1. What are the key differences between Procedural Programming and Object Oriented Programming (OOP)?

=>

|  |  |  |
| --- | --- | --- |
| Key | Procedural Programming | Object-Oriented Programming (OOP) |
| Structure | Programs are divided into functions or procedures. | Programs are divided into classes and objects. |
| Data and Functions | Data and functions are separate; functions operate on external data. | Data and methods (functions) are encapsulated within objects. |
| Reusability | Limited reusability; functions can be reused within a program. | High reusability through inheritance, polymorphism, and encapsulation. |
| Inheritance | No inheritance concept. | Supports inheritance, allowing code reuse between classes. |
| Code Maintenance | Complex | Easy |
| Examples | C | Java, C++, Python, C#, Ruby. |

2. List and explain the main advantages of OOP over POP

=> It has following Advantages: -

1. **Code reusability**: OOP allows developers to reuse code blocks, which can improve efficiency.

2. **Modularity**: OOP breaks down code into smaller, more manageable pieces, which makes it easier to maintain, update, and add new features.

3. **Security**: OOP supports access control and data hiding, making it more secure than POP

4. **Code maintenance**: OOP makes it easier to maintain code, and developers can create new classes based on existing ones.

5. **Code flexibility**: OOP makes code more flexible and adaptable.

6. **Clean code**: OOP uses classes and objects to make code clean which can help others understand it more easily.

3. Explain the steps involved in setting up a C++ development environment.

=> Step 1: Install a C++ Compiler: - For Windows there is Microsoft Visual C++ Compiler for 64 bit.

Step 2: Install IDE: - There is popular Editor is visual Studio. Download Visual Studio Code for 64 bit.

Step 3: Configure IDE: - there is few Steps: -

1 Create a tasks.json file to define build tasks:

* Press Ctrl + Shift + P, type "Tasks: Configure Task" → Select C/C++: g++.exe build active file.

2 Create a launch.json file for debugging.

3 Ensure your compiler path is set in the C/C++ extension settings.

Step 4: Write a Sample Code and Run

#include <iostream>

int main() {

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

return 0;

}

Then Click on Run to run the Program.

4. What are the main input/output operations in C++? Provide examples.

=> In C++, input and output operations are typically performed using the Standard Input/Output (I/O) Library, which includes streams such as cin for input and cout for output.

The I/O operations are facilitated by the iostream library.

1. Standard Output (cout): - It is used to display data on the console.

The insertion operator (<<) is used with cout to send data to the output stream.

e.g.

#include <iostream>

using namespace std;

int main() {

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

cout << "The result is: " << 42 << endl;

return 0;

}

Output: -

Hello, World!

The result is: 42

1. Standard input (cin): - It is used to read data from the console.

The extraction operator (>>) is used with cin to take input.

e.g.

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age;

cout << "You entered: " << age << endl;

return 0;

}

Input: 25

Output: - You entered: 25

1. Standard Error (cerr): - It is used to display error messages.

e.g.

#include <iostream>

using namespace std;

int main() {

cerr << "An error occurred!" << endl;

return 1;

}

Output: - An error occurred!

1. Standard Log (clog): - It is used to output log messages.

It is **buffered**, meaning it may delay output until the buffer is flushed.

e.g.

#include <iostream>

using namespace std;

int main() {

clog << "This is a log message." << endl;

return 0;

}

Output: - This is a log message.

1. File Input/Output: - It is used for reading from and writing to files, and this is provided by fstream library.
2. ifstream for reading files.
3. ofstream for writing files.
4. fstream for both reading and writing.

5. What are the different data types available in C++? Explain with examples.

=> It is of 3 types: -

1. Primitive Data Types: - int(4 bytes), float(4 bytes), double(8 bytes), long double(10-16 bytes), char(1 byte), bool(1 byte)

e.g.

#include <iostream>

using namespace std;

int main() {

int a = 42;

float num1 = 3.14f;

double num2 = 2.71828;

long double num3 = 1.41421356237L;

char letter = 'B';

bool isComplete = false;

cout << "a: " << a << , << "num1: " << num1 << ", num2: " << num2

<< ", num3: " << num3 << , << "Letter: " << letter << ,

<< "Task complete? " << (isComplete ? "Yes" : "No") << endl;

return 0;

}

2. Derived Date Types : -

1. Arrays: - It is a collection of elements of the same data type.

e.g.

#include <iostream>

using namespace std;

int main() {

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

cout << "First element: " << numbers[0] << endl;

return 0;

}

2. Pointers: - It is a variables that store memory addresses.

e.g.

#include <iostream>

using namespace std;

int main() {

int x = 10;

int \*ptr = &x;

cout << "Value: " << \*ptr << endl;

return 0;

}

3. User-Defined Data Types: - It allow for creating complex data types.

1. Structures (struct): - It combines different types of data.

e.g.

#include <iostream>

using namespace std;

struct Person {

string name;

int age;

};

int main() {

Person p = {"Alice", 30};

cout << p.name << " is " << p.age << " years old." << endl;

return 0;

}

2. Classes: - It is the blueprints for creating objects.

e.g.

#include <iostream>

using namespace std;

class Car {

public:

string brand;

void drive() {

cout << "Driving " << brand << endl;

}

};

int main() {

Car myCar;

myCar.brand = "Toyota";

myCar.drive();

return 0;

}

6. Explain the difference between implicit and explicit type conversion in C++.

=> In C++, type conversion means to changing the data type of a variable or expression to another type.

It is of 2 types: -

1. Implicit Type (Type Promotion): - Implicit type conversion happens automatically when the compiler converts one data type to another without explicit instruction from the programmer.

It typically occurs when performing operations between different data types or assigning a value to a variable of a different type.

e.g.

#include <iostream>

using namespace std;

int main() {

int intVal = 10;

double doubleVal = intVal; // Implicit conversion from int to double

cout << "Integer: " << intVal << endl;

cout << "Double: " << doubleVal << endl; // Output: 10.0

double result = 5 / 2; // Integer division, implicitly converts to double

cout << "Result: " << result << endl; // Output: 2.0 (due to integer division)

return 0;

}

1. Explicit Type (Type Casting): - Explicit type conversion requires the programmer to manually specify the desired type conversion.

It is used when automatic (implicit) conversion doesn't happen or when precision loss is acceptable.

There are 2 methods: -

1. C-style Casting: - syntax is (type) expression
2. C++ static\_cast Operator: - static\_cast<type>(expression)

e.g.

#include <iostream>

using namespace std;

int main() {

double pi = 3.14159;

int piAsInt = (int)pi; // C-style cast

int piAsInt2 = static\_cast<int>(pi); // C++ style cast

cout << "Original double: " << pi << endl;

cout << "Converted to int (C-style): " << piAsInt << endl;

cout << "Converted to int (C++ style): " << piAsInt2 << endl;

int a = 5, b = 2;

double result = static\_cast<double>(a) / b; // Explicit conversion to double

cout << "Result of division: " << result << endl; // Output: 2.5

return 0;

}

7. What are the different types of operators in C++? Provide examples of each.

=> There are different types of operators: -

1. Arithmetic Operators: - Operators are Addition +, Subtraction -,

Multiplication \*, Division /, Modulus %

e.g.

#include <iostream>

using namespace std;

int main() {

int a = 10, b = 3;

cout << "Addition: " << a + b << endl;

cout << "Subtraction: " << a - b << endl;

cout << "Multiplication: " << a \* b << endl;

cout << "Division: " << a / b << endl;

cout << "Modulus: " << a % b << endl;

return 0;

}

2. Relational Operators: - These operators compare values and return a boolean result (true or false).

Operators are Equal to (==), Not Equal to (!=), Greater than (>), Smaller than (<). Greater than or equal to (>=), Smaller than or equal to (<=).

e.g.

#include <iostream>

using namespace std;\

int main() {

int a = 5, b = 10;

cout << (a == b) << endl; // Output: 0 (false)

cout << (a < b) << endl; // Output: 1 (true)

return 0;

}

3. Logical Operators: - These operators perform logical operations and return boolean results.

Operators are Logical AND (&&), Logical OR (||) , Logical NOT (!).

e.g.

#include <iostream>

using namespace std;

int main() {

bool a = true, b = false;

cout << (a && b) << endl; // Output: 0 (false)

cout << (a || b) << endl; // Output: 1 (true)

cout << !a << endl; // Output: 0 (false)

return 0;

}

4. Bitwise Operators: - These operators perform bit-level operations.

Operators are Bitwise AND (&), Bitwise OR (`), Bitwise XOR (^), Bitwise NOT (~), Left shift (<<), Right shift (>>).

e.g.

#include <iostream>

using namespace std;

int main() {

int a = 5; // Binary: 0101

int b = 3; // Binary: 0011

cout << (a & b) << endl; // Output: 1 (Binary: 0001)

cout << (a | b) << endl; // Output: 7 (Binary: 0111)

cout << (a ^ b) << endl; // Output: 6 (Binary: 0110)

return 0;

}

5. Assignment Operators: - These operators assign values to variables.

Operators are Assign (=), Add and assign (+=) , Subtrat and assign (-=), Multiply and assign (\*=) . Divide and assign (/=) , Modulus and assign (%=).

e.g.

#include <iostream>

using namespace std;

int main() {

int a = 10;

a += 5; // Equivalent to a = a + 5

cout << a << endl; // Output: 15

return 0;

}

6. . Unary Operators: - These operators operate on a single operand.

Operators are Increment (++) , Decrement (--) , Unary minus (-) , Unary plus (+).

e.g.

#include <iostream>

using namespace std;

int main() {

int a = 5;

cout << ++a << endl; // Output: 6 (pre-increment)

cout << a-- << endl; // Output: 6 (post-decrement)

cout << a << endl; // Output: 5

return 0;

}

7. Ternary Operator: - It is the shortcut of If-else Statement. Operators are ?:

e.g.

#include <iostream>

using namespace std;

int main() {

int a = 10, b = 20;

int max = (a > b) ? a : b;

cout << "Max: " << max << endl; // Output: 20

return 0;

}

8. Explain the purpose and use of constants and literals in C++.

=> In C++, constants and literals play an important role in ensuring the reliability and maintainability of your code.

They are used to represent fixed values that do not change during the execution of a program.

1. Constants: - Constants are fixed values that cannot be altered once defined. There are many ways to define constants: -

* Using const Keyword: -

#include <iostream>

using namespace std;

int main() {

const double PI = 3.14159;

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

// PI = 3.14; // Error: Cannot modify a const variable

return 0;

}

* Using #define Preprocessor Directive: -

#include <iostream>

using namespace std;

#define PI 3.14159

int main() {

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

// PI = 3.14; // Error: Macros cannot be reassigned

return 0;

}

1. Literals: - Literals are fixed values that appear directly in the code. They represent constant data like numbers, characters, and strings.

Types of Literals are as follows: -

* Integer Literals: - Represent whole numbers.

e.g.

int a = 42; // Decimal literal

int b = 0x2A; // Hexadecimal literal (0x or 0X prefix)

int c = 052; // Octal literal (0 prefix)

int d = 0b101010; // Binary literal (0b or 0B prefix)

* Floating-Point Literals: - Represent numbers with a fractional part.

e.g.

double pi = 3.14159;

float e = 2.71f; // 'f' or 'F' suffix for float

double large = 2.5e10; // Scientific notation

* Character Literals: - Represent single characters enclosed in single quotes.

e.g.

char letter = 'A';

char newline = '\n'; // Escape sequence for new line

* String Literals: - Represent sequences of characters enclosed in double quotes.

e.g.

const char\* greeting = "Hello, World!";

* Boolean Literals: - Represent true or false.

e.g.

bool flag = true;

9. What are conditional statements in C++? Explain the if-else and switch statements.

=> Conditional statements in C++ allow the program to make decisions and execute specific blocks of code based on certain conditions.

* if Statement: - Executes a block if the condition is true, otherwise does nothing.

Eg: #include <iostream>

using namespace std;

int main() {

int num;

cout << "Enter a number: ";

cin >> num;

if (num % 2 == 0) {

cout << num << " is even." << endl; // Executes if condition is true

}

return 0;

}

* if-else Statement: - The if-else statement is used to execute a block of code based on whether a condition is true or false.

Eg: #include <iostream>

using namespace std;

int main() {

int num;

cout << "Enter a number: ";

cin >> num;

if (num % 2 == 0) {

cout << num << " is even." << endl; // Executes if condition is true

} else {

cout << num << " is odd." << endl; // Executes if condition is false

}

return 0;

}

* Nested if-else: - Allows checking multiple conditions.

Eg: #include <iostream>

using namespace std;

int main() {

int marks;

cout << "Enter the marks: ";

cin >> marks;

if (marks >= 90) {

cout << "Grade A";

} else if (marks >= 80) {

cout << "Grade B";

} else {

cout << "Grade C";

}

return 0;

}

* switch Statement: - The switch statement is used for multi-way branching based on a variable's value. It is an alternative to the else-if ladder or nested if-else.

Eg: #include <iostream>

using namespace std;

int main() {

int choice;

cout << "Enter a number (1-3): ";

cin >> choice;

switch (choice) {

case 1:

cout << "You chose One." << endl;

break;

case 2:

cout << "You chose Two." << endl;

break;

case 3:

cout << "You chose Three." << endl;

break;

default:

cout << "Invalid choice!" << endl;

}

return 0;

}

10. What is the difference between for, while, and do-while loops in C++?

=> In C++, loops are used to execute a block of code repeatedly based on a condition.

* For Loop: - The for loop is used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and update. It is suitable for count-based iterations.

Eg: #include <iostream>

using namespace std;

int main() {

// For loop

cout << "For loop: ";

for (int i = 1; i <= 3; i++) {

cout << i << " ";

}

return 0;

}

* While Loop: - The while loop is used when the number of iterations is not known in advance. It executes as long as the condition is true.It is suitable for condition-based iterations. Checks the condition before executing the code block.

Eg: #include <iostream>

using namespace std;

int main() {

// While loop

cout << "\nWhile loop: ";

int j = 1;

while (j <= 3) {

cout << j << " ";

j++;

}

return 0;

}

* Do-while Loop: - The do-while loop is similar to the while loop, but it always executes at least once, even if the condition is false. It is suitable when the code must execute at least once. Checks the condition after executing the code block.

Eg: #include <iostream>

using namespace std;

int main() {

// Do-while loop

cout << "\nDo-while loop: ";

int k = 1;

do {

cout << k << " ";

k++;

} while (k <= 3);

return 0;

}

11. How are break and continue statements used in loops? Provide examples.

=> In C++, break and continue statements are used to alter the flow of loops.

* Break statement: - The break statement is used to terminate the loop immediately and exit. It is commonly used when a specific condition is met, and further iterations are not required.

Eg: #include <iostream>

using namespace std;

int main() {

for (int i = 1; i <= 10; i++) {

if (i == 6) {

break; // Exit the loop when i equals 6

}

cout << i << " ";

}

return 0;

}

* Continue statement: - The continue statement is used to skip the current iteration of the loop and move to the next iteration. It is used when certain conditions are met, and we need to skip processing for that iteration but continue looping.

Eg: #include <iostream>

using namespace std;

int main() {

for (int i = 1; i <= 10; i++) {

if (i % 2 == 0) {

continue; // Skip even numbers

}

cout << i << " ";

}

return 0;

}

12. What is a function in C++? Explain the concept of function declaration, definition, and calling.

=> A function in C++ is a block of reusable code that performs a specific task. Functions make programs modular, easier to read, and reduce code duplication.

=> Types of Functions: -

* Built-In Functions: - These are predefined in libraries like sqrt(), pow(), cin, cout.
* User-defined Functions**:** Created by the programmer for specific tasks.

=> **Function Declaration: -** A function declaration tells the compiler about the function name, return type, and parameters (if any) before its actual definition.  
It is usually written before the main() function.

Eg: int add(int a, int b);

where **int** → Return type of the function.

**add** → Function name.

**int a, int b** → Parameters passed to the function.

=> **Function Definition:** - A function definition contains the actual implementation of the function. It specifies what the function does when called.

Eg: int add(int a, int b) { // Function definition

return a + b; // Adds two numbers and returns the result

}

=> **Function Call: -** A function call is used to execute the function. It passes required values (arguments) to the function and retrieves the return value (if any).

Eg: int result = add(5, 3); // Function call with arguments 5 and 3

**Complete Example: -**

#include <iostream>

using namespace std;

// Function declaration

int add(int a, int b);

int main() {

int x, y;

cout << "Enter two numbers: ";

cin >> x >> y;

// Function call

int result = add(x, y);

cout << "Sum = " << result << endl;

return 0;

}

// Function definition

int add(int a, int b) {

return a + b; // Returns sum of a and b

}

13. What is the scope of variables in C++? Differentiate between local and global scope

=> The scope of a variable in C++ defines the region or part of the program where the variable is accessible or visible. It determines whether a variable can be accessed inside or outside a particular block of code.

**Types of Scope in C++:**

1. Local Scope
2. Global Scope

1. Local Scope: -

* A local variable is declared inside a function or block (e.g., within { }).
* It is accessible only within that function or block where it is declared.
* The variable gets created when the block is entered and is destroyed when the block exits.

Eg: #include <iostream>

using namespace std;

void display() {

int x = 10; // Local variable

cout << "Inside display: " << x << endl;

}

int main() {

// cout << x; // Error: x is not accessible here

display(); // Valid

return 0;

}

Output: - Inside display: 10

2. Global Scope**: -**

* A global variable is declared outside all functions (usually at the top).
* It is accessible from any function throughout the program.
* Its lifetime extends until the program terminates.

Eg: #include <iostream>

using namespace std;

int x = 20; // Global variable

void display() {

cout << "Global variable inside display: " << x << endl;

}

int main() {

cout << "Global variable in main: " << x << endl;

display(); // Access in another function

return 0;

}

Output: - Global variable in main: 20

Global variable inside display: 20

14. Explain recursion in C++ with an example.

=> Recursion is a programming technique where a function calls itself either directly or indirectly to solve a problem. It is commonly used to solve problems that can be divided into smaller, similar subproblems.

**=>** Key Components of Recursion:

1. Base Case:
   * The condition that terminates recursion.
   * It prevents infinite function calls.
2. Recursive Case:
   * The part of the function where it calls itself to solve a smaller subproblem.

=> Advantages of Recursion: -

1. Simplifies complex problems
2. Readable Code

=> Disadvantage of recursion: -

1. Memory Overhead
2. Slower Execution
3. Difficult Debugging

Eg: #include <iostream>

using namespace std;

// Recursive function to calculate factorial

int factorial(int n) {

if (n == 0 || n == 1) { // Base Case

return 1;

} else { // Recursive Case

return n \* factorial(n - 1);

}

}

int main() {

int num;

cout << "Enter a positive number: ";

cin >> num;

if (num < 0) {

cout << "Factorial is not defined for negative numbers!" << endl;

} else {

cout << "Factorial of " << num << " is: " << factorial(num) << endl;

}

return 0;

}

Output: - Enter a positive number: 5

Factorial of 5 is: 120

15. What are function prototypes in C++? Why are they used?

=> A function prototype in C++ is a declaration of a function that specifies its return type, name, and parameters without providing the actual implementation. It serves as a forward declaration and is typically written before the main() function or in a header file.

Syntax: - returnType functionName(parameterType1, parameterType2, ...);

Eg: - int add(int, int); // Function prototype

=> Functions Prototype used for: -

1 Forward Declaration:

* Allows defining a function after the main() function while letting the compiler recognize its usage before its definition.

2 Type Checking:

* Ensures type safety by verifying the types of arguments passed during function calls.

3 Separation of Declaration and Definition:

* Helps organize code in multiple files or header files for better modularity and readability.

4 Ease of Use in Libraries:

* Allows reusability and simplifies the inclusion of predefined functions in libraries.

Eg: - #include <iostream>

using namespace std;

// Function prototype

int add(int, int);

int main() {

int result = add(5, 3); // Valid: Compiler knows about 'add'

cout << "Sum = " << result << endl;

return 0;

}

// Function definition

int add(int a, int b) {

return a + b;

}

Output: - Sum = 8

16. What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays

=> An array in C++ is a collection of elements of the same data type stored in memory locations. It allows storing multiple values under a single variable name, making data management easier.

=> Types of Arrays: -

1. Single-Dimensional Array

* It is a list-like structure that stores elements in a linear sequence.
* Accessed using index numbers starting from 0.

Eg: - #include <iostream>

using namespace std;

int main() {

int numbers[5] = {10, 20, 30, 40, 50}; // Single-dimensional array

cout << "Elements of the array:" << endl;

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

cout << numbers[i] << " "; // Accessing elements using indices

}

return 0;

}

Output: - Elements of the array:

10 20 30 40 50

2. Multi-Dimensional Array

* A multi-dimensional array can store data in tabular form (rows and columns).
* The most common type is a 2-dimensional array (matrix), but arrays can have more dimensions.

Eg: #include <iostream>

using namespace std;

int main() {

int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} }; // 2 rows, 3 columns

cout << "Elements of 2D array:" << endl;

for (int i = 0; i < 2; i++) { // Rows

for (int j = 0; j < 3; j++) { // Columns

cout << matrix[i][j] << " "; // Access using two indices

}

cout << endl;

}

return 0;

}

Output: - Elements of 2D array:

1 2 3

4 5 6

17. Explain string handling in C++ with examples.

=> A string in C++ is a sequence of characters used to represent textual data. C++ provides two main approaches for string handling:

1. C-style Strings (Character Arrays): - C-style strings are implemented as character arrays and terminated with a null character (\0).

Eg: #include <iostream>

#include <cstring> // For string handling functions

using namespace std;

int main() {

char str1[20] = "Hello";

char str2[20] = "World";

// Concatenation

strcat(str1, str2);

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

// String length

cout << "Length of String: " << strlen(str1) << endl;

// String copy

strcpy(str1, str2);

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

// String comparison

if (strcmp(str1, str2) == 0)

cout << "Strings are equal." << endl;

else

cout << "Strings are not equal." << endl;

return 0;

}

Output: - Concatenated String: HelloWorld

Length of String: 10

Copied String: World

Strings are equal.

1. C++ Strings (Standard Template Library string class): - C++ provides a string class in the Standard Template Library (STL), which is more flexible and easier to use compared to C-style strings. It automatically manages memory allocation and provides many built-in functions.

Eg: #include <iostream>

#include <string> // For C++ string class

using namespace std;

int main() {

string str1 = "Hello"; // Declaration

string str2 = "World";

// Concatenation

string result = str1 + " " + str2;

cout << "Concatenated String: " << result << endl;

// Length of string

cout << "Length of String: " << result.length() << endl;

// Substring

cout << "Substring: " << result.substr(0, 5) << endl;

// Find a character or substring

cout << "Position of 'World': " << result.find("World") << endl;

// Replace a substring

result.replace(6, 5, "Universe");

cout << "Replaced String: " << result << endl;

return 0;

}

Output: - Concatenated String: Hello World

Length of String: 11

Substring: Hello

Position of 'World': 6

Replaced String: Hello Universe

18. How are arrays initialized in C++? Provide examples of both 1D and 2D arrays

=> Arrays in C++ can be initialized during declaration or assigned values later. They can be 1-dimensional (1D) or 2-dimensional (2D) based on their structure.

=> Initialization of 1D Arrays: - A 1D array stores elements in a linear sequence.

* Direct Initialization: -

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

Output: - 1 2 3

4 5 6

* Row-wise Initialization: -

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

Output: - 1 2 3

4 5 6

* Partial Initialization: -

Eg: int matrix[2][3] = { {1, 2}, {3} }; // Remaining values set to 0

Output: - 1 2 0

3 0 0

* Default Initialization: -

Eg: int matrix[2][3] = {}; // All elements initialized to 0

* Dynamic Assignment: -

Eg: int matrix[2][3];

matrix[0][0] = 1; // Assigning values individually

matrix[0][1] = 2;

matrix[1][2] = 5;

=> 1D Array Example: -

Eg: - #include <iostream>

using namespace std;

int main() {

int numbers[5] = {10, 20, 30, 40, 50}; // Initialization

cout << "1D Array Elements:" << endl;

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

cout << numbers[i] << " "; // Access elements using index

}

return 0;

}

Output: -

1D Array Elements:

10 20 30 40 50

=> 2D Array Example:

Eg: - #include <iostream>

using namespace std;

int main() {

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

cout << "2D Array Elements:" << endl;

for (int i = 0; i < 2; i++) { // Rows

for (int j = 0; j < 3; j++) { // Columns

cout << matrix[i][j] << " ";

}

cout << endl; // Move to next row

}

return 0;

}

Output: -

2D Array Elements:

1 2 3

4 5 6

19. Explain string operations and functions in C++.

=> In C++, **strings** are used to handle **text data**, and operations can be performed using:

1. **C-style strings** (character arrays)
2. **C++ string class** (from the Standard Template Library - STL)

**1. String Operations with C-style Strings (char[]): -**

C-style strings are arrays of characters terminated with a null character (\0). The cstring library provides functions to manipulate these strings.

Common Operations:

1. Copy a String (strcpy): -

Eg: char str1[20], str2[20] = "Hello";

strcpy(str1, str2); // Copies str2 into str1

1. Concatenate Strings (strcat): -

Eg: char str1[20] = "Hello ";

char str2[] = "World!";

strcat(str1, str2); // Appends str2 to str1

1. Find Length (strlen): -

Eg: char str[] = "Hello";

cout << strlen(str); // Output: 5

1. Compare Strings (strcmp): -

Eg: char str1[] = "Apple";

char str2[] = "Banana";

if (strcmp(str1, str2) == 0)

cout << "Equal";

else

cout << "Not Equal";

1. Find Substring (strstr): -

Eg: char str[] = "Hello World";

char \*ptr = strstr(str, "World"); // Finds the substring

**2. String Operations with C++ string Class: -**

C++ provides a string class in the <string> header, which is easier to use and supports dynamic memory allocation. It includes several member functions for string manipulation.

1. Initialization and Assignment: -

Eg: string str1 = "Hello"; // Initialization

string str2; // Declaration

str2 = "World"; // Assignment

1. Concatenation (+ or append()): -

Eg: string str1 = "Hello";

string str2 = "World";

string result = str1 + " " + str2; // Concatenation using '+'

str1.append(" World"); // Concatenation using append()

1. Length of a String (length() or size()): -

Eg: string str = "Hello";

cout << str.length(); // Output: 5

cout << str.size(); // Equivalent to length()

1. Access Characters ([] or at()): -

Eg: string str = "Hello";

cout << str[0]; // Output: H

cout << str.at(1); // Output: e

=> Complete Example of String Funtions: -

#include <iostream>

#include <string>

using namespace std;

int main() {

string str1 = "Hello";

string str2 = "World";

// Concatenation

string str3 = str1 + " " + str2;

cout << "Concatenated: " << str3 << endl;

// Length

cout << "Length: " << str3.length() << endl;

// Substring

cout << "Substring: " << str3.substr(6, 5) << endl;

// Find

cout << "Find 'World': " << str3.find("World") << endl;

// Replace

str3.replace(6, 5, "Universe");

cout << "Replaced: " << str3 << endl;

return 0;

}

Output: - Concatenated: Hello World

Length: 11

Substring: World

Find 'World': 6

Replaced: Hello Universe

20. Explain the key concepts of Object-Oriented Programming (OOP)

=> The key concepts of OOP are:

1. **Encapsulation: -** Encapsulation means binding data (attributes) and methods (functions) into a single unit (class) and restricting direct access to the internal data to ensure data security. Protects data using access specifiers (e.g., private, public, protected).

Eg**:** #include <iostream>

using namespace std;

class Student {

private:

int age; // Private data member

public:

// Setter function

void setAge(int a) {

if (a > 0) // Validation

age = a;

else

cout << "Invalid age!" << endl;

}

// Getter function

int getAge() {

return age;

}

};

int main() {

Student s;

s.setAge(20); // Set value using setter

cout << "Age: " << s.getAge(); // Get value using getter

return 0;

}

Output: - Age: 20

1. **Abstraction: -** Abstraction hides the complex implementation details and shows only the essential features of an object.  
   It allows working with interfaces without worrying about internal implementation. Achieved using classes and abstract classes.

Eg**:** #include <iostream>

using namespace std;

// Abstract class

class Shape {

public:

virtual void area() = 0; // Pure virtual function

};

class Rectangle : public Shape {

private:

int length, width;

public:

Rectangle(int l, int w) {

length = l;

width = w;

}

void area() override {

cout << "Area of Rectangle: " << length \* width << endl;

}

};

int main() {

Shape\* s; // Pointer to abstract class

Rectangle r(10, 5);

s = &r;

s->area(); // Calls Rectangle's area()

return 0;

}

Output: - Area of Rectangle: 50

1. **Inheritance: -** Inheritance allows a class (child) to acquire the properties and methods of another class (parent).  
   It promotes code reusability and supports the hierarchical organization of classes.

**Types of Inheritance:** -

* Single Inheritance - Derived from one base class.
* Multiple Inheritance - Derived from multiple base classes.
* Multilevel Inheritance - Derived from a class that is itself derived.
* Hierarchical Inheritance - Multiple derived classes inherit from a single base class.
* Hybrid Inheritance - Combination of two or more types.

Eg: #include <iostream>

using namespace std;

// Base class

class Animal {

public:

void eat() {

cout << "Animal is eating" << endl;

}

};

// Derived class

class Dog : public Animal {

public:

void bark() {

cout << "Dog is barking" << endl;

}

};

int main() {

Dog d;

d.eat(); // Inherited from base class

d.bark(); // Defined in derived class

return 0;

}

Output: - Animal is eating

Dog is barking

1. **Polymorphism: -** Polymorphism means many forms—the same function or operator behaves differently based on the context.  
   It allows dynamic behavior and method overriding during runtime.

**Types of Polymorphism:**

1. Compile-time (Overloading): - Achieved using function overloading and operator overloading.
2. Run-time (Overriding): - Achieved using virtual functions and function overriding.

Eg: #include <iostream>

using namespace std;

// Base class

class Animal {

public:

virtual void makeSound() { // Virtual function

cout << "Animal sound" << endl;

}

};

// Derived class

class Dog : public Animal {

public:

void makeSound() override { // Overriding

cout << "Dog barks" << endl;

}

};

int main() {

Animal\* a; // Pointer to base class

Dog d;

a = &d;

a->makeSound(); // Calls Dog's makeSound() due to polymorphism

return 0;

}

Output: - Dog barks

21. What are classes and objects in C++? Provide an example.

=> Class: - A class is declared using the keyword class. A class is a blueprint or template for creating objects. It defines data members (attributes) and member functions (methods) to represent the state and behaviour of an object.

Eg: - class ClassName {

private: // Private members (only accessible within the class)

int data;

public: // Public members (accessible outside the class)

void setData(int d) { // Member function to set data

data = d;

}

int getData() { // Member function to get data

return data;

}

};

Object: - An object is an instance of a class that stores data and provides functionalities defined by the class. An object is created using the class name. It allows access to public members of the class.

Eg: #include <iostream>

using namespace std;

// Define a class

class Student {

private:

string name;

int age;

public:

// Member function to set student details

void setDetails(string n, int a) {

name = n;

age = a;

}

// Member function to display details

void displayDetails() {

cout << "Name: " << name << endl;

cout << "Age: " << age << endl;

}

};

int main() {

Student s1; // Create an object of class Student

// Set details using member function

s1.setDetails("John", 20);

// Display details using member function

s1.displayDetails();

return 0;

}

Output: - Name: John

Age: 20

22. What is inheritance in C++? Explain with an example.

=> Inheritance is one of the core concepts of Object-Oriented Programming (OOP) in C++. It allows a class (child or derived) to inherit properties and behaviours (data members and member functions) from another class (parent or base).

Purpose of Inheritance: -

1. Code Reusability - Avoids code duplication by reusing existing code.
2. Extensibility - Allows extension and modification of existing functionality.
3. Hierarchy - Supports a natural hierarchy of classes (e.g., Animal → Dog).

Types of Inheritance in C++: -

1. Single Inheritance - A class inherits from one base class.
2. Multiple Inheritance - A class inherits from more than one base class.
3. Multilevel Inheritance - A class is derived from another derived class.
4. Hierarchical Inheritance - Multiple derived classes inherit from a single base class.
5. Hybrid Inheritance - Combination of two or more types of inheritance.

Eg of Single Inheritance: -

#include <iostream>

using namespace std;

// Base class

class Animal {

public:

void eat() {

cout << "This animal eats food." << endl;

}

};

// Derived class

class Dog : public Animal {

public:

void bark() {

cout << "The dog barks." << endl;

}

};

int main() {

Dog d; // Create object of derived class

d.eat(); // Inherited method from base class

d.bark(); // Method from derived class

return 0;

}

Output: - This animal eats food.

The dog barks.

Eg of Multilevel Inheritance: -

#include <iostream>

using namespace std;

// Base class

class Person {

public:

void info() {

cout << "This is a person." << endl;

}

};

// Derived class

class Employee : public Person {

public:

void job() {

cout << "This person is an employee." << endl;

}

};

// Derived from Employee

class Manager : public Employee {

public:

void role() {

cout << "This employee is a manager." << endl;

}

};

int main() {

Manager m;

m.info(); // Access from Person class

m.job(); // Access from Employee class

m.role(); // Access from Manager class

return 0;

}

Output: - This is a person.

This person is an employee.

This employee is a manager.

Eg of Multiple Inheritance: -

#include <iostream>

using namespace std;

// Base class 1

class Father {

public:

void profession() {

cout << "Father is a Doctor." << endl;

}

};

// Base class 2

class Mother {

public:

void hobby() {

cout << "Mother likes painting." << endl;

}

};

// Derived class

class Child : public Father, public Mother {

public:

void activity() {

cout << "Child loves sports." << endl;

}

};

int main() {

Child c;

c.profession(); // From Father

c.hobby(); // From Mother

c.activity(); // From Child

return 0;

}

Output: - Father is a Doctor.

Mother likes painting.

Child loves sports.

23. What is encapsulation in C++? How is it achieved in classes?

=> Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP) in C++. It is the process of binding data (attributes) and methods (functions) that operate on the data into a single unit, called a class.

=> Encapsulation ensures data hiding and security by restricting direct access to the internal details of an object and only exposing the necessary functionalities through public methods.

Eg: - #include <iostream>

using namespace std;

class Student {

private:

// Private data members (hidden from outside)

string name;

int age;

public:

// Public member function to set data (setter)

void setDetails(string n, int a) {

if (a > 0) { // Validation

name = n;

age = a;

} else {

cout << "Invalid age!" << endl;

}

}

// Public member function to get name (getter)

string getName() {

return name;

}

// Public member function to get age (getter)

int getAge() {

return age;

}

// Display details

void displayDetails() {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

int main() {

Student s;

// Access private data using public methods

s.setDetails("John", 20); // Setter method

cout << "Name: " << s.getName() << endl; // Getter method

cout << "Age: " << s.getAge() << endl; // Getter method

s.displayDetails(); // Display details directly

return 0;

}

Output: Name: John

Age: 20

Name: John, Age: 20

Advantages of Encapsulation: -

1. Data Security:
2. Improved Flexibility:
3. Code Reusability:
4. Data Integrity:
5. Maintenance: