException("Custom Exception: Something went wrong!");
    } catch (const MyException& e) {
        std::cerr << "Error: " << e.what() << std::endl;
}
    return 0; }
```

Output: Error: Custom Exception: Something went wrong!

Example 3: Using try, throw, and catch to handle standard library exceptions.

```
#include <iostream>
#include <stdexcept>
int main() {
    try {
        int x = 10;
        int y = 0;
        if (y == 0)
            throw std::runtime_error("Division by zero!");
        std::cout << "Result: " << x / y << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
}
    return 0; }
```

Output: Error: Division by zero!

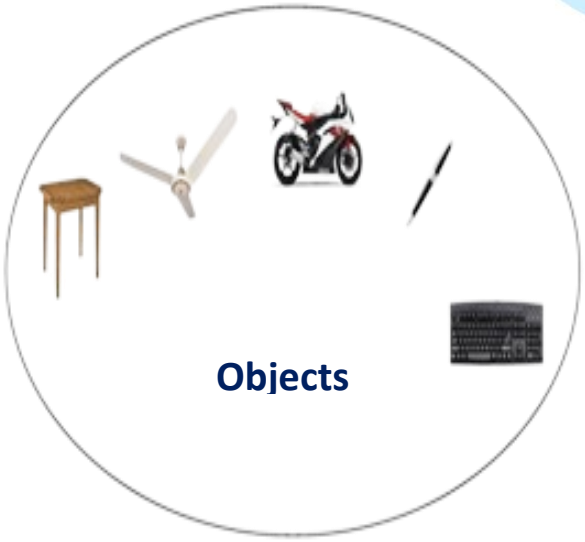
OOPS CONCEPTS
Object Oriented Programming System

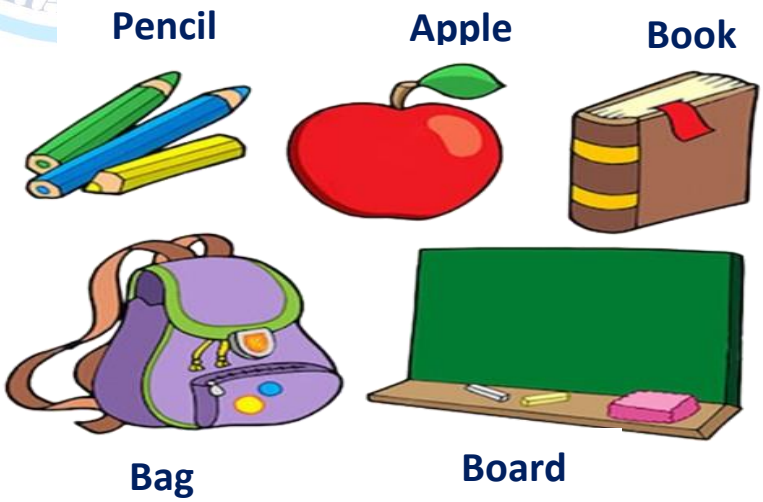
- Object-Oriented Programming is a methodology or paradigm to design a program using **classes** and **objects.**
- The programming paradigm where everything is represented as an **object** is known as truly **object-oriented programming** language. Smalltalk is considered as the first truly object-oriented programming language.
- It simplifies the software development and maintenance by providing following concepts.
    1. Object
    2. Class
    3. Inheritance
    4. Polymorphism
    5. Abstraction
    6. Encapsulation

1. Object:

- Object means a real word entity such as pen, chair, table etc.
- Any entity that has **state** and **behavior** is known as an object.
- For example: chair, pen, table, keyboard, bike etc. It can be **physical** and **logical.**
- **State** tells us how the object looks or what properties it has. **Behavior** tells us what the object does.
- object is a specific **instance** of a class.
- **instance** is a specific realization of any objects.

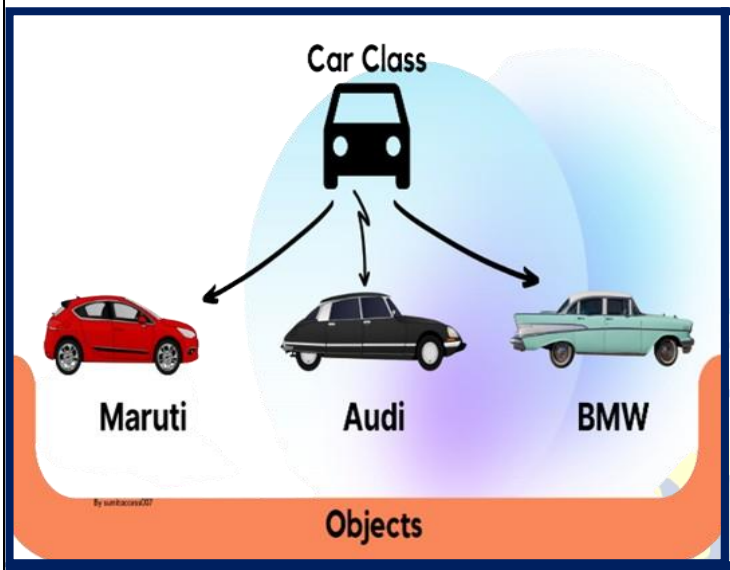
Example:

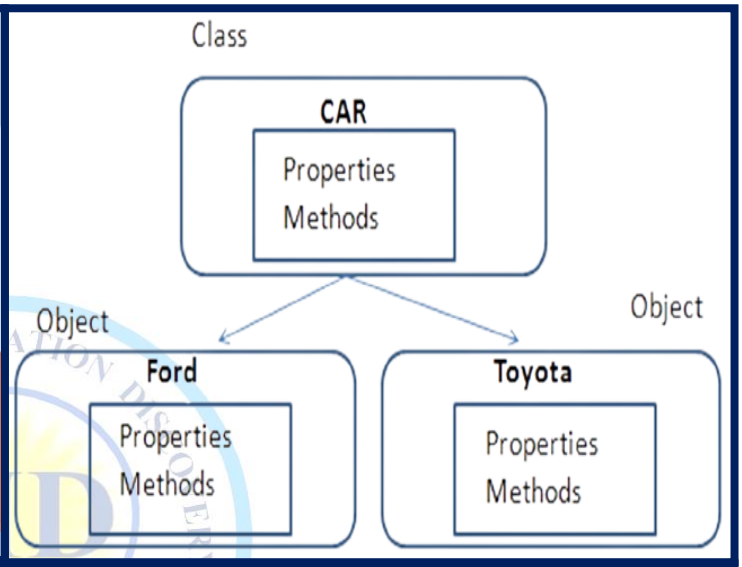


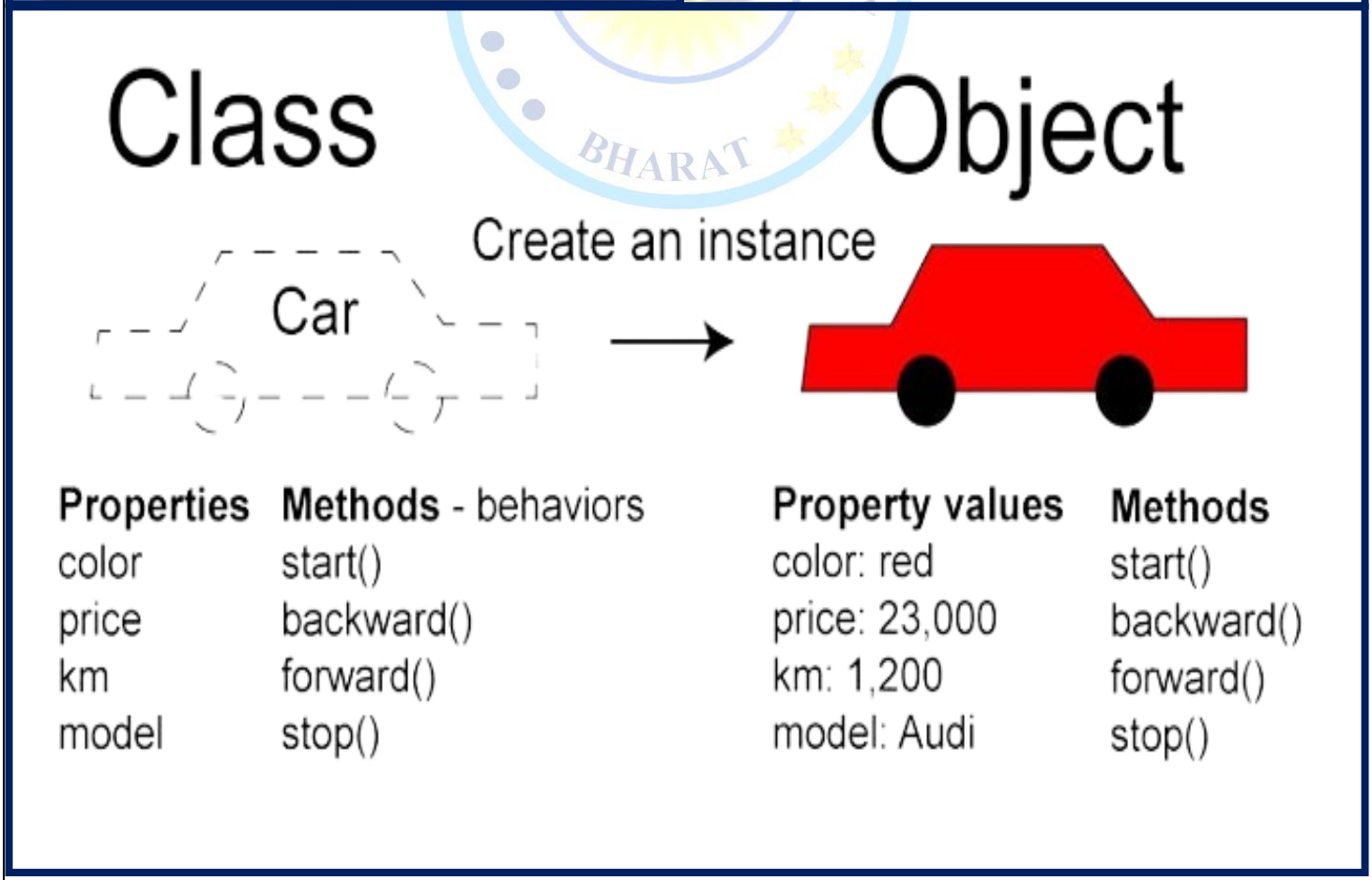


2. Class:

- Collection of **objects** is called class. It is a logical entity.
- To create objects, we required some model or plan or template or blue print, which is nothing but class.
- We can write a class to represent **properties(attributes)** and **actions (behaviour)** of object.
- **Properties** can be represented by **variable**
- **Action** can be represented by **methods.**
- Hence class contains both **variables** and **method.**
- a class is a template definition of the methods and variables in a particular kind of object. Thus, an object is a specific **instance of a class**.

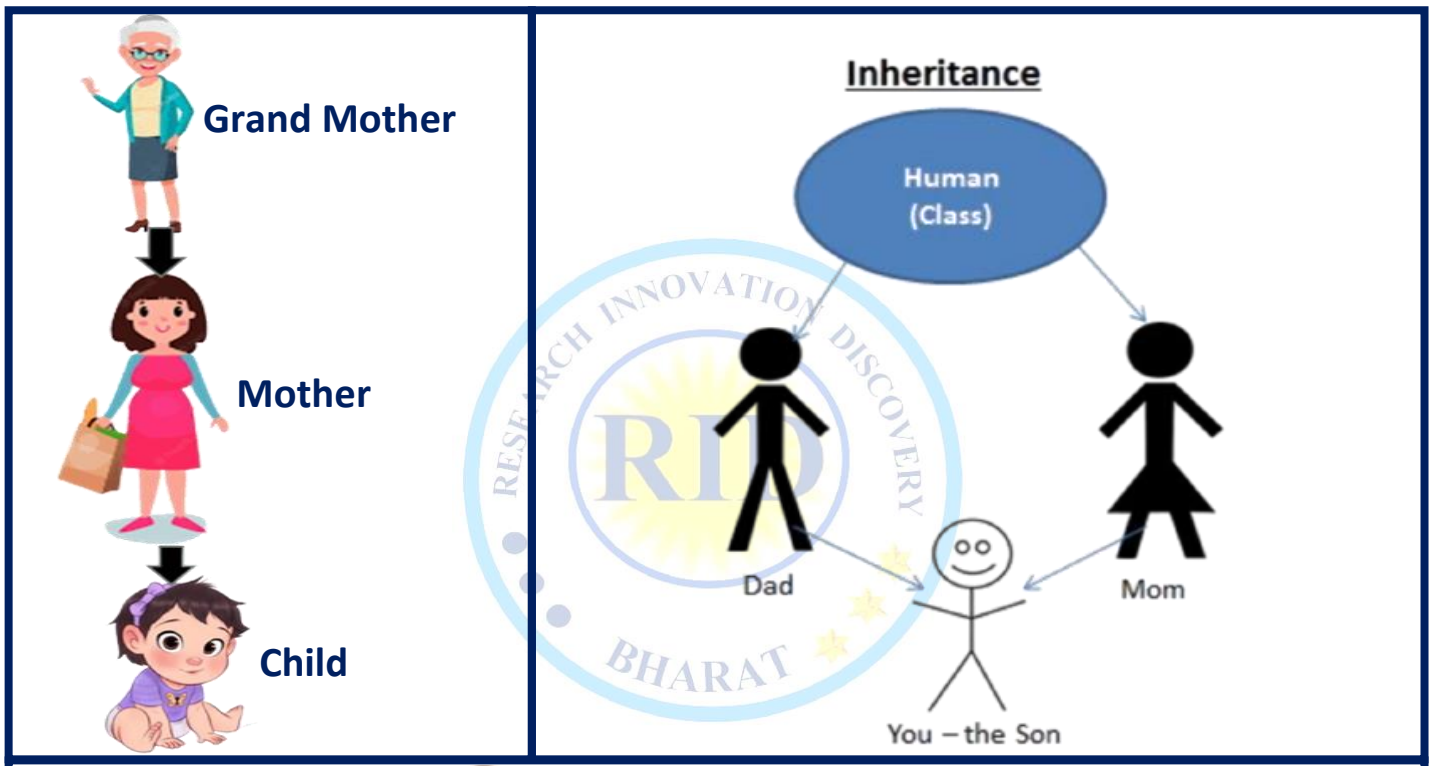


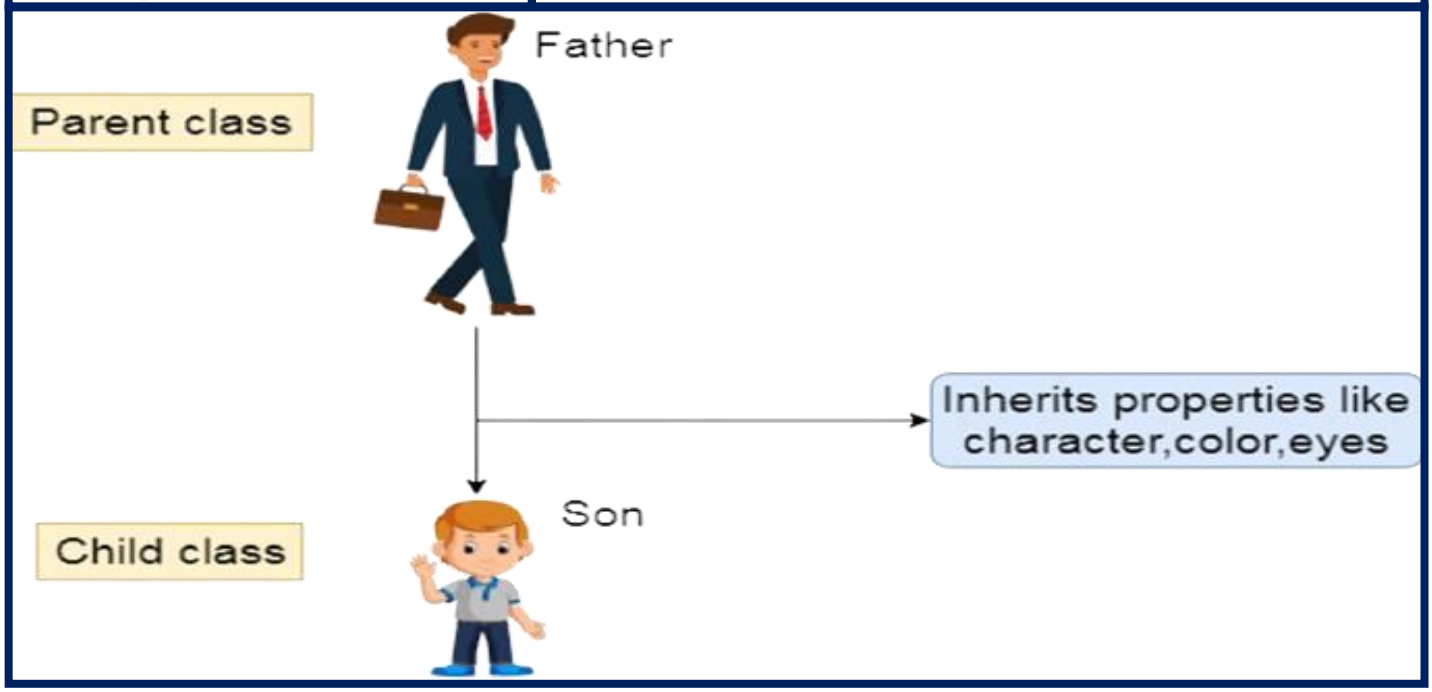




3. Inheritance:

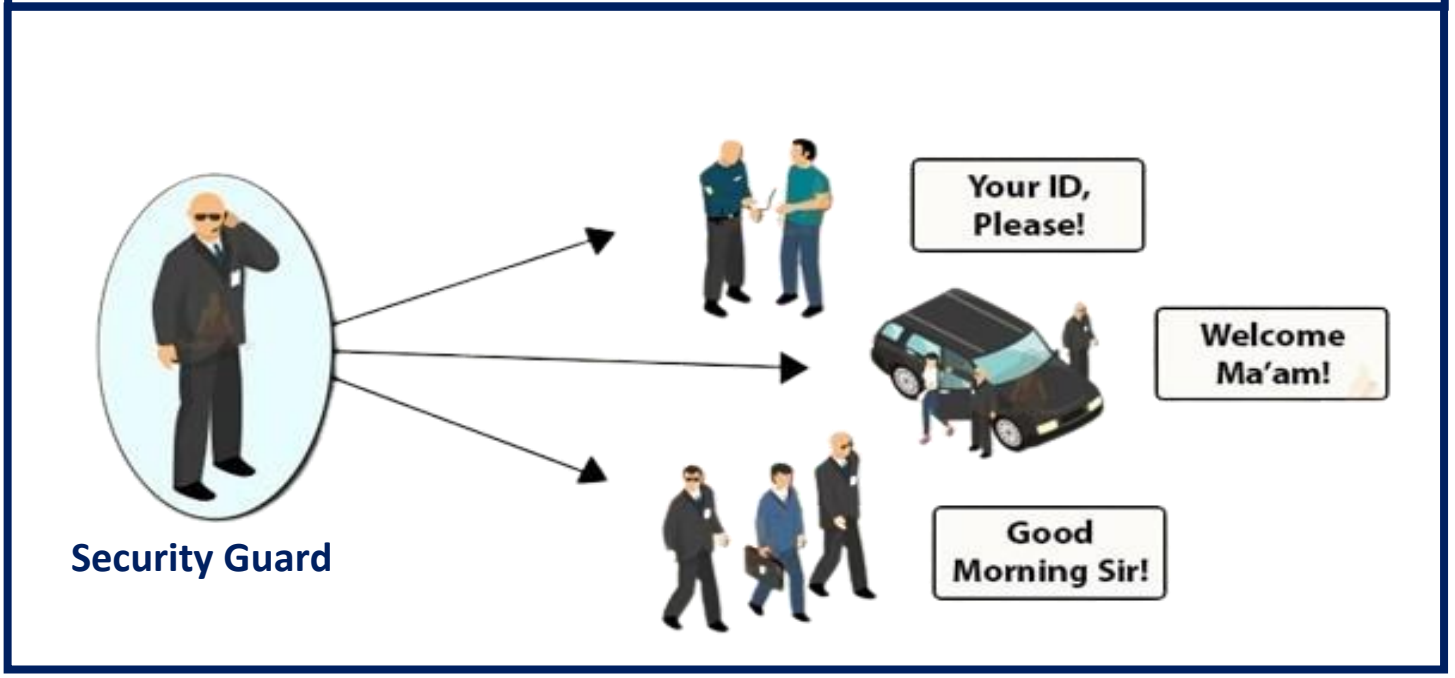
- When one object acquires all the **properties** and **behaviours** of **parent object** i.e. known as inheritance. It provides code reusability. It is used to achieve runtime **polymorphism.**
- **Sub class -** Subclass or Derived Class refers to a class that receives properties from another class.
- **Super class -** The term "Base Class" or "Super Class" refers to the class from which a subclass inherits its properties.
- **Reusability -** when we wish to create a new class, but an existing class already contains some of the code we need, we can generate our new class from the old class that is inheritance.





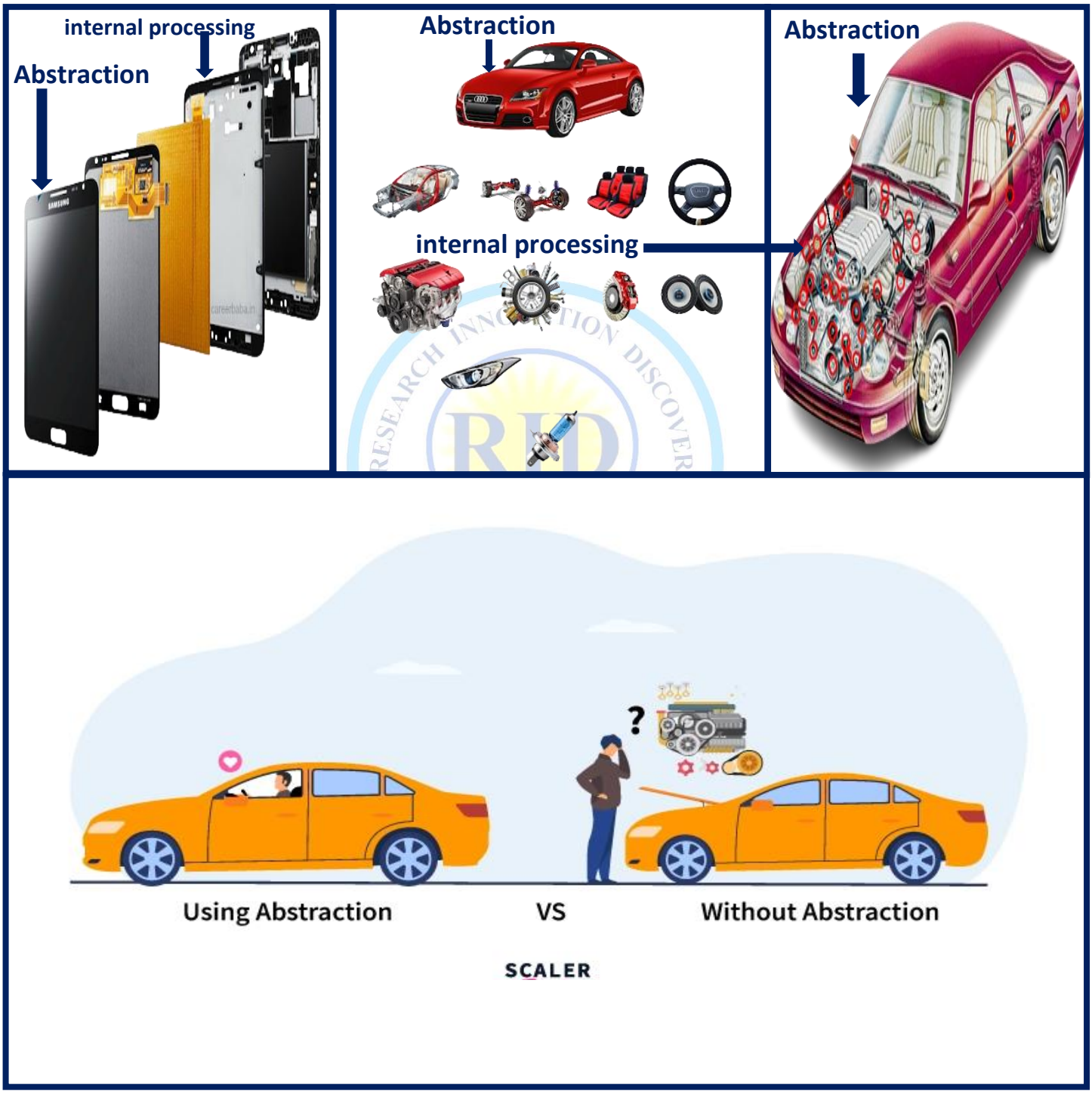
4. Polymorphism:

- When one task is performed by different ways i.e. known as polymorphism.
- Polymorphism means "many forms"
- Imagine a man named Sangam. Sangam has different roles depending on the context:
  - ✓ At work, Sangam is an Engineer.
  - ✓ At home, Sangam is a father.
  - ✓ In a social gathering, Sangam is a Friend.
  - ✓ In flight, Sangam is passenger.



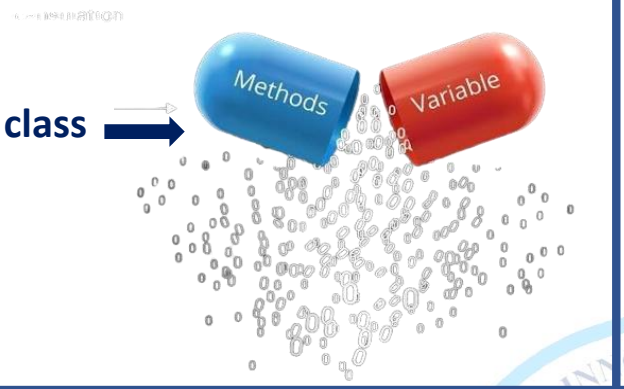
5. Abstraction:

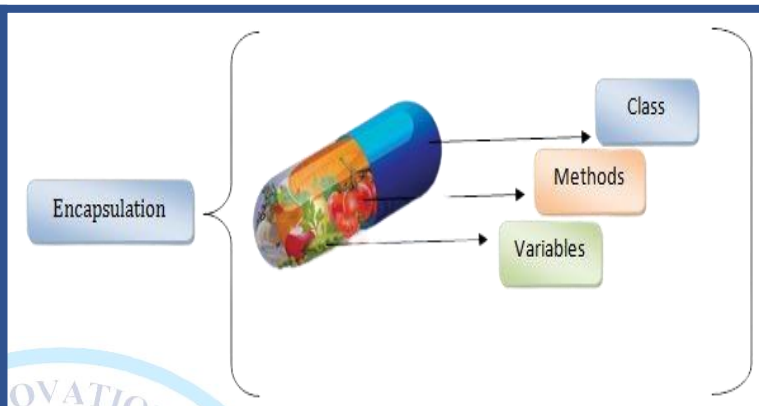
- Hiding **internal** details and showing **functionality** is known as abstraction.
- Data abstraction is the process of exposing to the outside world only the information that is absolutely necessary while concealing implementation or background information. For example: phone call and Car. we don't know the internal processing.
- n C++, we use abstract class and interface to achieve abstraction.

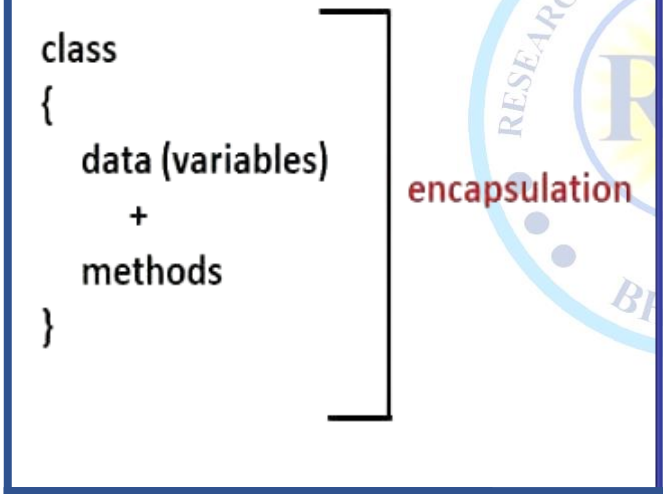


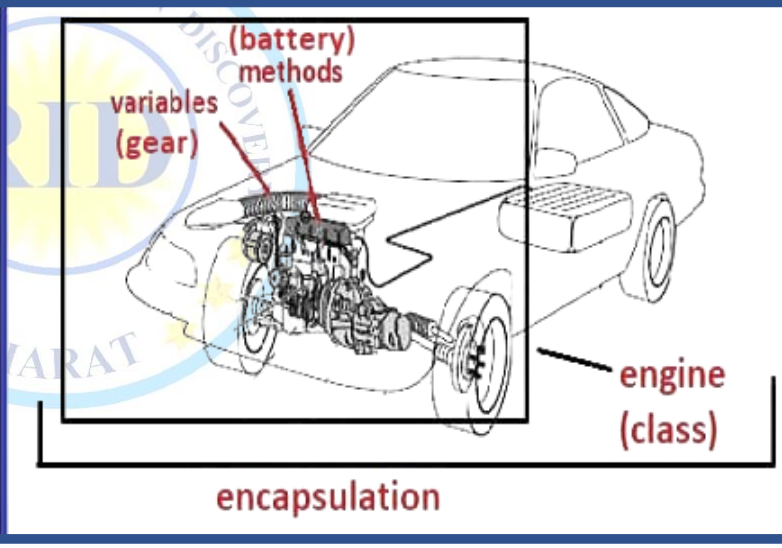
6. Encapsulation:

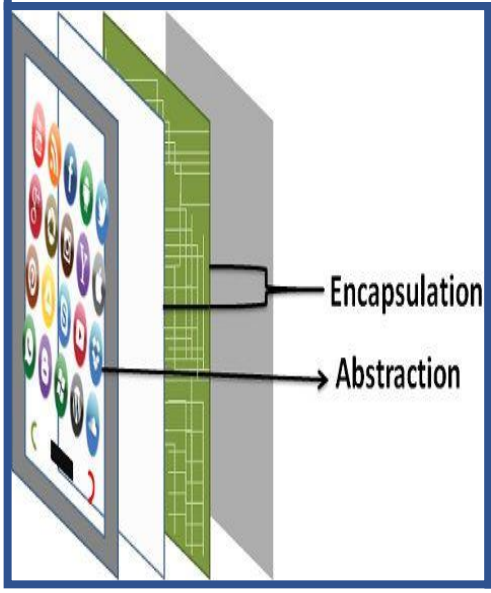
- **Binding** (or wrapping) **code** and **data** together into a single unit is known as encapsulation.
- encapsulation is described as the process of combining code (**methods or functions**) and data (**variables or attributes**) into a single unit.
- **Methods** or functions represent **executable** code, while data or variables represent **stored** information within a program.











Encapsulation:

For the mathematical equation, shown in the figure assume that complex functions are required - >But I the end we obtain result for it

Calculator shows the result of equation but hides the implementation (calculating the result) involved.

Abstraction:

The calculator shown in the figure has to be powered by a battery source. How the battery module works for the calculator is not necessary to know for the user who uses the calculator.

Using Battery module along with other modules we use calculator -> Thus using Abstraction encapsulation is performed

ADVANTAGES OF OOP

1) We can build the programs from standard working modules that communicate with one another, rather than having to start writing the code from scratch which leads to saving of development time and higher productivity,
2) OOP language allows to break the program into the bit-sized problems that can be solved easily (one object at a time).
3) The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.
4) OOP systems can be easily upgraded from small to large systems.
5) It is possible that multiple instances of objects co-exist without any interference,
6) It is very easy to partition the work in a project based on objects.
7) It is possible to map the objects in problem domain to those in the program.
8) The principle of data hiding helps the programmer to build secure programs which cannot be invaded by the code in other parts of the program.
9) By using inheritance, we can eliminate redundant code and extend the use of existing classes.
10) Message passing techniques is used for communication between objects which makes the interface descriptions with external systems much simpler.
11) The data-centered design approach enables us to capture more details of model in an implementable form.

Disadvantages of OOP

1) The length of the programmes developed using OOP language is much larger than the procedural approach. Since the programme becomes larger in size, it requires more time to be executed that leads to slower execution of the programme.
2) We can not apply OOP everywhere as it is not a universal language. It is applied only when it is required. It is not suitable for all types of problems.
3) Programmers need to have brilliant designing skill and programming skill along with proper planning because using OOP is little bit tricky.
4) OOPs take time to get used to it. The thought process involved in object-oriented programming may not be natural for some people.
5) Everything is treated as object in OOP so before applying it we need to have excellent thinking in terms of objects.

OBJECT AND CLASS

Object

- Since C++ is an object-oriented language, program is designed using objects and classes.
- In C++, Object is a real-world entity, for example, chair, car, pen, mobile, laptop etc.
- In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.
- Object is a runtime entity; it is created at runtime.
- Object is an instance of a class. All the members of the class can be accessed through object.

  Example: to create object of student class using s1 as the reference variable.

  Student s1; //creating an object of Student

  - In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.

Class

- In C++, class is a **group** of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.
- A class is a **blueprint** for creating objects. It defines the **properties** (data members) and **behaviors** (member functions) that objects of that class will have.
- Classes provide a way to encapsulate **data and methods** into a single unit, promoting code organization, reusability, and maintainability.

❖ Components of a class in c++:

1. **Data Members:** These are variables declared within the class, representing the properties or attributes of objects created from the class.
2. **Member Functions:** These are functions declared within the class, defining the behaviors or actions that objects created from the class can perform.
3. **Access Specifiers:** C++ provides three access specifiers: **public, private, & protected.** These specifiers determine accessibility of class members from outside class:
   - **public:** Members declared as `public` are accessible from outside the class.
   - **private:** Members declared as `private` are only accessible from within the class itself. They cannot be accessed directly from outside the class.
   - **protected:** Similar to private, but with additional access rights for derived classes.

Example of a class:

```
class MyClass {
public:
   int myInt;   // Data members
   double myDouble;
   void myFunction() {  // Member functions
      // Function implementation
   } };
```

In this example:

- **MyClass** is the name of the class.
- **myInt** and **myDouble** are data members of the class.
- **myFunction()** is a member function of the class.

Example-2:

```
class Student
{
   public:
   int id;  //field or data member
   float salary; //field or data member
   String name;//field or data member
}
```

Example-3:

This example demonstrating a class called **Rectangle,** which represents a rectangle **shape**. It has **data members** for storing the **width and height** of the rectangle, as well as **member functions** to set the dimensions and calculate the area of the rectangle. I'll also include a **main()** function to demonstrate how to create **objects** of the Rectangle **class** and use its **member functions.**

```
#include <iostream>
using namespace std;
class Rectangle {
private:
   double width;
   double height;
public:
   Rectangle(double w, double h) {  // Constructor
      width = w;
      height = h;
   }
   // Member function to set dimensions
   void setDimensions(double w, double h) {
      width = w;
      height = h;
   }
   // Member function to calculate area
   double calculateArea() {
      return width * height;
   } };
int main() {
   // Creating objects of Rectangle class
   Rectangle rectangle1(4.0, 5.0);
   Rectangle rectangle2(6.0, 3.0);
   // Using member functions
   cout << "Rectangle 1 area: " << rectangle1.calculateArea() << endl;
   cout << "Rectangle 2 area: " << rectangle2.calculateArea() << endl;
   // Changing dimensions and recalculating area
   rectangle1.setDimensions(7.0, 2.0);
   rectangle2.setDimensions(3.0, 8.0);
   cout << "After changing dimensions:" << endl;
   cout << "Rectangle 1 area: " << rectangle1.calculateArea() << endl;
   cout << "Rectangle 2 area: " << rectangle2.calculateArea() << endl;
   return 0; }
```

Output:

```
Rectangle 1 area: 20
Rectangle 2 area: 18
After changing dimensions:
Rectangle 1 area: 14
Rectangle 2 area: 24
```

Example-4:

```
#include <iostream>
using namespace std;
class Student {
  public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
};
int main() {
   Student s1; //creating an object of Student
   s1.id = 303;
   s1.name = "Sangam Kumar";
   cout<<s1.id<<endl;
   cout<<s1.name<<endl;
   return 0;
}
```

Output: 303
Sangam Kumar

Example-5: Initialize and Display data through method.

```
#include <iostream>
#include <string>
using namespace std;
class Student {
private:
   string name;
   int age;
public:
   // Method to initialize student data
   void initialize(string studentName, int studentAge) {
      name = studentName;
      age = studentAge;
   }
   void display() {    // Method to display student information
      cout << "Name: " << name << endl;
      cout << "Age: " << age << endl;
   } };
int main() {
   Student student1;  // Creating an object of Student class
   student1.initialize("Sangam Kumar", 15);   // Initializing data
   cout << "Student Information:" << endl;  // Displaying student information
   student1.display();
   return 0;
}
```

Output:

Student Information:
Name: Sangam Kumar
Age: 15

Example-6:

```
#include <iostream>
using namespace std;
class Student {
  public:
     int id;//data member (also instance variable)
     string name;//data member(also instance variable)
     void insert(int i, string n)
     {
        id = i;
        name = n;
     }
     void display()
     {
        cout<<id<<"  "<<name<<endl;
     }
};
int main(void) {
   Student s1; //creating an object of Student
   Student s2; //creating an object of Student
   s1.insert(301, "Sangam Kumar");
   s2.insert(302, "Sujeet Kumar");
   s1.display();
   s2.display();
   return 0;
}
```

Output:

```
301  Sangam Kumar
302  Sujeet Kumar
```

Example-7: Store and Display Employee Information

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id;//data member (also instance variable)
     string name;//data member(also instance variable)
     float salary;
     void insert(int i, string n, float s)
     {
        id = i;
        name = n;
        salary = s;
     }
     void display()
     {
        cout<<id<<"  "<<name<<"  "<<salary<<endl;
     }
};
int main(void) {
   Employee e1; //creating an object of Employee
   Employee e2; //creating an object of Employee
   e1.insert(301, " Sangam Kumar ",960000);
   e2.insert(302, " Sujeet Kumar ", 39000);
   e1.display();
   e2.display();
   return 0;
}
```

Output:

```
301  Sangam Kumar  960000
302  Sujeet Kumar    39000
```

Example-8:

```
#include <iostream>
#include <string>
using namespace std;
class Employee {
private:
   string name;
   int id;
   double salary;
public:
   // Method to initialize employee data
   void initialize(string employeeName, int employeeId,
double employeeSalary) {
      name = employeeName;
      id = employeeId;
      salary = employeeSalary;
   }
   void display() {    // Method to display employee
information
      cout << "Employee Name: " << name << endl;
      cout << "Employee ID: " << id << endl;
      cout << "Employee Salary: " << salary << endl;
   } };
int main() {
   // Creating an object of Employee class
   Employee employee1;
   // Initializing data
   employee1.initialize("Sangam Kumar", 339, 90000.0);
   // Displaying employee information
   cout << "Employee Information:" << endl;
   employee1.display();
   return 0;
}
```

Output:

```
Employee Information:
Employee Name: Sangam Kumar
Employee ID: 339
Employee Salary: 90000
```

CONSTRUCTOR

- constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.
- Its primary purpose is to initialize the object's data members or perform any other setup necessary for the object to be used.

❖ Key points about constructors:

1. **Name:** The constructor function has the same name as the class.
2. **No Return Type:** Constructors do not have a return type, not even void.
3. **Automatic Invocation:** Constructors are automatically invoked when an object is created.
4. **Initialization:** Constructors are commonly used to initialize the data members of the object to specific values.
5. **Overloading:** Like other functions, constructors can be overloaded, meaning a class can have multiple constructors with different parameter lists.

Example:

```
#include <iostream>
using namespace std;
class MyClass {
public:
   int myInt;
   MyClass() { // Constructor
      cout << "Constructor called" << endl;
      myInt = 0; // Initialize myInt to 0
   } };
int main() {
   MyClass obj; // Constructor called automatically
   cout << "Value of myInt: " << obj.myInt << endl;
   return 0;
}
```

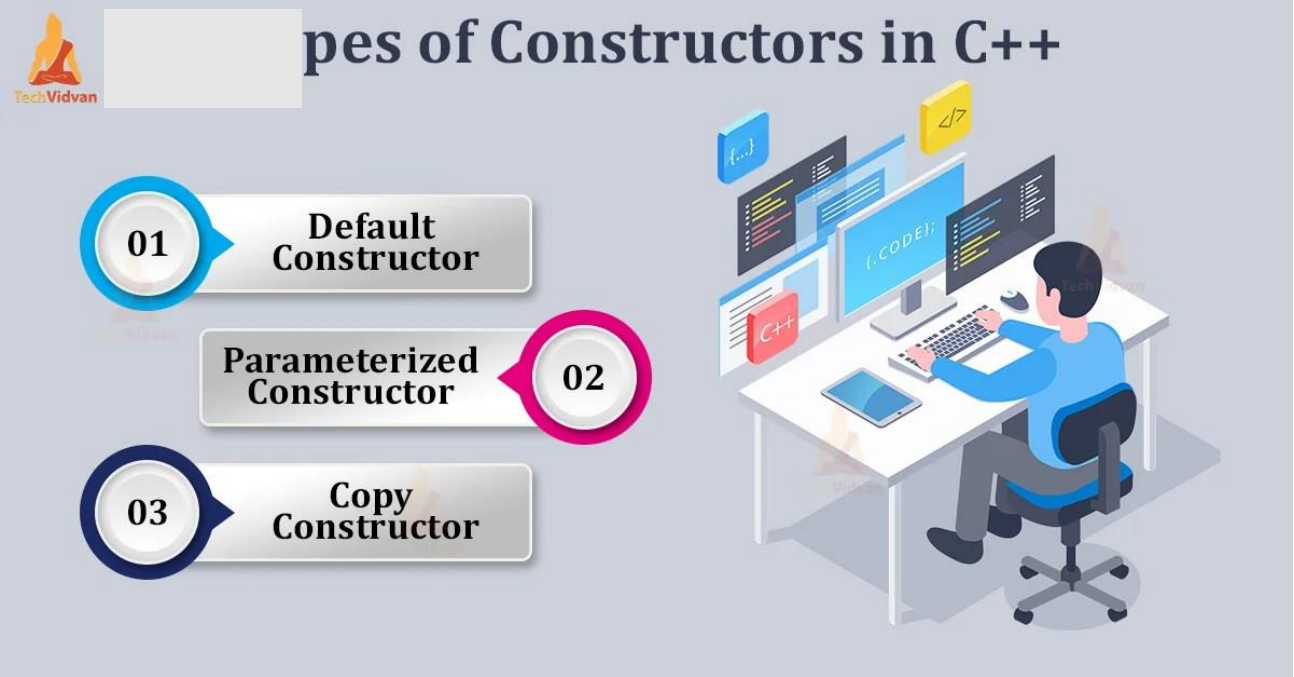
Output: Constructor called
Value of myInt: 0

❖ What distinguishes constructors from a typical member function?

- Constructor's name is the same as the class's
- Default There isn't an input argument for constructors. However, input arguments are available for copy and parameterized constructors.
- There is no return type for constructors.
- An object's constructor is invoked automatically upon creation.
- It must be shown in the classroom's open area.
- The C++ compiler creates a default constructor for the object if a constructor is not specified (expects any parameters and has an empty body).

❖ What are the characteristics of a constructor?

- The constructor has the same name as the class it belongs to.
- Although it is possible, constructors are typically declared in the class's public section. However, this is not a must.
- Because constructors don't return values, they lack a return type.
- When we create a class object, the constructor is immediately invoked.
- Overloaded constructors are possible.
- Declaring a constructor virtual is not permitted.
- One cannot inherit a constructor.
- Constructor addresses cannot be referenced to.
- When allocating memory, the constructor makes implicit calls to the new and delete operators.



- **There following types of constructors.**
    1. Default constructor
    2. Parameterized constructor
    3. Copy Constructor
    4. Constructor Overloading
    5. Destructor

1. Default Constructor: A constructor that doesn't take any arguments is called a default constructor. If a class doesn't explicitly define any constructors, the compiler automatically generates a default constructor. Its purpose is to initialize the object's data members to default values (e.g., zero or null).

Example:

```
MyClass() {
  // Default constructor
}
```

2. **Parameterized Constructor:** A constructor that accepts one or more parameters is called a parameterized constructor. It allows you to initialize the object's data members with specific values provided during object creation.

Example:

```
MyClass(int x, double y) {
  myInt = x;  // Parameterized constructor
  myDouble = y;
}
```

3. **Copy Constructor:** A constructor that creates an object by initializing it with an existing object of the same class is called a copy constructor. It creates a copy of the object passed to it. If a class doesn't define a copy constructor, the compiler generates a default copy constructor.

Example:

```
MyClass(const MyClass& obj) {
  myInt = obj.myInt;  // Copy constructor
  myDouble = obj.myDouble;
}
```

4. **Constructor Overloading:** Like other functions, constructors can be overloaded, meaning a class can have multiple constructors with different parameter lists. This allows you to create objects in different ways depending on the parameters provided.

Example:

```
MyClass() {      // Default constructor
}
MyClass(int x) {
  myInt = x;      // Parameterized constructor
}
```

5. **Destructor:** Though not technically a constructor, a destructor is another special member function of a class. It is called automatically when an object is destroyed (e.g., when it goes out of scope or when `delete` is called on a dynamically allocated object).

Example:

```
~MyClass() {
  // Destructor
}
```

- A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.
- A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).
- Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

Example-1: Default Constructor

```
#include <iostream>
using namespace std;
class Employee
 {
  public:
      Employee()
      {
         cout<<"Default Constructor
Invoked"<<endl;
      }
};
int main(void)
{
   Employee e1; //creating an object of
Employee
   Employee e2;
   return 0;
}
```

Output

```
Default Constructor Invoked
Default Constructor Invoked
```

Example-2 Parameterized Constructor

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id;//data member (also instance variable)
     string name;//data member
     Employee(int i, string n, float s)
      {
         id = i;
         name = n;
         salary = s;
      }
     void display()
      {
         cout<<id<<"  "<<name<<"  "<<salary<<endl;
      }  };
int main(void) {
   Employee e1 =Employee(301, "Sangam", 690000);
   Employee e2=Employee(302, "Sujeet", 39000);
   e1.display();
   e2.display();
   return 0;
}
```

Output: 301 Sangam 690000
302 Sujeet 39000

Example-3: Default Constructor

```
#include <iostream>
using namespace std;
class MyClass {
public:
   int myInt;
   // Default constructor
   MyClass() {
      cout << "Default constructor called" << endl;
      myInt = 0; // Initialize myInt to 0
   }
};
int main() {
   MyClass obj; // Default constructor called
automatically
 cout << "Value of myInt: " << obj.myInt << endl;
   return 0;
}
```

Output:

```
Default constructor called
Value of myInt: 0
```

Example-4 Parameterized Constructor

```
#include <iostream>
using namespace std;
class MyClass {
public:
   int myInt;
   // Parameterized constructor
   MyClass(int x) {
      cout << "Parameterized constructor called" << endl;
      myInt = x; // Initialize myInt with parameter value
   } };
int main() {
   MyClass obj(10); // Parameterized constructor called
automatically
   cout << "Value of myInt: " << obj.myInt << endl;
   return 0;
}
   return 0;
}
```

Output: Parameterized constructor called
Value of myInt: 10

Example-5: Copy constructor

```
#include <iostream>
using namespace std;
class MyClass {
public:
   int myInt;
   MyClass(const MyClass& obj) {
      cout << "Copy constructor called" << endl;
      myInt = obj.myInt; // Copy value from the original object
   }
};

int main() {
   MyClass obj1;
   obj1.myInt = 5;
   MyClass obj2 = obj1; // Copy constructor called automatically
cout << "Value of myInt in obj2: " << obj2.myInt << endl;
   return 0;
}
```

Output

```
Copy constructor called
Value of myInt in obj2: 5
```

Example-7: Destructor

```
#include <iostream>
using namespace std;
class MyClass {
public:
   int* arr;
   // Constructor
   MyClass() {
      cout << "Constructor called" << endl;
      arr = new int[5]; // Allocate memory
   }
   ~MyClass() {  // Destructor
      cout << "Destructor called" << endl;
      delete[] arr; // Free allocated memory
   }
};
int main() {
   MyClass obj;
   // Do something with obj...
   return 0;
}
```

Output:

```
Constructor called
Destructor called
```

Example-6 Constructor Overloading

```
#include <iostream>
using namespace std;
class MyClass {
public:
   int myInt;
   MyClass() {  // Default constructor
      cout << "Default constructor called" << endl;
      myInt = 0; // Initialize myInt to 0
   }
   MyClass(int x) {  // Parameterized constructor
      cout << "Parameterized constructor called" << endl;
      myInt = x; // Initialize myInt with parameter value
   } };
int main() {
   MyClass obj1; // Default constructor called automatically
   cout << "Value of myInt in obj1: " << obj1.myInt << endl;
   MyClass obj2(10); // Parameterized constructor called
automatically
   cout << "Value of myInt in obj2: " << obj2.myInt << endl;
   return 0; }
```

Output:

```
Default constructor called
Value of myInt in obj1: 0
Parameterized constructor called
Value of myInt in obj2: 10
```

Example-7: Destructor

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
      Employee()
      {
         cout<<"Constructor Invoked"<<endl;
      }
      ~Employee()
      {
         cout<<"Destructor Invoked"<<endl;
      } };
int main(void)
{
   Employee e1; //creating an object of Employee
   Employee e2; //creating an object of Employee
   return 0; }
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

Example-9:

```
#include <iostream>
#include <string>
using namespace std;
class Car {
private:
   string make;
   string model;
   int year;
public:
   // Default constructor
   Car() {
      make = "Unknown";
      model = "Unknown";
      year = 0;
   }
   // Parameterized constructor
   Car(string carMake, string carModel, int carYear) {
      make = carMake;
      model = carModel;
      year = carYear;
   }
   // Destructor
   ~Car() {
      cout << "Destructor called for " << make << " " << model << endl;
   }
   // Method to display car details
   void display() {
      cout << "Make: " << make << endl;
      cout << "Model: " << model << endl;
      cout << "Year: " << year << endl;
   }
};
int main() {
   // Creating objects using default constructor
   Car car1;
   cout << "Car 1 details:" << endl;
   car1.display();
   cout << endl;
   // Creating objects using parameterized constructor
   Car car2("Toyota", "Camry", 2020);
   cout << "Car 2 details:" << endl;
   car2.display();
   cout << endl;
   // Destructor will be called automatically when objects go out of scope
   return 0;
}
```

Output:

```
Car 1 details:
Make: Unknown
Model: Unknown
Year: 0
Car 2 details:
Make: Toyota
Model: Camry
Year: 2020
Destructor called for Toyota Camry
Destructor called for Unknown Unknown
```

this pointer

- In C++, the this pointer is a special pointer available inside non-static member functions of a class. It points to the object for which the member function is called. Essentially, this refers to the current object instance on which the member function is operating.

❖ key points about the this pointer:

- **Purpose:** The this pointer is used to access the members (data members and member functions) of the current object within a member function. It helps distinguish between local variables and object members when they have the same name.
- **Usage:** Inside a member function, you can use this to access the data members and call other member functions of the current object.
- **Implicit Usage:** When you call a member function on an object,this pointer is automatically passed to function by the compiler. You don't need to explicitly pass it.

Example-1:

```
class MyClass {
private:
   int myVar;
public:
   void setMyVar(int value) {
      // Using 'this' pointer to access the member variable
      this->myVar = value;
   }
   int getMyVar() {
      // Using 'this' pointer to return the member variable
      return this->myVar;
   } };
int main() {
   MyClass obj;
   obj.setMyVar(10);
   cout << "MyVar: " << obj.getMyVar() << endl;
   return 0;
}
```

Output: MyVar: 10

- In this example, this->myVar inside the member functions setMyVar() and getMyVar() is used to access the myVar member variable of the current object.

- **Use Cases:**
  - ✓ Avoiding name conflicts: When the names of local variables shadow the names of data members, using this helps to differentiate them.
  - ✓ Passing the current object to other member functions or constructors.
  - ✓ Returning a reference to the current object from a member function, allowing method chaining.
- **Const Member Functions:** Inside a const member function, this pointer is of type pointer to const, meaning you cannot modify the object's data members using this within a const member function.

Example-2:

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id; //data member (also instance variable)
     string name; //data member(also instance variable)
     float salary;
     Employee(int id, string name, float salary)
     {
         this->id = id;
        this->name = name;
        this->salary = salary;
     }
     void display()
     {
        cout<<id<<"  "<<name<<"  "<<salary<<endl;
     }
};
int main(void) {
   Employee e1 =Employee(301, "Sangam", 90000);
   Employee e2=Employee(302, "Sujeet", 93000); //creating an object of
Employee
   e1.display();
   e2.display();
   return 0;
}
```

Output:

```
301  Sangam  90000
302  Sujeet   93000
```

Static Keyword

- In C++, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C++, static can be field, method, constructor, class, properties, operator and event

❖ Advantage of C++ static keyword:

- Memory efficient: Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

❖ Static Field:

- A field which is declared as static is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.
- It is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.

❖ Example of static field:

```
#include <iostream>
using namespace std;
class Account {
  public:
     int accno; //data member (also instance variable)
     string name; //data member(also instance variable)
     static float rateOfInterest;
     Account(int accno, string name)
     {
         this->accno = accno;
        this->name = name;
     }
     void display()
     {
        cout<<accno<< "<<name<< " "<<rateOfInterest<<endl;
     }
};
float Account::rateOfInterest=6.5;
int main(void) {
   Account a1 =Account(201, "Sanjay"); //creating an object of Employee
   Account a2=Account(202, "Nakul"); //creating an object of Employee
   a1.display();
   a2.display();
   return 0;
}
```

Output:

```
201 Sanjay 6.5
202 Nakul 6.5
```

Example: static field Counting Objects

```
#include <iostream>
using namespace std;
class Account {
  public:
     int accno; //data member (also instance variable)
     string name;
     static int count;
     Account(int accno, string name)
     {
        this->accno = accno;
       this->name = name;
       count++;
     }
     void display()
     {
       cout<<accno<<" "<<name<<endl;
     }
};
int Account::count=0;
int main(void) {
   Account a1 =Account(201, "Sanjay"); //creating an object of Account
   Account a2=Account(202, "Nakul");
    Account a3=Account(203, "Ranjana");
   a1.display();
   a2.display();
   a3.display();
   cout<<"Total Objects are: "<<Account::count;
   return 0;
}
```

Output:

```
201 Sanjay
202 Nakul
203 Ranjana
Total Objects are: 3
```

Structs keyword

- In C++, classes and structs are blueprints that are used to create the instance of a class. Structs are used for lightweight objects such as Rectangle, color, Point, etc.
- Unlike class, structs in C++ are value type than reference type. It is useful if you have data that is not intended to be modified after creation of struct.
- C++ Structure is a collection of different data types. It is similar to the class that holds different types of data.

Syntax:

```
struct structure_name
{
   // member declarations.
}
```

- In the above declaration, a structure is declared by preceding the struct keyword followed by the identifier(structure name). Inside the curly braces, we can declare the member variables of different types. Consider the following situation:

```
struct Student
{
   char name[20];
    int id;
    int age;
}
```

- In the above case, Student is a structure contains three variables name, id, and age. When the structure is declared, no memory is allocated. When the variable of a structure is created, then the memory is allocated. Let's understand this scenario.

❖ How to create the instance of Structure?

- Structure variable can be defined as:

```
Student s;
```

- Here, s is a structure variable of type Student. When the structure variable is created, the memory will be allocated. Student structure contains one char variable and two integer variable. Therefore, the memory for one char variable is 1 byte and two ints will be 2*4 = 8. The total memory occupied by the s variable is 9 byte.

❖ How to access the variable of Structure:

- ✓ The variable of the structure can be accessed by simply using the instance of the structure followed by the dot (.) operator and then the field of the structure.

Example:

```
s.id = 4;
```

- In the above statement, we are accessing the id field of the structure Student by using the dot(.) operator and assigns the value 4 to the id field.

Example of Struct keyword

```
#include <iostream>
using namespace std;
 struct Rectangle
{
  int width, height;
```

```
};
int main(void) {
   struct Rectangle rec;
   rec.width=8;
   rec.height=5;
  cout<<"Area of Rectangle is: "<<(rec.width * rec.height)<<endl;
 return 0;
}
```

Output:

Area of Rectangle is: 40

Example: Using Constructor and Method

```
#include <iostream>
using namespace std;
 struct Rectangle   {
  int width, height;
 Rectangle(int w, int h)
  {
     width = w;
     height = h;
  }
 void areaOfRectangle() {
  cout<<"Area of Rectangle is: "<<(width*height); }
};
int main(void) {
   struct Rectangle rec=Rectangle(4,6);
   rec.areaOfRectangle();
  return 0;
}
```

Output:

Area of Rectangle is: 24

❖ Structure v/s Class:

❖ Structure:

- If access specifier is not declared explicitly, then by default access specifier will be **public.**

Syntax:

```
struct structure_name
{
// body of the structure.
}
```

- The instance of the structure is known as "Structure variable".

❖ Class:

- If access specifier is not declared explicitly, then by default access specifier will be **private.**

Syntax:

```
class class_name
{
// body of the class.
}
```

- The instance of the class is known as "Object of the class".

C++ ENUMERATION

- Enum in C++ is a data type that contains fixed set of constants.
- It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The C++ enum constants are static and final implicitly.
- C++ Enums can be thought of as classes that have fixed set of constants.
- Points to remember for C++ Enum:
  - enum improves type safety
  - enum can be easily used in switch
  - enum can be traversed
  - enum can have fields, constructors and methods
  - enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Example:

```
#include <iostream>
using namespace std;
enum week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
int main()
{
   week day;
   day = Friday;
   cout << "Day: " << day+1<<endl;
   return 0;
}
```

Output:

```
Day: 5
```

FRIEND FUNCTION

- If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
- By using the keyword friend compiler knows the given function is a friend function.
- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

❖ Declaration of friend function in C++:

```
class class_name
{
   friend data_type function_name(argument/s);          // syntax of friend function.
};
```

❖ Characteristics of a Friend function:

- ✓ The function is not in the scope of the class to which it has been declared as a friend.
- ✓ It cannot be called using the object as it is not in the scope of that class.
- ✓ It can be invoked like a normal function without using the object.
- ✓ It cannot access the member names directly and has to use an object name and dot membership operator with the member's name.
- ✓ It can be declared either in the private or public part.

Example-1

```
#include <iostream>
using namespace std;
class Box
{
   private:
      int length;
   public:
      Box(): length(0) { }
      friend int printLength(Box); //friend function
};
int printLength(Box b)
{
  b.length += 10;
   return b.length;
}
int main()
{
   Box b;
   cout<<"Length of box: "<< printLength(b)<<endl;
   return 0;
}
```

Output: Length of box: 10

Example-2:

```
#include <iostream>
using namespace std;
class B;        // forward declarartion.
class A
{   int x;
   public:
   void setdata(int i)
   {
      x=i;   }
   friend void min(A,B); };
class B
{
   int y;
   public:
   void setdata(int i)
   {
      y=i;   }
   friend void min(A,B); };
void min(A a,B b)
{  if(a.x<=b.y)
   std::cout << a.x << std::endl;
   else
   std::cout << b.y << std::endl; }
  int main()
{  A a;
  B b;
  a.setdata(10);
  b.setdata(20);
  min(a,b);
  return 0;  }
```

Output: 10

❖ Simple example when the function is friendly to two classes.

- In the above example, min() function is friendly to two classes, i.e., the min() function can access the private members of both the classes A and B.

❖ C++ Friend class:

- A friend class can access both private and protected members of the class in which it has been declared as friend.

Example.

```
#include <iostream>
using namespace std;
class A
{
   int x =5;
   friend class B;          // friend class.
};
class B
{
 public:
   void display(A &a)
   {
      cout<<"value of x is : "<<a.x;
   }
};
int main()
{
   A a;
   B b;
   b.display(a);
   return 0;
}
```

Output:

```
value of x is : 5
```

INHERITANCE

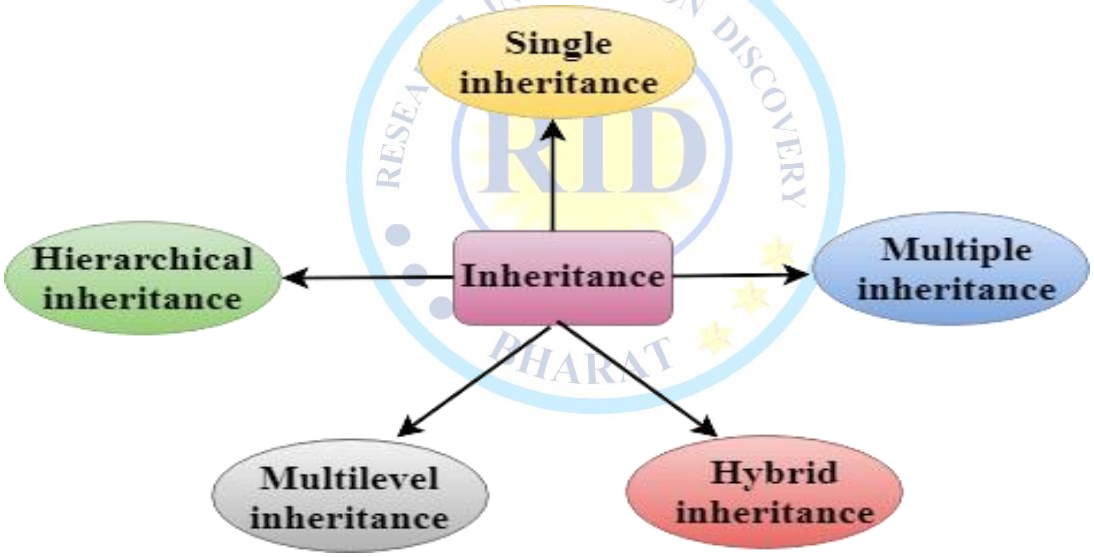
- inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.
- In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

❖ Advantage of Inheritance:

- Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

❖ Types Of Inheritance:

- C++ supports five types of inheritance:
  1. Single inheritance
  2. Multiple inheritance
  3. Hierarchical inheritance
  4. Multilevel inheritance
  5. Hybrid inheritance



❖ A Derived class is defined as the class derived from the base class.

- **Syntax of Derived class:**

```
class derived_class_name :: visibility-mode base_class_name
{
   // body of the derived class.
}
```

Where:

- **derived_class_name:** It is the name of the derived class.
- **visibility mode:** The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.
- **base_class_name:** It is the name of the base class.

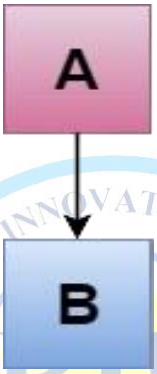
- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

1. Single Inheritance:

- Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



✓ Where 'A' is the base class, and 'B' is the derived class.

❖ Single Level Inheritance Example: Inheriting Fields

- When one class inherits another class, it is known as single level inheritance.

Example:

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
 };
   class Programmer: public Account {
   public:
   float bonus = 5000;
   };
int main(void) {
    Programmer p1;
    cout<<"Salary: "<<p1.salary<<endl;
    cout<<"Bonus: "<<p1.bonus<<endl;
   return 0;
}
```

Output: Salary: 60000
Bonus: 5000

- In the above example, Employee is the base class and Programmer is the derived class.

❖ C++ Single Level Inheritance Example: Inheriting Methods.

Example.

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
  };
   class Dog: public Animal
   {
      public:
    void bark(){
    cout<<"Barking...";
     }
   };
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```

Output: Eating...
Barking...

Example.

```
#include <iostream>
using namespace std;
class A
{
   int a = 4;
   int b = 5;
   public:
   int mul()
   {
      int c = a*b;
      return c;
   }  };
class B : private A
{
   public:
   void display()
   {
      int result = mul();
   std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
   } };
int main()
{
  B b;
  b.display();
   return 0;
}
```

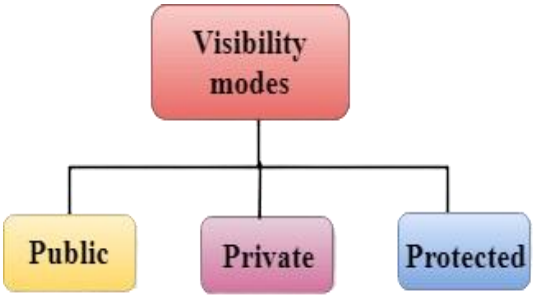
Output: Multiplication of a and b is : 20

- In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by object of class B. It can only be accessed by member function of class B.

❖ How to make a Private Member Inheritable:

- The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.
- C++ introduces a third visibility modifier, i.e., protected. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

❖ Visibility modes can be classified into three categories:



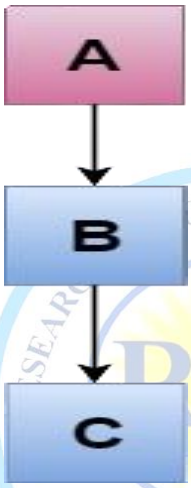
- **Public:** When the member is declared as public, it is accessible to all the functions of the program.
- **Private:** When the member is declared as private, it is accessible within the class only.
- **Protected:** When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

❖ Visibility of Inherited Members:

Base class visibility	Derived class visibility		
	Public	**Private**	**Protected**
• Private	Not Inherited	Not Inherited	Not Inherited
• Protected	Protected	Private	Protected
• Public	Public	Private	Protected

❖ Multilevel Inheritance:

➢ Multilevel inheritance is a process of deriving a class from another derived class.



Example:

- When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Example:

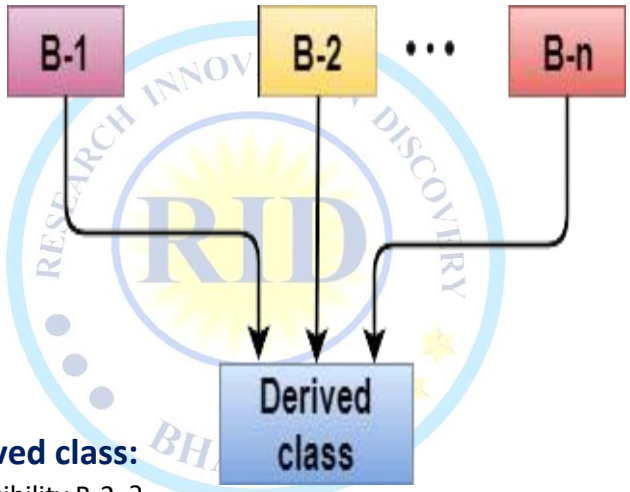
```
#include <iostream>
using namespace std;
 class Animal {
  public:
 void eat() {
   cout<<"Eating..."<<endl;
}    };
  class Dog: public Animal
  {
     public:
   void bark(){
  cout<<"Barking..."<<endl;
   }   };
  class BabyDog: public Dog
  {
```

```
    public:
   void weep() {
   cout<<"Weeping...";
    }   };
int main(void) {
   BabyDog d1;
   d1.eat();
   d1.bark();
    d1.weep();
    return 0;
}
```

Output: Eating...
Barking...
Weeping...

❖ Multiple Inheritance:

- Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



❖ Syntax of the Derived class:

```
class D : visibility B-1, visibility B-2, ?
{
   // Body of the class;
}
```

Example.

```
#include <iostream>
using namespace std;
class A
{
   protected:
    int a;
   public:
   void get_a(int n)
   {
      a = n;
   } };
class B
{
```

```
    protected:
    int b;
    public:
    void get_b(int n)
    {
       b = n;
    } };
class C : public A,public B
{
  public:
    void display()
    {
       std::cout << "The value of a is : " <<a<< std::endl;
       std::cout << "The value of b is : " <<b<< std::endl;
       cout<<"Addition of a and b is : "<<a+b;
    } };
int main()
{
  C c;
  c.get_a(10);
  c.get_b(20);
  c.display();
   return 0;
}
```

Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

- In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

❖ Ambiquity Resolution in Inheritance:

- Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Example:

```
#include <iostream>
using namespace std;
class A
{
   public:
   void display()
   {
      std::cout << "Class A" << std::endl;
   }
};
class B
{
   public:
   void display()
```

```
    {
        std::cout << "Class B" << std::endl;
    }
};
class C : public A, public B
{
    void view()
    {
        display();
    } };
int main()
{
    C c;
    c.display();
    return 0;
}
```

Output:

```
error: reference to 'display' is ambiguous
        display();
```

- The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B
{
    void view()
    {
        A :: display();       // Calling the display() function of class A.
        B :: display();       // Calling the display() function of class B.
    } };
```

- An ambiguity can also occur in single inheritance.
- Consider the following situation:

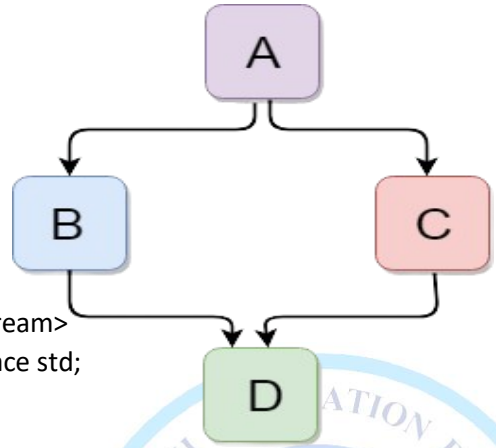
```
class A
{
  public:
void display()
{
  cout<<?Class A?;
}  } ;
class B
{
  public:
 void display()
{
 cout<<?Class B?;
} } ;
```

- In the above case, the function of the derived class overrides the method of the base class. Therefore, call to the display() function will simply call the function defined in the derived class. If we want to invoke the base class function, we can use the class resolution operator.

```
int main()
{
   B b;
  b.display();          // Calling the display() function of B class.
  b.B :: display();     // Calling the display() function defined in B class.
}
```

❖ C++ Hybrid Inheritance:

- Hybrid inheritance is a combination of more than one type of inheritance.



Example:

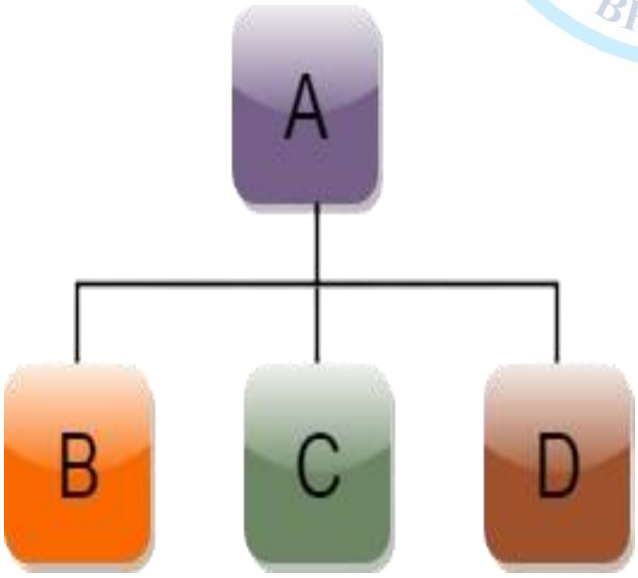
```
#include <iostream>
using namespace std;
class A
{
   protected:
   int a;
   public:
   void get_a()
   {
     std::cout << "Enter the value of 'a' : " << std::endl;
     cin>>a;
   } };
class B : public A
{
   protected:
   int b;
   public:
   void get_b()
   {
      std::cout << "Enter the value of 'b' : " << std::endl;
     cin>>b;
   } };
class C
{
   protected:
   int c;
   public:
   void get_c()
   {
      std::cout << "Enter the value of c is : " << std::endl;
      cin>>c;
```

```
    } };
class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
        get_a();
        get_b();
        get_c();
        std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
    } };
int main()
{
    D d;
    d.mul();
    return 0;
}
```

Output: Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000

❖ Hierarchical Inheritance:

- Hierarchical inheritance is defined as process of deriving more than one class from a base class.



Syntax of Hierarchical inheritance:

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

Example:

```
#include <iostream>
using namespace std;
class Shape              // Declaration of base class.
{  public:
   int a;
   int b;
   void get_data(int n,int m)
   {  a= n;
      b = m;
   } };
class Rectangle : public Shape  // inheriting Shape class
{  public:
   int rect_area()
   {  int result = a*b;
      return result;
   } };
class Triangle : public Shape    // inheriting Shape class
{  public:
   int triangle_area()
   {   float result = 0.5*a*b;
      return result;
   } };
int main()
{  Rectangle r;
   Triangle t;
   int length,breadth,base,height;
   std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
   cin>>length>>breadth;
   r.get_data(length,breadth);
   int m = r.rect_area();
   std::cout << "Area of the rectangle is : " <<m<< std::endl;
   std::cout << "Enter the base and height of the triangle: " << std::endl;
   cin>>base>>height;
   t.get_data(base,height);
   float n = t.triangle_area();
   std::cout <<"Area of the triangle is : "  << n<<std::endl;
   return 0;  }
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

AGGREGATION (HAS-A RELATIONSHIP)

- In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

❖ Example:

```
#include <iostream>
using namespace std;
class Address {
   public:
  string addressLine, city, state;
    Address(string addressLine, string city, string state)
   {
      this->addressLine = addressLine;
      this->city = city;
      this->state = state;
   }  };
class Employee
   {
      private:
      Address* address;  //Employee HAS-A Address
      public:
      int id;
      string name;
      Employee(int id, string name, Address* address)
     {
        this->id = id;
        this->name = name;
        this->address = address;
     }
    void display()
     {
        cout<<id <<" "<<name<< " "<<
         address->addressLine<< " "<< address->city<< " "<<address->state<<endl;
      }  };
int main(void) {
   Address a1= Address("C-146, Sec-15","Noida","UP");
   Employee e1 = Employee(101,"Nakul",&a1);
        e1.display();
  return 0;
}
```

Output:

101 Nakul C-146, Sec-15 Noida UP

POLYMORPHISM

- The term **"Polymorphism"** is the combination of **"poly" + "morphs"** which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

❖ Real Life Example of Polymorphism:

- Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

❖ There are two types of polymorphism:

- Compile time polymorphism: The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```
class A                          // base class declaration.
{
    int a;
    public:
    void display()
    {
        cout<< "Class A ";
    }
};
class B : public A               // derived class declaration.
{
   int b;
   public:
   void display()
   {
       cout<<"Class B";
   }
};
```

- In the above case, the prototype of display() function is the same in both the base and derived class. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as run time polymorphism.
- **Run time polymorphism:** Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

❖ Differences b/w compile time and run time polymorphism:

❖ Compile-time Polymorphism:

- Also known as static polymorphism or early binding.
- Occurs during the compilation phase of the program.
- Examples include function overloading and operator overloading.
- Resolution of which function to call is done by the compiler based on the function signature at compile time.
- Compile-time polymorphism provides better performance as there is no overhead during runtime for determining which function to call.

❖ Run-time Polymorphism:

- Also known as dynamic polymorphism or late binding.
- Occurs during the execution phase of the program.
- Achieved through virtual functions and inheritance.
- Resolution of which function to call is deferred until runtime based on the type of the object being referred to.
- Run-time polymorphism allows for more flexible and extensible code as decisions are made dynamically based on the actual type of the object.
- Requires the use of pointers or references to base class objects and virtual functions to achieve polymorphic behavior.

Example: Run time polymorphism.

```
#include <iostream>
using namespace std;
class Animal {
   public:
void eat(){
cout<<"Eating...";
   }
};
class Dog: public Animal
{
 public:
 void eat()
   {        cout<<"Eating bread...";
   }
};
int main(void) {
  Dog d = Dog();
  d.eat();
  return 0;
}
```

Output:

Eating bread...

Example: Run time Polymorphism By using two derived class

```
#include <iostream>
using namespace std;
class Shape {                          //  base class
   public:
virtual void draw(){                   // virtual function
cout<<"drawing..."<<endl;
   }
};
class Rectangle: public Shape              //  inheriting Shape class.
{
 public:
 void draw()
  {
     cout<<"drawing rectangle..."<<endl;
   }
};
class Circle: public Shape                 //  inheriting Shape class.

{
 public:
 void draw()
  {
    cout<<"drawing circle..."<<endl;
  }
};
int main(void) {
   Shape *s;                          //  base class pointer.
   Shape sh;                          // base class object.
      Rectangle rec;
       Circle cir;
     s=&sh;
    s->draw();
       s=&rec;
    s->draw();
   s=?
   s->draw();
}
```

Output:

```
drawing...
drawing rectangle...
drawing circle...
```

❖ Runtime Polymorphism with Data Members:

- Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

Example:

```
#include <iostream>
using namespace std;
class Animal {                          //  base class declaration.
   public:
   string color = "Black";
};
class Dog: public Animal                // inheriting Animal class.
{
 public:
   string color = "Grey";
};
int main(void) {
    Animal d= Dog();
   cout<<d.color;
}
```

Output:

Black

OVERLOADING

- If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload.
  - methods,
  - constructors, and
  - indexed properties
  - ✓ It is because these members have parameters only.

❖ Types of overloading:

1. Function overloading
2. Operator overloading

1. Function Overloading

- Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.
- **Advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

Example-1: Function Overloading.

```
#include <iostream>
using namespace std;
class Cal {
   public:
static int add(int a,int b){
      return a + b;
   }
static int add(int a, int b, int c)
   {
      return a + b + c;
   }
};
```

```
int main(void) {
   Cal C;                              //   class object declaration.
   cout<<C.add(10, 20)<<endl;
   cout<<C.add(12, 20, 23);
  return 0;
}
```

Output:

```
30
55
```

Example-2: **when the type of the arguments vary.**

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);
int mul(int a,int b)
{
   return a*b;
}
float mul(double x, int y)
{
   return x*y;
}
int main()
{
   int r1 = mul(6,7);
   float r2 = mul(0.2,3);
   std::cout << "r1 is : " <<r1<< std::endl;
   std::cout <<"r2 is : "  <<r2<< std::endl;
   return 0;
}
```

Output:

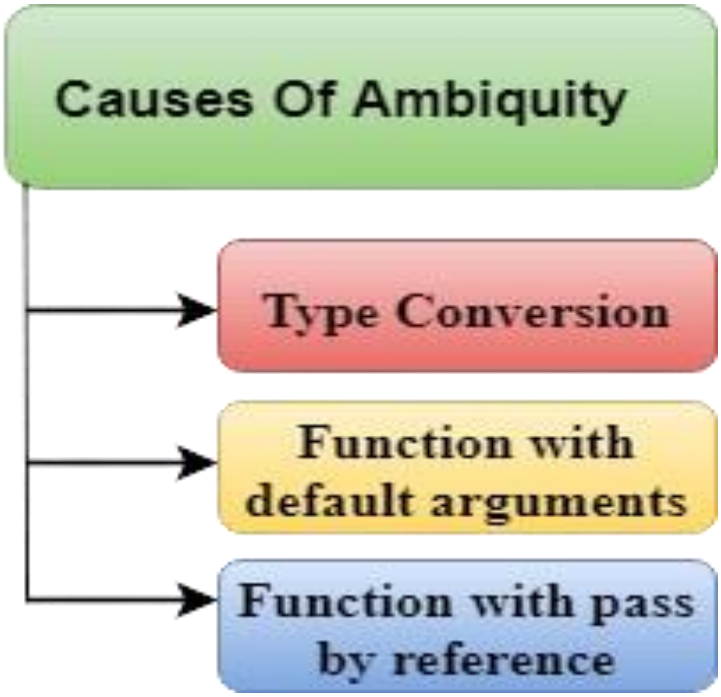
```
r1 is : 42
r2 is : 0.6
```

❖ Function Overloading and Ambiguity:

he compiler is unable to decide which function is to be invoked among the overloaded function, this situation
n as function overloading.
he compiler shows the ambiguity error, the compiler does not run program.

❖ Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.



❖ Type Conversion:

Example:

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
   std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(float j)
{
   std::cout << "Value of j is : " <<j<< std::endl;
}
int main()
{
   fun(12);
   fun(1.2);
   return 0;
}
```

- The above example shows an error "call of overloaded 'fun(double)' is ambiguous". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

❖ Function with Default Arguments:

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
   std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
   std::cout << "Value of a is : " <<a<< std::endl;
   std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
   fun(12);
   return 0;
}
```

❖ The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5).

❖ Function with pass by reference:

Example:

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
int a=10;
fun(a); // error, which f()?
return 0;
}
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
}
```

- In this example shows an error "call of overloaded 'fun(int&)' is ambiguous". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

❖ Operators Overloading:

- Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.
- **Advantage** of Operators overloading is to perform different operations on the same operand.

❖ Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

❖ Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

- Where the return type is the type of value returned by the function.
- class_name is the name of the class.
- operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

❖ Rules for Operator Overloading:

- ✓ Existing operators can only be overloaded, but new operators cannot be overloaded.
- ✓ The overloaded operator contains atleast one operand of the user-defined data type.
- ✓ We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- ✓ When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- ✓ When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

Example of operators overloading:

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
  public:
```

```
    Test(): num(8){}
    void operator ++()      {
      num = num+2;
    }
    void Print() {
       cout<<"The Count is: "<<num;
    }  };
int main()
{
   Test tt;
   ++tt;  // calling of a function "void operator ++()"
   tt.Print();
   return 0;
}
```

Output: The Count is: 10

Example-2 of overloading the binary operators.

```
#include <iostream>
using namespace std;
class A
{
   int x;
     public:
     A(){}
   A(int i)
   {
     x=i;
   }
   void operator+(A);
   void display();
};
void A :: operator+(A a)
{
   int m = x+a.x;
   cout<<"The result of the addition of two objects is : "<<m;
}
int main()
{
   A a1(5);
   A a2(4);
   a1+a2;
   return 0;
}
```

Output: The result of the addition of two objects is : 9

FUNCTION OVERRIDING

- If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

Example: In this example, we are overriding the eat() function.

```
#include <iostream>
using namespace std;
class Animal {
   public:
void eat(){
cout<<"Eating...";
   }
};
class Dog: public Animal
{
 public:
 void eat()
   {
      cout<<"Eating bread...";
   }
};
int main(void) {
  Dog d = Dog();
  d.eat();
  return 0;
}
```

Output:

```
Eating bread...
```

VIRTUAL FUNCTION

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

❖ Late binding or Dynamic linkage:

- In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

❖ Rules of Virtual Function:

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

Program:

```
#include <iostream>
using namespace std;
class A
{
  int x=5;
   public:
   void display()
   {
      std::cout << "Value of x is : " << x<<std::endl;
   }
};
class B: public A
{
   int y = 10;
   public:
   void display()
   {
      std::cout << "Value of y is : " <<y<< std::endl;
```

```
    }
};
int main()
{
   A *a;
   B b;
   a = &b;
  a->display();
   return 0;
}
```

- In this above example, * a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

Output: Value of x is : 5

Example: virtual function:.

```
#include <iostream>
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }  };
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }  };
int main()
{
 A* a;   //pointer of base class
 B b;    //object of derived class
 a = &b;
 a->display();  //Late Binding occurs
}
```

Output: Derived Class is invoked

❖ Pure Virtual Function:

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "do-nothing" function.
- The "do-nothing" function is known as a pure virtual function. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

❖ Pure virtual function can be defined as:

virtual void display() = 0;

Example:

```
#include <iostream>
using namespace std;
class Base
{
   public:
   virtual void show() = 0;
};
class Derived : public Base
{
   public:
   void show()
   {
      std::cout << "Derived class is derived from the base class." << std::endl;
   }
};
int main()
{
   Base *bptr;
   //Base b;
   Derived d;
   bptr = &d;
   bptr->show();
   return 0;
}
```

Output:

Derived class is derived from the base class.

- In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

Interfaces in C++ (Abstract Classes)

- Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only.
- Abstraction can be achieved by two ways:
  1. Abstract class
  2. Interface
- Abstract class and interface both can have abstract methods which are necessary for abstraction.

❖Abstract class:

- In C++ class is made abstract by declaring at least one of its functions as <>strong>pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration. Its implementation must be provided by derived classes.

Example of abstract class which has one abstract method draw(). Its implementation is provided by derived classes: Rectangle & Circle. Both classes have different implementation.

```
#include <iostream>
using namespace std;
 class Shape
{
   public:
   virtual void draw()=0;
};
 class Rectangle : Shape
{   public:
    void draw()
   {
      cout < <"drawing rectangle..." < <endl;
   }  };
class Circle : Shape
{   public:
    void draw()
   {
      cout <<"drawing circle..." < <endl;
   }  };
int main( ) {
   Rectangle rec;
   Circle cir;
   rec.draw();
   cir.draw();
  return 0;
}
```

Output: drawing rectangle...
drawing circle...

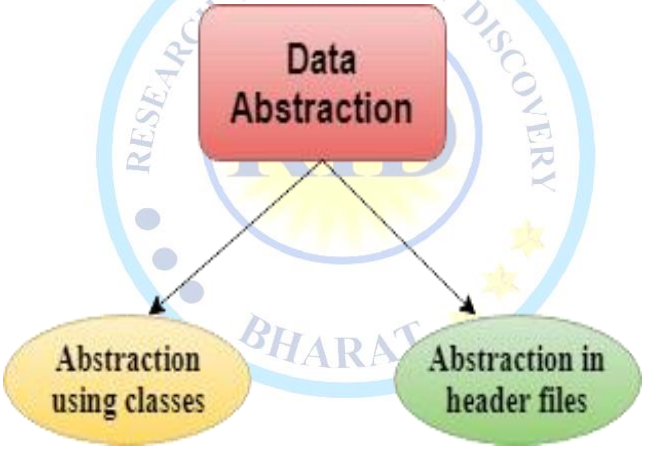
❖ Data Abstraction:

- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.
- Data Abstraction is a programming technique that depends on the seperation of the interface and implementation details of the program.
- Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC seperates the implementation details from the external interface.
- C++ provides a great level of abstraction. For example, pow() function is used to calculate the power of a number without knowing the algorithm the function follows.

➢ In C++ program if we implement class with private and public members then it is an example of data abstraction.

❖ Data Abstraction can be achieved in two ways:

1. Abstraction using classes
2. Abstraction in header files.



1. **Abstraction using classes:** An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.
2. **Abstraction in header files:** An another type of abstraction is header file. For example, pow() function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hides all the implementation details from the user.

❖ Access Specifiers Implement Abstraction:

- **Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.
- **Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

Example of abstraction in header files.

```
// program to calculate the power of a number.
#include <iostream>
#include<math.h>
using namespace std;
int main()
{
 int n = 4;
  int power = 3;
  int result = pow(n,power);        // pow(n,power) is the  power function
  std::cout << "Cube of n is : " <<result<< std::endl;
  return 0;
}
```

Output: Cube of n is : 64

- In the above example, pow() function is used to calculate 4 raised to the power 3. The pow() function is present in the math.h header file in which all the implementation details of the pow() function is hidden.

Example of data abstraction using classes.

```
#include <iostream>
using namespace std;
 class Sum
{
private: int x, y, z; // private variables
public:
void add()
{
cout<<"Enter two numbers: ";
cin>>x>>y;
z= x+y;
cout<<"Sum of two number is: "<<z<<endl;
}
};
int main()
{
Sum sm;
sm.add();
return 0;
}
```

Output:

```
Enter two numbers:
3
6
Sum of two number is: 9
```

❖Advantages Of Abstraction:

- Implementation details of the class are protected from the inadvertent user level errors.
- A programmer does not need to write the low level code.
- Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
- The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.
- Internal implementation can be changed without affecting the user level code.

NAMESPACES

- Namespaces in C++ are used to organize too many classes so that it can be easy to handle the application.
- For accessing the class of a namespace, we need to use namespacename::classname. We can use using keyword so that we don't have to use complete name all the time.
- In C++, global namespace is the root namespace. The global::std will always refer to the namespace "std" of C++ Framework.

Example: namespace which include variable and functions.

```
#include <iostream>
using namespace std;
namespace First {
   void sayHello() {
      cout<<"Hello First Namespace"<<endl;
   }
}
namespace Second  {
     void sayHello() {
        cout<<"Hello Second Namespace"<<endl;
     }  }
int main()
{
 First::sayHello();
 Second::sayHello();
return 0;
}
```

Output:

```
Hello First Namespace
Hello Second Namespace
```

Example: by using keyword namespace where we are using "using" keyword so that we don't have to use complete name for accessing a namespace program.

```
#include <iostream>
using namespace std;
namespace First{
  void sayHello(){
    cout << "Hello First Namespace" << endl;
  } }
namespace Second{
  void sayHello(){
    cout << "Hello Second Namespace" << endl;
  } }
using namespace First;
int main () {
  sayHello();
  return 0;
}
```

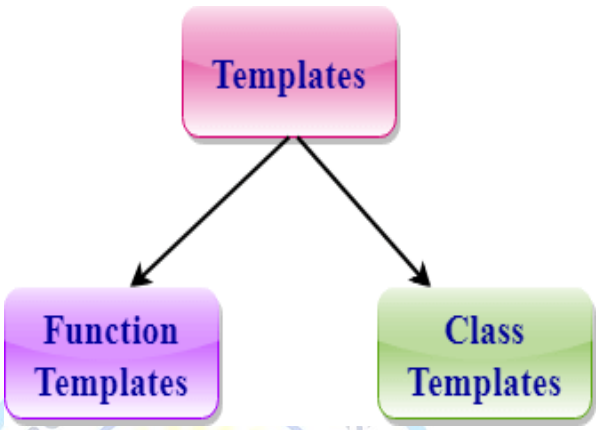
Output: Hello First Namespace

TEMPLATES

- A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

❖ Templates can be represented in two ways:

1. Function templates
2. Class templates



1. Function Templates:

- We can define a template for a function. For example, if we have an add() function, we can create versions add function for adding the int, float or double type values.

2. Class Template:

- We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

❖Function Template:

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

❖ Syntax of Function Template

```
template < class Ttype> ret_type func_name(parameter_list)
{
   // body of function.
}
```

- Where Ttype: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

Example of a function template:

```
#include <iostream>
using namespace std;
template<class T> T add(T &a,T &b)
{
   T result = a+b;
   return result;

}
int main()
{
 int i =2;
 int j =3;
 float m = 2.3;
 float n = 1.2;
 cout<<"Addition of i and j is :"<<add(i,j);
 cout<<'\n';
 cout<<"Addition of m and n is :"<<add(m,n);
 return 0;
}
```

Output:

```
Addition of i and j is :5
Addition of m and n is :3.5
```

- In the above example, we create the function template which can perform the addition operation on any type either it can be integer, float or double.

❖ Function Templates with Multiple Parameters

- We can use more than one generic type in the template function by using the comma to separate the list.

Syntax

```
template<class T1, class T2,.....>
return_type function_name (arguments of type T1, T2....)
{
   // body of function.
}
```

- In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

Example:

```
#include <iostream>
using namespace std;
template<class X,class Y> void fun(X a,Y b)
{
   std::cout << "Value of a is : " <<a<< std::endl;
   std::cout << "Value of b is : " <<b<< std::endl;
}

int main()
{
  fun(15,12.3);

  return 0;
}
```

Output: Value of a is : 15
Value of b is : 12.3

- In the above example, we use two generic types in the template function, i.e., X and Y.

❖ Overloading a Function Template

➢ We can overload the generic function means that the overloaded template functions can differ in the parameter list.

Example:

```
#include <iostream>
using namespace std;
template<class X> void fun(X a)
{
   std::cout << "Value of a is : " <<a<< std::endl;
}
template<class X,class Y> void fun(X b ,Y c)
{
   std::cout << "Value of b is : " <<b<< std::endl;
   std::cout << "Value of c is : " <<c<< std::endl;
}
int main()
{
  fun(10);
  fun(20,30.5);
  return 0;
}
```

Output: Value of a is : 10
Value of b is : 20
Value of c is : 30.5

- In the above example, template of fun() function is overloaded.

❖Restrictions of Generic Functions:

- Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

Example:

```
#include <iostream>
using namespace std;
void fun(double a)
{
   cout<<"value of a is : "<<a<<'\n';
}
void fun(int b)
{
   if(b%2==0)
   {
      cout<<"Number is even";
   }
   else
   {
      cout<<"Number is odd";
   } }
int main()
{
  fun(4.6);
  fun(6);
  return 0;
}
```

Output: value of a is : 4.6
Number is even

- In this example, we overload the ordinary functions. We cannot overload the generic functions as both the functions have different functionalities. First one is displaying the value and the second one determines whether the number is even or not.

❖Class template

- Class Template can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

Syntax

```
template<class Ttype>
class class_name
{
  .
  .
}
```

- Ttype is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

Now, we create an instance of a class

```
class_name<type> ob;
```

- **where class_name:** It is the name of the class.
- **type:** It is the type of the data that the class is operating on.
- **ob:** It is the name of the object.

Example:

```
#include <iostream>
using namespace std;
template<class T>
class A
{  public:
   T num1 = 5;
   T num2 = 6;
   void add()
   {
      std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
   }  };
int main()
{  A<int> d;
   d.add();
   return 0;
}
```

- In the above example, we create a template for class A. Inside the main() method, we create the instance of class A named as, 'd'.

Output: Addition of num1 and num2 : 11

➢ Class template with multiple parameters

- We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, ......>
class class_name
{
  // Body of the class.
}
```

Example when class template contains two generic data types.

```
#include <iostream>
    using namespace std;
    template<class T1, class T2>
   class A
   {
      T1 a;
      T2 b;
      public:
```

```
A(T1 x,T2 y)
{
  a = x;
  b = y;
}
  void display()
  {
      std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;
  } };
int main()
{
  A<int,float> d(5,6.5);
  d.display();
  return 0;
}
```

Output: Values of a and b are : 5,6.5

➢ Nontype Template Arguments

- The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types.

Example:

```
template<class T, int size>
class array
{
    T arr[size];  // automatic array initialization.
};
```

- In this case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

➢ Arguments are specified when the objects of a class are created:

```
array<int, 15> t1;          // array of 15 integers.
array<float, 10> t2;        // array of 10 floats.
array<char, 4> t3;          // array of 4 chars.
```

Example of nontype template arguments.

```
#include <iostream>
using namespace std;
template<class T, int size>
class A
{
  public:
  T arr[size];
  void insert()
  {
    int i =1;
    for (int j=0;j<size;j++)
    {
```

```
            arr[j] = i;
            i++;
        }  }
    void display()
    {
        for(int i=0;i<size;i++)
        {
            std::cout << arr[i] << " ";
        } } };
int main()
{
    A<int,10> t1;
    t1.insert();
    t1.display();
    return 0;
}
```

- In this example, the class template is created which contains the nontype template argument, i.e., size. It is specified when the object of class 'A' is created.

Output: 1 2 3 4 5 6 7 8 9 10

❖ Key Points:

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use nontype arguments such as built-in or derived data types as template arguments.

FILES AND STREAMS

- In C++ programming we are using the iostream standard library, it provides cin and cout methods for reading from input and writing to output respectively.
- To read and write from a file we are using the standard C++ library called fstream. Let us see the data types define in fstream library is:

Data Type **Description**

- **fstream** : It is used to create files, write information to files & read information from files.
- **ifstream** : It is used to read information from files.
- **ofstream** : It is used to create files and write information to the files.

Example: writing to a file

Example of writing to a text file testout.txt using C++ FileStream programming.

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
 ofstream filestream("testout.txt");
 if (filestream.is_open())
 {
  filestream << "Welcome to T3 skills center.\n";
  filestream << "C++ Tutorial.\n";
  filestream.close();
 }
 else cout <<"File opening is fail.";
 return 0;
}
```

Output: The content of a text file testout.txt is set with the data:

Welcome to T3 skills center.

Example: reading from a file

Example of reading from a text file testout.txt using C++ FileStream programming.

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
 string srg;
 ifstream filestream("testout.txt");
 if (filestream.is_open())
 {
  while ( getline (filestream,srg) )
  {
   cout << srg <<endl;
  }
  filestream.close();
```

```
    }
    else {
       cout << "File opening is fail."<<endl;
     }
    return 0;
  }
```

Note: Before running the code a text file named as "testout.txt" is need to be created and the content of a text file is given below: Welcome to T3 skills center.

Output: Welcome to T3 skills center.

Example: Read and Write

Example of writing the data to a text file testout.txt and then reading the data from the file using C++ FileStream programming.

```
#include <fstream>
#include <iostream>
using namespace std;
int main () {
  char input[75];
  ofstream os;
  os.open("testout.txt");
  cout <<"Writing to a text file:" << endl;
  cout << "Please Enter your name: ";
  cin.getline(input, 100);
  os << input << endl;
  cout << "Please Enter your age: ";
  cin >> input;
  cin.ignore();
  os << input << endl;
  os.close();
  ifstream is;
  string line;
  is.open("testout.txt");
  cout << "Reading from a text file:" << endl;
  while (getline (is,line))
  {
  cout << line << endl;
  }
  is.close();
  return 0;
}
```

Output: Writing to a text file:

```
Please Enter your name: Nakul Jain
Please Enter your age: 22
Reading from a text file:   Nakul Jain
 22
```

❖ getline():

- The cin is an object which is used to take input from the user but does not allow to take the input in multiple lines. To accept the multiple lines, we use the getline() function. It is a pre-defined function defined in a <string.h> header file used to accept a line or a string from the input stream until the delimiting character is encountered.

❖ Syntax of getline() function:

- There are two ways of representing a function:

➢ The first way of declaring is to pass three parameters.

istream& getline(istream& is, string& str, char delim);

✓ The above syntax contains three parameters, i.e., is, str, and delim.

Where:

- **is:** It is an object of the istream class that defines from where to read the input stream.
- **str:** It is a string object in which string is stored.
- **delim:** It is the delimiting character.

Return value:

- This function returns the input stream object, which is passed as a parameter to the function.

➢ The second way of declaring is to pass two parameters.

istream& getline(istream& is, string& str);

✓ The above syntax contains two parameters, i.e., is and str. This syntax is almost similar to the above syntax; the only difference is that it does not have any delimiting character.

Where:

- **is:** It is an object of the istream class that defines from where to read the input stream.
- **str:** It is a string object in which string is stored.

Return value

- This function also returns the input stream, which is passed as a parameter to the function.

Example.

- First, we will look at an example where we take the user input without using getline() function.

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
string name; // variable declaration
std::cout << "Enter your name :" << std::endl;
cin>>name;
cout<<"\nHello "<<name;
```

```
return 0;
}
```

- In the above code, we take the user input by using the statement cin>>name, i.e., we have not used the getline() function.

Output

```
Enter your name :
Sangam Kumar
Hello Sangam
```

- In the above output, we gave the name ' Sangam Kumar ' as user input, but only 'John' was displayed. Therefore, we conclude that cin does not consider the character when the space character is encountered.

Let's resolve the above problem by using getline() function.

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
string name; // variable declaration.
std::cout << "Enter your name :" << std::endl;
getline(cin,name); // implementing a getline() function
cout<<"\nHello "<<name;
return 0;}
```

- In the above code, we have used the getline() function to accept the character even when the space character is encountered.

Output

```
Enter your name :
Sangam Kumar
Hello Sangam Kumar
```

- In the above output, we can observe that both the words, i.e., Sangam Kumar, are displayed, which means that the getline() function considers the character after the space character also.

When we do not want to read the character after space then we use the following code:

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
string profile; // variable declaration
std::cout << "Enter your profile :" << std::endl;
getline(cin,profile,' '); // implementing getline() function with a delimiting character.
cout<<"\nProfile is :"<<profile;
}
```

- In the above code, we take the user input by using getline() function, but this time we also add the delimiting character('') in a third parameter. Here, delimiting character is a space character, means the character that appears after space will not be considered.

Output

Enter your profile :
Software Developer
Profile is: Software

❖ Getline Character Array:

- We can also define the getline() function for character array, but its syntax is different from the previous one.

Syntax

istream& getline(char* , int size);

- In the above syntax, there are two parameters; one is char*, and the other is size.

Where:

- char*: It is a character pointer that points to the array.
- Size: It acts as a delimiter that defines the size of the array means input cannot cross this size.

➢ **Let's understand through an example.**

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
char fruits[50]; // array declaration
cout<< "Enter your favorite fruit: ";
cin.getline(fruits, 50); // implementing getline() function
std::cout << "\nYour favorite fruit is :"<<fruits << std::endl;
return 0;
}
```

Output

Enter your favorite fruit: Watermelon
Your favorite fruit is: Watermelon

C++ ITERATORS

- Iterators are just like pointers used to access the container elements.

❖ Important Points:

- Iterators are used to traverse from one element to another element, a process is known as iterating through the container.
- The main advantage of an iterator is to provide a common interface for all the containers type.
- Iterators make the algorithm independent of the type of the container used.
- Iterators provide a generic approach to navigate through the elements of a container.

Syntax

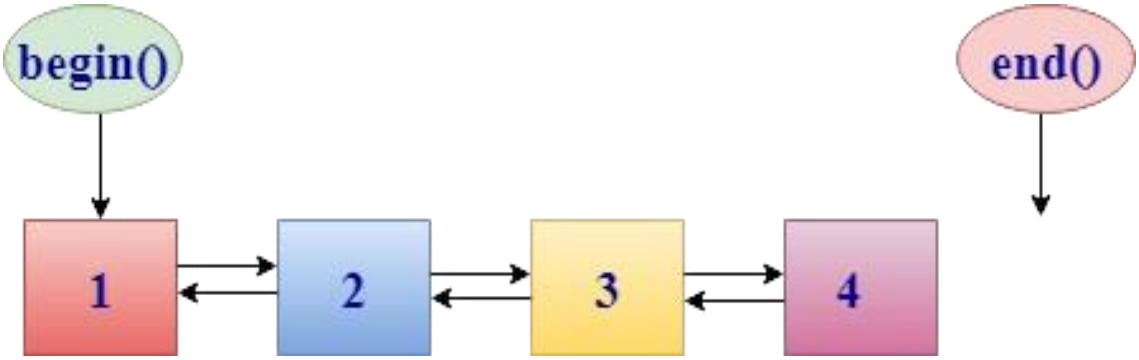
```
<ContainerType> :: iterator;
<ContainerType> :: const_iterator;
```

❖ Operations Performed on the Iterators:

- **Operator (*) :** The '*' operator returns the element of the current position pointed by the iterator.
- **Operator (++) :** The '++' operator increments the iterator by one. Therefore, an iterator points to the next element of the container.
- **Operator (==) and Operator (!=) :** Both these operators determine whether the two iterators point to the same position or not.
- **Operator (=) :** The '=' operator assigns the iterator.

❖ Difference b/w Iterators & Pointers:

- Iterators can be smart pointers which allow to iterate over the complex data structures. A Container provides its iterator type. Therefore, we can say that the iterators have the common interface with different container type.
- The container classes provide two basic member functions that allow to iterate or move through the elements of a container:
- **begin():** The begin() function returns an iterator pointing to the first element of the container.
- **end():** The end() function returns an iterator pointing to the past-the-last element of the container.



Example:

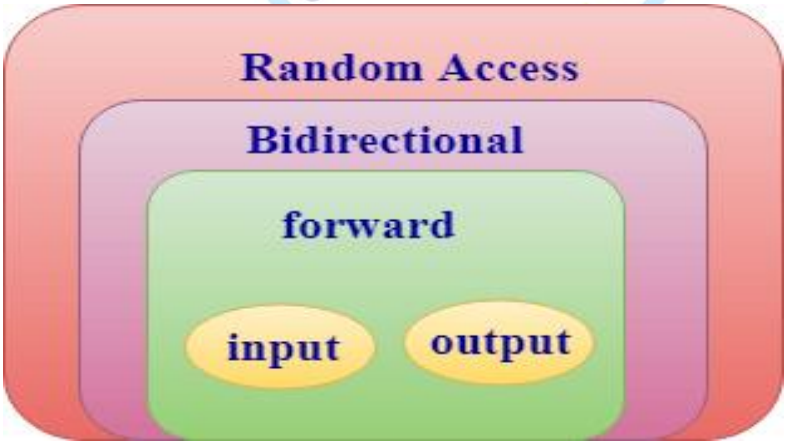
```
#include <iostream>
#include<iterator>
#include<vector>
using namespace std;
int main()
{
  std::vector<int> v{1,2,3,4,5};
  vector<int>::iterator itr;
  for(itr=v.begin();itr!=v.end();itr++)
  {
     std::cout << *itr <<" ";
  }
   return 0;
}
```

Output:

1 2 3 4 5

❖Iterator Categories:

- An iterator can be categorized in the following ways:
  1. Input Iterator
  2. Output Iterator
  3. Forward Iterator
  4. Bidirectional Iterator
  5. Random Access Iterator



❖ **Input Iterator:** An input iterator is an iterator used to access the elements from the container, but it does not modify the value of a container.

- Operators used for an input iterator are:
  - ✓ Increment operator(++)
  - ✓ Equal operator(==)
  - ✓ Not equal operator(!=)
  - ✓ Dereference operator(*)

- **Output Iterator:** An output iterator is an iterator used to modify the value of a container, but it does not read the value from a container. Therefore, we can say that an output iterator is a write-only iterator.
  - Operators used for an output iterator are:
    - ✓ Increment operator(++)
    - ✓ Assignment operator(=)
- **Forward Iterator:** A forward iterator is an iterator used to read and write to a container. It is a multi-pass iterator.
  - Operators used for a Forward iterator are:
    - ✓ Increment operator(++)
    - ✓ Assignment operator(=)
    - ✓ Equal operator(=)
    - ✓ Not equal operator(!=)
- **Bidirectional iterator:** A bidirectional iterator is an iterator supports all the features of a forward iterator plus it adds one more feature, i.e., decrement operator(--). We can move backward by decrementing an iterator.
  - Operators used for a Bidirectional iterator are:
    - ✓ Increment operator(++)
    - ✓ Assignment operator(=)
    - ✓ Equal operator(=)
    - ✓ Not equal operator(!=)
    - ✓ Decrement operator(--)
- **Random Access Iterator:** A Random Access iterator is an iterator provides random access of an element at an arbitrary location. It has all the features of a bidirectional iterator plus it adds one more feature, i.e., pointer addition and pointer subtraction to provide random access to an element.

Providers Of Iterators

Iterator categories	Provider
1. Input iterator	istream
2. Output iterator	ostream
3. Forward iterator	
4. Bidirectional iterator	List, set, multiset, map, multimap
5. Random access iterator	Vector, deque, array

Disadvantages of iterator:

- If we want to move from one data structure to another at the same time, iterators won't work.
- If we want to update the structure which is being iterated, an iterator won?t allow us to do because of the way it stores the position.
- If we want to backtrack while processing through a list, the iterator will not work in this case.

Advantages of iterator:

- **Ease in programming:** It is convenient to use iterators rather than using a subscript operator[] to access the elements of a container.

Example:

```
#include <iostream>
#include<vector>
#include<iterator>
using namespace std;
int main()
{
   vector<int> v{1,2,3,4,5};
   vector<int>::iterator itr;
   for(int i=0;i<5;i++)          // Traversal without using an iterator.
   {
      cout<<v[i]<<" ";
   }
   cout<<'\n';
   for(itr=v.begin();itr!=v.end();itr++)  // Traversal by using an iterator.
   {
      cout<<*itr<<" ";
   }
   v.push_back(10);
   cout<<'\n';
    for(int i=0;i<6;i++)
   {
      cout<<v[i]<<" ";
   }
   cout<<'\n';
    for(itr=v.begin();itr!=v.end();itr++)
   {
      cout<<*itr<<" ";
   }
    return 0;
}
```

Output:

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5 10
1 2 3 4 5 10
```

- **Code Reusability:** A code can be reused if we use iterators. In the above example, if we replace vector with the list, and then the subscript operator[] would not work to access the elements as the list does not support the random access. However, we use iterators to access the elements, then we can also access the list elements.

- **Dynamic Processing:** C++ iterators provide the facility to add or delete the data dynamically.

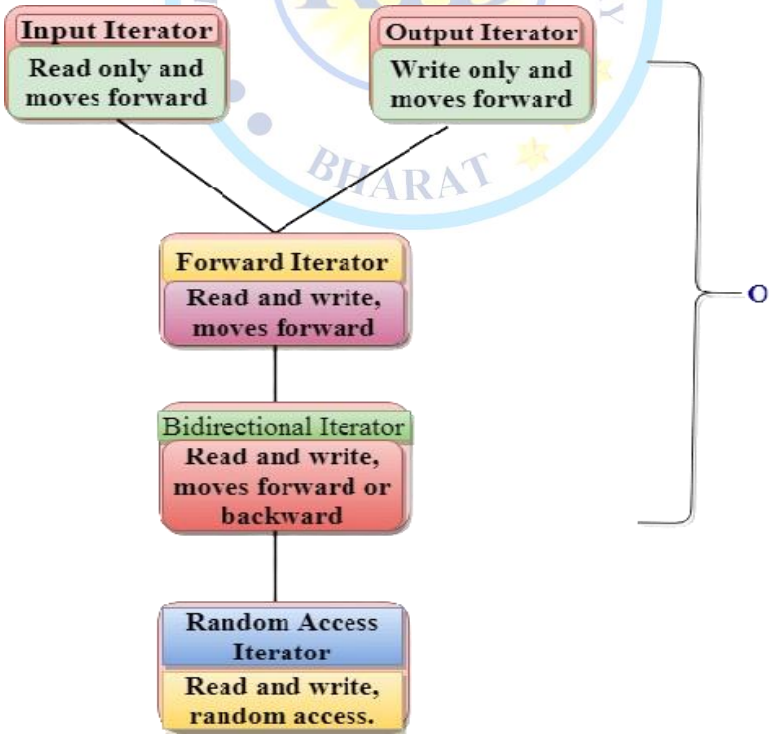
Example:

```
#include <iostream>
#include<vector>
#include<iterator>
using namespace std;
int main()
{
   vector<int> v{1,2,3,4,5};  // vector declaration
   vector<int>::iterator itr;
   v.insert(v.begin()+1,10);
   for(itr=v.begin();itr!=v.end();itr++)
   {
      cout<<*itr<<" ";
   }
   return 0;
}
```

Output: 1 10 2 3 4 5

- In the above example, we insert a new element at the second position by using insert() function and all other elements are shifted by one.

❖ Difference b/w Random Access Iterator and Other Iterators:



- The most important difference between the Random access iterator and other iterators is that random access iterator requires '1' step to access an element while other iterators require 'n' steps.

STL (Standard Template Library)

- "STL," which stands for Standard Template Library The Standard Template Library (STL) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, stacks, maps, sets, etc.

❖ some key components of the C++ STL:

1. Containers:

➢ Containers are objects that store data. Examples include:

- **Vector:** Dynamic array.
- **List:** Doubly linked list.
- **Deque:** Double-ended queue.
- **Queue:** FIFO (First-In-First-Out) data structure.
- **Stack:** LIFO (Last-In-First-Out) data structure.
- **Map:** Associative container that stores key-value pairs sorted by keys.
- **Set:** Associative container that stores unique elements sorted in ascending order.

2. Iterators:

- Iterators provide a way to access elements of containers in a sequential manner. They act as pointers to elements within the container.

3. Algorithms:

- The STL provides a wide range of algorithms that operate on containers. These include sorting, searching, modifying, and manipulating algorithms like sort(), find(), transform(), etc.

4. Function Objects (Functors):

- Functors are objects that can be called as if they were functions. They are often used as arguments to algorithms to define custom behavior.

5. Utilities:

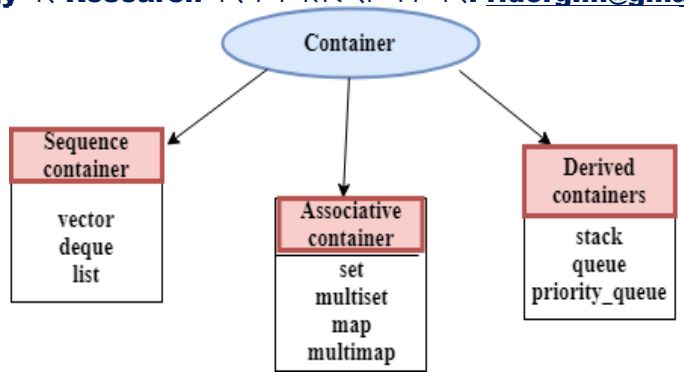
- The STL includes utility classes and functions like pair (for storing two heterogeneous objects), swap(), move(), etc.

CONTAINERS

➢ Containers can be described as the objects that hold the data of the same type. Containers are used to implement different data structures

➢ Classification of containers :

- Sequence containers
- Associative containers
- Derived containers



1. Sequence Containers:

- Sequence containers are data structures that maintain the order of elements as they are inserted. Elements are stored in a linear sequence, and operations like insertion, deletion, and traversal are performed in a linear manner.
- Common sequence containers in C++ STL include:
  - ✓ **vector:** Dynamic array that grows and shrinks automatically.
  - ✓ **deque (double-ended queue):** Similar to a vector but allows efficient insertion and deletion at both ends.
  - ✓ **list:** Doubly linked list that supports efficient insertion and deletion at any position.
  - ✓ **forward_list:** Singly linked list that supports efficient insertion and deletion at the beginning and after an element.

2. Associative Containers:

- Associative containers are data structures that store elements in a sorted order based on some criteria (e.g., key) and provide efficient searching, insertion, and deletion operations.
- Common associative containers in C++ STL include:
  - ✓ **set:** Collection of unique elements sorted in ascending order.
  - ✓ **multiset:** Collection of elements sorted in ascending order, allowing duplicate elements.
  - ✓ **map:** Collection of key-value pairs sorted by key. Keys are unique.
  - ✓ **multimap:** Collection of key-value pairs sorted by key, allowing duplicate keys.

3. Derived Containers:

- Derived containers are specialized containers that are built on top of sequence or associative containers to provide additional functionalities or optimizations.
- Common derived containers in C++ STL include:
  - ✓ **stack:** Adapter class that provides a Last-In-First-Out (LIFO) data structure using a sequence container (e.g., deque or list) as the underlying container.
  - ✓ **queue:** Adapter class that provides a First-In-First-Out (FIFO) data structure using a sequence container as the underlying container.
  - ✓ **priority_queue:** Adapter class that provides a priority queue using a sequence container as the underlying container. Elements are dequeued based on their priority.

C++ VECTOR

- A vector is a sequence container class that implements dynamic array, means size automatically changes when appending elements. A vector stores the elements in contiguous memory locations and allocates the memory as needed at run time.

❖ Difference between vector and array:

- An array follows static approach, means its size cannot be changed during run time while vector implements dynamic array means it automatically resizes itself when appending elements.

Syntax

Consider a vector 'v1'. Syntax would be:

vector<object_type> v1;

Example

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<string> v1;
v1.push_back("t3skillscenter ");
v1.push_back("RID ORG");
for(vector<string>::iterator itr=v1.begin();itr!=v1.end();++itr)
cout<<*itr;
return 0;
}
```

Output: t3skillscenter RID ORG

❖C++ Vector Functions

Function	Description
1) at()	It provides a reference to an element.
2) back()	It gives a reference to the last element.
3) front()	It gives a reference to the first element.
4) swap()	It exchanges the elements between two vectors.
5) push_back()	It adds a new element at the end.
6) pop_back()	It removes a last element from the vector.
7) empty()	It determines whether the vector is empty or not.
8) insert()	It inserts new element at the specified position.
9) erase()	It deletes the specified element.
10) resize()	It modifies the size of the vector.
11) clear()	It removes all the elements from the vector.
12) size()	It determines a number of elements in the vector.
13) capacity()	It determines the current capacity of the vector.
14) assign()	It assigns new values to the vector.

15) operator=()	It assigns new values to the vector container.
16) operator[]()	It access a specified element.
17) end()	It refers to the past-lats-element in the vector.
18) emplace()	It inserts a new element just before the position pos.
19) emplace_back()	It inserts a new element at the end.
20) rend()	It points the element preceding the first element of the vector.
21) rbegin()	It points the last element of the vector.
22) begin()	It points the first element of the vector.
23) max_size()	It determines the maximum size that vector can hold.
24) cend()	It refers to the past-last-element in the vector.
25) cbegin()	It refers to the first element of the vector.
26) crbegin()	It refers to the last character of the vector.
27) crend()	It refers to the element preceding the first element of the vector.
28) data()	It writes the data of the vector into an array.
29) shrink_to_fit()	It reduces the capacity and makes it equal to the size of the vector.

1. at() - Accessing Element by Index:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> numbers = {10, 20, 30, 40, 50};
   // Accessing the element at index 2
   std::cout << "Element at index 2: " << numbers.at(2) << std::endl;
   return 0;
}
```

Output: Element at index 2: 30

2. back() - Accessing the Last Element:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> numbers = {10, 20, 30, 40, 50};
std::cout << "Last element: " << numbers.back() << std::endl;
   return 0;
}
```

Output: Last element: 50

3. swap() - Exchanging Elements Between Vectors:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> numbers1 = {1, 2, 3};
   std::vector<int> numbers2 = {10, 20, 30};
   numbers1.swap(numbers2);
   std::cout << "Numbers1 after swap: ";
   for (int num : numbers1) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   std::cout << "Numbers2 after swap: ";
   for (int num : numbers2) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0; }
```

Output: Numbers1 after swap: 10 20 30
Numbers2 after swap: 1 2 3

4. front() - Accessing the First Element:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> numbers = {10, 20, 30, 40, 50};
   // Accessing the first element
   std::cout << "First element: " << numbers.front() << std::endl;
   return 0;
}
```

Output: First element: 10

6. push_back(): Adds a new element at the end of the vector:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> numbers;
   numbers.push_back(10);
   numbers.push_back(20);
   numbers.push_back(30);
   std::cout << "Vector elements after push_back(): ";
   for (int num : numbers) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Vector elements after push_back(): 10 20 30

5. pop_back(): Removes the last element from vector:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> numbers = {10, 20, 30};
   numbers.pop_back();
   std::cout << "Vector elements after pop_back(): ";
   for (int num : numbers) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Vector elements after pop_back(): 10 20

8. insert(): Inserts a new element at the specified position:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> numbers = {10, 20, 30};
   // Inserting a new element at position 1
   numbers.insert(numbers.begin() + 1, 15);
   // Output the vector elements
   std::cout << "Vector elements after insert(): ";
   for (int num : numbers) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Vector elements after insert(): 10 15 20 30

7. empty(): Determines whether the vector is empty or not:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> numbers;
   // Check if the vector is empty
   if (numbers.empty()) {
      std::cout << "Vector is empty." << std::endl;
   } else {
      std::cout << "Vector is not empty." << std::endl;
   }
   return 0;
}
```

Output: Vector is empty.

9. erase(): Deletes the specified element:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> numbers = {10, 15, 20, 30};
   // Erasing the element at position 2
   numbers.erase(numbers.begin() + 2);
   // Output the vector elements
   std::cout << "Vector elements after erase(): ";
   for (int num : numbers) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Vector elements after erase(): 10 15 30

10. resize(): Modifies the size of the vector:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> numbers = {10, 20, 30};
   // Resizing the vector to have 5 elements
   numbers.resize(5, 0); // Fill the new elements with 0
   // Output the vector elements
   std::cout << "Vector elements after resize(): ";
   for (int num : numbers) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Vector elements after resize(): 10 20 30 0 0

11. clear(): Removes all elements from the vector:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   std::cout << "Before clear, size: " << vec.size() << std::endl;
   vec.clear();
   std::cout << "After clear, size: " << vec.size() << std::endl;
   return 0;
}
```

Output: Before clear, size: 5
After clear, size: 0

12. size(): Determines the number of elements in the vector:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   std::cout << "Size of the vector: " << vec.size() << std::endl;
   return 0;
}
```

Output: Size of the vector: 5.

13. capacity(): Determines the current capacity of the vector.:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   std::cout << "Capacity of the vector: " << vec.capacity() << std::endl;
   return 0;
}
```

Output: Capacity of the vector: 5.

14. assign(): Assigns new values to the vector:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   vec.assign({10, 20, 30});
   for (int num : vec) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: 10 20 30

15. operator=(): Assigns new values to the vector container:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec1 = {1, 2, 3, 4, 5};
   std::vector<int> vec2;
   vec2 = vec1;
   for (int num : vec2) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: 1 2 3 4 5

16. operator[]():

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   // Accessing elements using operator[]
   std::cout << "Element at index 2: " << vec[2] <<
std::endl;
   return 0;
}
```

Output: Element at index 2: 3

17. end():

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   // Using end() to refer to the past-the-last-element
   auto it = vec.end();
   std::cout << "Value of past-the-last-element: " <<
*(it - 1) << std::endl;
   return 0;
}
```

Output: Vector is empty.

18. emplace():

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
// Inserting a new element just before the position pos
   auto it = vec.emplace(vec.begin() + 2, 10); // insert
10 before index 2
   for (int num : vec) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: 1 2 10 3 4 5.

19. emplace_back():

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   vec.emplace_back(10);
   for (int num : vec) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: 1 2 3 4 5 10.

20. rend():

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   auto it = vec.rend(); // it points to the element preceding the first element
   std::cout << "Value of element preceding the first element: " << *(it - 1) << std::endl;
   return 0;
}
```

Output: Value of element preceding the first element: 1

21. rbegin():

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   auto it = vec.rbegin();
   std::cout << "Last element of the vector: " << *it << std::endl;
   return 0;
}
```

Output: Last element of the vector: 5

22. begin() - It points the first element of the vector:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   // Using begin() to get iterator to the first element
   std::vector<int>::iterator it = vec.begin();
   // Output the first element of the vector
   std::cout << "First element of the vector: " << *it << std::endl;
   return 0;
}
```

Output: First element of the vector: 1.

23. max_size() - It determines the maximum size that vector can hold:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec;
   // Output the maximum size of the vector
   std::cout << "Maximum size of the vector: " << vec.max_size() << std::endl;
   return 0;
}
```

Output: Maximum size of the vector: 4611686018427387903

24. cend() - It refers to the past-last-element in the vector:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   std::vector<int>::const_iterator it = vec.cend();
   // Output the value of the past-last-element
   std::cout << "Value of the past-last-element: " << *(--it) << std::endl;
   return 0;
}
```

Output: Value of the past-last-element: 5.

25. cbegin() - It refers to the first element of the vector:

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   std::vector<int>::const_iterator it = vec.cbegin();
   // Output the value of the first element
   std::cout << "Value of the first element: " << *it << std::endl;
   return 0;
}
```

Output: Value of the first element: 1

26 crend() - It refers to the element preceding the first element of the vector

Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   // Using crend() to get const reverse iterator to the element preceding the first element
   std::vector<int>::const_reverse_iterator it = vec.crend();
   std::cout << "Value of the element preceding the first element: " << *(--it) << std::endl;
   return 0;
}
```

Output: Value of the element preceding the first element: 5

27. data() - It writes the data of the vector into an array:

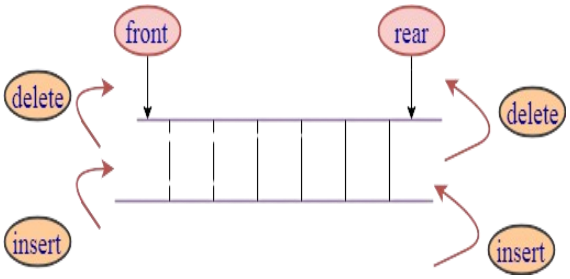
Program:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4, 5};
   int* ptr = vec.data()
  std::cout << "Elements of the vector using data(): ";
   for (size_t i = 0; i < vec.size(); ++i) {
      std::cout << *(ptr + i) << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements of the vector using data(): 1 2 3 4 5

C++ DEQUE

- Deque stands for double ended queue. It generalizes the queue data structure i.e insertion and deletion can be performed from both the ends either front or back.



❖ Syntax for creating a deque object:

deque<object_type> deque_name;

❖ Deque Functions:

Method	Description
1. assign()	It assigns new content and replacing the old one.
2. emplace()	It adds a new element at a specified position.
3. emplace_back()	It adds a new element at the end.
4. emplace_front()	It adds a new element in the beginning of a deque.
5. insert()	It adds a new element just before the specified position.
6. push_back()	It adds a new element at the end of the container.
7. push_front()	It adds a new element at the beginning of the container.
8. pop_back()	It deletes the last element from the deque.
9. pop_front()	It deletes the first element from the deque.
10. swap()	It exchanges the contents of two deques.
11. clear()	It removes all the contents of the deque.
12. empty()	It checks whether the container is empty or not.
13. erase()	It removes the elements.
14. max_size()	It determines the maximum size of the deque.
15. resize()	It changes the size of the deque.
16. shrink_to_fit()	It reduces the memory to fit the size of the deque.
17. size()	It returns the number of elements.
18. at()	It access the element at position pos.
19. operator[]()	It access the element at position pos.
20. operator=()	It assigns new contents to the container.
21. back()	It access the last element.
22. begin()	It returns an iterator to the beginning of the deque.
23. cbegin()	It returns a constant iterator to the beginning of the deque.
24. end()	It returns an iterator to the end.
25. cend()	It returns a constant iterator to the end.
26. rbegin()	It returns a reverse iterator to the beginning.
27. crbegin()	It returns a constant reverse iterator to the beginning.
28. rend()	It returns a reverse iterator to the end.
29. crend()	It returns a constant reverse iterator to the end.
30. front()	It access the last element.

1. assign() - It assigns new content, replacing the old one:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> deque1 = {1, 2, 3};
   // Assign new content to the deque
   deque1.assign({4, 5, 6});
   std::cout << "Elements after assigning new content:";
   for (int num : deque1) {
      std::cout << " " << num;
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements after assigning new content: 4 5 6

2. emplace() - It adds a new element at a specified position:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> deque1 = {1, 2, 3};
   // Emplace a new element at the second position
   deque1.emplace(deque1.begin() + 1, 4);
   std::cout << "Elements after emplacing:";
   for (int num : deque1) {
      std::cout << " " << num;
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements after emplacing: 1 4 2 3

3.emplace_back() - It adds a new element at the end:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> deque1 = {1, 2, 3};
   deque1.emplace_back(4);
  std::cout << "Elements after emplacing at the end:";
   for (int num : deque1) {
      std::cout << " " << num;
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements after emplacing at the end: 1 2 3 4

4.emplace_front() - It adds a new element at the beginning of a deque:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> deque1 = {1, 2, 3};
   // Emplace a new element at the beginning
   deque1.emplace_front(0);
   // Output the elements of the deque after emplacing at the beginning
   std::cout << "Elements after emplacing at the beginning:";
   for (int num : deque1) {
      std::cout << " " << num;
   }
   std::cout << std::endl;
   return 0;
}
```

Output:

Elements after emplacing at the beginning: 0 1 2 3

5. insert() - It adds a new element just before the specified position:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> deque1 = {1, 2, 3};
   // Insert a new element just before the second position
   deque1.insert(deque1.begin() + 1, 4)
  // Output the elements of the deque after insertion
   std::cout << "Elements after insertion:";
   for (int num : deque1) {
      std::cout << " " << num;
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements after insertion: 1 4 2 3

6. push_back() - It adds a new element at the end of the container:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> deque1 = {1, 2, 3};
   // Push a new element at the end
   deque1.push_back(4);
   std::cout << "Elements after pushing at the end:";
   for (int num : deque1) {
      std::cout << " " << num;
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements after pushing at the end: 1 2 3 4

7. push_front() - It adds a new element at the beginning of the container.:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> deque1 = {1, 2, 3};
   deque1.push_front(0);
   std::cout << "Elements after pushing at the beginning:";
   for (int num : deque1) {
      std::cout << " " << num;
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements after pushing at the beginning: 0 1

8. pop_back() - It deletes the last element from the deque:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> deque1 = {1, 2, 3};
   // Pop the last element from the deque
   deque1.pop_back();
   std::cout << "Elements after popping the last element:";
   for (int num : deque1) {
      std::cout << " " << num;
   }
   std::cout << std::endl;
   return 0;
}
```

Output:

Elements after popping the last element: 1 2

9. pop_front() - It deletes the first element from the deque: :

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> deque1 = {1, 2, 3};
   // Pop the first element from the deque
   deque1.pop_front();
   std::cout << "Elements after popping the first element:";
   for (int num : deque1) {
      std::cout << " " << num;
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements after popping the first element: 2 3

10. push_back() - It adds a new element at the end of the container:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> deque1 = {1, 2, 3};
   std::deque<int> deque2 = {4, 5, 6};
   // Swap the contents of deque1 and deque2
   deque1.swap(deque2);
   std::cout << "Elements of deque1 after swapping:";
   for (int num : deque1) {
      std::cout << " " << num;
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements of deque1 after swapping: 4 5 6

11. clear() - It removes all the contents of the deque:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq = {1, 2, 3, 4, 5};
   // Clear the deque
   dq.clear();
   std::cout << "Size of the deque after clearing: " << dq.size() << std::endl;
   return 0;
}
```

Output: Size of the deque after clearing: 0

12. empty() - It checks whether the container is empty or not:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq;
   // Check if the deque is empty
   if (dq.empty()) {
      std::cout << "Deque is empty" << std::endl;
   } else {
      std::cout << "Deque is not empty" << std::endl;
   }
   return 0;
}
```

Output: Deque is empty

13. erase() - It removes the elements:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq = {1, 2, 3, 4, 5};
   // Erase the second element
   dq.erase(dq.begin() + 1);
   // Output the elements of the deque after erasing
   std::cout << "Elements of the deque after erasing: ";
   for (int num : dq) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements of the deque after erasing: 1 3 4 5

14. max_size() - It determines the maximum size of the deque: :

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq;
   std::cout << "Maximum size of the deque: " << dq.max_size() << std::endl;
   return 0;
}
```

Output: Maximum size of the deque: 1152921504606846975

15. resize() - It changes the size of the deque:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq = {1, 2, 3};
   dq.resize(5);
   std::cout << "Elements of the deque after resizing: ";
   for (int num : dq) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements of the deque after resizing: 1 2 3 0 0

16. shrink_to_fit() - It reduces the memory to fit the size of the deque:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq = {1, 2, 3, 4, 5};
   dq.resize(3);
   dq.shrink_to_fit();
   std::cout << "Capacity of the deque after shrinking: " << dq.capacity() << std::endl;
   return 0;
}
```

Output: Capacity of the deque after shrinking: 3

17. size() - It returns the number of elements:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq = {1, 2, 3, 4, 5};
   // Output the number of elements in the deque
   std::cout << "Number of elements in the deque: " << dq.size() << std::endl;
   return 0;
}
```

Output: Number of elements in the deque: 5

18. at() - It accesses the element at position pos:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq = {1, 2, 3, 4, 5};
   // Accessing the element at position 2 using at()
   int element = dq.at(2);
   // Output the accessed element
   std::cout << "Element at position 2: " << element << std::endl;
   return 0;
}
```

Output: Element at position 2: 3

19. operator[]() - It accesses the element at position pos:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq = {1, 2, 3, 4, 5};
   // Accessing the element at position 3 using operator[]
   int element = dq[3];
   // Output the accessed element
   std::cout << "Element at position 3: " << element << std::endl;
   return 0;
}
```

Output: Element at position 3: 4

20. operator=() - It assigns new contents to the containe

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq1 = {1, 2, 3};
   std::deque<int> dq2 = {4, 5, 6};
   dq1 = dq2;
  std::cout << "Elements of dq1 after assignment: ";
   for (int elem : dq1) {
      std::cout << elem << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements of dq1 after assignment: 4 5 6

21. back() - It accesses the last element:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq = {1, 2, 3, 4, 5};
   // Accessing the last element using back()
   int last_element = dq.back();
   std::cout << "Last element of the deque: " << last_element << std::endl;
   return 0;
}}
```

Output: Last element of the deque: 5

22. begin() - It returns an iterator to the beginning of the deque:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq = {1, 2, 3, 4, 5};
   std::deque<int>::iterator it = dq.begin();
   std::cout << "Element at the beginning of the deque: " << *it << std::endl;
   return 0;
}
```

Output: Element at the beginning of the deque: 1

23. cbegin() - It returns a constant iterator to the beginning of the deque:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq = {1, 2, 3, 4, 5};
   std::deque<int>::const_iterator it = dq.cbegin();
   std::cout << "Value of the first element: " << *it << std::endl;
   return 0;
}
```

Output: Value of the first element: 1

24. end() - It returns an iterator to the end:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq = {1, 2, 3, 4, 5};
   // Using end() to get an iterator to the end
   std::deque<int>::iterator it = dq.end();
   std::cout << "Value of the element after the last element: " << *(--it) << std::endl;
   return 0;
}
```

Output: Value of the element after the last element: 5

25. cend() - It returns a constant iterator to the end:

Program:

```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq = {1, 2, 3, 4, 5};
   // Using cend() to get a constant iterator to the end
   std::deque<int>::const_iterator it = dq.cend();
   // Output the value of the element after the last element using the constant iterator
   std::cout << "Value of the element after the last element: " << *(--it) << std::endl;
   return 0;
}
```

Output: Value of the element after the last element: 5

C++ LIST

- List is a contiguous container while vector is a non-contiguous container i.e list stores the elements on a contiguous memory and vector stores on a non-contiguous memory.
- Insertion and deletion in the middle of the vector is very costly as it takes lot of time in shifting all the elements. Linklist overcome this problem and it is implemented using list container.
- List supports a bidirectional and provides an efficient way for insertion and deletion operations.
- Traversal is slow in list as list elements are accessed sequentially while vector supports a random access.

❖ Template for list

```
#include<iostream>
#include<list>
using namespace std;
int main()
{
  list<int> l;
}
```

➢ It creates an empty list of integer type values.

- List can also be initalised with the parameters.

```
#include<iostream>
#include<list>
using namespace std;
int main()
{
  list<int> l{1,2,3,4};
}
```

❖ List can be initialised in two ways.

1. list<int> new_list{1,2,3,4}; or
2. list<int> new_list = {1,2,3,4};

❖ C++ List Functions

- Following are the member functions of the list:

Method	Description
1) insert()	It inserts the new element before the position pointed by the iterator.
2) push_back()	It adds a new element at the end of the vector.
3) push_front()	It adds a new element to the front.
4) pop_back()	It deletes the last element.
5) pop_front()	It deletes the first element.
6) empty()	It checks whether the list is empty or not.
7) size()	It finds the number of elements present in the list.
8) max_size()	It finds the maximum size of the list.
9) front()	It returns the first element of the list.

10) back()It returns the last element of the list.
11) swap()It swaps two list when the type of both the list are same.
12) reverse()It reverses the elements of the list.
13) sort()It sorts the elements of the list in an increasing order.
14) merge()It merges the two sorted list.
15) splice()It inserts a new list into the invoking list.
16) unique()It removes all the duplicate elements from the list.
17) resize()It changes the size of the list container.
18) assign()It assigns a new element to the list container.
19) emplace()It inserts a new element at a specified position.
20) emplace_back()It inserts a new element at the end of the vector.
21) emplace_front()It inserts a new element at the beginning of the list.

1. insert() - It inserts the new element before the position pointed by the iterator:

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   std::list<int>::iterator it = ++myList.begin();
   myList.insert(it, 10);
   for (const auto& element : myList) {
      std::cout << element << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: 1 10 2 3 4 5

2. push_back() - It adds a new element at the end of the list.

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   myList.push_back(6);
   // Output the list after push_back()
   for (const auto& element : myList) {
      std::cout << element << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: 1 2 3 4 5 6

3. push_front() - It adds a new element to the front:

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   myList.push_front(0);
   for (const auto& element : myList) {
      std::cout << element << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: 0 1 2 3 4 5

4. pop_back() - It deletes the last element.

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   // Using pop_back() to delete the last element
   myList.pop_back();
   // Output the list after pop_back()
   for (const auto& element : myList) {
      std::cout << element << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: 1 2 3 4

5. pop_front() - It deletes the first element:

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   myList.pop_front();
   for (const auto& element : myList) {
      std::cout << element << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: 2 3 4 5

6. empty() - It checks whether the list is empty or not.

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   // Check if the list is empty
   if (myList.empty()) {
      std::cout << "The list is empty." << std::endl;
   } else {
      std::cout << "The list is not empty." << std::endl;
   }
   return 0;
}
```

Output: The list is not empty.

7. size() - It finds the number of elements present in the list:

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   // Output the number of elements in the list
   std::cout << "Number of elements in the list: " << myList.size() << std::endl;
   return 0;
}
```

Output: Number of elements in the list: 5

8. max_size() - It finds the maximum size of the list.

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList;
   std::cout << "Maximum size of the list: " << myList.max_size() << std::endl;
   return 0;
}
```

Output: Maximum size of the list: 1152921504606846975

9. front() - It returns the first element of the list:

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   // Output the first element of the list
   std::cout << "First element of the list: " << myList.front() << std::endl;
   return 0;
}
```

Output: First element of the list: 1

10. back() - It returns the last element of the list.

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   // Output the last element of the list
   std::cout << "Last element of the list: " << myList.back() << std::endl;
   return 0;
}
```

Output: Last element of the list: 5

11. swap() - It swaps two lists when the type of both lists is same:

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> list1 = {1, 2, 3};
   std::list<int> list2 = {4, 5, 6};
   list1.swap(list2);
  std::cout << "Elements of the first list after swap: ";
   for (int num : list1) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;}
```

Output: Elements of the first list after swap: 4 5 6

12. reverse() - It reverses the elements of the list.

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   myList.reverse();
   std::cout << "Elements of the list after reverse: ";
   for (int num : myList) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Elements of the list after reverse: 5 4 3 2 1.

13. sort() - It sorts the elements of the list in increasing order

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {5, 2, 8, 3, 1};
   myList.sort();
   std::cout << "Elements of the list after sorting: ";
   for (int num : myList) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0; }
```

Output: Elements of the list after sorting: 1 2 3 5 8

14. merge() - It merges two sorted lists.

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> list1 = {1, 3, 5};
   std::list<int> list2 = {2, 4, 6};
   list1.merge(list2);
   std::cout << "Elements of list1 after merge: ";
   for (int num : list1) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;}
```

Output: Elements of list1 after merge: 1 2 3 4 5 6

15. splice() - It inserts a new list into the invoking list:

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> list1 = {1, 2, 3};
   std::list<int> list2 = {4, 5, 6};
   auto it = list1.begin();
   std::advance(it, 2);
   list1.splice(it, list2);
   std::cout << "Merged List: ";
   for (const auto& elem : list1) {
      std::cout << elem << " ";
   }
   std::cout << std::endl;
   return 0;}
```

Output: Merged List: 1 2 4 5 6 3

16. unique() - It removes all the duplicate elements from the list.

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 2, 3, 3, 4, 5, 5};
   myList.unique();
   std::cout << "List after removing duplicates: ";
   for (const auto& elem : myList) {
      std::cout << elem << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: List after removing duplicates: 1 2 3 4 5

17. resize() - It changes the size of the list container:

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3};
   myList.resize(5);
   std::cout << "Resized List: ";
   for (const auto& elem : myList) {
      std::cout << elem << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Resized List: 1 2 3 0 0

18. assign() - It assigns a new element to the list container.

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList;
   myList.assign({1, 2, 3, 4, 5});
   std::cout << "Assigned List: ";
   for (const auto& elem : myList) {
      std::cout << elem << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: Assigned List: 1 2 3 4 5 1.

19. emplace() - It inserts a new element at a specified position

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   auto it = std::next(myList.begin(), 1);
   myList.emplace(it, 10);
   std::cout << "List after emplacing element: ";
   for (const auto& elem : myList) {
      std::cout << elem << " ";}
   std::cout << std::endl;
   return 0; }
```

Output: List after emplacing element: 1 10 2 3 4 5

20. emplace_back() - It inserts a new element at the end of the list. Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   myList.emplace_back(6);
   std::cout << "List after emplacing element at the end: ";
   for (const auto& elem : myList) {
      std::cout << elem << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output: List after emplacing element at the end: 1 2 3 4 5 6

21. emplace_front() - It inserts a new element at the beginning of the list.:

Program:

```
#include <iostream>
#include <list>
int main() {
   std::list<int> myList = {1, 2, 3, 4, 5};
   myList.emplace_front(0);
   std::cout << "List after emplacing element at the beginning: ";
   for (const auto& elem : myList) {
      std::cout << elem << " ";
   }
   std::cout << std::endl;
   return 0; }
```

Output: List after emplacing element at the beginning: 0 1 2 3 4 5

C++ SET

- Sets are part of the C++ STL (Standard Template Library). Sets are the associative containers that stores sorted key, in which each key is unique and it can be inserted or deleted but cannot be altered.

Syntax

```
template < class T,                     // set::key_type/value_type
        class Compare = less<T>,        // set::key_compare/value_compare
        class Alloc = allocator<T>      // set::allocator_type
        > class set;
```

Parameter

- **T:** Type of element stored in the container set.
- **Compare:** A comparison class that takes two arguments of the same type bool and returns a value. This argument is optional and the binary predicate less<T>, is the default value.
- **Alloc:** Type of the allocator object which is used to define the storage allocation model.

❖ Member Functions:

- Below is the list of all member functions of set:

❖ Constructor/Destructor:

Functions	Description
1. (constructor)	Construct set
2. (destructor)	Set destructor
3. operator=	Copy elements of the set to another set.

❖ Iterators:

Functions	Description
1. Begin	Returns an iterator pointing to the first element in the set.
2. cbegin	Returns a const iterator pointing to the first element in the set.
3. End	Returns an iterator pointing to the past-end.
4. Cend	Returns a constant iterator pointing to the past-end.
5. rbegin	Returns a reverse iterator pointing to the end.
6. Rend	Returns a reverse iterator pointing to the beginning.
7. crbegin	Returns a constant reverse iterator pointing to the end.
8. Crend	Returns a constant reverse iterator pointing to the beginning.

❖ Capacity

Functions	Description
1) empty	Returns true if set is empty.
2) Size	Returns the number of elements in the set.
3) max_size	Returns the maximum size of the set.

❖ Modifiers

Functions	Description
1) insert	Insert element in the set.
2) Erase	Erase elements from the set.
3) Swap	Exchange the content of the set.
4) Clear	Delete all the elements of the set.
5) emplace	Construct and insert the new elements into the set.
6) emplace_ hint	Construct and insert new elements into the set by hint.

❖ Observers:

Functions	Description
1) key_comp	Return a copy of key comparison object.
2) value_comp	Return a copy of value comparison object.

❖ Operations:

Functions	Description
1) Find	Search for an element with given key.
2) count	Gets the number of elements matching with given key.
3) lower_bound	Returns an iterator to lower bound.
4) upper_bound	Returns an iterator to upper bound.
5) equal_range	Returns the range of elements matches with given key.

❖ Allocator:

Functions	Description
1) get_allocator	Returns an allocator object that is used to construct the set.
2) Non-Member	Overloaded Functions
3) Functions	Description
4) operator==	Checks whether the two sets are equal or not.
5) operator!=	Checks whether the two sets are equal or not.
6) operator<	Checks whether the first set is less than other or not.
7) operator<=	Checks whether the first set is less than or equal to other or not.
8) operator>	Checks whether the first set is greater than other or not.
9) operator>=	Checks whether the first set is greater than equal to other or not.
10) swap()	Exchanges the element of two sets.

C++ STACK

- In computer science we go for working on a large variety of programs. Each of them has their own domain and utility. Based on the purpose and environment of the program creation, we have a large number of data structures available to choose from. One of them is 'stack'. Before discussing about this data type let us take a look at its syntax.

Syntax

template<class T, class Container = deque<T> > class stack;

- This data structure works on the LIFO technique, where LIFO stands for Last In First Out. The element which was first inserted will be extracted at the end and so on. There is an element called as 'top' which is the element at the upper most position. All the insertion and deletion operations are made at the top element itself in the stack.
- Stacks in the application areas are implied as the container adaptors.
- The containers should have a support for the following list of operations:
    1. empty
    2. size
    3. back
    4. push_back
    5. pop_back

❖ Template Parameters:

- **T:** The argument specifies the type of the element which the container adaptor will be holding.
- **Container:** The argument specifies an internal object of container where the elements of the stack are hold.

❖ Member Types:

- Given below is a list of the stack member types with a short description of the same.

Member Types	Description
1. value_type	Element type is specified.
2. container_type	Underlying container type is specified.
3. size_type	It specifies the size range of the elements.

❖ Functions:

- With the help of functions, an object or variable can be played with in the field of programming. Stacks provide a large number of functions that can be used or embedded in the programs. A list of the same is given below:

Function **Description**

1. **(constructor)**→The function is used for the construction of a stack container.
2. **Empty**→ it is used to test for emptiness of a stack. If the stack is empty function returns true else false.
3. **Size**→it is returns size of stack container, which is a measure of number of elements stored in stack.
4. **top**→it is used to access the top element of the stack. The element plays a very important role as all the insertion and deletion operations are performed at the top element.
5. **Push**→ The function is used for the insertion of a new element at the top of the stack.
6. **Pop**→ The function is used for deletion of element, the element in stack is deleted from top.

7. **Emplace→** function is used for insertion of new elements in stack above current top element.
8. **Swap→** The function is used for interchanging the contents of two containers in reference.
9. **relational operators→** The non member function specifies the relational operators that are needed for the stacks.
10. **uses allocator<stack>→** As the name suggests the non member function uses the allocator for the stacks.

Example: A simple program to show the use of basic stack functions.

```
#include <iostream>
#include <stack>
using namespace std;
void newstack(stack <int> ss)
{
   stack <int> sg = ss;
   while (!sg.empty())
   {
      cout << '\t' << sg.top();
      sg.pop();
   }
   cout << '\n';
}
int main ()
{
   stack <int> newst;
   newst.push(55);
   newst.push(44);
   newst.push(33);
   newst.push(22);
   newst.push(11);
   cout << "The stack newst is : ";
   newstack(newst);
   cout << "\n newst.size() : " << newst.size();
   cout << "\n newst.top() : " << newst.top();
   cout << "\n newst.pop() : ";
   newst.pop();
   newstack(newst);
   return 0;
}
```

Output:

```
The stack newst is :    11      22      33      44      55
 newst.size() : 5
 newst.top() : 11
 newst.pop() :  22      33      44      55
```

C++ QUEUE

- In computer science we go for working on a large variety of programs. Each of them has their own domain and utility. Based on the purpose and environment of the program creation, we have a large number of data structures available to choose from. One of them is 'queues. Before discussing about this data type let us take a look at its syntax.

Syntax

template<class T, class Container = deque<T> > class queue;

- This data structure works on the FIFO technique, where FIFO stands for First In First Out. The element which was first inserted will be extracted at the first and so on. There is an element called as 'front' which is the element at the front most position or say the first position, also there is an element called as 'rear' which is the element at the last position. In normal queues insertion of elements take at the rear end and the deletion is done from the front.
- Queues in the application areas are implied as the container adaptors.
- The containers should have a support for the following list of operations:
  1. empty
  2. size
  3. push_back
  4. pop_front
  5. front
  6. back

❖ Template Parameters:

- **T:** The argument specifies the type of the element which the container adaptor will be holding.
- **Container:** The argument specifies an internal object of container where the elements of the queues are held.

❖ Member Types:

- Given below is a list of the queue member types with a short description of the same.

Member Types	Description
1. value_type	Element type is specified.
2. container_type	Underlying container type is specified.
3. size_type	It specifies the size range of the elements.
4. reference	It is a reference type of a container.
5. const_reference	It is a reference type of a constant container.

❖ Functions:

- With the help of functions, an object or variable can be played with in the field of programming. Queues provide a large number of functions that can be used or embedded in the programs. A list of the same is given below:

Function	Description
1. (constructor)→	The function is used for the construction of a queue container.

2. **Empty→** The function is used to test for the emptiness of a queue. If the queue is empty the function returns true else false.
3. **Size→** The function returns the size of the queue container, which is a measure of the number of elements stored in the queue.
4. **Front→** The function is used to access the front element of the queue. The element plays a very important role as all the deletion operations are performed at the front element.
5. **Back→** The function is used to access the rear element of the queue. The element plays a very important role as all the insertion operations are performed at the rear element.
6. **Push→** The function is used for the insertion of a new element at the rear end of the queue.
7. **Pop→** The function is used for the deletion of element; the element in the queue is deleted from the front end.
8. **Emplace→** The function is used for insertion of new elements in the queue above the current rear element.
9. **Swap→** The function is used for interchanging the contents of two containers in reference.
10. **relational operators→** The non member function specifies the relational operators that are needed for the queues.
11. **uses allocator<queue>→** As the name suggests the non member function uses the allocator for the queues.

Example: A simple program to show the use of basic queue functions.

```
#include <iostream>
#include <queue>
using namespace std;
void showsg(queue <int> sg)
{
   queue <int> ss = sg;
   while (!ss.empty())
   {
      cout << '\t' << ss.front();
      ss.pop();
   }
   cout << '\n';
}
int main()
{
   queue <int> fquiz;
   fquiz.push(10);
   fquiz.push(20);
   fquiz.push(30);
   cout << "The queue fquiz is : ";
   showsg(fquiz);
```

```
    cout << "\nfquiz.size() : " << fquiz.size();
    cout << "\nfquiz.front() : " << fquiz.front();
    cout << "\nfquiz.back() : " << fquiz.back();
    cout << "\nfquiz.pop() : ";
    fquiz.pop();
    showsg(fquiz);
    return 0;
}
```

Output:

```
The queue fquiz is :    10      20      30
fquiz.size() : 3
fquiz.front() : 10
fquiz.back() : 30
fquiz.pop() :    20      30
```

PRIORITY QUEUE IN C++

- The priority queue in C++ is a derived container in STL that considers only the highest priority element. The queue follows the FIFO policy while priority queue pops the elements based on the priority, i.e., the highest priority element is popped first.
- It is similar to the ordinary queue in certain aspects but differs in the following ways:
- In a priority queue, every element in the queue is associated with some priority, but priority does not exist in a queue data structure.
- The element with the highest priority in a priority queue will be removed first while queue follows the FIFO(First-In-First-Out) policy means the element which is inserted first will be deleted first.
- If more than one element exists with the same priority, then the order of the element in a queue will be taken into consideration.
- Note: The priority queue is the extended version of a normal queue except that the element with the highest priority will be removed first from the priority queue.

❖Syntax of Priority Queue:

```
priority_queue<int> variable_name;
```

❖ Member Function of Priority Queue:

	Function	Description
1.	**push()**	It inserts a new element in a priority queue.
2.	**pop()**	It removes the top element from the queue, which has the highest priority.
3.	**top()**	This function is used to address the topmost element of a priority queue.
4.	**size()**	It determines the size of a priority queue.
5.	**empty()**	It verifies whether the queue is empty or not. Based on the verification, it returns the status.

6. **swap()** It swaps the elements of a priority queue with another queue having the same type and size.
7. **emplace()** It inserts a new element at the top of the priority queue.

Let's create a simple program of priority queue.

```
#include <iostream>
#include<queue>
using namespace std;
int main()
{
 priority_queue<int> p;  // variable declaration.
 p.push(10); // inserting 10 in a queue, top=10
 p.push(30); // inserting 30 in a queue, top=30
 p.push(20); // inserting 20 in a queue, top=20
 cout<<"Number of elements available in 'p' :"<<p.size()<<endl;
 while(!p.empty())
 {
    std::cout << p.top() << std::endl;
    p.pop();
 }
 return 0;
}
```

- In the above code, we have created a priority queue in which we insert three elements, i.e., 10, 30, 20. After inserting the elements, we display all the elements of a priority queue by using a while loop.

Output

```
Number of elements available in 'p' :3
30
20
10 zzzzz/
```

Let's see another example of a priority queue.

```
#include <iostream>
#include<queue>
using namespace std;
int main()
{
  priority_queue<int> p;  // priority queue declaration
  priority_queue<int> q;  // priority queue declaration
  p.push(1); // inserting element '1' in p.
  p.push(2); // inserting element '2' in p.
  p.push(3); // inserting element '3' in p.
  p.push(4); // inserting element '4' in p.
  q.push(5); // inserting element '5' in q.
  q.push(6); // inserting element '6' in q.
```

```
    q.push(7); // inserting element '7' in q.
    q.push(8); // inserting element '8' in q.
    p.swap(q);
    std::cout << "Elements of p are : " << std::endl;
    while(!p.empty())
    {
      std::cout << p.top() << std::endl;
       p.pop();
    }
    std::cout << "Elements of q are :" << std::endl;
     while(!q.empty())
    {
      std::cout << q.top() << std::endl;
       q.pop();
    }
     return 0;
  }
```

- In the above code, we have declared two priority queues, i.e., p and q. We inserted four elements in 'p' priority queue and four in 'q' priority queue. After inserting the elements, we swap the elements of 'p' queue with 'q' queue by using a swap() function.

Output

```
Elements of p are :
8
7
6
5
Elements of q are :
4
3
2
1
```

Important programming question and answer in C++

1. Fibonacci Series
2. Prime Number
3. Palindrome Number
4. Factorial
5. Armstrong Number
6. Sum of digits
7. Reverse Number
8. Swap Number
9. Matrix Multiplication
10. Decimal to Binary
11. Number in Characters
12. Alphabet Triangle
13. Number Triangle
14. Fibonacci Triangle
15. Char array to string in C++
16. Calculator Program in C++
17. Program to convert infix to postfix expression in C++ using the Stack Data Structure
18. C++ program to merge two unsorted arrays
19. C++ coin change program
20. C++ program to add two complex numbers using class
21. C++ program to find the GCD of two numbers
22. C++ program to find greatest of four numbers
23. Delete Operator in C++
24. How to concatenate two strings in C++

Solution:

1. Fibonacci Series:

Program:

```
#include <iostream>
int main() {
   int n, first = 0, second = 1, next;
   std::cout << "Enter the number of terms: ";
   std::cin >> n;
   std::cout << "Fibonacci Series: ";
   for (int i = 0; i < n; ++i) {
      std::cout << first << " ";
      next = first + second;
      first = second;
      second = next;
   }
```

```
    std::cout << std::endl;

    return 0;
}
```

Output (for n = 10):

```
Enter the number of terms: 10
Fibonacci Series: 0 1 1 2 3 5 8 13 21 34
```

2. Prime Number

Program:

```
#include <iostream>
bool isPrime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i * i <= n; ++i) {
        if (n % i == 0) return false;
    }
    return true;
}
int main() {
    int num;
    std::cout << "Enter a number: ";
    std::cin >> num;
    if (isPrime(num))
        std::cout << num << " is a prime number." << std::endl;
    else
        std::cout << num << " is not a prime number." << std::endl;
    return 0;
}
```

Output:

```
Enter a number: 13
13 is a prime number.
```

3. Palindrome Number

Program:

```
#include <iostream>
bool isPalindrome(int n) {
    int original = n, reversed = 0, remainder;
    while (n != 0) {
        remainder = n % 10;
        reversed = reversed * 10 + remainder;
        n /= 10;
    }
    return original == reversed;
}
int main() {
```

```
    int num;
    std::cout << "Enter a number: ";
    std::cin >> num;

    if (isPalindrome(num))
        std::cout << num << " is a palindrome number." << std::endl;
    else
        std::cout << num << " is not a palindrome number." << std::endl;
    return 0;
}
```

Output:

```
Enter a number: 12321
12321 is a palindrome number.
```

4. Factorial:

Program:

```
#include <iostream>
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
int main() {
    int n;
    std::cout << "Enter a number: ";
    std::cin >> n;
    std::cout << "Factorial of " << n << " is " << factorial(n) << std::endl;
    return 0;
}
```

Output:

```
Enter a number: 5
Factorial of 5 is 120
```

5. Armstrong Number

Program:

```
#include <iostream>
#include <cmath>
bool isArmstrong(int num) {
    int originalNum, remainder, n = 0, result = 0;
    originalNum = num;
    while (originalNum != 0) {
        originalNum /= 10;
        ++n;
    }
```

```
    originalNum = num;
    while (originalNum != 0) {
        remainder = originalNum % 10;
        result += pow(remainder, n);
        originalNum /= 10;
    }
    return result == num;
}
int main() {
    int num;
    std::cout << "Enter a number: ";
    std::cin >> num;
    if (isArmstrong(num))
        std::cout << num << " is an Armstrong number." << std::endl;
    else
        std::cout << num << " is not an Armstrong number." << std::endl;
    return 0;
}
```

Output:

```
Enter a number: 153
153 is an Armstrong number.
```

6. Sum of digits:

Program:

```
#include <iostream>
int sumOfDigits(int n) {
    int sum = 0;
    while (n != 0) {
        sum += n % 10;
        n /= 10;
    }
    return sum;
}
int main() {
    int num;
    std::cout << "Enter a number: ";
    std::cin >> num;
    std::cout << "Sum of digits of " << num << " is " << sumOfDigits(num) << std::endl;
    return 0;
}
```

Output:

```
Enter a number: 123
Sum of digits of 123 is 6
```

7. Reverse Number:

Program:

```
#include <iostream>
int reverseNumber(int n) {
   int reversed = 0;
   while (n != 0) {
      reversed
      reversed = reversed * 10 + n % 10;
      n /= 10;
   }
   return reversed;
}
int main() {
   int num;
   std::cout << "Enter a number: ";
   std::cin >> num;
   std::cout << "Reverse of " << num << " is " << reverseNumber(num) << std::endl;
   return 0;
}
```

Output:

```
Enter a number: 12345
Reverse of 12345 is 54321
```

8. Swap Number:

Program:

```
#include <iostream>
void swap(int &a, int &b) {
   int temp = a;
   a = b;
   b = temp;
}
int main() {
   int a = 5, b = 10;
   std::cout << "Before swapping: a = " << a << ", b = " << b << std::endl;
   swap(a, b);
   std::cout << "After swapping: a = " << a << ", b = " << b << std::endl;
   return 0;
}
```

Output:

```
Before swapping: a = 5, b = 10
After swapping: a = 10, b = 5
```

9. Matrix Multiplication

Program:

```
#include <iostream>
#include <vector>
```

```
std::vector<std::vector<int>> matrixMultiply(const std::vector<std::vector<int>>& mat1,
const std::vector<std::vector<int>>& mat2) {
   int row1 = mat1.size(), col1 = mat1[0].size();
   int row2 = mat2.size(), col2 = mat2[0].size();
   if (col1 != row2) {
      std::cerr << "Matrix multiplication not possible!" << std::endl;
      return {};
   }
   std::vector<std::vector<int>> result(row1, std::vector<int>(col2, 0));
   for (int i = 0; i < row1; ++i) {
      for (int j = 0; j < col2; ++j) {
         for (int k = 0; k < col1; ++k) {
            result[i][j] += mat1[i][k] * mat2[k][j];
         }}}
   return result;
}
void printMatrix(const std::vector<std::vector<int>>& mat) {
   for (const auto& row : mat) {
      for (int elem : row) {
         std::cout << elem << " ";
      }
      std::cout << std::endl;
   }}
int main() {
   std::vector<std::vector<int>> mat1 = {{1, 2, 3}, {4, 5, 6}};
   std::vector<std::vector<int>> mat2 = {{7, 8}, {9, 10}, {11, 12}};
   std::cout << "Matrix 1:" << std::endl;
   printMatrix(mat1);
   std::cout << "Matrix 2:" << std::endl;
   printMatrix(mat2);
   std::vector<std::vector<int>> result = matrixMultiply(mat1, mat2);
   std::cout << "Resultant Matrix:" << std::endl;
   printMatrix(result);
   return 0;
}
```

Output:

```
Matrix 1:
1 2 3
4 5 6
Matrix 2:
7 8
9 10
11 12
Resultant Matrix:
58 64
139 154
```

10. Decimal to Binary

Program:

```
#include <iostream>
#include <stack>
std::string decimalToBinary(int n) {
   std::stack<int> binaryDigits;

   while (n > 0) {
      binaryDigits.push(n % 2);
      n /= 2;
   }
   std::string binaryString;
   while (!binaryDigits.empty()) {
      binaryString += std::to_string(binaryDigits.top());
      binaryDigits.pop();
   }
   return binaryString;
}
int main() {
   int decimalNumber;
   std::cout << "Enter a decimal number: ";
   std::cin >> decimalNumber;
   std::string binaryNumber = decimalToBinary(decimalNumber);
   std::cout << "Binary representation: " << binaryNumber << std::endl;
   return 0;
}
```

Output (for input 10):

```
Enter a decimal number: 10
Binary representation: 1010
```

11. Number in Characters:

Program:

```
#include <iostream>
#include <string>
int main() {
   int number;
   std::cout << "Enter a number: ";
   std::cin >> number;
   std::string numberString = std::to_string(number);
   std::cout << "Number in characters: " << numberString << std::endl;
   return 0;
}
```

Output (for input 123):

```
Enter a number: 123
Number in characters: 123
```

12. Alphabet Triangle

Program:

```
#include <iostream>
int main() {
   int rows;
   std::cout << "Enter the number of rows: ";
   std::cin >> rows;
   for (int i = 0; i < rows; ++i) {
      char alphabet = 'A';
      for (int j = 0; j <= i; ++j) {
         std::cout << alphabet++ << " ";
      }
      std::cout << std::endl;
   }
   return 0;
}
```

Output (for input 5):

```
A
A B
A B C
A B C D
A B C D E
```

13. Number Triangle

Program:

```
#include <iostream>
int main() {
   int rows;
   std::cout << "Enter the number of rows: ";
   std::cin >> rows;
   for (int i = 1; i <= rows; ++i) {
      for (int j = 1; j <= i; ++j) {
         std::cout << j << " ";
      }
      std::cout << std::endl;
   }
   return 0;
}
```

Output (for input 5):

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

14. Fibonacci Triangle

Program:

```
#include <iostream>
int main() {
   int rows;
```

```
    std::cout << "Enter the number of rows: ";
    std::cin >> rows;
    int a = 0, b = 1, c;
    for (int i = 1; i <= rows; ++i) {
        for (int j = 1; j <= i; ++j) {
            std::cout << a << " ";
            c = a + b;
            a = b;
            b = c;
        }
        std::cout << std::endl;
    }
    return 0;
}
```

Output (for input 5):

```
0
1 1
2 3 5
8 13 21 34
55 89 144 233 377
```

15. Char array to string in C++

Program:

```
#include <iostream>
#include <string>
int main() {
    char charArray[] = {'H', 'e', 'l', 'l', 'o', '\0'};
    std::string str(charArray);
    std::cout << "String from char array: " << str << std::endl;
    return 0;
}
```

Output:

```
String from char array: Hello
```

16. Calculator Program in C++

Program:

```
#include <iostream>
int main() {
    char op;
    double num1, num2;
    std::cout << "Enter operator (+, -, *, /): ";
    std::cin >> op;
    std::cout << "Enter two operands: ";
    std::cin >> num1 >> num2;
    switch(op) {
        case '+':
            std::cout << "Sum = " << num1 + num2;
            break;
```

```
        case '-':
            std::cout << "Difference = " << num1 - num2;
            break;
        case '*':
            std::cout << "Product = " << num1 * num2;
            break;
        case '/':
            if(num2 != 0)
                std::cout << "Quotient = " << num1 / num2;
            else
                std::cout << "Error! Division by zero!";
            break;
        default:
            std::cout << "Invalid operator!";
    }
    std::cout << std::endl;
    return 0;
}
```

Output (example input: '+', 5, 3):

```
Enter operator (+, -, *, /): +
Enter two operands: 5 3
Sum = 8
```

17. Program to convert infix to postfix expression in C++ using the Stack Data Structure:

Program:

```
#include <iostream>
#include <stack>
#include <string>
int precedence(char op) {
    if(op == '+' || op == '-') return 1;
    if(op == '*' || op == '/') return 2;
    return 0;
}
std::string infixToPostfix(const std::string& infix) {
    std::string postfix = "";
    std::stack<char> operators;
    for(char ch : infix) {
        if(isalnum(ch))
            postfix += ch;
        else if(ch == '(')
            operators.push(ch);
        else if(ch == ')') {
            while(!operators.empty() && operators.top() != '(') {
                postfix += operators.top();
                operators.pop();
            }
            operators.pop();
        } else {
```

```
            while(!operators.empty() && precedence(ch) <= precedence(operators.top())) {
               postfix += operators.top();
               operators.pop();
            }
            operators.push(ch);
         }
      }
      while(!operators.empty()) {
         postfix += operators.top();
         operators.pop();
      }
      return postfix;
   }

   int main() {
      std::string infix;
      std::cout << "Enter infix expression: ";
      std::cin >> infix;
      std::cout << "Postfix expression: " << infixToPostfix(infix) << std::endl;
      return 0;
   }
```

Output (example input: "a+b*c"):

```
Enter infix expression: a+b*c
Postfix expression: abc*+
```

18. C++ program to merge two unsorted arrays:

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
int main() {
   std::vector<int> arr1 = {5, 2, 8, 1};
   std::vector<int> arr2 = {7, 3, 6, 4};
   arr1.insert(arr1.end(), arr2.begin(), arr2.end());
   std::cout << "Merged array: ";
   for(int num : arr1) {
      std::cout << num << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

Output:

```
Merged array: 5 2 8 1 7 3 6 4
```

19. C++ coin change program

Program:

```
#include <iostream>
#include <vector>
```

```
int countWays(int amount, const std::vector<int>& coins) {
    std::vector<int> dp(amount + 1);
    dp[0] = 1;
    for(int coin : coins) {
        for(int i = coin; i <= amount; ++i) {
            dp[i] += dp[i - coin];
        }
    }
    return dp[amount];
}
int main() {
    int amount;
    std::cout << "Enter amount: ";
    std::cin >> amount;
    std::vector<int> coins = {1, 2, 5}; // denominations of coins
    std::cout << "Number of ways to make change: " << countWays(amount, coins) <<
std::endl;
    return 0;
}
```

Output (example input: 5):

```
Enter amount: 5
Number of ways to make change: 4
```

20. C++ program to add two complex numbers using class

Program:

```
#include <iostream>
class Complex {
private:
    float real;
    float imag;
public:
    Complex(float r = 0.0, float i = 0.0) : real(r), imag(i) {}
    Complex operator+(const Complex& c) {
        return Complex(real + c.real, imag + c.imag);
    }
    void display() {
        std::cout << "Sum = " << real << " + " << imag << "i" << std::endl;
    }
};
int main() {
    Complex c1(3.5, 2.7), c2(8.1, 4.2), result;
    result = c1 + c2;
    result.display();
    return 0;
}
```

Output:

```
Sum = 11.6 + 6.9i
```

21. C++ program to find the GCD of two numbers

Program:

```
#include <iostream>
int gcd(int a, int b) {
   if (b == 0)
      return a;
   return gcd(b, a % b);
}
int main() {
   int num1, num2;
   std::cout << "Enter two numbers: ";
   std::cin >> num1 >> num2;
   std::cout << "GCD of " << num1 << " and " << num2 << " is " << gcd(num1, num2) <<
std::endl;
   return 0; }
```

Output (example input: 12, 18):

```
Enter two numbers: 12 18
GCD of 12 and 18 is 6
```

22. C++ program to find greatest of four numbers

Program:

```
#include <iostream>
int main() {
   int num1, num2, num3, num4;
   std::cout << "Enter four numbers: ";
   std::cin >> num1 >> num2 >> num3 >> num4;
   int max = num1;
   if (num2 > max)
      max = num2;
   if (num3 > max)
      max = num3;
   if (num4 > max)
      max = num4;
   std::cout << "Greatest number: " << max << std::endl;
   return 0;
}
```

Output (example input: 5, 8, 2, 7):

```
Enter four numbers: 5 8 2 7
Greatest number: 8
```

23. Delete Operator in C++

Program:

```
#include <iostream>
int main() {
   int* ptr = new int(5); // dynamically allocate memory for an integer
   std::cout << "Value before deletion: " << *ptr << std::endl;
   delete ptr; // delete the dynamically allocated memory
   // Accessing the memory after deletion may lead to undefined behavior
```

```
    // std::cout << "Value after deletion: " << *ptr << std::endl;
    return 0;
}
```

Output:

```
Value before deletion: 5
```

24. How to concatenate two strings in C++

Program:

```
#include <iostream>
#include <string>
int main() {
    std::string str1 = "Hello, ";
    std::string str2 = "world!";
    // Using + operator to concatenate strings
    std::string result = str1 + str2;
    std::cout << "Concatenated string: " << result << std::endl;
    return 0;
}
```

Output:

```
Concatenated string: Hello, world!
```

Interview Question and answer

1. **What is C++?**
- C++ is a general-purpose programming language developed as an extension of the C programming language. It provides features such as classes and objects, inheritance, polymorphism, encapsulation, and more.

2. **What are the key features of C++?**
   - Object-oriented
   - Platform-independent
   - Strongly typed
   - Compiled
   - Rich library support
   - Pointers
   - Dynamic memory allocation
   - Exception handling
   - Templates
   - Standard Template Library (STL)

3. **What is the difference between C and C++?**
   - C is a procedural programming language, while C++ is an object-oriented programming language.
   - C does not support classes and objects, while C++ does.
   - C supports only procedural programming paradigm, whereas C++ supports both procedural and object-oriented programming paradigms.

4. **What is a class in C++?**
   - A class in C++ is a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of the class will have.

5. **What is an object in C++?**
   - An object is an instance of a class. It is created using the class blueprint and represents a specific entity in a program.

6. **What is inheritance in C++?**
   - Inheritance is the process by which one class can acquire the properties and behaviors of another class. It promotes code reusability and allows the creation of hierarchical relationships between classes.

7. **What are access specifiers in C++?**
   - Access specifiers (public, private, protected) control the visibility and accessibility of class members (attributes and methods) in C++.
   - Public members are accessible from outside the class, private members are accessible only within the class, and protected members are accessible within the class and its derived classes.

8. **What is polymorphism in C++?**
   - Polymorphism refers to the ability of objects to take multiple forms. In C++, polymorphism is achieved through function overloading and function overriding.

9. What is function overloading?

- Function overloading is the ability to define multiple functions with the same name but different parameter lists. The compiler selects the appropriate function to call based on the arguments provided.

10. What is function overriding?

- Function overriding occurs when a derived class provides a specific implementation of a method that is already defined in its base class. It is used to achieve runtime polymorphism.

11. What is encapsulation in C++?

- Encapsulation is the bundling of data (attributes) and methods (behaviors) that operate on the data into a single unit called a class. It helps in hiding the internal details of a class and protects the data from unauthorized access.

12. What is a constructor in C++?

- A constructor is a special member function of a class that is automatically called when an object of the class is created. It is used to initialize the object's state.

13. What is a destructor in C++?

- A destructor is a special member function of a class that is automatically called when an object goes out of scope or is explicitly deleted. It is used to release resources acquired by the object.

14. What is the difference between a constructor and a destructor?

- A constructor is called when an object is created, while a destructor is called when an object is destroyed.
- Constructors initialize the object's state, while destructors release resources acquired by the object.

15. What is a copy constructor?

- A copy constructor is a special constructor that creates a new object by copying the state of an existing object of the same class. It is invoked when an object is passed by value, returned by value, or initialized with another object of the same class.

16. What is the difference between shallow copy and deep copy?

- Shallow copy copies the memory address of the original object's data members into the new object, resulting in two objects sharing the same memory. Deep copy creates a new copy of the original object's data members in the new object, resulting in two independent objects.

17. What is a friend function in C++?

- A friend function is a function that is not a member of a class but has access to the private and protected members of the class. It is declared using the friend keyword inside the class.

18. What is the this pointer in C++?

- The this pointer is a special pointer that holds the address of the current object. It is used to access the members of the current object within its member functions.

19. What is a static member in C++?

- A static member is a member (variable or function) of a class that belongs to the class itself rather than individual objects of the class. It is shared among all instances of the class.

20. What is a static function in C++?

- A static function is a member function of a class that can be called without creating an object of the class. It operates on the class's static data members and does not have access to non-static members.

21. What is a namespace in C++?

- A namespace is a declarative region that provides a scope for the identifiers (variables, functions, classes) declared within it. It helps in organizing the code and avoiding naming conflicts.

22. What is the Standard Template Library (STL) in C++?

- The Standard Template Library (STL) is a collection of generic algorithms and data structures provided by C++ to facilitate programming tasks. It includes containers (like vectors, lists, maps), iterators, algorithms, and function objects.

23. What are iterators in C++?

- Iterators are objects that allow traversing the elements of a container (like arrays, vectors, lists) in a sequential manner. They provide a uniform way to access the elements of a container, regardless of its underlying implementation.

24.What are the types of inheritance supported in C++?

- C++ supports the following types of inheritance:
- Single inheritance: Derived class inherits from only one base class.
- Multiple inheritance: Derived class inherits from multiple base classes.
- Multilevel inheritance: Derived class inherits from a base class, and another class inherits from this derived class.
- Hierarchical inheritance: Multiple derived classes inherit from a single base class.
- Hybrid inheritance: Combination of multiple and multilevel inheritance.

25.What is a virtual function in C++?

- A virtual function is a member function of a class that is declared with the virtual keyword and can be overridden in derived classes. It enables dynamic binding or late binding, allowing the correct function to be called at runtime based on the object's actual type.
- What is dynamic binding in C++?
- Dynamic binding, also known as late binding, is the process of determining which function to call at runtime instead of compile time. It is achieved through virtual functions and polymorphism.

26. What is a pure virtual function?

- A pure virtual function is a virtual function that is declared in a base class but has no implementation. It is marked as pure virtual by appending "= 0" to its declaration. Classes containing pure virtual functions are called abstract classes and cannot be instantiated.

27.What is an abstract class in C++?

- An abstract class is a class that contains at least one pure virtual function. It cannot be instantiated and serves as a base class for derived classes to provide common interfaces and behaviors.

28.What is a virtual destructor?

- A virtual destructor is a destructor that is declared as virtual in the base class. It ensures that the appropriate destructor is called when an object is deleted through a pointer to a base class that points to a derived class object. It prevents memory leaks and ensures proper cleanup of resources in derived classes.

29.What are access specifiers in a class?

- Access specifiers (public, private, protected) control the accessibility of class members.
- Public members are accessible from outside the class.
- Private members are accessible only within the class.
- Protected members are accessible within the class and its derived classes.

30.What is a friend class in C++?

- A friend class is a class that is granted access to the private and protected members of another class. It is declared using the friend keyword in the class declaration.

31.What is the difference between reference and pointer?

- A reference is an alias to an existing variable and must be initialized when declared. It cannot be null and cannot be reassigned to refer to another object.
- A pointer is a variable that stores the memory address of another variable. It can be null and can be reassigned to point to different objects.

32.What is the new operator in C++?

- The new operator is used to dynamically allocate memory for objects on the heap. It returns a pointer to the allocated memory. For example: int* ptr = new int;

33.What is the delete operator in C++?

- The delete operator is used to deallocate memory that was dynamically allocated using the new operator. It releases the memory occupied by the object and returns it to the system's free memory pool. For example: delete ptr;

34. What is a smart pointer in C++?

- A smart pointer is a class template that provides automatic memory management for dynamically allocated objects. It automatically deallocates memory when the pointer goes out of scope, preventing memory leaks. Examples include std::unique_ptr, std::shared_ptr, and std::weak_ptr.

35. What are the differences between std::unique_ptr and std::shared_ptr?

- std::unique_ptr is a smart pointer that owns a dynamically allocated object exclusively. It cannot be copied or shared.
- std::shared_ptr is a smart pointer that allows multiple pointers to refer to the same dynamically allocated object. It uses reference counting to manage the object's lifetime.

36. What is the Standard Template Library (STL) in C++?

- The Standard Template Library (STL) is a collection of generic algorithms and data structures provided by C++ to facilitate programming tasks. It includes containers (like vectors, lists, maps), iterators, algorithms, and function objects.

37. What are containers in the STL?

- Containers are data structures provided by the STL to store and manipulate collections of objects. They include sequences (like vectors, lists, arrays), associative containers (like maps, sets), and container adapters (like stacks, queues).

38.What is an iterator in the STL?

- An iterator is an object that allows traversing the elements of a container (like arrays, vectors, lists) in a sequential manner. It provides a uniform way to access the elements of a container, regardless of its underlying implementation.

39. What are algorithms in the STL?

- Algorithms are functions provided by the STL to perform various operations on containers. They include sorting, searching, modifying, and operating on elements of containers.

40. What is a lambda function in C++?

- A lambda function is an anonymous function defined using the lambda expression syntax. It allows defining a small function inline without the need for a separate function declaration. Lambda functions are often used as arguments to higher-order functions like std::sort and std::for_each.

41. What is the auto keyword in C++?

- The auto keyword is used to declare variables with automatic type deduction. The compiler deduces the type of the variable based on the initializer expression.

42. What is RAII (Resource Acquisition Is Initialization)?

- RAII is a programming idiom in C++ where resource management is tied to object lifetime. Resources are acquired during object initialization and released during object destruction, ensuring proper cleanup even in the presence of exceptions.

43.What are move semantics in C++?

- Move semantics is a feature introduced in C++11 that enables the efficient transfer of resources from one object to another. It is implemented using move constructors and move assignment operators to perform shallow copies instead of deep copies.

44. What is a rvalue reference in C++?

- An rvalue reference is a new type of reference introduced in C++11 that can bind to temporary objects (rvalues). It is denoted by && and allows move semantics to be implemented efficiently.

45. What are the differences between std::vector and std::array?

- std::vector is a dynamic array that can resize itself dynamically. It is part of the STL and provides dynamic memory allocation.
- std::array is a fixed-size array introduced in C++11 with a size known at compile time. It does not support resizing and provides a more efficient alternative to built-in arrays.

44. What is the const qualifier in C++?

- The const qualifier is used to declare variables that cannot be modified after initialization. It ensures that the variable

45. What is the difference between const and constexpr in C++?

- const is used to declare variables that are not modifiable after initialization. It does not necessarily result in a compile-time constant.
- constexpr is used to declare variables or functions that can be evaluated at compile time. It guarantees compile-time evaluation and can be applied to variables, functions, and constructors.

46. What is the purpose of the volatile keyword in C++?

- The volatile keyword is used to indicate to the compiler that a variable's value can be changed unexpectedly by external factors not known to the compiler, such as hardware interrupts or concurrently running threads. It prevents the compiler from performing optimizations that could result in incorrect behavior due to assumptions about the variable's value.

47. What is the override keyword in C++?

- The override keyword is used to explicitly specify that a member function in a derived class overrides a virtual function declared in a base class. It helps in ensuring that the intended overriding actually occurs and improves code clarity and maintainability.

48. What is the final keyword in C++?

- The final keyword is used to indicate that a class or virtual function cannot be derived from or overridden, respectively. When applied to a class, it prevents inheritance, and when applied to a virtual function, it prevents further overriding in derived classes.

49. What is the difference between std::sort and std::stable_sort in C++?

- std::sort is an algorithm in the STL that sorts the elements of a container in ascending order. It may not preserve the relative order of equal elements.
- std::stable_sort is an algorithm in the STL that also sorts the elements of a container in ascending order but preserves the relative order of equal elements. It is slower than std::sort but maintains stability.

50. What is the std::move function in C++?

- The std::move function is used to convert an lvalue (an object with a named identifier) into an rvalue (an object that can be moved). It allows efficient transfer of resources, such as ownership of dynamically allocated memory or unique pointers, from one object to another.

51. What are the advantages of using exceptions in C++?

- Exceptions provide a structured way to handle errors and exceptional conditions in a program.
- They separate error-handling logic from normal program flow, improving code readability and maintainability.
- They allow propagating errors across function and method boundaries without using error codes or return values.
- They support stack unwinding, ensuring that resources are properly cleaned up even in the presence of exceptions.

52. What is the difference between std::map and std::unordered_map in C++?

- std::map is an associative container in the STL that stores elements in a sorted order based on keys. It uses a red-black tree for internal storage and provides O(log n) time complexity for most operations.
- std::unordered_map is an associative container that stores elements in an unordered manner based on keys. It uses a hash table for internal storage and provides O(1) average time complexity for most operations but does not maintain any specific order of elements.

53. What is a template specialization in C++?

- Template specialization is the process of providing a specific implementation for a template for a particular set of template arguments. It allows customizing the behavior of templates for specific types or values.

54. What are the differences between class templates and function templates in C++?

- Class templates are used to define generic classes that can work with any data type. They are instantiated with specific data types to create concrete classes.
- Function templates are used to define generic functions that can operate on any data type. They are instantiated with specific data types to generate concrete function definitions.

55. What is the std::forward function in C++?

- The std::forward function is used to forward arguments exactly as received, preserving their value category (lvalue or rvalue). It is primarily used in perfect forwarding scenarios, where arguments need to be forwarded to another function without losing their value category.

56. What is the std::tie function in C++?

- The std::tie function is used to create a tuple of lvalue references from individual variables or tuple-like objects. It allows unpacking the elements of a tuple or a tuple-like object into individual variables or references.

57. What are the differences between std::thread and std::async in C++?

- std::thread is a class in the C++ Standard Library that represents a single thread of execution. It is used for creating and managing concurrent execution of tasks in a multithreaded environment.
- std::async is a function template in the C++ Standard Library that is used to launch asynchronous tasks and obtain their results. It provides a high-level interface for launching asynchronous operations and supports lazy evaluation.

58. What are move constructors and move assignment operators in C++?

- Move constructors and move assignment operators are special member functions in C++ that enable efficient transfer of resources, such as memory or ownership, from one object to another. They are used in move semantics to perform shallow copies instead of deep copies, improving performance and reducing unnecessary resource duplication.

59. What is the rule of three (or rule of five) in C++?

- The rule of three (or rule of five in C++11 and later) is a guideline in C++ that states that if a class requires a custom destructor, copy constructor, or copy assignment operator, it likely requires all three This ensures proper resource management and prevents issues like memory leaks and dangling pointers.

60. What are the differences between std::unique_lock and std::lock_guard in C++?

- std::unique_lock is a class in the C++ Standard Library that provides exclusive ownership of a mutex. It allows more flexibility, such as deferred locking and manual unlocking, but incurs slightly higher overhead.
- std::lock_guard is a class in the C++ Standard Library that provides exclusive ownership of a mutex for the duration of its scope. It is simpler and more lightweight than std::unique_lock but does not support deferred locking or manual unlocking.

What is RID Organization (RID संस्था क्या है)?

- **RID Organization** यानि **Research, Innovation and Discovery Organization एक** संस्था हैं जो TWF (TWKSAA WELFARE FOUNDATION) **NGO** द्वारा RUN किया जाता है | जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले **NEW (RID, PMS & TLR)** की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW (RID, PMS & TLR)** के माध्यम से किया जाये इसके लिए ही **इस RID Organization** की स्थपना 30.09.2023 किया गया है | जो TWF द्वारा संचालित किया जाता है |
- TWF (TWKSAA WELFARE FOUNDATION) NGO की स्थपना 26-10-2020 में बिहार की पावन धरती सासाराम में Er. RAJESH PRASAD एवं Er. SUNIL KUMAR द्वारा किया गया था जो की भारत सरकार द्वारा मान्यता प्राप्त संस्था हैं |
- Research, Innovation & Discovery में रूचि रखने वाले आप सभी विधार्थियों, शिक्षकों एवं बुधीजिवियो से मैं आवाहनं करता हूँ की आप सभी **इस RID संस्था** से जुड़ें एवं अपने बुध्दि, विवेक एवं प्रतिभा से दुनियां को कुछ नई **(RID, PMS & TLR)** की खोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें |

MISSION, VISSION & MOTIVE OF "RID ORGANIZATION"	
मिशन	हर एक ONE भारत के संग
विजन	TALENT WORLD KA SHRESHTM AB AAYEGA भारत में और भारत का TALENT भारत में
मकसद	NEW (RID, PMS, TLR)

MOTIVE OF RID ORGANIZATION NEW (RID, PMS, TLR)		
NEW (RID)		
R	I	D
Research	Innovation	Discovery
NEW (TLR)		
T	L	R
Technology, Theory, Technique	Law	Rule
NEW (PMS)		
P	M	S
Product, Project, Production	Machine	Service

RID रीड संस्था की मिशन, विजन एवं मकसद को सार्थक हमें बनाना हैं |
भारत के वर्चस्व को हर कोने में फैलना हैं |
कर के नया कार्य एक बदलाव समाज में लाना हैं |
रीड संस्था की कार्य-सिध्दांतों से ही, हमें अपनी पहचान बनाना हैं |

Er. Rajesh Prasad (B.E, M.E)
Founder:
TWF & RID Organization

C++ Programming Language के इस **E-Book** में अगर मिलती त्रुटी मिलती है तो कृपया हमें सूचित करें | **WhatsApp's No: 9202707903 or Email Id: ridorg.in@gmail.com**

|| धन्यवाद ||