



CODTECH IT SOLUTIONS PVT.LTD
IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana



C++ Programming Material



OUR PARTNERS & CERTIFICATIONS



**M MINISTRY OF
C CORPORATE
A AFFAIRS**
GOVERNMENT OF INDIA



INDEX FOR C++ PROGRAMMING

1. INTRODUCTION TO C++

- 1.1 OVERVIEW OF C++
- 1.2 FEATURES OF C++
- 1.3 HISTORY OF C++
- 1.4 APPLICATIONS OF C++
- 1.5 SETTING UP A C++ DEVELOPMENT ENVIRONMENT
- 1.6 FIRST C++ PROGRAM
- 1.7 COMPILED AND EXECUTION PROCESS

2. BASIC CONCEPTS

- 2.1 C++ SYNTAX AND STRUCTURE
- 2.2 VARIABLES AND DATA TYPES
- 2.3 CONSTANTS AND LITERALS
- 2.4 INPUT/OUTPUT IN C++ (CIN, COUT)
- 2.5 OPERATORS IN C++ (ARITHMETIC, LOGICAL, RELATIONAL, BITWISE)

3. CONTROL FLOW STATEMENTS

- 3.1 DECISION-MAKING STATEMENTS (IF, IF-ELSE, SWITCH)
- 3.2 LOOPING STATEMENTS (FOR, WHILE, DO-WHILE)
- 3.3 JUMP STATEMENTS (BREAK, CONTINUE, GOTO, RETURN)

4. FUNCTIONS IN C++

- 4.1 FUNCTION DECLARATION AND DEFINITION
- 4.2 FUNCTION PARAMETERS AND RETURN TYPES
- 4.3 CALL BY VALUE VS CALL BY REFERENCE
- 4.4 FUNCTION OVERLOADING
- 4.5 INLINE FUNCTIONS
- 4.6 RECURSIVE FUNCTIONS

5. ARRAYS AND STRINGS

- 5.1 INTRODUCTION TO ARRAYS**
- 5.2 ONE-DIMENSIONAL ARRAYS**
- 5.3 TWO-DIMENSIONAL ARRAYS**
- 5.4 MULTI-DIMENSIONAL ARRAYS**
- 5.5 STRING HANDLING IN C++ (STRING CLASS VS C-STYLE STRINGS)**
- 5.6 STRING MANIPULATION FUNCTIONS**

6. POINTERS IN C++

- 6.1 INTRODUCTION TO POINTERS**
- 6.2 POINTER ARITHMETIC**
- 6.3 POINTERS AND ARRAYS**
- 6.4 POINTERS TO FUNCTIONS**
- 6.5 POINTERS TO OBJECTS**

7. DYNAMIC MEMORY ALLOCATION

- 7.1 NEW AND DELETE OPERATORS**
- 7.2 DYNAMIC ARRAYS**
- 7.3 MEMORY LEAKS AND PREVENTION**

8. OBJECT-ORIENTED PROGRAMMING (OOP) IN C++

- 8.1 INTRODUCTION TO OOP**
- 8.2 CLASSES AND OBJECTS**
- 8.3 ACCESS SPECIFIERS (PUBLIC, PRIVATE, PROTECTED)**

9. INHERITANCE IN C++

9.1 TYPES OF INHERITANCE (SINGLE, MULTIPLE, MULTILEVEL, HIERARCHICAL, HYBRID)

9.2 CONSTRUCTOR AND DESTRUCTOR IN INHERITANCE

9.3 FUNCTION OVERRIDING

9.4 VIRTUAL BASE CLASS

9.5 ACCESS CONTROL IN INHERITANCE

10. POLYMORPHISM IN C++

10.1 COMPILE-TIME POLYMORPHISM (FUNCTION OVERLOADING, OPERATOR OVERLOADING)

10.2 RUNTIME POLYMORPHISM (VIRTUAL FUNCTIONS, ABSTRACT CLASSES, PURE VIRTUAL FUNCTIONS)

11. ENCAPSULATION AND ABSTRACTION

11.1 DATA HIDING AND ACCESS MODIFIERS

11.2 ABSTRACT CLASSES AND INTERFACES

12. FILE HANDLING IN C++

12.1 FILE STREAMS (IFSTREAM, OFSTREAM, FSTREAM)

12.2 READING AND WRITING TO FILES

12.3 FILE HANDLING MODES

12.4 FILE POINTERS AND MANIPULATORS

13. EXCEPTION HANDLING IN C++

13.1 INTRODUCTION TO EXCEPTION HANDLING

13.2 TRY, CATCH, AND THROW MECHANISM

13.3 STANDARD EXCEPTIONS (STD::EXCEPTION, STD::RUNTIME_ERROR, ETC.)

13.4 USER-DEFINED EXCEPTIONS



CODTECH IT SOLUTIONS PVT.LTD
IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

14. STANDARD TEMPLATE LIBRARY (STL)

14.1 INTRODUCTION TO STL

14.2 CONTAINERS (VECTOR, LIST, DEQUE, STACK, QUEUE, SET, MAP)

14.3 ITERATORS IN STL

14.4 ALGORITHMS IN STL

14.5 FUNCTION OBJECTS AND LAMBDA EXPRESSIONS

15. MULTITHREADING IN C++

15.1 INTRODUCTION TO MULTITHREADING

15.2 CREATING THREADS USING STD::THREAD

15.3 THREAD SYNCHRONIZATION (MUTEX, LOCKS)

15.4 CONDITION VARIABLES

16. TEMPLATES IN C++

16.1 FUNCTION TEMPLATES

16.2 CLASS TEMPLATES

16.3 VARIADIC TEMPLATES



CODTECH IT SOLUTIONS PVT.LTD

IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

INDEX FOR C++ PROGRAMMING

1. INTRODUCTION TO C++

C++ is a powerful, high-performance, general-purpose programming language developed by Bjarne Stroustrup in 1979 as an extension of the C language. It combines the efficiency of procedural programming with the flexibility of object-oriented programming (OOP), making it widely used for system software, game development, real-time applications, and large-scale software engineering. One of the key strengths of C++ is its support for multiple programming paradigms, including procedural, object-oriented, and generic programming using templates. It provides features like encapsulation, polymorphism, inheritance, and abstraction, which help in writing modular and reusable code. Additionally, C++ supports dynamic memory management through pointers and the Standard Template Library (STL), which offers built-in data structures and algorithms.

C++ is also widely used in performance-critical applications such as operating systems, embedded systems, and financial applications due to its low-level memory access and high execution speed. Over the years, it has evolved significantly with newer standards like C++11, C++14, C++17, and C++20, introducing features like smart pointers, lambda expressions, concurrency support, and concepts.

Overall, C++ remains one of the most versatile and widely used programming languages, providing a perfect balance between control, efficiency, and flexibility for software development.



8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

1.1 OVERVIEW OF C++

C++ is a powerful, high-performance, general-purpose programming language that extends the C language with object-oriented programming (OOP) features. Developed by Bjarne Stroustrup in 1979, C++ provides a balance between low-level system programming and high-level application development, making it widely used in fields such as software engineering, game development, operating systems, and real-time systems.

C++ supports multiple programming paradigms, including procedural, object-oriented, and generic programming. It allows code reuse through classes and objects, enhances code flexibility with polymorphism and inheritance, and provides encapsulation for data protection. The Standard Template Library (STL) simplifies complex operations with built-in data structures and algorithms like vectors, lists, maps, and sets.

Another key feature of C++ is its dynamic memory management, which enables efficient resource handling using pointers, new/delete operators, and smart pointers introduced in C++11. With each new standard (C++11, C++14, C++17, C++20), modern C++ continues to evolve, introducing features like lambda expressions, concurrency support, and concepts for better performance and usability.

Due to its speed, efficiency, and flexibility, C++ remains a preferred language for high-performance applications, game engines, embedded systems, and financial systems, solidifying its place in modern software development.



1.2 FEATURES OF C++

C++ is a powerful and versatile programming language that offers a combination of high performance, flexibility, and rich feature set, making it suitable for various applications. Some of its key features include:

1. Object-Oriented Programming (OOP) – C++ supports classes, objects, encapsulation, inheritance, polymorphism, and abstraction, enabling modular and reusable code.
2. Multi-Paradigm Language – It supports procedural, object-oriented, and generic programming, allowing developers to choose the best approach for their projects.
3. High Performance – C++ is a compiled language with direct access to system resources, making it faster than many other high-level languages.
4. Memory Management – It provides manual memory management with new and delete and automatic memory handling using smart pointers (unique_ptr, shared_ptr).
5. Standard Template Library (STL) – STL offers built-in data structures (vectors, lists, maps) and algorithms (sorting, searching, etc.), improving efficiency.
6. Portability – C++ programs can run on multiple platforms with minimal changes.
7. Rich Library Support – It includes both standard and third-party libraries for graphics, networking, and database handling.
8. Concurrency and Multithreading – Modern C++ (C++11 and later) provides threading support for parallel execution.
9. Low-Level Programming Support – It allows direct hardware interaction, making it ideal for system programming and embedded systems.

These features make C++ a preferred choice for game development, operating systems, real-time applications, and large-scale software development.



CODTECH IT SOLUTIONS PVT.LTD
IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

1.3 HISTORY OF C++

C++ was developed by Bjarne Stroustrup at Bell Labs in 1979 as an extension of the C programming language. Initially called "C with Classes," it was designed to enhance C by introducing object-oriented programming (OOP) concepts like classes, inheritance, and polymorphism, which improve code reusability and maintainability.

In 1983, the language was renamed C++, where "++" signifies an increment, indicating an improved version of C. The first official C++ release (C++98) came in 1998, standardizing its syntax and features. C++03 followed in 2003 with minor improvements and bug fixes.

The C++11 standard, released in 2011, introduced major features like auto keyword, lambda expressions, smart pointers, move semantics, multithreading support, and enhanced STL features. This was followed by C++14 (2014) with minor updates and C++17 (2017), which improved performance, added new libraries, and streamlined syntax.

The latest versions, C++20 (2020) and C++23 (2023), brought features like concepts, ranges, coroutines, and modules, making C++ more efficient and modern.

Today, C++ remains widely used in game development, operating systems, real-time applications, and high-performance computing, continuing to evolve to meet modern programming needs.

1.4 APPLICATIONS OF C++

C++ is a powerful, high-performance programming language widely used in various domains due to its speed, efficiency, and versatility. Some key applications include:

1. System Software Development – C++ is used to build operating systems (Windows, macOS, Linux components) and device drivers due to its low-level memory access and hardware interaction capabilities.
2. Game Development – Game engines like Unreal Engine are built using C++ because of its ability to handle complex real-time graphics, physics simulations, and AI.
3. Embedded Systems – C++ is used in embedded systems, including automotive software, medical devices, and IoT applications, where performance and memory control are crucial.
4. High-Performance Applications – It is used in banking, trading, and stock market applications for high-speed computations (e.g., Bloomberg Terminal, financial modeling tools).
5. Web Browsers & GUI Applications – Browsers like Google Chrome and Mozilla Firefox use C++ for rendering engines and performance-critical tasks.
6. Scientific Computing & AI – C++ is used in machine learning frameworks like TensorFlow (backend) and high-performance scientific simulations.
7. Database Management Systems (DBMS) – Databases like MySQL and MongoDB use C++ for efficient data handling.

With its wide range of applications, C++ continues to be a crucial language in modern software development.



1.5 SETTING UP A C++ DEVELOPMENT ENVIRONMENT

To start coding in C++, you need a compiler, an integrated development environment (IDE), and a code editor. Follow these steps to set up your C++ development environment:

1. Install a C++ Compiler

A compiler translates C++ code into machine code. Popular C++ compilers include:

- GCC (GNU Compiler Collection) – Best for Linux/macOS, can be installed via MinGW on Windows.
- MSVC (Microsoft Visual C++ Compiler) – Comes with Visual Studio.
- Clang – A fast and modern compiler used on macOS and Linux.

2. Choose an IDE or Text Editor

An IDE provides tools like debugging, syntax highlighting, and auto-completion. Popular IDEs include:

- Visual Studio (Windows)
- Code::Blocks (Lightweight, cross-platform)
- Dev-C++ (Simple for beginners)
- Eclipse CDT (Cross-platform)
- CLion (Powerful but paid)

For lightweight coding, use VS Code or Sublime Text with a C++ extension.

3. Install a Build System (Optional)

- CMake – Helps manage complex projects.
- Make – Standard build tool for Linux.

4. Write and Run a C++ Program

Create a simple hello.cpp file and compile it using:

```
g++ hello.cpp -o hello
```

```
./hello
```

Now, you're ready to code in C++!



1.6 FIRST C++ PROGRAM

Writing and running your first C++ program is the first step toward learning the language. A simple C++ program prints "Hello, World!" on the screen.

1. Writing the Program

Create a new file named hello.cpp and write the following code:

```
#include <iostream> // Include input-output library
using namespace std; // Use standard namespace
int main() {
    cout << "Hello, World!" << endl; // Output message
    return 0; // Indicate successful program execution
}
```

2. Understanding the Code

- `#include <iostream>` – Includes the input-output stream library for printing text.
- `using namespace std;` – Allows direct use of `cout`, `cin`, and other standard library components.
- `int main()` – The main function, the entry point of a C++ program.
- `cout << "Hello, World!"`; – Prints text to the console.
- `return 0;` – Indicates successful program execution.

3. Compiling and Running the Program

To compile and run your program:

- Using g++ (GCC Compiler):
 - `g++ hello.cpp -o hello`
 - `./hello`
 - Using Microsoft Visual Studio: Open the IDE, create a new C++ project, and run the program.



1.7 COMPIILATION AND EXECUTION PROCESS

In C++, the process of converting source code into an executable program involves compilation and execution. This process occurs in several stages:

1. Writing the Source Code

A C++ program is written in a file with a .cpp extension. Example:

```
#include <iostream>using namespace std;  
int main() {  
    cout << "Hello, World!" << endl;  
    return 0;  
}
```

2. Compilation Process (Translation to Machine Code)

The compiler converts the human-readable C++ code into machine code in multiple steps:

- Preprocessing: Handles #include, #define, and macros.
- Compilation: Translates preprocessed code into assembly language.
- Assembly: Converts assembly language into machine code.
- Linking: Combines object files (.o or .obj) with libraries to create the final executable.

3. Execution Process

Once compiled, the program can be run on the system.

4. Compiling and Running a C++ Program

Using GCC (g++), compile and execute:

sh

```
g++ hello.cpp -o hello
```

```
./hello
```

Output:

Hello, World!

This process ensures that your code is correctly translated into a format the computer can execute



8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

2. BASIC CONCEPTS

C++ is a structured, object-oriented, and high-performance programming language. Before diving into advanced topics, understanding the basic concepts is essential.

1. C++ Program Structure

A basic C++ program consists of:

- Preprocessor Directives (#include <iostream>)
- Main Function (int main())
- Statements (cout << "Hello, World!");
- Return Statement (return 0;)

2. Variables and Data Types

Variables store values, and C++ supports different data types:

- Integer (int): Stores whole numbers (e.g., int a = 10;).
- Floating-Point (float, double): Stores decimals (e.g., double pi = 3.14;).
- Character (char): Stores a single character (e.g., char grade = 'A';).
- Boolean (bool): Stores true or false.
- String (string): Stores text (e.g., string name = "John";).

3. Input and Output

C++ uses cin for input and cout for output:

```
int age;  
cout << "Enter your age: ";  
cin >> age;  
cout << "You are " << age << " years old.";
```

4. Operators

C++ supports:

- Arithmetic Operators (+, -, *, /, %)
- Relational Operators (==, !=, <, >, <=, >=)
- Logical Operators (&&, ||, !)

Understanding these basics lays a strong foundation for learning C++ programming.

2.1 C++ SYNTAX AND STRUCTURE

C++ syntax and structure follow a well-defined format similar to C but with additional features like object-oriented programming.

Basic Structure

A C++ program consists of:

1. Preprocessor Directives: Begin with #, such as #include <iostream> for input/output.
2. Main Function: The entry point of the program, int main() { }.
3. Statements: End with a semicolon (;), enclosed within { }.
4. Comments: // for single-line and /* */ for multi-line comments.

Data Types & Variables

C++ supports fundamental types like int, float, char, double, and bool. Variables must be declared before use.

Operators

Arithmetic (+ - * / %), relational (== != > <), logical (&& || !), bitwise, and assignment operators exist.

Control Structures

- Conditional: if, if-else, switch-case.
- Loops: for, while, do-while.

Functions

Defined using returnType functionName(parameters) { }, functions enable code reuse.

Classes & Objects

Encapsulation via class allows defining attributes and methods. Objects instantiate classes.

Pointers & References

Pointers (*, &) enable dynamic memory handling.

C++ is powerful, supporting procedural, object-oriented, and generic programming paradigms.

2.2 VARIABLES AND DATA TYPES

In C++, a variable is a named storage location in memory that holds a value. Every variable must have a data type, which determines the kind of data it can store.

Basic Data Types in C++:

1. Integer Types (int) – Stores whole numbers (e.g., int age = 25;).
2. Floating-Point Types (float, double) – Stores decimal numbers (e.g., double pi = 3.14159;).
3. Character Type (char) – Holds a single character (e.g., char grade = 'A';).
4. Boolean Type (bool) – Holds true or false values (e.g., bool isPassed = true;).
5. Void (void) – Represents no value, mainly used for functions.

Variable Declaration and Initialization:

Variables must be declared before use. They can also be initialized at the time of declaration:

```
int x = 10; // Integer variable
float y = 5.5; // Floating-point variable
char letter = 'C'; // Character variable
bool flag = true; // Boolean variable
```

User-Defined Data Types:

C++ also supports struct, class, and enum for complex data structures.

Proper data type selection ensures efficient memory usage and correct program behavior.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

2.3 CONSTANTS AND LITERALS

CONSTANTS IN C++

A constant is a fixed value that does not change during program execution. There are three main ways to define constants in C++:

1. Using const Keyword

2. `const int MAX_AGE = 100;`

- The value of MAX_AGE cannot be changed after initialization.

3. Using #define Preprocessor Directive

4. `#define PI 3.14159`

- Replaces PI with 3.14159 before compilation.

5. Using constexpr (C++11 and later)

6. `constexpr int SIZE = 50;`

- Ensures the value is a compile-time constant.

LITERALS IN C++

Literals are fixed values directly used in the code.

1. Integer Literals (int)

2. `int x = 10; // Decimal (base 10)int y = 0xA; // Hexadecimal (base 16)`

3. Floating-Point Literals (float, double)

4. `double pi = 3.14;`

5. `float rate = 2.5f; // 'f' indicates float`

6. Character and String Literals

7. `char grade = 'A';`

8. `const char* name = "John"; // String literal`

9. Boolean Literals

10. `bool status = true; // Can be true or false`

Constants and literals improve program readability, safety, and efficiency.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

2.4 INPUT/OUTPUT IN C++ (CIN, COUT)

C++ provides standard input and output operations using the iostream library. The two main objects used are cin for input and cout for output.

1. Output using cout

The cout (character output) object is used to display data on the console. It is part of the std namespace and works with the insertion operator (<<).

```
#include <iostream>using namespace std;  
int main() {  
    cout << "Hello, C++!" << endl; // Output with newline  
    cout << "Age: " << 25;      // Multiple outputs  
    return 0;  
}
```

- endl is used to insert a newline (alternative: \n).

2. Input using cin

The cin (character input) object is used to take input from the user. It works with the extraction operator (>>).

```
#include <iostream>using namespace std;  
int main() {  
    int age;  
    cout << "Enter your age: ";  
    cin >> age; // Takes input from the user  
    cout << "You are " << age << " years old.";  
    return 0;  
}
```

- cin ignores spaces and stops input at whitespace.

For multiple inputs:

```
int a, b;  
cin >> a >> b;
```

cin and cout help in user interaction, making programs dynamic and interactive.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

2.5 OPERATORS IN C++ (ARITHMETIC, LOGICAL, RELATIONAL, BITWISE)

Operators in C++ are symbols that perform operations on variables and values. The four main types are Arithmetic, Logical, Relational, and Bitwise operators.

1. Arithmetic Operators

These operators perform mathematical calculations.

- + (Addition): $a + b$
- - (Subtraction): $a - b$
- * (Multiplication): $a * b$
- / (Division): a / b (for integers, only quotient is returned)
- % (Modulus): $a \% b$ (returns remainder of division)

Example:

```
int x = 10, y = 3;  
cout << x + y; // Output: 13  
cout << x % y; // Output: 1
```

2. Relational Operators

Used to compare values and return a boolean result (true or false).

- == (Equal to): $a == b$
- != (Not equal to): $a != b$
- > (Greater than): $a > b$
- < (Less than): $a < b$
- >= (Greater than or equal to): $a >= b$
- <= (Less than or equal to): $a <= b$

Example:

```
cout << (5 > 3); // Output: 1 (true)
```

3. Logical Operators

Used in conditions and return boolean values.

- && (Logical AND): $a \&\& b$ (true if both are true)
- || (Logical OR): $a || b$ (true if at least one is true)



CODTECH IT SOLUTIONS PVT.LTD

IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

! (Logical NOT): !a (negates the boolean value)

Example:

```
bool x = true, y = false;  
cout << (x && y); // Output: 0 (false)  
cout << (!x); // Output: 0 (false)
```

4. Bitwise Operators

These work at the bit level on integers.

- & (Bitwise AND)
- | (Bitwise OR)
- ^ (Bitwise XOR)
- ~ (Bitwise NOT)
- << (Left shift)
- >> (Right shift)

Example:

```
int a = 5, b = 3;  
cout << (a & b); // Output: 1 (0101 & 0011 = 0001)
```

Operators are essential for performing operations and making decisions in C++.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

3. CONTROL FLOW STATEMENTS

Control flow statements in C++ are used to determine the order in which individual statements are executed in a program. These statements help direct the program's flow based on certain conditions or iterations. There are three main types of control flow statements in C++: conditional statements, looping statements, and branching statements.

1. Conditional Statements: These allow the program to execute certain code based on a condition.

- if: Executes a block of code if the condition is true.
- else: Executes a block of code if the condition in the if statement is false.
- else if: Checks additional conditions if the initial if condition is false.
- switch: Used to select one of many code blocks to be executed based on the value of an expression.

2. Looping Statements: These repeat a block of code as long as a specified condition is true.

- for: Executes a block of code a fixed number of times.
- while: Executes a block of code as long as the condition remains true.
- do-while: Similar to while, but the block of code executes at least once before the condition is tested.

3. Branching Statements: These control the flow of execution.

- break: Exits from a loop or switch statement.
- continue: Skips the current iteration of a loop and moves to the next iteration.
- return: Exits from a function and optionally returns a value.

These control flow statements help manage the program's execution and create dynamic, efficient code.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

3.1 DECISION-MAKING STATEMENTS (IF, IF-ELSE, SWITCH)

In C++, decision-making statements are used to execute certain blocks of code based on specific conditions. The primary decision-making statements are if, if-else, and switch.

1. if Statement: The if statement checks a condition and executes the associated block of code if the condition is true. If the condition is false, the code inside the if block is skipped.

```
2. if (condition) {  
3.   // Code to execute if condition is true  
4. }
```

5. if-else Statement: The if-else statement provides an alternative execution path. If the condition is true, the code inside the if block is executed; if the condition is false, the code inside the else block is executed.

```
6. if (condition) {  
7.   // Code to execute if condition is true  
8. } else {  
9.   // Code to execute if condition is false  
10.}
```

11. switch Statement: The switch statement is used to select one of many blocks of code to execute based on the value of a variable or expression. It is typically used when there are multiple possible conditions to check. The switch statement uses case labels to define various options and a default case for unmatched values.

```
12. switch (variable) {  
13.   case 1:  
14.     // Code for case 1  
15.     break;  
16.   case 2:  
17.     // Code for case 2  
18.     break;  
19.   default:  
20.     // Code if no case matches  
21. }
```

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

3.2 LOOPING STATEMENTS (FOR, WHILE, DO-WHILE)

In C++, looping statements allow you to repeatedly execute a block of code as long as a specified condition is true. The primary looping statements are for, while, and do-while.

1. for Loop: The for loop is used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and increment/decrement. The loop executes as long as the condition is true.

2. for (initialization; condition; increment) {

3. // Code to execute

4. }

5. Example:

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

7. std::cout << i << " ";

8. }

9. while Loop: The while loop is used when the number of iterations is not known in advance. It keeps executing the block of code as long as the given condition evaluates to true.

10. while (condition) {

11. // Code to execute

12. }

13. Example:

14. int i = 0;

15. while (i < 5) {

16. std::cout << i << " ";

17. i++;

18. }



8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

do-while Loop: The do-while loop is similar to the while loop, but it guarantees that the code block will be executed at least once, as the condition is checked after the block is executed.

```
1. do {  
2.   // Code to execute  
3. } while (condition);  
4. Example:  
5. int i = 0;  
6. do {  
7.   std::cout << i << " ";  
8.   i++;  
9. } while (i < 5);
```

These looping statements help repeat code efficiently based on conditions.

3.3 JUMP STATEMENTS (BREAK, CONTINUE, GOTO, RETURN)

jump statements control the flow of execution by transferring control to different parts of the program. The primary jump statements are break, continue, goto, and return.

1. break: The break statement is used to exit from a loop (for, while, do-while) or a switch statement immediately. It terminates the current loop or switch, and control moves to the next statement after the loop or switch.

```
2. for (int i = 0; i < 5; i++) {  
3.   if (i == 3) {  
4.     break; // Exits the loop when i equals 3  
5.   }  
6.   std::cout << i << " ";  
7. }
```



8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

continue: The continue statement skips the current iteration of a loop and moves to the next iteration.

It only affects loops.

```
1. for (int i = 0; i < 5; i++) {  
2.   if (i == 3) {  
3.     continue; // Skips the iteration when i equals 3  
4.   }  
5.   std::cout << i << " ";  
6. }
```

7.goto: The goto statement transfers control to a labeled statement in the code. It is rarely used due to potential issues with program readability and maintainability.

8.goto label; // Jump to the label

9.label:

10. std::cout << "Jumped!";

11.return: The return statement exits from a function and optionally returns a value. It transfers control to the calling function.

12.int add(int a, int b) {

13. return a + b; // Exits the function and returns the sum

14.}

These jump statements help manage program flow, but they should be used carefully to avoid making the code hard to follow.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

4. FUNCTIONS IN C++

In C++, functions are blocks of code that perform a specific task and can be called to execute at any point in a program. Functions help in organizing code, improving reusability, and making the program modular. A function in C++ typically consists of a function declaration (or prototype), function definition, and function call.

1. Function Declaration: This provides the compiler with the function's name, return type, and parameters before the function is called. It acts as a forward declaration.

2. `int add(int, int); // Function declaration`

3. Function Definition: This is where the actual implementation of the function is written, including the body that performs the task.

4. `int add(int a, int b) {`

5. `return a + b; // Function definition`

6. }

7. Function Call: To execute the function, you call it by name, passing the required arguments (if any).

8. `int result = add(3, 4); // Function call`

Functions can have parameters (input values) and a return type (output value). A function can return a value of any type, or it can be of type void if it does not return anything.

Functions can also be overloaded (multiple functions with the same name but different parameters) and recursively called (a function calls itself). Functions help organize complex programs and make them easier to maintain.



CODTECH IT SOLUTIONS PVT.LTD
IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

4.1 FUNCTION DECLARATION AND DEFINITION

function declaration and function definition are essential components of a function. They help define the structure of a function and allow it to be used in different parts of the program.

1. Function Declaration (Prototype): A function declaration provides information to the compiler about the function's name, return type, and parameters (if any), without the function body. It tells the compiler that the function exists and will be defined later in the code. The declaration must appear before the function is called in the program. This is especially useful when the function is defined after its use or in another file.

2. int add(int, int); // Declaration of the function

3. In this example, int is the return type, and the function add takes two int parameters.

4. Function Definition: The function definition includes both the declaration and the body of the function, where the actual logic is implemented. The definition provides the complete functionality of the function and is where the behavior of the function is described.

5. int add(int a, int b) {

6. return a + b; // Function body

7. }

8. In this example, the add function is defined to take two integer parameters (a and b), add them, and return the result.

The declaration and definition are often separated, particularly in large programs or libraries, to allow functions to be used across different files.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

4.2 FUNCTION PARAMETERS AND RETURN TYPES

function parameters and return types are key concepts that define how a function operates, takes inputs, and provides outputs.

1. Function Parameters: Function parameters are variables listed in the function declaration and definition that accept values when the function is called. They act as input values to the function, enabling the function to operate on dynamic data. Parameters are defined in the function's parentheses and can be of any data type (e.g., int, float, char, etc.). A function can have no parameters (empty parentheses) or multiple parameters, depending on the logic needed.

2. Example:

3. void displayMessage(std::string message); // Parameter: message of type std::string

4. Function call:

5. displayMessage("Hello, C++!"); // Passing a string as an argument

6. C++ also allows default parameters. If no argument is passed, the default value is used:

7. void print(int x = 10); // Default value of x is 10

8. Return Type: The return type specifies the type of value a function will return to the caller after execution. It can be any valid data type (int, float, void, etc.). If the function does not return a value, the return type is specified as void.

9. Example:

10. int add(int a, int b) { // Return type is int
return a + b;

11. }

12. The return type ensures that the caller knows what kind of value to expect when the function completes.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

4.3 CALL BY VALUE VS CALL BY REFERENCE

Call by Value and Call by Reference are two different methods used to pass arguments to a function. They affect how the arguments are handled inside the function.

1. Call by Value: In Call by Value, a copy of the actual argument is passed to the function. This means that any changes made to the parameter inside the function do not affect the original argument outside the function. It is useful when you want to protect the original data from being modified.

2. Example:

```
3. void addTen(int x) {  
4.     x = x + 10; // Modifies the copy of x  
5. }  
6. int main() {  
7.     int a = 5;  
8.     addTen(a);  
9.     std::cout << a; // Outputs 5, as a was not modified  
10.}
```

11. In this case, a remains unchanged after the function call.

12. Call by Reference: In Call by Reference, a reference (memory address) of the actual argument is passed to the function. This allows the function to modify the original variable's value, as it directly accesses the memory where the variable is stored.

13. Example:

```
14. void addTen(int &x) {  
15.     x = x + 10; // Modifies the original variable  
16. }  
17. int main() {  
18.     int a = 5;  
19.     addTen(a); std::cout << a; // Outputs 15, as a was modified }Here, the value of a is changed because the function operates directly on the original variable.
```

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

4.4 FUNCTION OVERLOADING

Function Overloading in C++ is a feature that allows multiple functions to have the same name but with different parameter lists. The compiler distinguishes between the functions based on the number or types of parameters. Function overloading improves code readability and allows the same function name to be used for different tasks, provided the parameters differ in type, number, or both.

Key Points:

1. Same Function Name: Overloaded functions must share the same name but must differ in their parameters.
2. Different Signatures: The function signature (the function name and the parameter list) must be different. The return type does not count toward overloading.

Example of Function Overloading

```
#include <iostream>using namespace std;  
  
int add(int a, int b) {  
    return a + b; // Adds two integers  
}  
  
double add(double a, double b) {  
    return a + b; // Adds two doubles  
}  
  
int main() {  
    cout << add(3, 4) << endl;    // Calls the integer version  
    cout << add(3.5, 4.5) << endl; // Calls the double versionreturn 0;  
}
```

Output:

7

8

In the above example, both add functions perform similar tasks, but one works with integers and the other with doubles. The appropriate function is called based on the arguments passed.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

4.4 FUNCTION OVERLOADING

Function Overloading in C++ is a feature that allows multiple functions to have the same name but with different parameter lists. The compiler distinguishes between the functions based on the number or types of parameters. Function overloading improves code readability and allows the same function name to be used for different tasks, provided the parameters differ in type, number, or both.

Key Points:

1. Same Function Name: Overloaded functions must share the same name but must differ in their parameters.
2. Different Signatures: The function signature (the function name and the parameter list) must be different. The return type does not count toward overloading.

Example of Function Overloading

```
#include <iostream>using namespace std;  
int add(int a, int b) {  
    return a + b; // Adds two integers  
}  
  
double add(double a, double b) {  
    return a + b; // Adds two doubles  
}  
  
int main() {  
    cout << add(3, 4) << endl;      // Calls the integer version  
    cout << add(3.5, 4.5) << endl;  // Calls the double versionreturn 0;  
}
```

Output:

7

8

In the above example, both add functions perform similar tasks, but one works with integers and the other with doubles. The appropriate function is called based on the arguments passed.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

4.5 INLINE FUNCTIONS

inline functions are functions that are expanded at the point of call, rather than being called through a function call mechanism. This can improve performance by eliminating the overhead of a function call, such as pushing arguments to the stack and jumping to the function code. The inline keyword is used to suggest to the compiler that it should replace the function call with the function code itself.

Syntax:

```
inline return_type function_name(parameters) {  
    // Function body  
}
```

Example:

```
#include <iostream>using namespace std;  
inline int add(int a, int b) {  
    return a + b;  
}  
  
int main() {  
    int result = add(3, 4); // Directly replaces with '3 + 4'  
    cout << result << endl; // Outputs 7return 0;  
}
```

Advantages:

1. Reduced Function Call Overhead: In small, frequently called functions, the function call overhead is eliminated.
2. Improved Performance: Especially for simple functions like add(), where the code is small and quick to execute.

Disadvantages:

1. Code Bloat: If the inline function is called many times, it can increase the size of the compiled code, potentially leading to memory inefficiency.
2. Limited Use: Inline functions are typically best for small, simple functions. Complex functions may not benefit from inlining.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

4.6 RECURSIVE FUNCTIONS

recursive functions are functions that call themselves in order to solve a problem. Recursive functions typically break down a problem into smaller, more manageable sub-problems until they reach a base case, which stops further recursive calls.

Structure of a Recursive Function:

1. Base Case: A condition that stops the recursion and prevents infinite calls.
2. Recursive Case: A part of the function where it calls itself with modified arguments, reducing the problem's size.

Example of a Recursive Function (Factorial):

The factorial of a number n is the product of all positive integers from 1 to n, and is defined as:

- $\text{factorial}(n) = n * \text{factorial}(n-1)$ with the base case $\text{factorial}(0) = 1$.
- `#include <iostream>using namespace std;`

```
int factorial(int n) {
    // Base case: if n is 0, return 1
    if (n == 0) {
        return 1;
    }
    // Recursive case: n * factorial of n-1
    return n * factorial(n - 1);
}
```

```
int main() {
    int result = factorial(5); // Calls factorial recursively
    cout << "Factorial of 5 is: " << result << endl; // Outputs 120
}
```

Output:

csharp

Factorial of 5 is: 120

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

5. ARRAYS AND STRINGS

Arrays:

An array is a collection of elements of the same type, stored in contiguous memory locations. It is used to store multiple values in a single variable. The size of an array is fixed once it is declared. You can access elements using their index, starting from 0.

Syntax:

```
type array_name[size];
```

Example:

```
#include <iostream>using namespace std;  
int main() {  
    int arr[5] = {1, 2, 3, 4, 5}; // Declaring an integer array  
    cout << arr[0] << endl; // Accesses the first element, outputs 1return 0; }
```

Arrays can also be multidimensional, such as 2D arrays:

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Strings:

In C++, a string is a sequence of characters. While C++ provides the `string` class (part of the C++ Standard Library), a C-style string is an array of characters terminated by a null character ('\0').

C++ String (Standard Library):

```
#include <iostream>#include <string>using namespace std;  
int main() {  
    string str = "Hello, World!";  
    cout << str << endl; // Outputs: Hello, World!return 0; }
```

C-Style String:

```
#include <iostream>using namespace std;  
int main() {  
    char str[] = "Hello";  
    cout << str << endl; // Outputs: Helloreturn 0;  
}
```

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

5.1 INTRODUCTION TO ARRAYS

an array is a data structure that allows you to store a fixed-size collection of elements of the same type in contiguous memory locations. Arrays enable you to group related data under a single name, making it easier to manage and manipulate collections of data.

Key Characteristics of Arrays:

1. Fixed Size: The size of an array is defined at the time of declaration and cannot be changed later.
2. Homogeneous Elements: All elements in an array must be of the same data type (e.g., all int, all float).
3. Zero-Based Indexing: Elements in an array are accessed using an index, starting from 0 for the first element.

Syntax:

```
type array_name[size];
```

- type: The data type of the array elements (e.g., int, float).
- array_name: The name of the array.
- size: The number of elements in the array.

Example:

```
#include <iostream>using namespace std;  
int main() {  
    int arr[5] = {1, 2, 3, 4, 5}; // Array of 5 integers  
    cout << arr[0] << endl; // Outputs the first element (1)  
    return 0;  
}
```

Accessing Elements:

Array elements are accessed by using the index, like arr[0], arr[1], etc. The index must be within the array's bounds, i.e., $0 \leq \text{index} < \text{size}$.

Arrays are useful when dealing with a known, fixed amount of data, and they provide efficient access and manipulation of the stored elements.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

5.2 ONE-DIMENSIONAL ARRAYS

A one-dimensional array in C++ is a collection of elements of the same data type stored in a contiguous memory block. It is essentially a list where each element is accessed using an index, starting from 0. One-dimensional arrays are useful for storing a fixed-size sequence of data, such as a list of numbers or characters.

Syntax:

To declare a one-dimensional array in C++, you specify the type of elements, the name of the array, and the size (number of elements):

type array_name[size];

- type: The data type of the array elements (e.g., int, float).
- array_name: The name used to refer to the array.
- size: The number of elements the array can hold.

Example:

```
#include <iostream>using namespace std;  
int main() {  
    int arr[5] = {10, 20, 30, 40, 50}; // One-dimensional array of 5 integers// Accessing and printing  
array elementsfor (int i = 0; i < 5; i++) {  
    cout << arr[i] << " "; // Outputs each element  
}  
return 0;  
}
```

Output:

10 20 30 40 50

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

5.3 TWO-DIMENSIONAL ARRAYS

A two-dimensional array in C++ is an array of arrays, essentially representing a matrix or table with rows and columns. Each element in a two-dimensional array is accessed using two indices: one for the row and one for the column.

Syntax:

To declare a two-dimensional array, you specify the data type, the name of the array, and the size of both dimensions (rows and columns):

```
type array_name[rows][columns];
```

- type: The data type of the array elements (e.g., int, float).
- rows: The number of rows in the array.
- columns: The number of columns in each row.

Example:

```
#include <iostream>using namespace std;
int main() {
    int matrix[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    }; // A 3x3 two-dimensional array// Accessing and printing array elementsfor (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix[i][j] << " "; // Outputs each element }
            cout << endl; // Newline after each row }
    return 0 }
```

Output:

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

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

5.4 MULTI-DIMENSIONAL ARRAYS

multi-dimensional arrays are arrays with more than two dimensions. These arrays extend the concept of two-dimensional arrays by allowing for multiple rows, columns, and additional dimensions, making them useful for representing data structures such as matrices, 3D grids, or even more complex structures like tensors.

Syntax:

To declare a multi-dimensional array, you specify the data type, array name, and the size for each dimension. For example, a three-dimensional array would be declared as:

```
type array_name[dim1][dim2][dim3];
```

- dim1, dim2, dim3: Sizes of each dimension (number of elements along each axis).
- type: The type of the elements (e.g., int, float).

Example (3D Array):

```
#include <iostream>using namespace std;  
int main() {  
    int arr[2][3][4] = {  
        {  
            {1, 2, 3, 4},  
            {5, 6, 7, 8},  
            {9, 10, 11, 12}  
        },  
        {  
            {13, 14, 15, 16},  
            {17, 18, 19, 20},  
            {21, 22, 23, 24}  
        }  
    }; // 2x3x4 array (2 blocks, 3 rows, 4 columns)// Accessing and printing elements  
    for (int i = 0; i < 2; i++) {
```

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

```
for (int j = 0; j < 3; j++) {  
    for (int k = 0; k < 4; k++) {  
        cout << arr[i][j][k] << " ";  
    }  
    cout << endl;  
}  
}  
return 0;  
}  
  
Output:  
1 2 3 4  
5 6 7 8  
9 10 11 12  
13 14 15 16  
17 18 19 20  
21 22 23 24
```

5.5 STRING HANDLING IN C++ (STRING CLASS VS C-STYLE STRINGS)

string handling can be done using two main approaches: the string class (from the C++ Standard Library) and C-style strings (character arrays). Both have their uses, but they differ in functionality, ease of use, and flexibility.

String Class (C++ Standard Library):

The string class, defined in the `<string>` header, provides a high-level abstraction for handling strings. It allows dynamic resizing, built-in methods for manipulation, and is more user-friendly compared to C-style strings.

Example:

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



8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

```
string str = "Hello, C++!";  
cout << str.length() << endl; // Outputs the length of the string  
str.append(" Welcome!");  
cout << str << endl; // Outputs the updated stringreturn 0;  
}
```

C-style Strings:

A C-style string is an array of characters terminated by a null character ('\0'). They are simpler but require manual memory management and do not support built-in functions for manipulation, making them prone to errors.

Example:

```
#include <iostream>using namespace std;  
int main() {  
    char str[] = "Hello, C!";  
    cout << strlen(str) << endl; // Outputs the length of the stringstrcat(str, " Welcome!"); // Concatenates  
    a string  
    cout << str << endl; // Outputs the concatenated stringreturn 0;  
}
```

Key Differences:

Ease of Use: The string class offers more functionality (e.g., dynamic sizing, built-in methods).

Memory Management: C-style strings require manual memory management and are more error-prone.

Performance: C-style strings can be faster for small tasks due to their simplicity, but string objects are more robust for general use.

In general, the string class is recommended for most applications due to its flexibility and ease of use.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

5.6 STRING MANIPULATION FUNCTION

string manipulation functions allow you to perform various operations like concatenation, comparison, searching, and modification on strings. The C++ Standard Library provides a powerful string class (from the `<string>` header) with built-in functions for easy string manipulation. Below are some common functions:

1. length():

Returns the number of characters in a string.

```
string str = "Hello";
cout << str.length(); // Outputs 5
```

2. append():

Adds characters to the end of a string.

```
string str = "Hello";
str.append(", World!");
cout << str; // Outputs "Hello, World!"
```

3. substr():

Extracts a substring starting from a specified position and optional length.

```
string str = "Hello, World!";
string sub = str.substr(7, 5); // "World"
cout << sub;
```

4. find():

Finds the first occurrence of a substring within the string, returning its position.

```
string str = "Hello, World!";
size_t pos = str.find("World"); // Returns position 7
cout << pos;
```



CODTECH IT SOLUTIONS PVT.LTD

IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

5. replace():

Replaces part of a string with another string.

```
string str = "Hello, World!";
str.replace(7, 5, "C++");
cout << str; // Outputs "Hello, C++!"
```

6. erase():

Removes characters from the string.

```
string str = "Hello, World!";
str.erase(5, 7); // Removes ", World!"
cout << str; // Outputs "Hello"
```

7. c_str():

Converts the string to a C-style string (null-terminated character array).

```
string str = "Hello";
const char* cstr = str.c_str();
cout << cstr; // Outputs "Hello"
```

These functions make string manipulation in C++ convenient and efficient, offering flexibility for handling text data in various ways.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

6. POINTERS IN C++

pointers are variables that store memory addresses of other variables. Instead of holding data directly, pointers hold the address of another variable, enabling powerful features such as dynamic memory allocation, array manipulation, and creating complex data structures like linked lists.

Declaring Pointers:

A pointer is declared by specifying the type of the variable it will point to, followed by an asterisk (*).

```
int* ptr; // Pointer to an integer
```

Initializing Pointers:

Pointers are initialized using the address-of operator (&), which retrieves the memory address of a variable.

```
int num = 5;
```

```
int* ptr = &num; // ptr now holds the address of num
```

Dereferencing Pointers:

Dereferencing a pointer means accessing the value stored at the memory address it points to, using the asterisk (*).

```
int value = *ptr; // Dereference ptr to get the value of num
```

Example:

```
#include <iostream>using namespace std;  
int main() {  
    int num = 10;  
    int* ptr = &num; // Pointer to num  
    cout << "Value of num: " << num << endl; // 10  
    cout << "Address of num: " << &num << endl; // Memory
```

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

address of num

```
cout << "Value through pointer: " << *ptr << endl; // 10return 0;  
}
```

Output:

yaml

Value of num: 10Address of num: 0x7ffee7bfa8acValue through pointer: 10

Key Concepts:

- Pointers are essential for efficient memory management and accessing dynamic data.
- NULL pointers (or nullptr) can indicate an invalid or uninitialized pointer.

Pointers provide flexibility and control over memory, making them a core concept in C++.

6.1 INTRODUCTION TO POINTERS

pointers are variables that store the memory address of another variable. Instead of holding data directly, pointers store the location in memory where the data is stored. This allows more flexible and efficient manipulation of data, particularly for dynamic memory allocation, arrays, and data structures like linked lists.

Basic Syntax:

A pointer is declared by specifying the type of data it points to, followed by an asterisk (*):

```
type* pointer_name;
```

For example, an integer pointer is declared as:

```
int* ptr;
```

Assigning Address to a Pointer:

To assign a memory address to a pointer, use the address-of operator (&):

```
int num = 5;
```

```
int* ptr = &num; // ptr stores the address of num
```

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

For example, an integer pointer is declared as:

```
int* ptr;
```

Assigning Address to a Pointer:

To assign a memory address to a pointer, use the address-of operator (&):

```
int num = 5;
```

```
int* ptr = &num; // ptr stores the address of num
```

6.2 POINTER ARITHMETIC

pointer arithmetic refers to performing operations on pointers, such as incrementing, decrementing, or adding an offset. This allows for navigating through memory locations in an efficient manner, particularly when working with arrays and dynamic memory.

Pointer Arithmetic Operations:

1. Increment (++): When you increment a pointer, it moves to the next memory location of its type.
If the pointer is of type int*, it moves by the size of an int (typically 4 bytes on most systems).
2.

```
int arr[] = {10, 20, 30};
```
3.

```
int* ptr = arr; // ptr points to arr[0]
```
4.

```
ptr++; // Now ptr points to arr[1]
```
5. Decrement (--): When you decrement a pointer, it moves backward by the size of its type.
6.

```
ptr--; // Now ptr points back to arr[0]
```
7. Addition (+): You can add an integer to a pointer to move it forward by that many elements (not bytes).
8.

```
ptr = arr + 2; // Now ptr points to arr[2]
```
9. Subtraction (-): You can subtract two pointers pointing to elements of the same array to get the number of elements between them.
10.

```
int diff = ptr - arr; // Number of elements between ptr and arr
```

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

_Example:

```
#include <iostream>using namespace std;  
int main() {  
    int arr[] = {10, 20, 30, 40};  
    int* ptr = arr;  
  
    cout << *ptr << endl; // 10  
    ptr++;  
    cout << *ptr << endl; // 20return 0;  
}
```

6.3 POINTERS AND ARRAYS

pointers and arrays are closely related, as arrays are essentially pointers to a contiguous block of memory. Understanding how they interact can enhance your ability to manipulate data efficiently.

Arrays and Pointers Relationship:

1. Array Name as a Pointer: The name of an array is a constant pointer to the first element of the array. For example, if you have:

2. `int arr[] = {10, 20, 30};`

3. `arr` is a pointer to `arr[0]`. So, `arr` and `&arr[0]` refer to the same memory address.

4. Accessing Array Elements Using Pointers: You can use pointers to access array elements by incrementing the pointer:

5. `int* ptr = arr; // ptr points to arr[0]`

6. `cout << *ptr << endl; // Outputs 10`

7. `ptr++; // Move the pointer to the next element`

8. `cout << *ptr << endl; // Outputs 20`

9. Pointer Arithmetic: Since arrays are contiguous blocks of memory, you can perform pointer arithmetic to navigate through them:

10. `int* ptr = arr;`

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

- `_cout << *(ptr + 2) << endl; // Outputs 30 (arr[2])`

Example:

```
#include <iostream>using namespace std;
int main() {
    int arr[] = {10, 20, 30, 40};
    int* ptr = arr;
    for (int i = 0; i < 4; i++) {
        cout << *(ptr + i) << " "; // Access array elements using pointer
    }
    return 0;
}
```

Output:
10 20 30 40

6.4 POINTERS TO FUNCTIONS

pointers to functions allow you to store the memory address of a function and use that pointer to call the function indirectly. This feature is useful for implementing callback mechanisms, handling dynamic behavior, and passing functions as arguments to other functions.

Declaring a Pointer to a Function:

A pointer to a function is declared by specifying the return type, followed by an asterisk (*), the pointer name, and the function signature (parameter types).

```
return_type (*pointer_name)(parameter_types);
```

For example, a pointer to a function that takes two int arguments and returns an int is declared as:

```
int (*funcPtr)(int, int);
```

Assigning a Function to the Pointer:

You assign a function's name (without parentheses) to the pointer:

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

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

- funcPtr = add; // Assign function 'add' to the pointer
- Calling a Function via Pointer:
- To call the function through the pointer, use the pointer followed by parentheses, just like calling a function.
- int result = funcPtr(3, 4); // Calls add(3, 4)
- Example:
- #include <iostream>using namespace std;
- int add(int a, int b) {
- return a + b;
- }
- int main() {
- int (*funcPtr)(int, int) = add; // Pointer to add function
- cout << "Result: " << funcPtr(5, 6) << endl; // Calls add(5, 6) return 0;
- }
- Output:
- makefile
- Result: 11

6.5 POINTERS TO OBJECTS

pointers to objects allow you to store the memory address of an object, enabling dynamic memory allocation and efficient manipulation of objects, especially when dealing with large data structures or object-oriented programming patterns like polymorphism.

Declaring a Pointer to an Object:

A pointer to an object is declared similarly to a pointer to a basic data type. You specify the class type followed by an asterisk (*).

ClassName* pointerName;

For example, if you have a class Person, you can declare a pointer to a Person object as:

Person* ptr;



8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

Creating Objects Dynamically:

You can create objects dynamically using the new operator, and the pointer will store the memory address of the object.

```
Person* ptr = new Person(); // Dynamically allocate memory for a Person object
```

Accessing Object Members via Pointer:

To access the members of an object through a pointer, you use the arrow operator (->).

```
ptr->name = "John"; // Set the name of the person
```

```
ptr->age = 30; // Set the age of the person
```

Example:

```
#include <iostream>
using namespace std;
class Person {
public:
    string name;
    int age;
};
int main() {
    Person* ptr = new Person; // Dynamically allocate memory for Person
    ptr->name = "Alice";
    ptr->age = 25;
    cout << "Name: " << ptr->name << ", Age: " << ptr->age << endl;
    delete ptr; // Free dynamically allocated memory
    return 0;
}
```

Output:

```
yaml
```

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

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

7. DYNAMIC MEMORY ALLOCATION

dynamic memory allocation allows you to allocate memory at runtime, providing flexibility for managing memory when the size of data structures is not known beforehand. This is crucial for situations where memory needs may change during the program's execution, such as for dynamic arrays or complex data structures.

Using new and delete Operators:

- **new:** Allocates memory for a single object or an array of objects dynamically. It returns a pointer to the allocated memory.
- **delete:** Frees memory allocated by new. It prevents memory leaks by releasing memory when it is no longer needed.

Syntax for Allocating Memory:

1.Single Object:

2.cpp

3.Copy

4.int* ptr = new int; // Allocates memory for one int

5.*ptr = 5; // Assign value to allocated memory

6.Array of Objects:

7.cpp

8.Copy

9.int* arr = new int[5]; // Allocates memory for an array of 5 integers

10.arr[0] = 1; // Access array elements

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

Deallocating Memory:

- For a Single Object:
- `delete ptr;` // Frees memory allocated for a single object
- For an Array:
- `delete[] arr;` // Frees memory allocated for an array of objects

Example:

```
#include <iostream>using namespace std;  
int main() {  
    int* arr = new int[3]; // Dynamically allocate an array of 3 integers  
    arr[0] = 10; arr[1] = 20; arr[2] = 30;  
    for (int i = 0; i < 3; i++) {  
        cout << arr[i] << " "; // Output the array elements  
    }  
    delete[] arr; // Deallocate memory  
    return 0;  
}
```

Output:

10 20 30

7.1 NEW AND DELETE OPERATORS

In C++, new and delete are operators used for dynamic memory management.

- new allocates memory on the heap and returns a pointer to the allocated memory.
- `int* ptr = new int;` // Allocates memory for one integer
- delete deallocates memory allocated with new, preventing memory leaks.
- `delete ptr;` // Frees memory for the single integer

For arrays, use `new[]` and `delete[]`:

- `new[]`: Allocates memory for an array.
- `delete[]`: Deallocates memory for an array.
- `int* arr = new int[5];` // Allocates an array of 5 integers
`delete[] arr;` .

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

7.2 DYNAMIC ARRAYS

Dynamic arrays in C++ allow for flexible memory allocation at runtime using pointers and the new operator. Unlike static arrays, whose size is fixed at compile time, dynamic arrays can be resized as needed.

To create a dynamic array, use:

```
int* arr = new int[size];
```

To deallocate memory and prevent leaks, use:

```
delete[] arr;
```

For resizing, a new array is allocated, and existing elements are copied:

```
int* newArr = new int[newSize];
std::copy(arr, arr + oldSize, newArr);
delete[] arr;
arr = newArr;
```

Alternatively, std::vector from the STL provides a safer and more efficient way to manage dynamic arrays, as it handles resizing and memory management automatically. Dynamic arrays are useful in scenarios requiring variable storage, such as handling user input or implementing data structures like lists and queues.

7.3 MEMORY LEAKS AND PREVENTION

Memory leaks in C++ occur when dynamically allocated memory is not properly deallocated, leading to wasted resources and potential program crashes. This happens when a pointer loses reference to allocated memory without calling delete or delete[].

Causes of Memory Leaks

- Forgetting to free memory after new allocation.
- Overwriting pointers without first releasing memory.
- Improper handling of exceptions leading to memory not being freed.

Prevention Techniques

1. Use delete or delete[]: Always deallocate memory when it's no longer needed.



CODTECH IT SOLUTIONS PVT.LTD

IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

- int* ptr = new int(5);
- delete ptr;
- Use Smart Pointers (std::unique_ptr, std::shared_ptr): They automatically manage memory.
- std::unique_ptr<int> ptr = std::make_unique<int>(5);
- Use Containers (std::vector, std::string): STL containers manage memory efficiently.
- Memory Leak Detection Tools: Tools like Valgrind help identify leaks.

Proper memory management ensures efficient resource utilization and program stability.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

8. OBJECT-ORIENTED PROGRAMMING (OOP) IN C++

Object-Oriented Programming (OOP) in C++ is a programming paradigm that organizes code using objects and classes, promoting modularity and reusability.

Key Concepts of OOP in C++

1. Classes and Objects – A class is a blueprint, and objects are instances of the class.
2. public:
3. string brand;
4. void show() { cout << brand; }
- 5.};
6. Car myCar; // Object creation
7. Encapsulation – Bundling data and methods to protect object integrity.
8. class BankAccount {
9. private:
10. double balance;
11. public:
12. void deposit(double amount) { balance += amount; }
- 13.};
14. Inheritance – Enables code reuse by deriving new classes from existing ones.

1. class ElectricCar : public Car {};
2. Polymorphism – Allows function overriding and dynamic behavior.
3. class Animal { virtual void speak() {} };
4. class Dog : public Animal { void speak() override { cout << "Bark"; } };

OOP enhances code organization, reusability, and maintainability in C++.

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

8.1 INTRODUCTION TO LOOP

Loops in C++ allow repetitive execution of a block of code, reducing redundancy and improving efficiency. They run until a specified condition is met.

Types of Loops in C++

1. for Loop – Used when the number of iterations is known.
2. `for(int i = 0; i < 5; i++) { cout << i << " "; }`
3. while Loop – Runs as long as a condition remains true.
4. `int i = 0; while(i < 5) { cout << i << " "; i++; }`
5. do-while Loop – Executes at least once before checking the condition.
6. `int i = 0; do { cout << i << " "; i++; } while(i < 5);`

Loops are essential for handling repetitive tasks efficiently.

8.2 CLASSES AND OBJECTS

Classes and Objects in C++

In C++, a class is a user-defined data type that encapsulates data (variables) and functions (methods). An object is an instance of a class, representing real-world entities with properties and behaviors.

Defining a Class and Creating Objects

```
#include <iostream>using namespace std;  
class Car {  
public:  
    string brand;  
    int year;  
void showDetails() {  
    cout << "Brand: " << brand << ", Year: " << year << endl;  
}  
};
```



8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

```
int main() {
    Car myCar; // Object creation
    myCar.brand = "Toyota";
    myCar.year = 2023;
    myCar.showDetails();
    return 0;
}
```

8.3 ACCESS SPECIFIERS (PUBLIC, PRIVATE, PROTECTED)

Access specifiers define the visibility and accessibility of class members in C++.

1. Public – Members are accessible from anywhere.
2. class Example {
3. public:
4. int x;
5. };
6. Example obj;
7. obj.x = 10; // Allowed
8. Private – Members are accessible only within the class.
9. class Example {
10. private:
11. int x;
12. };
13. Example obj;
14. obj.x = 10; // Error
15. Protected – Like private but accessible in derived classes.
16. class Base {
17. protected:
18. int x;
19. };



9 . Inheritance in C++

9.1 Types of Inheritance (Single, Multiple, Multilevel, Hierarchical, Hybrid)

Inheritance is a core concept in object-oriented programming (OOP), enabling classes to inherit attributes and methods from other classes, promoting code reusability and modularity. There are several types of inheritance:

- Single Inheritance: In this type, a child class inherits from only one parent class. It is the simplest form of inheritance, where the child can access methods and properties of the single parent class. Example: A class Dog inheriting from Animal.
- Multiple Inheritance: Here, a class inherits from more than one parent class. This allows the child class to acquire features from multiple classes. However, it can lead to ambiguity issues, especially if parent classes have methods with the same name. Example: A class Bat inheriting from both Mammal and Bird.
- Multilevel Inheritance: In multilevel inheritance, a class inherits from a parent class, and another class inherits from this child class, forming a chain. This creates a hierarchy of inheritance. Example: A class Puppy inheriting from Dog, which inherits from Animal.
- Hierarchical Inheritance: Multiple classes inherit from a single parent class. All child classes share the properties and methods of the parent class. Example: Dog and Cat inheriting from Animal.
- Hybrid Inheritance: A combination of multiple inheritance types, such as multilevel and multiple inheritance, leading to a more complex structure. Example: A class inheriting from multiple parent classes in different hierarchies.



9.2 Constructor and Destructor in Inheritance

In object-oriented programming (OOP), constructors and destructors play crucial roles in the lifecycle of objects. When inheritance is involved, these special methods are also inherited and can be modified or extended by child classes.

Constructor in Inheritance:

A constructor is a special method used to initialize objects when they are created. In inheritance, constructors of the parent class are inherited by the child class. However, the child class can also define its own constructor. When a child class object is instantiated, the constructor of the parent class is called first (implicitly or explicitly) before the child class constructor is executed.

- Implicit Call: In most languages like Python, if the child class doesn't define its constructor, the parent class constructor is called automatically.
- Explicit Call: If the child class has a constructor, it can explicitly call the parent class constructor using `super()` (in Python) or equivalent keywords in other languages.

Example:

class Animal:

```
def __init__(self, name):
    self.name = name
    print(f"Animal {self.name} created")
```

class Dog(Animal):

```
def __init__(self, name, breed):
    super().__init__(name) # Calling parent class constructor
    self.breed = breed
    print(f"Dog {self.breed} created")
```

```
d = Dog("Rex", "Bulldog")
```

Destructor in Inheritance:

A destructor is used to clean up resources when an object is destroyed. In inheritance, the destructor of the child class calls the parent class destructor (if defined) to ensure proper cleanup. If the child class doesn't define a destructor, the parent class destructor is automatically called.

Example:

```
class Animal:
```

```
    def __del__(self):  
        print("Animal destroyed")
```

```
class Dog(Animal):
```

```
    def __del__(self):  
        print("Dog destroyed")  
        super().__del__()
```

```
d = Dog()
```

```
del d # Destructor is called
```

9.3 Function Overriding

Function Overriding is a feature in object-oriented programming (OOP) that allows a subclass (child class) to provide a specific implementation of a method that is already defined in its superclass (parent class). This is done by defining a method in the child class with the same name, signature, and parameters as the method in the parent class, but with a different implementation.

The primary purpose of function overriding is to allow the child class to modify or extend the behavior of an inherited method, providing functionality that is more specific to the child class. When an object of the child class calls the overridden method, the child class's version of the method is executed, rather than the parent class's version.

Key Points:

- Method Signature: The method name, parameters, and return type in the child class must match those in the parent class.
- Polymorphism: Function overriding is a form of runtime polymorphism, where the method that gets called is determined at runtime, based on the object's type.
- super(): In some cases, the child class can call the parent class method using super() to retain the parent's functionality while adding or altering behavior.

Example:

```
class Animal:  
    def speak(self):  
        print("Animal speaks")
```

```
class Dog(Animal):  
    def speak(self):  
        print("Dog barks")
```

```
d = Dog()  
d.speak() # Calls Dog's overridden method
```

In this example, the speak() method is overridden in the Dog class, changing its behavior compared to the Animal class.

9.4 Virtual Base Class

A Virtual Base Class is a concept in C++ used to solve issues related to diamond inheritance (when a class inherits from multiple classes that share a common ancestor). Without virtual inheritance, a class could inherit multiple copies of the same base class, leading to redundancy and ambiguity.

By declaring the common base class as virtual, it ensures that only one instance of the base class is inherited, regardless of how many paths the inheritance takes. This avoids duplication of data members and methods from the base class.

Example:

```
class Base { ... };  
class Derived1 : virtual public Base { ... };  
class Derived2 : virtual public Base { ... };  
class Final : public Derived1, public Derived2 { ... };
```

9.5 Access Control in Inheritance:

Access Control in Inheritance refers to how the visibility and accessibility of members (variables and methods) of a parent class are managed in a child class. In object-oriented programming, access modifiers define the level of access for class members. The primary access control modifiers are:

1. Public: Members declared as public in the parent class are accessible in the child class and outside it, even if inherited.
2. Protected: Members declared as protected are accessible within the child class and its subclasses, but not outside the class hierarchy.
3. Private: Members declared as private are not accessible directly in the child class, though they can still be accessed through public or protected getter/setter methods.

In inheritance, access control affects how derived classes interact with base class members. For example, public inheritance allows public members to remain accessible in the derived class, while private inheritance hides the base class's public and protected members in the derived class.

Example:

class Animal:

```
def __init__(self):
```

```
    self.public = "Public"
```

```
    self._protected = "Protected"
```

```
    self.__private = "Private"
```

```
class Dog(Animal):
```

```
def __init__(self):
```

```
    super().__init__()
```

```
    print(self.public) # Accessible
```

```
    print(self._protected) # Accessible#
```

```
    print(self.__private) # Not accessible, raises AttributeError
```



10. Polymorphism in C++

10.1 Compile-Time Polymorphism (Function Overloading, Operator Overloading)

Compile-Time Polymorphism refers to the ability to resolve method calls at compile time, allowing a program to exhibit different behaviors based on the context. This type of polymorphism is achieved through Function Overloading and Operator Overloading, which enables flexibility in how functions and operators are used.

Function Overloading:

Function overloading occurs when multiple functions with the same name exist in a class but have different parameters (number or type). The compiler determines the correct function to call based on the argument types or the number of arguments passed. This allows different types of operations to be performed using the same function name.

Example (C++):

```
class Print {  
public:  
    void display(int i) { std::cout << "Integer: " << i << std::endl; }  
    void display(double d) { std::cout << "Double: " << d << std::endl; }  
};
```

In this example, the `display()` function is overloaded to handle both integer and double types.

Operator Overloading:

Operator overloading allows you to redefine the behavior of operators (like `+`, `-`, `*`) for user-defined objects. It enables objects to interact using familiar syntax, similar to built-in types. The compiler resolves which operator function to call based on the object types involved.



CODTECH IT SOLUTIONS PVT.LTD
IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

Example (C++):

```
class Complex {  
public:  
    int real, imag;  
    Complex operator + (const Complex& obj) {  
        return Complex(real + obj.real, imag + obj.imag);  
    }  
};
```

In this case, the `+` operator is overloaded to perform addition on `Complex` objects. Both function overloading and operator overloading are forms of compile-time polymorphism, where the correct method or operator is selected during compilation, improving code readability and flexibility.

10.2 Runtime Polymorphism (Virtual Functions, Abstract Classes, Pure Virtual Functions)

Runtime Polymorphism is a feature in object-oriented programming (OOP) that allows a function or method to behave differently based on the object calling it, determined at runtime. This enables flexibility and dynamic method resolution, which is particularly useful in scenarios involving inheritance. Runtime polymorphism is typically implemented through virtual functions, abstract classes, and pure virtual functions.

Virtual Functions:

Virtual functions are methods defined in a base class and are meant to be overridden in derived classes. When a virtual function is called through a base class pointer or reference, the derived class's version of the function is executed, even if the pointer/reference is of the base class type. This behavior is known as dynamic dispatch.

Example (C++):

```
class Animal {  
public:  
    virtual void sound() {  
        std::cout << "Animal sound" << std::endl;};
```



CODTECH IT SOLUTIONS PVT.LTD
IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

```
class Dog : public Animal {  
public:  
    void sound() override {  
        std::cout << "Bark" << std::endl;  
    }  
};
```

```
int main() {  
    Animal* animal = new Dog();  
    animal->sound(); // Outputs "Bark"  
}
```

Abstract Classes:

An abstract class cannot be instantiated directly and may include pure virtual functions. It serves as a blueprint for other classes.

Pure Virtual Functions:

A pure virtual function is a method in an abstract class that has no implementation and must be overridden by derived classes.

Example (C++):

```
class Shape {  
public:  
    virtual void draw() = 0; // Pure virtual function  
}
```

Runtime polymorphism allows for more flexible and scalable code by enabling behavior changes based on the object type at runtime, thus supporting dynamic method resolution.



11. Encapsulation and Abstraction

11.1 Data Hiding and Access Modifiers

Data Hiding is a key concept in object-oriented programming (OOP) that refers to the practice of restricting direct access to certain details of an object's internal state. This ensures that the object's data is protected from unintended or unauthorized modifications. The main goal of data hiding is to safeguard the object's integrity, improve maintainability, and reduce complexity by exposing only the necessary functionalities to the outside world.

To implement data hiding, access modifiers are used to control the visibility of class members (variables and methods). These modifiers define the level of access that other classes or objects have to a class's internal data. The common access modifiers are:

1. Public: Members declared as public are accessible from anywhere, both inside and outside the class.
2. Private: Members declared as private are only accessible within the class itself. They cannot be accessed directly from outside the class, which is the core of data hiding.
3. Protected: Members declared as protected are accessible within the class and by derived (child) classes, but not by other classes outside the hierarchy.
4. Default/Package (in some languages like Java): If no access modifier is specified, members are accessible only within the same package or module.

By using these access modifiers, developers can encapsulate the internal workings of an object and expose only the necessary interface to the outside world, promoting better security and code management.



11.2 Abstract Classes and Interfaces

Data Hiding is a key concept in object-oriented programming (OOP) that refers to the practice of restricting direct access to certain details of an object's internal state. This ensures that the object's data is protected from unintended or unauthorized modifications. The main goal of data hiding is to safeguard the object's integrity, improve maintainability, and reduce complexity by exposing only the necessary functionalities to the outside world.

To implement data hiding, access modifiers are used to control the visibility of class members (variables and methods). These modifiers define the level of access that other classes or objects have to a class's internal data. The common access modifiers are:

1. Public: Members declared as public are accessible from anywhere, both inside and outside the class.
2. Private: Members declared as private are only accessible within the class itself. They cannot be accessed directly from outside the class, which is the core of data hiding.
3. Protected: Members declared as protected are accessible within the class and by derived (child) classes, but not by other classes outside the hierarchy.
4. Default/Package (in some languages like Java): If no access modifier is specified, members are accessible only within the same package or module.

By using these access modifiers, developers can encapsulate the internal workings of an object and expose only the necessary interface to the outside world, promoting better security and code management.

You said:

Abstract Classes and Interfaces give 200 words on above topic

ChatGPT said:

Abstract Classes and Interfaces are fundamental concepts in object-oriented programming (OOP) used to define abstract behaviors that must be implemented by derived classes. They both provide a way to achieve abstraction, but they have distinct characteristics and use cases.

Abstract Classes:

An abstract class allows for partial implementation of its behavior and can provide some default functionality to its derived classes. It is used when a base class shares common functionality but also requires the child classes to implement specific behaviors.

Example (C++):

```
class Shape {  
public:  
    virtual void draw() = 0; // Abstract method  
    void display() {  
        std::cout << "Displaying shape" << std::endl;  
    }  
};
```

Interfaces:

An interface is a contract that a class must adhere to. It only contains abstract methods (no method implementations). A class that implements an interface must provide concrete implementations for all the methods declared in the interface. Interfaces are commonly used to define a common behavior across unrelated classes.

Example (Java):

```
java  
interface Drawable {  
    void draw(); // Abstract method  
}  
  
class Circle implements Drawable {  
    public void draw() {  
        System.out.println("Drawing Circle");  
    }  
}
```

Key Differences:

- Abstract Class: Can have both abstract and concrete methods, can have member variables.
- Interface: Can only have abstract methods (in languages like Java), and classes must implement all methods.



12.File Handling in C++

12.1 File Streams (ifstream, ofstream, fstream)

File Streams are a part of the C++ Standard Library used to handle file input and output (I/O) operations. File streams allow the program to read from and write to files, facilitating persistent storage of data. The primary file stream classes in C++ are ifstream, ofstream, and fstream, each serving different purposes.

1. ifstream:

The ifstream (input file stream) is used for reading data from files. It allows you to open a file and read its contents. The file is opened in read mode by default.

Example (ifstream):

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

int main() {
    ifstream infile("example.txt");
    string line;
    while (getline(infile, line)) {
        cout << line << endl;
    }
    infile.close();
    return 0;
}
```

2. ofstream:

The ofstream (output file stream) is used for writing data to files. It opens a file in write mode and creates a new file if it doesn't already exist. If the file exists, it is overwritten by default.

Example (ofstream):

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

int main() {
    ofstream outfile("output.txt");
    outfile << "Hello, world!" << endl;
    outfile.close();
    return 0;
}
```



3. fstream:

The fstream class is a combination of ifstream and ofstream, allowing both reading and writing to a file. It can open a file in both input and output modes.

Example (fstream):

```
#include <fstream>#include <iostream>using namespace std;
int main() {
    fstream file("example.txt", ios::in | ios::out);
    string content;
    file << "Writing data to the file." << endl;
    file.seekg(0); // Move to the beginning to readgetline(file, content);
    cout << content << endl;
    file.close();
    return 0;
}
```

In summary, ifstream, ofstream, and fstream are essential for performing file-based I/O in C++, allowing for flexible reading, writing, and manipulation of files.

12.2 Reading and Writing to Files

Reading and Writing to Files are essential operations in programming for handling persistent data. In C++, file I/O operations are facilitated by classes such as ifstream, ofstream, and fstream.

Reading from a File:

To read from a file, use the ifstream class, which opens the file in read mode. You can use functions like getline() to read line by line or the extraction operator (>>) to read word by word.

Example (Reading):

```
#include <fstream>#include <iostream>using namespace std;
int main() {
    ifstream infile("data.txt");
    string line;
    while (getline(infile, line)) {
        cout << line << endl; // Print each line
    }
    infile.close();
    return 0;
}
```



Writing to a File:

To write to a file, use the `ofstream` class, which opens the file in write mode. The file is overwritten by default. The `<<` operator is used to write data.

Example (Writing):

```
#include <fstream>#include <iostream>using namespace std;  
int main() {  
    ofstream outfile("output.txt");  
    outfile << "Hello, world!" << endl;  
    outfile.close();  
    return 0;  
}
```

Both reading and writing operations are performed by opening the file in the appropriate mode, ensuring data is transferred between the program and the file.

12.3 File Handling Modes

In C++, file handling modes are used to specify how a file should be opened (whether for reading, writing, or appending). These modes are passed as arguments to file stream objects (`ifstream`, `ofstream`, or `fstream`) to determine the type of file operation.

Common File Handling Modes:

1. `ios::in`: Opens a file for reading. If the file doesn't exist, the operation fails.
2. `ios::out`: Opens a file for writing. It creates a new file if it doesn't exist, or truncates (overwrites) the file if it does.
3. `ios::app`: Opens a file for appending. Data is written to the end of the file without modifying existing content.
4. `ios::ate`: Opens a file and moves the file pointer to the end. Data is written at the end, but unlike `app`, you can seek to other positions.
5. `ios::binary`: Opens a file in binary mode (default is text mode), used for reading/writing non-text data.

Example:

```
ofstream outfile("file.txt", ios::out | ios::app); // Open for writing and appending
```

By combining these modes, you control how files are accessed and manipulated during program execution.



12.4 File Pointers and Manipulators

File Pointers and Manipulators are important features in file handling that allow more control over reading and writing operations in C++.

File Pointers:

File pointers are used to track the current position in a file. They are automatically managed by the file stream objects, but you can manipulate them using functions like:

- `seekg(offset, direction)`: Moves the input file pointer (for reading) to a specific location. `offset` specifies how many bytes to move, and `direction` (such as `ios::beg`, `ios::cur`, or `ios::end`) defines the reference point.
- `seekp(offset, direction)`: Moves the output file pointer (for writing) in the same way.

Example:

```
ifstream infile("data.txt");
infile.seekg(0, ios::end); // Move pointer to the end of the file
```

Manipulators:

Manipulators are functions that modify the formatting of input or output in C++.

Common manipulators include:

- `endl`: Adds a newline and flushes the output buffer.
- `setw(width)`: Sets the width of the next output field.
- `setprecision(n)`: Sets the number of decimal places for floating-point output.

Example:

```
ofstream outfile("output.txt");
outfile << setw(10) << 123 << endl; // Writes number with width of 10
```

File pointers and manipulators enhance control over file and output formatting in C++.



13.Exception Handling in C++

13.1 Introduction to Exception Handling

Exception Handling in programming is a mechanism that allows a program to deal with unexpected events or errors during execution without crashing. Instead of the program terminating abruptly when an error occurs, exception handling provides a structured way to respond to errors, maintain control, and allow the program to continue or fail gracefully.

In C++, exception handling is managed using try, catch, and throw blocks:

- 1.try: A block of code that might generate an exception is placed inside a try block.
- 2.throw: If an error occurs inside the try block, an exception is thrown using the throw keyword.
- 3.catch: Once an exception is thrown, control is passed to the catch block, where specific actions (such as logging the error or providing user feedback) can be taken.

Example:

```
try {  
    int result = 10 / 0; // Error: division by zero  
} catch (const exception& e) {  
    cout << "Error: " << e.what() << endl;  
}
```

This approach enhances code robustness by managing runtime errors efficiently, improving program reliability.

13.2 Try, Catch, and Throw Mechanism

In C++, the try, catch, and throw mechanism is used for exception handling, allowing programs to handle errors gracefully without crashing. This mechanism involves three main components:

1. try block:

The try block contains the code that might throw an exception. If any error occurs within this block, the program will transfer control to the corresponding catch block.

2. throw keyword:

When an error is detected, the throw keyword is used to signal an exception. It can throw built-in exceptions (like int, std::exception) or user-defined ones.

3. catch block:

The catch block is used to catch and handle the exception thrown by the try block. It specifies the type of exception it can catch and contains code to manage the exception (e.g., logging the error, cleaning up resources).

Example:

```
try {  
    int result = 10 / 0; // Division by zero error  
} catch (const exception& e) {  
    cout << "Exception caught: " << e.what() << endl; // Handle error  
}
```

This mechanism allows developers to handle runtime errors more flexibly and maintain the program's flow even in case of exceptions.

13.3 Standard Exceptions (`std::exception`, `std::runtime_error`, etc.)

In C++, the Standard Library provides several predefined exceptions to handle common error conditions. These exceptions are part of the `<stdexcept>` header and are derived from the base class `std::exception`.

1. `std::exception`:

The base class for all standard exceptions. It defines a `what()` method that returns a description of the exception. This class is rarely thrown directly but serves as a foundation for other exceptions.

```
try {
    throw std::exception();
} catch (const std::exception& e) {
    std::cout << e.what() << std::endl; // Output: "std::exception"
}
```

2. `std::runtime_error`:

A subclass of `std::exception`, it represents errors that occur during program execution, like invalid operations or resource unavailability. It's commonly used for runtime errors that are outside the programmer's control.

```
throw std::runtime_error("Runtime error occurred");
```

3. `std::logic_error`:

Represents errors in logic, such as invalid arguments passed to a function.

Other exceptions include `std::out_of_range`, `std::invalid_argument`, and `std::bad_alloc`. These exceptions provide robust error handling, enabling better program stability and debugging.

13.4 User-Defined Exceptions

User-defined exceptions in C++ allow developers to create custom exception types that are specific to their application's needs. While C++ provides standard exceptions like std::runtime_error, user-defined exceptions provide more flexibility by allowing you to create exceptions that better reflect the unique errors your program may encounter.

To define a user-defined exception, you typically create a new class that inherits from the std::exception class or one of its derived classes, like std::runtime_error. You can then override the what() method to provide a custom error message.

Example:

```
#include <iostream>
#include <stdexcept>
class MyException : public std::exception {
public:
    const char* what() const noexcept override {
        return "My custom exception occurred!";
    }
};

int main() {
    try {
        throw MyException();
    } catch (const MyException& e) {
        std::cout << e.what() << std::endl; // Output: My custom exception occurred!
    }
    return 0;
}
```

In this example, MyException is a user-defined exception that provides a custom error message. User-defined exceptions help provide clearer, more context-specific error handling in complex programs.



14.Standard Template Library (STL)

14.1 Introduction to STL

The Standard Template Library (STL) in C++ is a powerful set of template classes and functions that provide general-purpose data structures and algorithms. It is designed to simplify programming by providing reusable components that handle common tasks like storing and managing data, sorting, and searching.

The STL consists of four key components:

1. Containers: Data structures that store collections of objects. Examples include vector, list, deque, map, set, and unordered_map.
2. Algorithms: Functions that perform operations on containers. These include searching, sorting, and modifying elements, such as sort(), find(), and reverse().
3. Iterators: Objects that allow traversal of elements in containers. Iterators provide a way to access container elements sequentially without exposing the underlying data structure.
4. Function Objects (Functors): Objects that can be called like functions. Functors are used with algorithms to define custom operations.

STL simplifies code by offering efficient and generic data structures and algorithms, promoting code reuse and reducing development time.

14.2 Containers (Vector, List, Deque, Stack, Queue, Set, Map)

In C++, containers are part of the Standard Template Library (STL) and are used to store and organize collections of objects. Each container type serves a different purpose, offering varying ways to store and manage data. Here are some common STL containers:

1. Vector:

A vector is a dynamic array that allows random access to its elements. It grows automatically when elements are added. It is ideal when frequent access to elements is needed, but it may be inefficient for insertions or deletions in the middle.

2. List:

A list is a doubly linked list, providing efficient insertions and deletions from both ends, but slower random access compared to a vector.

3. Deque:

A deque (double-ended queue) allows fast insertions and deletions at both ends, and random access to elements, making it a flexible option for many use cases.

4. Stack:

A stack is a container that follows the Last In, First Out (LIFO) principle. Elements can only be added or removed from the top.

5. Queue:

A queue follows the First In, First Out (FIFO) principle, where elements are added to the back and removed from the front.

6. Set:

A set is an ordered collection of unique elements. It automatically sorts elements and prevents duplicates.

7. Map:

A map is a collection of key-value pairs where keys are unique. It stores data in sorted order based on the keys.

Each container type in STL is optimized for different use cases, and choosing the right one can significantly improve program efficiency.

14.3 Iterators in STL

In the Standard Template Library (STL), iterators are objects that allow traversal through the elements of containers, providing a uniform way to access container elements regardless of the underlying structure. Iterators function similarly to pointers but are more flexible. They can be used with various containers like vector, list, map, etc.

There are several types of iterators:

- Input Iterator: Allows reading elements in one direction.
- Output Iterator: Used for writing elements.
- Forward Iterator: Allows reading and writing, only in one direction.
- Bidirectional Iterator: Allows movement both forward and backward.
- Random Access Iterator: Provides access to elements at any position with constant-time complexity.



14.4 Algorithms in STL

In the Standard Template Library (STL), algorithms are a collection of functions that perform operations on containers, such as searching, sorting, manipulating, and modifying elements. These algorithms are designed to be highly efficient and work with various container types, making them a powerful tool for developers.

Some key STL algorithms include:

1. Sorting: `std::sort()` and `std::stable_sort()` are used to arrange elements in ascending or descending order.
2. Searching: `std::find()` locates an element, while `std::binary_search()` searches in a sorted container.
3. Modifying: `std::reverse()` reverses the order of elements, and `std::fill()` assigns a specific value to a range of elements.
4. Counting: `std::count()` counts occurrences of a specific value in a container.
5. Merging: `std::merge()` merges two sorted ranges into a single sorted range.
6. Transforming: `std::transform()` applies a function to a range of elements, creating a new result.

Example:

```
std::vector<int> vec = {3, 1, 4, 1, 5};  
std::sort(vec.begin(), vec.end());
```

STL algorithms are generic and work with iterators, making them versatile across different container types like vector, list, and map.

14.5 Function Objects and Lambda Expressions

Function Objects (or Functors) and Lambda Expressions are advanced features in C++ that enhance the flexibility and expressiveness of the language, particularly when used with algorithms and other high-level operations.

Function Objects (Functors):

A function object is an object of a class or struct that overloads the operator(), allowing it to be used as a callable function.

They are more powerful than regular function pointers because they can maintain state, such as data members, between calls.

Example:

```
struct Add {  
    int value;  
    Add(int v) : value(v) {}  
    int operator()(int x) {  
        return x + value;  
    }  
};  
std::vector<int> vec = {1, 2, 3, 4};  
std::transform(vec.begin(), vec.end(), vec.begin(), Add(5)); // Adds 5 to each element
```

Lambda Expressions:

A lambda expression is an anonymous function that can be defined inline. It allows for more concise, flexible, and readable code, especially when used with STL algorithms. Lambdas can capture variables from the surrounding scope by value or reference and can be passed directly into functions.

Example:

```
std::vector<int> vec = {1, 2, 3, 4};  
std::for_each(vec.begin(), vec.end(), [](int x) { std::cout << x << " "; });  
// Output: 1 2 3 4
```

Both functors and lambdas allow custom logic in STL algorithms, making them ideal for writing clean, efficient, and flexible C++ code.



15.Multithreading in C++

15.1 Introduction to Multithreading

Multithreading is a programming technique that allows a program to execute multiple threads concurrently, enabling better utilization of CPU resources and improving performance, especially in modern multi-core processors. Each thread represents an independent unit of execution, and multithreading allows these threads to run in parallel, handling different tasks simultaneously.

In C++, multithreading is facilitated by the `<thread>` header introduced in C++11. Threads are created using the `std::thread` class, and each thread can execute a function or callable object. Multithreading is commonly used to perform tasks like data processing, I/O operations, or background computations without blocking the main program flow.

However, multithreading also introduces complexity, such as the need for synchronization between threads to avoid race conditions and ensure safe access to shared data. Tools like mutexes, locks, and condition variables are used to manage synchronization and prevent conflicts between threads.

Example:

```
#include <iostream>
#include <thread>
void printHello() {
    std::cout << "Hello from thread!" << std::endl;
}
int main() {
    std::thread t(printHello); // Start a new thread
    t.join(); // Wait for the thread to finish
    return 0;
}
```

Multithreading improves performance by enabling concurrent execution, but it requires careful management of shared resources and synchronization.



15.2 Creating Threads using std::thread

In C++, the `std::thread` class, introduced in C++11, provides a simple and powerful way to create and manage threads. It allows programs to run multiple functions or tasks concurrently, leveraging the power of multi-core processors.

Creating Threads:

To create a thread, you instantiate an object of the `std::thread` class, passing the function (or callable object) to be executed as a parameter. The function will then be executed in a separate thread of execution.

Basic Example:

```
#include <iostream>
#include <thread>

void hello() {
    std::cout << "Hello from thread!" << std::endl;
}

int main() {
    std::thread t(hello); // Create and run a new thread
    t.join();            // Wait for the thread to complete
    return 0;
}
```

In this example, a thread is created using `std::thread t(hello);`, which runs the `hello` function in a new thread. The `join()` function is called to ensure that the main thread waits for the new thread to complete before continuing.

Passing Arguments to Threads:

You can pass arguments to a thread function by providing them as additional parameters when creating the `std::thread` object.

```
void printMessage(const std::string& msg) {
    std::cout << msg << std::endl;
}

int main() {
    std::string message = "Hello, thread!";
    std::thread t(printMessage, message);
    t.join();
    return 0;
}
```



15.3 Thread Synchronization (Mutex, Locks)

Thread synchronization is crucial in multithreaded programming to ensure that threads do not interfere with each other when accessing shared resources, leading to issues like race conditions. In C++, synchronization is achieved through mechanisms like mutexes and locks, which manage access to shared data and ensure that only one thread can modify or access the resource at a time.

Mutex:

A mutex (short for mutual exclusion) is a synchronization primitive used to protect shared resources. When a thread locks a mutex, other threads attempting to lock it are blocked until the mutex is unlocked. This ensures exclusive access to the resource.

Example:

```
#include <iostream>#include <thread>#include <mutex>
std::mutex mtx;
void printMessage(int i) {
    mtx.lock(); // Lock the mutex to protect shared resource
    std::cout << "Message " << i << std::endl;
    mtx.unlock(); // Unlock the mutex after accessing the resource
}
int main() {
    std::thread t1(printMessage, 1);
    std::thread t2(printMessage, 2);
    t1.join();
    t2.join();
    return 0;
}
```

Locks:

Locks provide a safer and more flexible way to work with mutexes. `std::lock_guard` and `std::unique_lock` are lock classes that automatically acquire and release locks, avoiding potential deadlocks or manual unlock errors.



15.4 Condition Variables

Condition variables in C++ are synchronization primitives used to block a thread until a specific condition is met. They are useful for scenarios where threads need to wait for a certain condition to change (e.g., when waiting for data to be processed or when a resource becomes available).

Condition variables are typically used in conjunction with mutexes to ensure that the thread waiting for a condition is properly synchronized with the shared data.

Key functions:

1. `wait()`: Blocks the current thread until notified by another thread.
2. `notify_one()`: Wakes up one thread that is waiting on the condition variable.
3. `notify_all()`: Wakes up all threads waiting on the condition variable.

Example:

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void print_id(int id) {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [] { return ready; });
    std::cout << "Thread " << id << std::endl;
}

int main() {
    std::thread threads[10];
    for (int i = 0; i < 10; ++i)
        threads[i] = std::thread(print_id, i);
    std::cout << "10 threads ready to print..." << std::endl;
    ready = true;
    cv.notify_all();
    for (auto& t : threads)
        t.join();
    return 0;
}
```

Condition variables help synchronize threads by allowing them to wait and wake up based on certain conditions, making them essential for managing thread dependencies.



CODTECH IT SOLUTIONS PVT.LTD
IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

16.Templates in C++

16.1 Function Templates

Function templates in C++ allow you to write a single function definition that can operate on different data types. Instead of creating multiple functions for different data types, a function template allows you to define a generic function that works with any type provided at compile-time. This is especially useful for writing reusable and flexible code.

A function template is declared using the `template` keyword followed by the type parameter, which acts as a placeholder for the data type.

Example:

```
#include <iostream>using namespace std;
template <typename T>
T add(T a, T b) {
    return a + b;
}
int main() {
    cout << add(5, 3) << endl;      // Works with int
    cout << add(3.5, 2.5) << endl;  // Works with double
    return 0;
}
```

In this example, the `add()` function is defined as a template that can handle any data type (like `int`, `double`, etc.). When called, the compiler automatically deduces the correct type based on the arguments passed. Function templates provide a powerful way to write type-independent functions while maintaining type safety.

16.2 Class Templates

Class templates in C++ allow you to create a blueprint for a class that can work with any data type. Instead of defining separate classes for different data types, class templates enable the creation of generic classes that can operate on any data type provided at compile-time. This enhances code reusability and flexibility. A class template is defined using the template keyword, followed by a type parameter that serves as a placeholder for the data type.

Example:

```
#include <iostream>using namespace std;
template <typename T>
class Box {
private:
    T value;
public:
    Box(T v) : value(v) {}
    T getValue() {
        return value;
    }
};
int main() {
    Box<int> intBox(5);
    Box<double> doubleBox(3.14);
    cout << intBox.getValue() << endl; // Output: 5
    cout << doubleBox.getValue() << endl; // Output: 3.14return 0;
}
```

In this example, the Box class is a template that can store any data type, such as int or double. Class templates provide a convenient way to design type-safe, reusable classes without repeating code.

16.3 Variadic Templates

Variadic templates in C++ are a powerful feature that allows functions or classes to accept a variable number of arguments. Introduced in C++11, variadic templates make it possible to define templates that can handle an arbitrary number of template parameters, enabling more flexible and generic code. Variadic templates use template parameter packs to represent the variable arguments. This allows you to define functions or classes that can accept any number of arguments, reducing the need for overloads or manual handling of different argument types.

Example of a variadic template function:

```
#include <iostream>using namespace std;
template <typename... Args>
void print(Args... args) {
    (cout << ... << args) << endl; // Fold expression for printing all arguments
}

int main() {
    print(1, 2.5, "Hello", true); // Output: 12.5Hello1return 0;
}
```

In this example, `print()` is a variadic template function that can accept any number and types of arguments. The `Args...` parameter pack collects all the arguments, and a fold expression is used to output them. Variadic templates simplify code that needs to handle variable argument lists.

This material is for reference to gain basic knowledge : don't rely solely on it, and also refer to other internet resources for competitive exams. Thank you from CodTech.

