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

Ans:- Here are the **key differences between Procedural Programming and Object-Oriented Programming (OOP):**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Procedural Programming** | **Object-Oriented Programming (OOP)** |
| **Approach** | Follows a top-down approach | Follows a bottom-up approach |
| **Focus** | Focuses on functions or procedures | Focuses on objects (instances of classes) |
| **Data** | Data is global and accessible by any function | Data is encapsulated within objects (via classes) |
| **Modularity** | Divides programs into functions | Divides programs into objects and classes |
| **Security** | Less secure due to global data access | More secure due to encapsulation and data hiding |
| **Code Reusability** | Limited code reuse; functions must be redefined or copied | High code reuse via inheritance and polymorphism |
| **Examples** | C, Pascal, Fortran | C++, Java, Python (OOP style), C# |
| **Ease of Maintenance** | Harder to maintain as code grows large | Easier to maintain due to modular structure |
| **Real-world Modeling** | Less effective for modeling real-world problems | Designed to simulate real-world entities and behaviors |
| **Inheritance/Polymorphism** | Not supported | Supported, allowing better scalability and flexibility |

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

Ans:- Here are the main advantages of Object-Oriented Programming (OOP) over Procedural-Oriented Programming (POP):

**1. Encapsulation**

* **Definition:** Bundling of data and methods that operate on that data within a single unit (class).
* **Advantage:** Keeps data safe from outside interference and misuse.
* **Comparison:** In POP, data is often global and can be modified from anywhere, making it harder to manage.

**2. Data Hiding**

* **Definition:** Using access modifiers (like private, public, protected) to restrict access to class members.
* **Advantage:** Prevents unauthorized access and accidental changes to data.
* **Comparison:** POP does not support access control like OOP does.

**3. Reusability through Inheritance**

* **Definition:** A class can inherit properties and behaviors from another class.
* **Advantage:** Promotes code reuse, reduces redundancy, and improves maintainability.
* **Comparison:** In POP, code reuse requires copying and modifying functions, increasing errors and maintenance work.

**4. Polymorphism**

* **Definition:** Ability to use a single function name to perform different tasks based on context (e.g., method overloading/overriding).
* **Advantage:** Enhances flexibility and scalability by allowing the same interface for different underlying data types.
* **Comparison:** POP requires different function names for different tasks, which can be confusing and harder to manage.

**5. Abstraction**

* **Definition:** Hiding internal implementation details and showing only the necessary features.
* **Advantage:** Simplifies complex systems and improves focus on essential tasks.
* **Comparison:** POP does not naturally support abstraction, making complex programs harder to design and understand.

**6. Modularity**

* **Definition:** Programs are divided into separate objects or classes.
* **Advantage:** Easier to test, debug, and maintain since each object is independent.
* **Comparison:** POP uses functions but lacks the structural modularity that OOP offers through classes.

**7. Better Modeling of Real-World Problems**

* **Advantage:** OOP allows you to create classes that represent real-world entities, making code more relatable and understandable.
* **Comparison:** POP uses procedures and data separately, which may not align well with real-world scenarios.

**8. Easier Maintenance and Scalability**

* **Advantage:** Changes in the program are easier to make without affecting the entire system, due to encapsulation and modularity.
* **Comparison:** POP systems become difficult to manage as they grow in size and complexity.

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

Ans:- Setting up a **C++ development environment** involves several steps to install and configure the necessary tools to write, compile, and run C++ programs:-

**1. Install a C++ Compiler**

A compiler is needed to convert C++ code into executable programs.

* **Windows:**
  + Use **MinGW** (Minimalist GNU for Windows) or **MSVC** (Microsoft Visual C++).
  + Recommended: Install **MinGW** via [MSYS2](https://www.msys2.org/) or use **TDM-GCC**.
* **macOS:**
  + Install **Xcode Command Line Tools** using:

xcode-select --install

* **Linux:**
  + Most distributions come with g++ pre-installed. If not, install it via:

sudo apt install g++

**2. Choose and Install a Text Editor or IDE**

An IDE (Integrated Development Environment) helps write, manage, and debug code easily.

* **Lightweight Text Editors:**
  + **Visual Studio Code (VS Code)** – popular and customizable.
  + **Sublime Text** – fast and user-friendly.
* **Full-featured IDEs:**
  + **Code::Blocks** – beginner-friendly and integrates with GCC.
  + **Dev C++** – simple C++ IDE (good for beginners).
  + **Visual Studio (Windows only)** – powerful IDE with MSVC.
  + **CLion** – professional C++ IDE by JetBrains.

**3. Configure the Environment**

* **Set the Compiler Path** (if using an editor like VS Code):
  + Ensure the compiler (e.g., g++) is in your system's PATH.
  + Check by running:

g++ --version

* **Install Extensions (for VS Code)**:
  + C/C++ Extension by Microsoft.
  + Code Runner (optional for quick code execution).

**4. Write a Sample C++ Program**

#include <iostream>

using namespace std;

main()

{

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

}

**5. Compile and Run the Program**

* **Command Line (All platforms):**

g++ program.cpp -o program

./program // On Windows, use: program.exe

* **Using IDE:**
  + Just press **Run** or **Build and Run**, depending on your IDE.

**6. (Optional) Debugging Setup**

* Use an IDE that supports debugging or install a debugger (like gdb) and configure breakpoints, watch variables, etc.

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

Ans:- In C++, **input and output** operations are primarily handled using the **cin** and **cout** objects, which are part of the **iostream** library.

**🔹 1. Output Operation (cout)**

* Used to display data on the screen (standard output).
* Uses the insertion operator (<<).

**Example:**

#include <iostream>

using namespace std;

main()

{

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

cout << "The value of x is: " << 10 << endl;

}

**🔹 2. Input Operation (cin)**

* Used to take input from the user (standard input).
* Uses the extraction operator (>>).

**Example:**

#include <iostream>

using namespace std;

main()

{

int age;

cout << "Enter your age: ";

cin >> age;

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

}

**3. Multiple Inputs**

You can take multiple inputs in one line using cin.

**Example:**

int a, b;

cin >> a >> b;

**🔹 4. Using getline() for Strings**

* cin stops at whitespace, so for full lines (including spaces), use getline().

**Example:**

#include <iostream>

#include <string>

using namespace std;

main()

{

string name;

cout << "Enter your full name: ";

getline(cin, name);

cout << "Hello, " << name << "!" << endl

}

**5. Formatted Output with iomanip**

For precision or alignment, use iomanip library.

**Example:**

#include <iostream>

#include <iomanip>

using namespace std;

main()

{

double pi = 3.14159265;

cout << "Value of pi (2 decimal): " << fixed << setprecision(2) << pi << endl;

}

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

Ans:- **1. Primary (Built-in) Data Types:**

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Description** | **Example** |
| int | Stores integers | int age = 25; |
| float | Stores decimal numbers (single precision) | float pi = 3.14f; |
| double | Stores decimal numbers (double precision) | double rate = 7.89; |
| char | Stores a single character | char grade = 'A'; |
| bool | Stores true or false | bool isOn = true; |
| void | Used for functions that return nothing | void display(); |

**2. Derived Data Types**

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Example** |
| **Array** | Collection of elements of same type | int marks[5]; |
| **Pointer** | Stores address of another variable | int\* ptr = &x; |
| **Function** | Function returning specific type | int add(int a, int b); |
| **Reference** | Alias for another variable | int& ref = x; |

**3. User-defined Data Types**

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Example** |
| **struct** | Groups different types of variables | struct Student {int id; char name[50];}; |
| **class** | Blueprint for objects in OOP | class Car { public: int speed; }; |
| **union** | Shares memory for all members | union Data { int i; float f; }; |
| **enum** | Defines named constants | enum Color {RED, GREEN, BLUE}; |
| **typedef / using** | Aliases for data types | typedef int Marks; or using Marks = int; |

**4. Constant and Literal Types**

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Example** |
| Integer | Whole numbers | 100, -42 |
| Floating | Decimal values | 3.14, -0.01 |
| Character | Single character in quotes | 'A', 'z' |
| String | Sequence of characters in quotes | "Hello" |
| Boolean | True or false | true, false |

**Example Program Demonstrating Different Data Types:**

#include <iostream>

using namespace std;

main()

{

int age = 20;

float height = 5.9;

double pi = 3.14159;

char grade = 'A';

bool isStudent = true;

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

cout << "Height: " << height << endl;

cout << "PI: " << pi << endl;

cout << "Grade: " << grade << endl;

cout << "Is Student: " << isStudent << endl;

}

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

Ans:- **1. Implicit Type Conversion (Type Promotion)**

**Definition:**

Automatic conversion performed by the compiler when types are compatible.

**Key Points:**

* Happens automatically.
* Usually occurs in expressions involving mixed data types.
* Converts to a "higher" or more precise type (e.g., int to float).

**Example:**

int a = 5;

float b = 6.5;

float result = a + b; // 'a' is implicitly converted to float

Here, a is converted to 5.0 before the addition.

**2. Explicit Type Conversion (Type Casting)**

**Definition:**

Manual conversion specified by the programmer using casting operators.

**Key Points:**

* Must be written by the programmer.
* Used when you want full control over conversion.
* Useful to avoid data loss or when compiler won’t do it automatically.

**Syntax:**

(type) expression

**Example:**

float x = 7.9;

int y = (int)x; // Explicit conversion: x is cast to int (value becomes 7)

**Comparison Table**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Implicit Conversion** | **Explicit Conversion** |
| Also called | Type Promotion | Type Casting |
| Done by | Compiler | Programmer |
| Syntax | No special syntax | (type) expression or type(expr) |
| Control | Less control | Full control |
| Risk of Data Loss | Possible but not always obvious | More noticeable and intentional |
| Example | int + float → float | (int)7.9 → 7 |

**Example Program Showing Both:**

#include <iostream>

using namespace std;

main()

{

int a = 10;

float b = 5.5;

// Implicit conversion: 'a' converted to float

float sum = a + b;

// Explicit conversion: 'b' converted to int

int truncated = (int)b;

cout << "Sum (implicit): " << sum << endl; // Output: 15.5

cout << "Truncated (explicit): " << truncated << endl; // Output: 5

}

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

Ans:-

**1. Arithmetic Operators**

Used to perform basic mathematical operations.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example (a = 10, b = 3)** | **Result** |
| + | Addition | a + b | 13 |
| - | Subtraction | a - b | 7 |
| \* | Multiplication | a \* b | 30 |
| / | Division | a / b | 3 |
| % | Modulus (remainder) | a % b | 1 |

**2. Relational (Comparison) Operators**

Used to compare two values.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example (a = 10, b = 3)** | **Result** |
| == | Equal to | a == b | false |
| != | Not equal to | a != b | true |
| > | Greater than | a > b | true |
| < | Less than | a < b | false |
| >= | Greater than or equal | a >= b | true |
| <= | Less than or equal | a <= b | false |

**3. Logical Operators**

Used to combine multiple conditions.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Result** |
| && | Logical AND | (a > 5 && b < 5) | true |
| || | Logical OR | (a > 5 || b > 5) | true |
| ! | Logical NOT | !(a == 10) | false |

**4. Assignment Operators**

Used to assign values to variables.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Equivalent To** |
| = | Assign | a = 5 | — |
| += | Add and assign | a += 3 | a = a + 3 |
| -= | Subtract and assign | a -= 2 | a = a - 2 |
| \*= | Multiply and assign | a \*= 4 | a = a \* 4 |
| /= | Divide and assign | a /= 2 | a = a / 2 |
| %= | Modulus and assign | a %= 3 | a = a % 3 |

**5. Increment and Decrement Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Result** |
| ++ | Increment (by 1) | a++ or ++a | a = a + 1 |
| -- | Decrement (by 1) | b-- or --b | b = b - 1 |

++a is *pre-increment*, a++ is *post-increment* (same for --).

**6. Bitwise Operators**

Operate on bits of data.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example (a=5, b=3)** | **Result** |
| & | AND | a & b (101 & 011) | 1 |
| | | OR | a | b (101 | 011) | 1 |
| ^ | XOR | a ^ b | 6 |
| ~ | NOT (one's complement) | ~a | -6 |
| << | Left shift | a << 1 | 10 |
| >> | Right shift | a >> 1 | 2 |

**7. Conditional (Ternary) Operator**

* Format: condition ? expr1 : expr2;
* Example:

int a = 10, b = 20;

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

**8. Special Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| sizeof | Returns the size of data type | sizeof(int) |
| typeid | Returns type info at runtime | typeid(a).name() |
| :: | Scope resolution operator | std::cout |
| . | Access members of object | obj.name |
| -> | Access through pointer | ptr->value |
| new/delete | Dynamic memory allocation | int\* p = new int; |

**Example Program Using Multiple Operators:**

#include <iostream>

using namespace std;

main()

{

int a = 10, b = 3;

cout << "Addition: " << (a + b) << endl; // Arithmetic

cout << "Is a > b? " << (a > b) << endl; // Relational

cout << "Both positive? " << (a > 0 && b > 0) << endl; // Logical

a += 5; // Assignment

cout << "New value of a: " << a << endl;

cout << "Size of int: " << sizeof(int) << " bytes" << endl; // Special

}

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

Ans:-

**1. Constants**

**Definition:**

Constants are named values that remain unchanged throughout the program.

**Ways to Declare Constants:**

**a) Using const keyword**

const int MAX\_USERS = 100;

* The value of MAX\_USERS cannot be changed after initialization.

**b) Using #define (preprocessor directive)**

#define PI 3.14159

* Used before compilation; no data type is specified.

**Advantages of Constants:**

* Prevent accidental modification.
* Improve code readability.
* Make it easier to update values (only in one place).

**2. Literals**

**Definition:**

Literals are fixed values used directly in the code without being stored in variables.

**Types of Literals:**

|  |  |  |
| --- | --- | --- |
| **Type** | **Example** | **Description** |
| **Integer** | 10, -5 | Whole numbers |
| **Floating** | 3.14, -0.001 | Numbers with decimal points |
| **Character** | 'A', '9' | A single character inside single quotes |
| **String** | "Hello" | Series of characters inside double quotes |
| **Boolean** | true, false | Logical literals |
| **Nullptr** | nullptr | Null pointer literal (C++11+) |

**Example Program Using Constants and Literals**

#include <iostream>

using namespace std;

main()

{

const float PI = 3.14159; // constant

int radius = 5; // literal: 5

float area = PI \* radius \* radius;

cout << "Radius: " << radius << endl;

cout << "Area of circle: " << area << endl;

cout << "Is circle large? " << (area > 50 ? "Yes" : "No") << endl; // boolean literal

}

**Constants vs. Literals**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Constant** | **Literal** |
| Definition | Named value | Raw, fixed value |
| Usage | Used via variable name | Used directly in code |
| Example | const int x = 10; | cout << 10; |
| Modifiable | No | No |

**Why Use Them?**

* **Safety**: Prevent accidental changes to important values.
* **Clarity**: Makes code easier to understand.
* **Maintenance**: Changing a constant in one place updates it everywhere.

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

Ans:-

**Conditional Statements in C++ :**

Conditional statements allow a program to make decisions and execute different blocks of code based on whether a condition is true or false.

1. **if, if-else, and if-else-if Statements**

**Syntax:**

**if statement:**

if (condition)

{

// code runs if condition is true

}

**if-else statement:**

if (condition)

{

// code runs if condition is true

}

else

{

// code runs if condition is false

}

**if-else-if ladder:**

if (condition1)

{

// runs if condition1 is true

}

else if (condition2)

{

// runs if condition2 is true

}

else

{

// runs if none are true

}

**Example:**

#include <iostream>

using namespace std;

main()

{

int marks;

cout << "Enter marks: ";

cin >> marks;

if (marks >= 90)

{

cout << "Grade A";

}

else if (marks >= 75)

{

cout << "Grade B";

}

else if (marks >= 50)

{

cout << "Grade C";

}

else

{

cout << "Fail";

}

}

**2. switch Statement**

Used to test multiple values of a single variable. It’s a cleaner alternative to long if-else-if chains when comparing the same variable to constants.

**Syntax:**

switch (expression)

{

case value1: // code for value1

break;

case value2: // code for value2

break;

...

default: // code if no case matches

}

break is used to exit the switch after a match. Without it, execution continues to the next case (*fall-through*).

**Example:**

#include <iostream>

using namespace std;

main()

{

int day;

cout << "Enter day number (1-7): ";

cin >> day;

switch (day)

{

case 1: cout << "Sunday";

break;

case 2: cout << "Monday";

break;

case 3: cout << "Tuesday";

break;

case 4: cout << "Wednesday";

break;

case 5: cout << "Thursday";

break;

case 6: cout << "Friday";

break;

case 7: cout << "Saturday";

break;

default: cout << "Invalid day";

}

}

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

Ans:-

**1. for Loop :**

Used when the number of iterations is known.

**Syntax:**

for (initialization; condition; update)

{

// loop body

}

**Example:**

for (int i = 1; i <= 5; i++)

{

cout << i << " ";

}

**Output:** 1 2 3 4 5

**2. while Loop :**

Used when the number of iterations is not known and depends on a condition**.**

**Syntax:**

while (condition)

{

// loop body

}

**Example:**

int i = 1;

while (i <= 5)

{

cout << i << " ";

i++;

}

**Output:** 1 2 3 4 5

**3. do-while Loop:**

Used when the loop must run at least once (condition is checked after executing).

**Syntax:**

Do

{

// loop body

} while (condition);

**Example:**

int i = 1;

do

{

cout << i << " ";

i++;

} while (i <= 5);

**Output:** 1 2 3 4 5

**Comparison Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **for Loop** | **while Loop** | **do-while Loop** |
| Condition checked | Before loop body | Before loop body | After loop body |
| Executes at least once | No | No | Yes |
| Best use case | Known number of iterations | Unknown iterations | Run at least once regardless |
| Syntax compactness | Most compact | Medium | Least compact |

**Example Showing Key Difference:**

int x = 6;

while (x < 5)

{

cout << "In while"; // Won't execute

}

Do

{

cout << "In do-while"; // Will execute once

} while (x < 5);

**Output:** In do-while

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

Ans:-

**1. break Statement**

**Purpose:**

Used to terminate the loop immediately, even if the loop condition is still true.

**Syntax:**

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

{

if (i == 5)

break;

cout << i << " ";

}

**Output:**

1 2 3 4

Loop stops completely when i == 5.

**2. continue Statement**

**Purpose:**

Used to skip the current iteration and move to the next iteration of the loop.

**Syntax:**

for (int i = 1; i <= 5; i++)

{

if (i == 3)

continue;

cout << i << " ";

}

**Output:**

1 2 4 5

When i == 3, cout is skipped, and the loop continues.

**Examples in Different Loops**

**break in a while loop:**

int i = 1;

while (i <= 10)

{

if (i == 6)

break;

cout << i << " ";

i++;

}

// Output: 1 2 3 4 5

**continue in a while loop:**

int i = 0;

while (i < 5)

{

i++;

if (i == 3)

continue;

cout << i << " ";

}

// Output: 1 2 4 5

# 12). Explain nested control structures with an example.

Ans:- **Nested Control Structures in C++ :**

Nested control structures are control structures (like if, for, while, switch) placed inside another control structure. They allow for more complex logic by combining decisions and loops.

**Types of Nested Structures:**

* if inside if (nested if)
* for inside for (nested loops)
* if inside for, while inside if, etc.

**Example 1: Nested if Statements**

#include <iostream>

using namespace std;

main()

{

int marks;

cout << "Enter your marks: ";

cin >> marks;

if (marks >= 50)

{

if (marks >= 90)

{

cout << "Excellent!";

}

else

{

cout << "Passed!";

}

}

else

{

cout << "Failed!";

}

}

**Output:**

* If marks = 92: Excellent!
* If marks = 75: Passed!
* If marks = 40: Failed!

**Example 2: Nested for Loops (Multiplication Table)**

#include <iostream>

using namespace std;

main()

{

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

{

for (int j = 1; j <= 3; j++)

{

cout << i << " x " << j << " = " << i \* j << "\t";

}

cout << endl;

}

}

**Output:**

1 x 1 = 1 1 x 2 = 2 1 x 3 = 3

2 x 1 = 2 2 x 2 = 4 2 x 3 = 6

3 x 1 = 3 3 x 2 = 6 3 x 3 = 9

**Key Points:**

|  |  |
| --- | --- |
| **Feature** | **Description** |
| Flexibility | Allows more complex conditions or repetition |
| Indentation | Crucial for readability |
| Use cases | Menus, tables, grading systems, etc. |

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

Ans:- A function in C++ is a block of code that performs a specific task. Functions help in modularizing code, reusing logic, and making programs easier to read, test, and debug.

**Components of a Function**

A function in C++ typically has three main parts:

1. **Function Declaration (Prototype)**
2. **Function Definition**
3. **Function Call**

**1. Function Declaration**

Also called a function prototype, it tells the compiler:

* Function's name
* Return type
* Number and type of parameters

**Syntax:**

return\_type function\_name(parameter\_list);

**Example:**

int add(int, int); // declares a function that returns int and takes two int arguments

**2. Function Definition**

This is the actual code that defines what the function does.

**Syntax:**

return\_type function\_name(parameter\_list) {

// body of the function

}

**Example:**

int add(int a, int b)

{

return a + b;

}

**3. Function Call**

To use a function, you "call" it by writing its name with the required arguments.

**Syntax:**

function\_name(arguments);

**Example:**

int result = add(5, 3); // Calls the add() function

**Complete Example**

#include <iostream>

using namespace std;

// Function Declaration

int add(int, int);

// Main Function

main()

{

int x = 10, y = 20;

int sum = add(x, y); // Function Call

cout << "Sum = " << sum;

}

// Function Definition

int add(int a, int b)

{

return a + b;

}

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

Ans:- Scope in C++ refers to the region of the program where a variable is accessible or visible. It defines the lifetime and accessibility of a variable within a program.

**Types of Variable Scope in C++**

1. **Local Scope**
2. **Global Scope**
3. (Also: Function Scope, Block Scope, Class Scope – but we'll focus on local vs. global for now)

**1. Local Scope**

A variable declared inside a function or block is called a local variable.  
It is accessible only within that function or block.

**Example:**

void display()

{

int x = 10; // local variable

cout << x;

}

**Key Points:**

* Declared inside a function or block
* Exists only while the function is executing
* Cannot be accessed outside that function

**2. Global Scope**

A variable declared outside all functions (typically at the top of the program) is a global variable.  
It is accessible from any function in the program.

**Example:**

int x = 100; // global variable

void display()

{

cout << x;

}

**Key Points:**

* Declared outside all functions
* Accessible from any part of the program
* Exists for the lifetime of the program

**Difference Between Local and Global Variables**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Local Variable** | **Global Variable** |
| **Declared in** | Inside a function or block | Outside all functions |
| **Scope** | Only within the function/block | Entire program |
| **Lifetime** | During function execution only | Entire program runtime |
| **Accessibility** | Not accessible outside the block | Accessible from any function |
| **Memory Usage** | Allocated during function call | Allocated when program starts |

**Example Showing Both**

#include <iostream>

using namespace std;

int x = 100; // Global variable

void show()

{

int x = 10; // Local variable

cout << "Local x = " << x << endl; // Outputs 10

cout << "Global x = " << ::x << endl; // Outputs 100 using scope resolution (::)

}

main()

{

show();

}

# 15. Explain recursion in C++ with an example.

Ans:- **Recursion** in C++ is a programming technique where a **function calls itself** either directly or indirectly to solve a smaller instance of the same problem.

**Key Concepts of Recursion**

1. **Base Case:**  
   The condition that **stops** the recursion (prevents infinite calls).
2. **Recursive Case:**  
   The part where the function **calls itself** with a simpler/smaller input.

**Syntax Example**

return\_type function\_name(parameters)

{

if (base\_condition)

return base\_value;

else

return function\_name(smaller\_problem);

}

**Example: Factorial of a Number**

The **factorial** of a number n is:  
**n! = n × (n-1) × (n-2) × ... × 1**

**Recursive formula:**  
factorial(n) = n \* factorial(n - 1)  
factorial(1) = 1 (base case)

**C++ Code Example:**

#include <iostream>

using namespace std;

int factorial(int n)

{

if (n == 1) // Base case

return 1;

else

return n \* factorial(n - 1); // Recursive call

}

main()

{

int num = 5;

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

}

🟦 **Output:**

Factorial of 5 is: 120

**How It Works Internally (For factorial(5)):**

ruby

factorial(5)

=> 5 \* factorial(4)

=> 5 \* 4 \* factorial(3)

=> 5 \* 4 \* 3 \* factorial(2)

=> 5 \* 4 \* 3 \* 2 \* factorial(1)

=> 5 \* 4 \* 3 \* 2 \* 1

=> 120

**Advantages of Recursion**

* Simplifies code for problems like factorial, Fibonacci, tower of Hanoi, etc.
* Elegant and close to mathematical definitions.

**Disadvantages**

* Uses more **stack memory**
* Can lead to **stack overflow** if the base case is missing or recursion is too deep

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

Ans:- A **function prototype** in C++ is a declaration of a function that tells the compiler:

* The function name
* The return type
* The parameters (types and order)

It does not contain the function body.

**Syntax of a Function Prototype:**

return\_type function\_name(parameter\_list);

**Example:**

int add(int, int); // Function prototype

**Why are Function Prototypes Used?**

|  |  |
| --- | --- |
| **Purpose** | **Explanation** |
| **Tell the compiler early** | So it knows how to handle the function call before it sees the definition. |
| **Enable top-down design** | You can write main() first and define functions later. |
| **Type checking** | Helps the compiler catch errors in argument type or count. |
| **Support modular programming** | Makes large code easier to manage and organize. |

**Example with and without Function Prototype:**

**With Function Prototype:**

#include <iostream>

using namespace std;

int add(int, int); // Function prototype

main()

{

cout << add(5, 3); // Call before definition

}

int add(int a, int b)

{

return a + b;

}

**Without Function Prototype (Incorrect if definition is below main):**

#include <iostream>

using namespace std;

int main()

{

cout << add(5, 3); // Error: 'add' is not declared yet

return 0;

}

int add(int a, int b) {

return a + b;

}

**To fix this**, add a **prototype before main()**, or move the function definition above main().

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

Ans:- An array in C++ is a collection of elements of the same data type stored at contiguous memory locations. It allows storing multiple values in a single variable using an index.

**Syntax of Array Declaration**

data\_type array\_name[size];

**Example:**

int marks[5]; // Declares an array of 5 integers

**Key Points:**

* Index starts from **0**
* The last element is at **size - 1**
* Arrays store homogeneous data (same type)

**Types of Arrays in C++**

|  |  |
| --- | --- |
| **Type** | **Description** |
| **Single-Dimensional** | A linear list of elements (like a row or line) |
| **Multi-Dimensional** | Arrays with more than one index (like a table/matrix) |

**1. Single-Dimensional Array**

A one-dimensional array is like a list.

**Example:**

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

cout << numbers[2]; // Output: 30

**2. Multi-Dimensional Array**

A multi-dimensional array is like a grid, matrix, or table.

**Example (2D Array):**

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

cout << matrix[1][2]; // Output: 6

2D array can be visualized as:

Row 0: 1 2 3

Row 1: 4 5 6

You can also have 3D arrays, like int arr[2][3][4];

**Difference Between Single and Multi-Dimensional Arrays:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Single-Dimensional Array** | **Multi-Dimensional Array** |
| Structure | Linear/List | Table/Grid/Matrix |
| Declaration | int arr[5]; | int arr[3][4]; |
| Access | arr[2] | arr[1][3] |
| Use Case | Simple list of items | Tables, matrices, game boards, etc. |

**Example Program (Both Types)**

#include <iostream>

using namespace std;

main()

{

int scores[3] = {85, 90, 95}; // 1D Array

int grades[2][2] = {{88, 92}, {75, 80}}; // 2D Array

cout << "Single-D Array: " << scores[1] << endl; // Output: 90

cout << "Multi-D Array: " << grades[1][0] << endl; // Output: 75

}

# 18. Explain string handling in C++ with examples.

Ans:- In C++, strings can be handled in two main ways:

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

**1. C-Style Strings (Character Arrays)**

A C-style string is an array of characters ending with a null character '\0'.

**Declaration:**

char name[10] = "John";

**Example:**

#include <iostream>

using namespace std;

main()

{

char name[20];

cout << "Enter your name: ";

cin >> name; // Takes input until first space

cout << "Hello, " << name << "!";

}

**Common C-string Functions (from <cstring>):**

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| strlen(s) | Length of string |
| strcpy(a, b) | Copy string b into a |
| strcat(a, b) | Concatenate b to end of a |
| strcmp(a, b) | Compare two strings |

Example:

#include <iostream>

#include <cstring>

using namespace std;

main()

{

char str1[20] = "Hello ";

char str2[] = "World";

strcat(str1, str2); // Combines str1 and str2

cout << str1; // Output: Hello World

}

**2. C++ String** Class

C++ provides a string type in the <string> library that is safer, easier, and more powerful than C-style strings.

**Declaration:**

#include <string>

string name = "John";

**Example:**

#include <iostream>

#include <string>

using namespace std;

main()

{

string name;

cout << "Enter your name: ";

getline(cin, name); // Allows spaces in input

cout << "Hello, " << name << "!";

}

**Common string Functions (Methods):**

|  |  |
| --- | --- |
| **Method** | **Description** |
| length() / size() | Returns the number of characters |
| empty() | Checks if string is empty |
| append(str) | Adds str to the end of the string |
| substr(start, len) | Extracts substring |
| find(str) | Finds index of str (or -1 if not found) |
| compare(str) | Compares strings |

Example:

#include <iostream>

#include <string>

using namespace std;

main()

{

string s = "Hello";

s.append(" World");

cout << s << endl; // Output: Hello World

cout << "Length: " << s.length() << endl; // Output: 11

cout << "Substring: " << s.substr(6, 5); // Output: World

}

**🔸 Difference: C-Style vs C++ Strings**

|  |  |  |
| --- | --- | --- |
| **Feature** | **C-style Strings** | **C++ string Class** |
| Declaration | char str[20] | string str |
| Ends with | '\0' (null terminator) | No need for null character |
| Flexibility | Limited operations | Rich built-in functions |
| Safety | Prone to errors (overflow) | Safer and easier to use |

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

Ans:- In C++, arrays can be initialized at the time of declaration using curly braces {}. Initialization can be partial, complete, or implicit.

**1. Initializing One-Dimensional (1D) Arrays**

**Syntax:**

data\_type array\_name[size] = {value1, value2, ..., valueN};

**Examples:**

**Full Initialization:**

int nums[5] = {10, 20, 30, 40, 50};

**Partial Initialization:**

int nums[5] = {1, 2}; // Remaining elements are set to 0

**Implicit Size:**

int nums[] = {5, 10, 15}; // Compiler sets size to 3

**Uninitialized (values are garbage in local scope):**

int nums[4]; // Contains garbage values if declared inside a function

**2. Initializing Two-Dimensional (2D) Arrays**

**Syntax:**

data\_type array\_name[rows][cols] = {

{val1, val2, ...},

{val3, val4, ...},

...

};

**Examples:**

**Full Initialization:**

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

**Row-Major Initialization:**

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

**Partial Initialization:**

int matrix[2][3] = {

{1, 2}, // third element in row is 0

{3} // second and third elements are 0

};

**Output Example:**

#include <iostream>

using namespace std;

main()

{

int a[3] = {10, 20, 30};

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

cout << "1D Array: ";

for (int i = 0; i < 3; i++)

cout << a[i] << " ";

cout << "\n2D Array:\n";

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

for (int j = 0; j < 2; j++)

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

cout << endl;

}

}

**Output:**

1D Array: 10 20 30

2D Array:

1 2

3 4

# 20. Explain string operations and functions in C++.

Ans:- In C++, string operations can be performed in two main ways:

1. **C-style strings** (char arrays with null terminator)
2. **C++ string class** (from <string> library — recommended)

**1. C++ string Class Operations**

C++ provides a powerful and easy-to-use string class with built-in functions.

**Common String Operations:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Example** |
| **Assignment** | Assign value to a string | str = "Hello"; |
| **Concatenation** | Combine strings using + or += | s1 + s2, s1 += " World"; |
| **Comparison** | Compare strings (==, !=, <, etc.) | if (s1 == s2) |
| **Length** | Get number of characters | s.length() or s.size() |
| **Access character** | Get or set a character at a position | s[2] = 'a'; |
| **Substring** | Extract part of a string | s.substr(pos, len) |
| **Insert** | Insert into string at a position | s.insert(3, "abc"); |
| **Erase** | Remove part of the string | s.erase(2, 3); |
| **Find** | Find a substring | s.find("abc"); |
| **Replace** | Replace a substring | s.replace(1, 3, "xyz"); |
| **Clear** | Remove all characters | s.clear(); |

**Example:**

#include <iostream>

#include <string>

using namespace std;

main()

{

string s1 = "Hello";

string s2 = "World";

string s3 = s1 + " " + s2; // Concatenation

cout << "Combined: " << s3 << endl;

cout << "Length: " << s3.length() << endl;

cout << "Substring (0-4): " << s3.substr(0, 5) << endl;

s3.replace(6, 5, "C++"); // Replace 'World' with 'C++'

cout << "After replace: " << s3 << endl;

}

**Output:**

Combined: Hello World

Length: 11

Substring (0-4): Hello

After replace: Hello C++

**2. C-style String Functions (from <cstring>)**

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| strlen(s) | Returns length of string |
| strcpy(a, b) | Copies string b into a |
| strcat(a, b) | Appends b to the end of a |
| strcmp(a, b) | Compares two strings (lexicographically) |
| strchr(s, c) | Finds first occurrence of char c |
| strstr(s, sub) | Finds first occurrence of substring |

**Example:**

#include <iostream>

#include <cstring>

using namespace std;

main()

{

char str1[20] = "Hello";

char str2[] = "World";

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

cout << "Combined: " << str1 << endl;

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

cout << "Comparison: " << strcmp(str1, str2) << endl;

}

**Output:**

Combined: HelloWorld

Length: 10

Comparison: >0 (positive value if str1 > str2)

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

Ans:- Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects rather than functions and logic. It helps in reusability, modularity, and maintainability of code.

**1. Class and Object**

* **Class**: A user-defined blueprint for creating objects. It defines data members and member functions.
* **Object**: An instance of a class. It represents a real-world entity.

Example:

class Car

{

public:

string brand;

void drive() {

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

}

};

main()

{

Car c1;

c1.brand = "Toyota";

c1.drive();

}

**2. Encapsulation**

* **Definition**: Bundling of **data** and **methods** that operate on the data into a single unit (class).
* Access is controlled using **access specifiers**: private, public, and protected.

Example:

class Student

{

private:

int age;

public:

void setAge(int a) { age = a; }

int getAge() { return age; }

};

**3. Abstraction**

* **Definition**: Hiding the complex implementation and showing only the essential features.
* Achieved using access specifiers and abstract classes or interfaces.

Example:

class Shape

{

public:

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

};

**4. Inheritance**

* **Definition**: One class (child/derived) can inherit properties and methods from another (parent/base) class.
* Promotes code reusability.

Example:

class Animal

{

public:

void eat() { cout << "Eating...\n"; }

};

class Dog : public Animal

{

public:

void bark() { cout << "Barking...\n"; }

};

**5. Polymorphism**

* **Definition**: Same function or operator behaves differently in different contexts.
* **Types**:
  + **Compile-time** (function overloading, operator overloading)
  + **Run-time** (function overriding via virtual functions)

Example – Function Overloading:

class Print

{

public:

void show(int a) { cout << a << endl; }

void show(string s) { cout << s << endl; }

};

Example – Function Overriding:

class Base

{

public:

virtual void display() { cout << "Base class\n"; }

};

class Derived : public Base

{

public:

void display() override { cout << "Derived class\n"; }

};

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

Ans:- In C++, **classes** and **objects** are the core concepts of Object-Oriented Programming (OOP). They allow you to create reusable, modular, and organized code.

**What is a Class?**

A class is a user-defined blueprint or template that describes the data (called data members) and the functions (called member functions) that operate on that data.

**Syntax:**

class ClassName

{

// Access specifier

public:

// Data members

// Member functions

};

**What is an Object?**

An **object** is an **instance** of a class. It represents a real-world entity and is used to access members of the class.

**Example: Class and Object in C++**

#include <iostream>

using namespace std;

// Class definition

class Car

{

public:

string brand;

int year;

// Member function

void displayInfo()

{

cout << "Brand: " << brand << endl;

cout << "Year: " << year << endl;

}

};

main()

{

// Object creation

Car car1;

// Accessing members

car1.brand = "Toyota";

car1.year = 2020;

car1.displayInfo(); // Calling member function

}

**Output:**

Brand: Toyota

Year: 2020

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

Ans:- **Inheritance** is one of the key features of Object-Oriented Programming (OOP) in C++.  
It allows a class (child/derived class) to inherit properties and behaviors (data members and functions) from another class (parent/base class).

**Why Use Inheritance?**

* **Code reusability**: Common code written in the base class can be reused in derived classes.
* **Extensibility**: You can extend base functionality without modifying original code.
* **Hierarchy**: Helps represent real-world relationships.

**Syntax of Inheritance**

class Base

{

// base class members

};

class Derived : access\_specifier Base

{

// derived class members

};

Common access\_specifier: public, protected, private

public is the most commonly used for general inheritance.

**Example: 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;

}

};

main()

{

Dog d1;

d1.eat(); // inherited from Animal

d1.bark(); // own method

}

**Output:**

This animal eats food.

The dog barks.

**Types of Inheritance in C++**

|  |  |
| --- | --- |
| **Type** | **Description** |
| **Single** | One base class → one derived class |
| **Multiple** | Multiple base classes → one derived class |
| **Multilevel** | Derived class becomes base for another |
| **Hierarchical** | One base class → multiple derived classes |
| **Hybrid** | Combination of above types |

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

Ans:- **Encapsulation** is one of the core principles of Object-Oriented Programming (OOP).  
It refers to the bundling of data and methods that operate on that data into a single unit, typically a class, and restricting direct access to some components to protect the integrity of the data.

**Key Features of Encapsulation:**

* Data hiding: Internal details are hidden from outside.
* Secure code: Prevents accidental modification of data.
* Controlled access: Achieved through access specifiers.

**How is Encapsulation Achieved in C++?**

Encapsulation is implemented using:

1. **Classes**
2. **Access Specifiers**:
   * private: Members are accessible only within the class.
   * public: Members are accessible from outside the class.
   * protected: Used for inheritance (not directly related to encapsulation).

**Example: Encapsulation in C++**

#include <iostream>

using namespace std;

class BankAccount {

private:

double balance; // private data member

public:

// Setter function

void setBalance(double b)

{

if (b >= 0)

balance = b;

else

cout << "Invalid balance!" << endl;

}

// Getter function

double getBalance() {

return balance;

}

};

main()

{

BankAccount acc;

acc.setBalance(1000.50); // setting value via public method

cout << "Balance: " << acc.getBalance() << endl; // getting value

}

**Output:**

Balance: 1000.5

**Benefits of Encapsulation**

|  |  |
| --- | --- |
| **Feature** | **Benefit** |
| **Data Hiding** | Prevents external code from tampering |
| **Modularity** | Clear separation of concerns |
| **Maintainability** | Easier to modify code safely |