**Introduction to OOP’s programming**

**Theory exercise: -**

* **Introduction to C++.**

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

**->**

**1). Procedural programming.**

* Developed by Denish Ritchey, 1972 at bell labs.
* Use header file <STDIO. h>.
* Use library function Print f (), Scan f ().
* Data not secure in procedural programming.
* Namespace functionality not supported.
* Inheritance not supported in procedural programming.

**2). Object -Oriented Programming (OOP’s)**

* Developed by B Jarn Stroustrup, 1985 at bell labs.
* Use header file <iostream>
* Use library function Cout <<, Cin >>.
* Data secure in object -oriented programming.
* Namespace functionality supported in object -oriented programming.
* Inheritance supported in procedural programming.

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

**->**

Main advantages of object-oriented programming (OOP) over procedural-oriented programming (POP)

**1. Improved code organization and maintainability**

* Modularity: OOP organizes code into self-contained units called objects, each representing a real-world entity with its data (attributes) and behaviours (methods). This modularity makes code easier to understand, manage, debug, and update.
* Separation of Concerns: Each object in OOP is responsible for a specific task, limiting the impact of changes to other parts of the system. This contrasts with POP, where functions often share global data, leading to a tighter coupling and potential for unintended side effects during modifications.

**2. Enhanced code reusability**

* Inheritance: OOP supports inheritance, allowing new classes to derive properties and methods from existing classes. This reduces code duplication, saving development time and effort.
* Polymorphism: This OOP principle enables a single interface to be used for different data types or classes. It provides flexibility by allowing methods to behave differently based on the object calling them, reducing the need for repetitive code.

**3. Better data security**

* Encapsulation: OOP bundles data and the methods that operate on that data within an object. This restricts direct access to an object's internal components, protecting data from unintended modifications and improving security. In contrast, POP often uses global variables, which are easily accessible and modifiable by any function.

**4. Improved scalability**

* Modularity and Inheritance: The modular structure and inheritance in OOP make it easier to add new features or functionalities to a system without altering existing code. For example, in a vehicle management system, adding a new vehicle type can be done by creating a new class that inherits from a base vehicle class, [according to upGrad](https://www.upgrad.com/blog/what-are-the-advantages-of-object-oriented-programming/). This approach simplifies scaling the program and ensures it remains organized as it expands.

In conclusion, OOP provides advantages in terms of code organization, reusability, data security, and scalability, making it a powerful paradigm for developing complex and large-scale software applications.

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

**->**

1. **Install a C++ compiler.**

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

1. Windows:

* MinGW (Minimalist GNU for Windows)
* Microsoft C++ compiler via Visual Studio

1. Linux:

* Use g++, typically pre-installed or install with:

Sudo apt install g++

1. MacOS:

* Install Xcode Command Line Tools:

Xcode – select -- install

1. **Choose and Install an IDE or Text Editor.**
2. Popular IDEs for C++:

* Visual Studio (Windows)
* Code Blocks
* Eclipse CDT

1. Lightweight Editors:

* Visual Studio Code (VS Code) — with C++ extensions
* Sublime Text
* Atom

1. **Set Up the IDE/Editor.**
2. For visual Studio Code:

* Install the C/C++ extension from Microsoft
* Configure tasks for build and debug by creating:
* Tasks. Json for build
* Launch. Json for debugging

1. In Code Block.

* Select the correct compiler in settings
* Set up new C++ project

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

**->**

**1.** Input using **Cin**

* Stand for **“console input”**.
* Used to get input from the user (typically via the keyboard).
* Syntax: “Cin >> variables”;
* Example:

#include <iostream>

Using namespace std;

Int main ()

{

Int a;

Cout << “Enter Your age is:” << Endl;

Cin >> a >> Endl;

Cout << “Your age is:” << a << Endl;

Return 0;

}

**2.** Output using **Cout**

* Stand for **“Console output”**.
* Used to display information on the screen.
* Syntax: “Cout << “Display”;
* Example:

#include <iostream>

Using namespace std;

Int main ()

{

Cout << “Hello World!” << Endl;

Return 0;

}

* **Variables, Data Types, and Operators.**

**(1). What are the different data types available in C++? Explain with examples.**

**->**

1. **Primary Data type.**

Int, char, float, double.

1. **Derived Data type.**

These are based on built-in types:

Array, structure, pointer, Function.

1. **Void Data type.**

Void means no value and no return type.

1. **Modifiers**

Modifiers change the size or behaviour of the basic data types:

Signed, Unsigned, short, long

**(2). Explain the difference between implicit and explicit type conversion in C++.**

**->**

1. **Implicit Type: -**

* An implicit type conversion is automatically performed by the compiler when differing data types are intermixed in an expression.
* An implicit type conversion is performed without programmer’s intervention.
* Example:

a, b =5, 25.5

c = a + b

1. **Explicit Type: -**

* An explicit type conversion is user -defined conversion thar forces an expression to be of specific type.
* An explicit type conversion is specified explicitly by the programmer.
* Example:

a, b = 5, 25.5

c = int (a + b)

**(3). What are the different types of operators in C++? Provide examples of each.**

**->**

1. Arithmetic operator

Used for mathematical operations.

+, -, /, \*, =, %

Example: -

int a = 10, b = 3;

Cout << a + b; // Output: 13

1. Relational operator

Used compare two Value.

==, !=, >, <, >=, <=.

Example: -

If(a>b)

{

Cout<< “A is greater”;

}

1. Logical operator

Used to combine conditional statement.

&&, ||, !

Example: -

If (a > 0 && b > 0)

{

Cout << “Both are positive”;

}

1. Assignment operator

Used to assign value.

=, +=, -=, \*=, /=, %=

1. Increment / Decrement operator

Used to increase and decrease value by 1.

++, --

1. Bitwise operator

Operate on bits and perform bit-by-bit operations.

&, ^, ~, <<, >>

1. Ternary operator

A shorthand for if-else.

Example: -

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

1. Special operator

* **Scope resolution (: :)** – Used to define a function outside the class.
* **Size of** – Returns the size of data type.
* **Comma (,)** – Separates expressions.
* **Pointer (\*)** – Pointer declaration/dereferencing.
* **Address-of (&)** – Gets the memory address.
* Example: -

int x = 10;

Cout << size of(x); // Output: 4

1. Member Access operator

Used to access members of classes, structures, or unions.

. , - >

**(4). Explain the purpose and use of constants and literals in C++.**

**->**

1. **Constants**

Constants are fixed values that cannot be modified during program execution. They are declared using the Constant keyword or with #define preprocessor directive.

Example using constant:

Constant in MAX = 100; // Constant integer

Example using #define:

#define PI 3.14 // Define a constant

1. **Literals**

Literals are constant values directly used in the code. They request fixed values of specific types, such as:

* **Integer literal**: 100
* **Floating-point literal**: 3.14
* **Character literals:** ‘A’
* **String literal**: “Hello”
* **Boolean literal**: true or false

Constants are literals help ensure that specific values remain unchanged and are directly usable in expressions.

* **Control flow Statement.**

**(1).** **What are conditional statements in C++? Explain the if-else and switch statements.**

**->**

1. **If-else**

If -else statement allows selecting any one of the two available options depending upon the output of the test condition.

**Syntax:**

If(condition)

{

Statements; // true statement

}

Else

{

Statements; // false statement

}

1. **Switch**

Switch case statements occurs when a switch statement is defined inside another switch statement.

**Syntax:**

Switch (variable)

{

Case 1:

Statement 1;

Break;

Case 2:

Statement 2;

Break;

Case 3:

Statement 3;

Break;

Default;

}

**(2).** **What is the difference between for, while, and do-while loops in C++?**

**->**

1. For loop

**Best for**: When the number of iterations is known in advance.

Example:

For (int I = 0; I < 5; i++)

{

Cout << I << “”;

}

1. While loop

**Best for**: When the number of iterations is **not** known in advance.

Example:

Int I = 0;

While (I < 5)

{

Cout << I << “”;

I++;

}

1. Do -while loop

**Best for**: When the loop should execute **at least once**, regardless of the condition.

Example:

Int I = 0;

Do

{

Cout << I << “”;

I++;

} While (I < 5);

**(3).** **How are break and continue statements used in loops? Provide examples.**

**->**

**(1). Break**

**Purpose**: Immediately exits the loop, even if the condition is still true.

**Use case**: To stop the loop when a specific condition is met.

**Example:**

#include <iostream>

using namespace std;

int main ()

{

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

{

if (i == 5)

{

break; // Exit loop when i is 5

}

Cout << I << " ";

}

return 0;

}

**(2). Continue**

Purpose: Skips the current iteration and moves to the next one.

**Use case**: To skip processing certain values without exiting the loop.

Example:

#include <iostream>

using namespace std;

int main ()

{

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

{

if (i == 3)

{

continue; // Skip the rest of loop body when i is 3

}

Cout << i << " ";

}

return 0;

}

**(4).** **Explain nested control structures with an example.**

**->**

A nested control structure means using one control structure (like if, for, while, switch, etc.) inside another.

1. **Nested if Statement.**

This involves placing one if or if-else statement inside another. The inner if or if-else statement is only evaluated if the outer if condition is true.

1. **Nested loop Statement.**

This entails placing one loop (e.g., for, while, or do-while) inside another loop. The inner loop completes all its iterations for every single iteration of the outer loop. This is particularly useful when working with multi-dimensional data structures like matrices or grids, or to solve problems requiring repeated nested iterations.

* **Functions and Scope**

**(1). What is a function in C++? Explain the concept of function declaration, definition, and calling.**

**->**

A function in C++ is a block of code that performs a specific task and can be reused. Functions help to break down large programs into smaller, manageable sections.

1. Function Declaration

Int add (int a, int b)

1. Function Definition

Int add (int a, int b)

{

Return a + b;

}

1. Function call

Int result = add (5, 3);

**(2).** **What is the scope of variables in C++? Differentiate between local and global scope.**

**->**

In C++, the scope of a variable refers to the region or context within the program where that variable is visible and accessible. This dictates where you can use a variable in your code and influences its lifetime.

**(1). Local**

* Inside a function or block.
* Only within the function/block declared.
* Created when function starts, destroyed when it ends.
* Uninitialized (garbage value).

**(2). Global**

* Outside all functions.
* Any function in the same file.
* Exists throughout the program run.
* Automatically initialized to 0.
* Can Use:: to access if shadowed.

**(3).** **Explain recursion in C++ with an example.**

**->**

Recursion is a programming technique where a function calls itself to solve a problem. It breaks a large, complex problem into smaller, identical subproblems until a simple base case is reached, which allows the recursion to terminate.

**(4). What are function prototypes in C++? Why are they used?**

**->**

A function prototype in C++ is a declaration of a function that specifies its name, return type, and parameters but does not contain the function body.

**Example: -**

int add (int a, int b); // This is a function prototype

* **Array and String**

**(1).** **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 contiguous memory locations. Arrays allow you to store and access multiple values using a single variable name with an index.

**1.Single -Dimensional Array.**

* A list of elements.
* Require one index.
* Contiguous block of memory.
* Storing list of data.
* Size of (data \_type) \* array \_size

**2.Multi -Dimensional Array.**

* An array of arrays (e.g., a grid or table for 2D).
* Require multiple indices.
* Contiguous block of memory, often in row -major order.
* Storing tabular data, matrices, or grids.
* Size of (data \_type) \* size1 \* size2 \* … \* size N

**(2). Explain string handling in C++ with examples.**

**->**

In C++, strings are sequences of characters used to store and manipulate text data. They can be handled in two main ways: C-style character arrays and the more powerful and flexible Std:: string class.

1. **C-style strings (character arrays)**

**Definition**: C-style strings are character arrays terminated by a null character (\0).

**Declaration:**You declare them like any other array, specifying the size, or letting the compiler determine it based on the assigned literal.

char name [20];

char city [] = "New York";

1. **C++ string class**

**Definition:**The std::string class (from the <string> header) provides dynamic string management, making operations like resizing, copying, and concatenation much simpler and safer.

**Declaration:** Declare STD :: string variables using the STD :: string type.

Std:: string greeting = "Hello";

Std::string message;

**(3).** **How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.**

**->**

Arrays in C++ are collections of elements of the same data type stored at contiguous memory locations. When you declare an array, its elements may initially contain undefined values (often called "garbage values"), so it's important to initialize them with meaningful data. Here's how to do it for both one-dimensional (1D) and two-dimensional (2D) arrays.

1. **One-dimensional (1D) arrays.**

A one-dimensional array is like a list or a row of elements. You can initialize it at the time of declaration using an initializer list:

**Explanation:** You can provide a list of initial values enclosed in curly braces {}. If the size is omitted, the compiler determines the size based on the number of initializers. If the size is specified and the initializer list is shorter, the remaining elements are initialized to 0

1. **Two-dimensional (2D) arrays**

A two-dimensional array is like a table or grid, organized in rows and columns. You initialize it using nested curly braces.

* Explanation: Each inner set of {} represents a row in the 2D array. The values are assigned sequentially. You can omit the row size (like int matrix [] [3]), but you must always specify the size of the columns.

**(4). Explain string operations and functions in C++.**

**->**

strings are sequences of characters used to store and manipulate textual data.

C++ String Class (std::string): This is a powerful and flexible class from the C++ Standard Library, defined in the <string> header file. It provides dynamic size, automatic memory management, and a rich set of member functions for manipulating strings.

1. **Input / Output**

* Cin >> string \_name: Reads a single word into the string, stopping at the first whitespace.
* Get line (Cin, String \_name): Reads an entire line, including spaces, until a newline character is encountered.

1. **String length**

* String \_name. length () or string \_name. size (): Returns the number of characters in the string

1. **Concatenate**

* + Operator: Combines two strings. For instance, full Name = first Name + “” + last Name.
* Append () function: Appends a string to the end of another string. Example: str1.append(str2).
* Str cat () function: Used for C-style strings (character arrays). Appends one string to another. Note: Requires including the <c string> header and careful handling to prevent buffer overflows.

1. **Comparison**

* Relational operators (==,!=, <, >, <=, >=): Compare strings lexicographically (based on dictionary order).
* Compare () function: Provides a more detailed comparison, returning 0 if strings are equal, a positive value if the first string is lexicographically greater, and a negative value if it's less.
* Str CMP () function: Used for C-style strings. Compares two strings lexicographically. Returns 0 if strings are equal, a positive value if the first is greater, and a negative value if it's less. Note: Requires including the <c string> header.
* **Introduction to Object-Oriented Programming**

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

**->**

1. **Class and object.**

**Class:** A blueprint for creating objects. It defines properties (data members) and behaviours (member functions).

**Object**: An instance of a class.

1. **Encapsulation.**

Wrapping data and functions into a single unit (class).

Data is protected using **access specifiers**: private, protected, and public.

Promotes **data hiding** and **security**.

1. **Inheritance.**

One class (child/derived) inherits features of another class (base).

Promotes **code reuse**.

Types: **Single, Multiple, Multilevel, Hierarchical**, and **Hybrid**.

1. **Polymorphism**

Same function or operator behaves differently based on context.

Types:

* **Compile-time (Static)**: Function overloading, Operator overloading.
* **Runtime (Dynamic)**: Function overriding using virtual functions.

1. **Abstraction**

Hides internal details and shows only essential features.

Achieved using **classes** and **access specifiers**.

Helps reduce complexity and increase usability.

**(2).** **What are classes and objects in C++? Provide an example.**

**->**

**(1). Class**

A class in C++ serves as a blueprint or template for creating objects. It acts as a user-defined data type that groups related data (variables) and functions (methods or member functions) into a single, cohesive unit.

**(2). Object**

An object is an actual instance of a class. Once a class is defined (like a blueprint), you can create multiple objects from it, each with its own set of values for the data members and capable of using the member functions defined in the class.

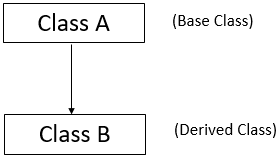
**(3).** **What is inheritance in C++? Explain with an example.**

**->**

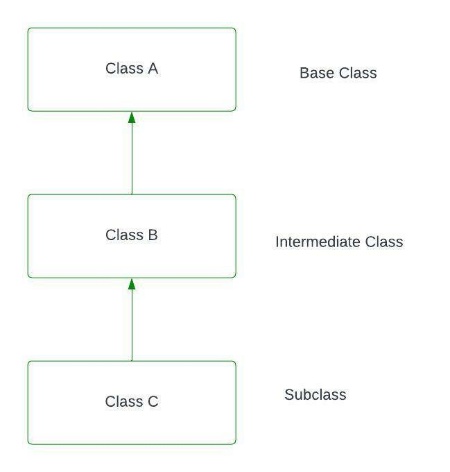
Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows a new class (the derived class or child class) to inherit properties and behaviour (data members and member functions) from an existing class (the base class or parent class).

**Types of inheritance:**

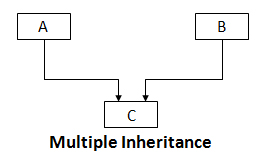
(1). Single level



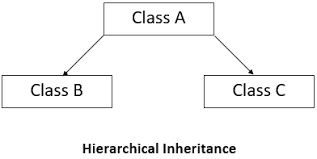
(2). Multilevel



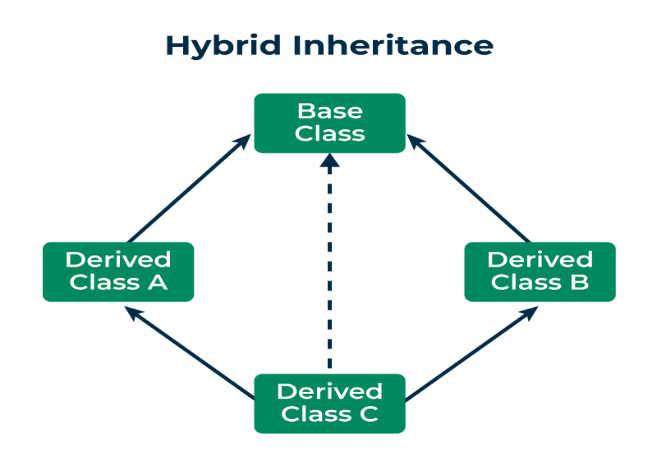
(3). Multiple level



(4). Hierarchical level



(5). Hybrid level



**(4).** **What is encapsulation in C++? How is it achieved in classes?**

**->**

Encapsulation is “wrapping up” of data & member functions in a single unit called class.

**(1). Defining Data Members (Attributes) as Private/Protected:**

The core of encapsulation is achieved by declaring data members (variables) within a class as private or protected.

Private members are only accessible from within the class itself, ensuring that external code cannot directly view or modify them.

By default, if no access specifier is explicitly mentioned for class members, they are considered private.

**(2). Providing Public Member Functions (Getters and Setters):**

While data members are hidden, access to them is provided through public member functions (methods).

These public functions act as a controlled interface to interact with the encapsulated data.

Getter methods (or accessor methods) allow external code to retrieve the value of a private or protected data member. For example, get Age () to retrieve a student's age.

Setter methods (or mutator methods) allow external code to modify the value of a private or protected data member, often including validation logic to maintain data consistency. For example, set Age () might ensure a student's age is a positive value.

The public functions are part of the class's interface and can be accessed using the dot operator with an object of the class.

**(3). Using Access Specifiers:**

C++ provides three access specifiers to control the visibility and accessibility of class members: public, private, and protected.

Public members form the interface of the class, allowing other parts of the program to interact with the object's functionality, [says Naukri Code 360](https://www.naukri.com/code360/library/access-specifier-in-cpp).

Private members encapsulate the internal state and implementation details, shielding them from external access.

Protected members offer a balance between public and private, allowing derived classes to access certain members while keeping them inaccessible to external code.