**Python – Full Stack Assignment**

**Module-3 Introduction to OOPS Programming:-**

* **Introduction to C++ :-**

1. **What are the key differences between Procedural Programming and ObjectOrientedProgramming (OOP)?**

rocedural programming focuses on procedures (functions) to manipulate data, while Object-Oriented Programming (OOP) organizes code around objects that encapsulate data and behavior, promoting modularity and reusability.

Here's a more detailed breakdown of the key differences:

Procedural Programming:

* + **Focus:** Procedures or functions that perform specific tasks on data.
  + **Data and Functions:** Data and functions are treated separately.
  + **Approach:** Top-down, where a problem is broken down into smaller procedures.
  + **Modularity:** Less modular than OOP, making code harder to maintain and reuse as projects grow.
  + **Examples:** Languages like C and FORTRAN.

Object-Oriented Programming (OOP):

* + **Focus:** Objects, which encapsulate data (attributes) and actions (methods) that operate on that data.
  + **Data and Functions:** Data and functions are bundled together within objects (classes).
  + **Approach:** Bottom-up, where objects interact to achieve a task.
  + **Modularity:** Promotes modularity, reusability, and maintainability through the use of classes and objects.

.

1. **List and explain the main advantages of OOP over POPWhat are the key steps involved in the programming process?**

Object-Oriented Programming (OOP) offers several advantages over Procedural Programming (POP), primarily through enhanced code organization, reusability, maintainability, and security, stemming from features like encapsulation, inheritance, and polymorphism.

Here's a breakdown of the main advantages:

* **Modularity and Code Organization:** OOP promotes a modular approach by grouping data and functions into objects, making code easier to understand, maintain, and debug. This modularity allows developers to isolate specific functionalities, leading to cleaner and more manageable codebases.
* **Code Reusability:** OOP facilitates code reuse through inheritance, where classes can inherit properties and methods from parent classes, reducing redundancy and saving development time. This inheritance allows for creating new classes based on existing ones, promoting efficiency and reducing the need to rewrite code.
* **Data Encapsulation and Security:** OOP uses encapsulation to bundle data and the methods that operate on that data into a single unit (object), protecting data from external access and improving security. This controlled access enhances data integrity and prevents unintended modifications.
* **Flexibility and Extensibility:** OOP's features like polymorphism and inheritance allow for greater flexibility and extensibility in software development. Polymorphism enables objects to take on different forms or behaviors, while inheritance allows for creating new classes based on existing ones, enabling easy addition of new features and functionalities.
* **Maintainability:** The modular and encapsulated nature of OOP makes it easier to maintain and update software. Changes can be made to specific objects or classes without impacting other parts of the system, reducing the risk of introducing errors and simplifying the debugging process.
* **Real-World Modeling:** OOP allows developers to model real-world systems and concepts more naturally, as objects represent real-world entities with their own attributes and behaviors. This intuitive approach makes it easier to understand and implement complex systems.

1. **What are the key differences between Procedural Programming and ObjectOrientedProgramming (OOP)?**

To set up a C++ development environment, you'll need a compiler (like GCC or Clang), an Integrated Development Environment (IDE) or text editor (like Visual Studio Code), and potentially a build system (like CMake).

Here's a more detailed breakdown:

**1**. **Choose a Compiler:**

* GCC (GNU Compiler Collection): A widely used and powerful compiler, often included with Linux distributions and available for Windows (through MinGW or MSYS2).
* Clang: Another popular compiler, known for its speed and diagnostics, also available for various platforms.
* Visual Studio (MSVC): Microsoft's IDE and compiler, primarily used on Windows.

**2. Select an IDE or Text Editor:**

* Visual Studio Code (VS Code): A versatile and highly customizable text editor with excellent C++ support through extensions.
* Visual Studio: A powerful IDE from Microsoft, offering a comprehensive development environment for C++.
* Eclipse: A popular open-source IDE with a C/C++ development plugin.

**3. Install the Compiler:**

* Linux:

Most distributions include GCC; you might need to install it using your package manager (e.g., sudo apt-get install build-essential on Debian/Ubuntu).

* macOS:

Xcode's command-line tools provide GCC; install them via xcode-select --install.

* Windows:
  + MinGW: Install MinGW (Minimalist GNU for Windows) and add its bin directory to your system's PATH environment variable.
  + MSYS2: Another option for Windows, providing a Linux-like environment with GCC.
  + Visual Studio: Download and install Visual Studio, selecting the C++ workload during installation.

**4. Install the IDE or Text Editor:**

* Download and install the IDE or text editor of your choice from their respective websites.

**5. (Optional) Install a Build System:**

* CMake: A cross-platform build system that simplifies the process of building C++ projects.

**6. Configure your IDE/Text Editor:**

* C/C++ Extension (VS Code):

Install the C/C++ extension for VS Code to enable features like syntax highlighting, IntelliSense, and debugging.

* Compiler Path:

Configure your IDE/text editor to point to the location of your C++ compiler.

**7. Create a C++ Project and Start Coding:**

* Create a new project or folder for your C++ code.
* Create a C++ file (e.g., main.cpp) and write your code.
* Compile and run your program using the compiler and build system (if using one).

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

In C++, the primary input/output (I/O) operations are handled using the iostream library and its stream objects, cin for input and cout for output, along with operators >> (extraction) and << (insertion).

Input:

* cin (standard input stream): Reads data from the keyboard (or standard input).
* >> (extraction operator): Used to extract data from the input stream (cin) and store it in a variable.
  + **Example:**

#include <iostream>  
 int main() {  
 int age;  
 std::string name;  
 std::cout << "What is your name? ";  
 std::cin >> name; *// Reads input from cin and stores it in 'name'*  
 std::cout << "What is your age? ";  
 std::cin >> age; *// Reads input from cin and stores it in 'age'*  
 std::cout << "Hello, " << name << "! You are " << age << " years old." << std::endl;  
 return 0;  
 }

Output:

* cout (standard output stream): Writes data to the console (or standard output).
* << (insertion operator): Used to insert data into the output stream (cout).
  + **Example:**

#include <iostream>  
 int main() {  
 std::cout << "Hello, world!" << std::endl; *// Prints the message to the console*  
 return 0;  
 }

* **Variables, Data Types, and Operators :-**

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

* **Primitive Data Types:**
* **int: Stores whole numbers (integers).**
  + int myNum = 5;               // Integer (whole number)
* **float:** Stores single-precision floating-point numbers.  
  float myFloatNum = 5.99;      // Floating point number
* **double:** Stores double-precision floating-point numbers  
  double myDoubleNum = 9.98;    // Floating point number
* **char:** Stores single characters.
  + char myLetter = 'D';          // Character
* **bool:** Stores boolean values (true or false).
  + bool myBoolean = true;        // Boolean

* **Derived Data Types:**
* **array:** A collection of elements of the same data type in contiguous memory.
* **pointer:** A variable that stores the memory address of another variable.
* **function:** A block of code that performs a specific task.
* **reference:** An alias for an existing variable.

User-Defined Data Types:

* **struct:** A collection of variables of different data types grouped under a single name.
* **class:** A blueprint for creating objects, encapsulating data and functions.
* **union:** A data type that can store different data types at the same memory location.
* **enum:** A set of named integer constants.
* **User-Defined Data Types:**
* **struct:** A collection of variables of different data types grouped under a single name.
* **class:** A blueprint for creating objects, encapsulating data and functions.
* **union:** A data type that can store different data types at the same memory location.
* **enum:** A set of named integer constants.

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

In C++, type conversion, or type casting, refers to the process of converting a value from one data type to another. There are two main types of type conversion: implicit and explicit.

* Implicit Type Conversion :-

Implicit type conversion, also known as automatic type conversion, is performed by the compiler automatically without any intervention from the programmer. It occurs when the compiler determines that a type conversion is necessary and safe.

|  |
| --- |
| int intValue = 10; double doubleValue = intValue; // Implicit conversion from int to double |

In this example, the integer value intValue is automatically converted to a double before being assigned to doubleValue. This is safe because there is no loss of data in converting an integer to a double. Implicit conversions typically happen when converting from a smaller data type (e.g., int) to a larger data type (e.g., double) or when using mixed data types in expressions.

* Explicit Type Conversion :-

Explicit type conversion, also known as type casting, is performed by the programmer using casting operators. It is used when the programmer wants to explicitly convert a value from one data type to another, even if it may involve data loss or potential errors.

|  |
| --- |
| double doubleValue = 10.78; int intValue = (int)doubleValue; // Explicit conversion from double to int |

In this example, the double value doubleValue is explicitly converted to an integer using the (int) cast operator. This will truncate the decimal part, resulting in intValue being assigned the value 10. Explicit conversions are necessary when converting from a larger data type to a smaller data type or when the compiler cannot automatically perform the conversion. C++ offers several casting operators for different scenarios, such as static\_cast, dynamic\_cast, reinterpret\_cast, and const\_cast.

Key Differences

* Initiation: Implicit conversion is done automatically by the compiler, while explicit conversion is done manually by the programmer.
* Safety: Implicit conversion is generally safe and does not result in data loss, while explicit conversion may lead to data loss or errors if not used carefully.
* Syntax: Implicit conversion does not require any special syntax, while explicit conversion uses casting operators.
* Use Cases: Implicit conversion is often used for convenience and when the conversion is obvious and safe. Explicit conversion is used when more control is needed over the conversion process or when converting between incompatible types.

|  |
| --- |
| #include <iostream>  int main() {  // Implicit conversion  int intValue = 100;  float floatValue = intValue; // int is implicitly converted to float  std::cout << "Implicit Conversion: " << floatValue << std::endl;   // Explicit conversion  double doubleValue = 3.14159;  int truncatedValue = (int)doubleValue; // double is explicitly cast to int  std::cout << "Explicit Conversion: " << truncatedValue << std::endl;   return 0; } |

In summary, implicit type conversion provides convenience and automatic handling of type conversions in safe scenarios, while explicit type conversion offers greater control and the ability to convert between any data types, albeit with the responsibility of ensuring data integrity.

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

* **Arithmetic Operators:**

Perform mathematical calculations.

* + (Addition): int sum = 5 + 3;
* - (Subtraction): int difference = 10 - 4;
* \* (Multiplication): int product = 6 \* 7;
* / (Division): int quotient = 20 / 5;
* % (Modulus): int remainder = 15 % 4;
* ++ (Increment): int x = 5; x++;
* -- (Decrement): int y = 8; y--;
* **Relational Operators:**

Compare two operands.

* == (Equal to): bool isEqual = (5 == 5);
* != (Not equal to): bool isNotEqual = (10 != 3);
* &gt; (Greater than): bool isGreater = (8 &gt; 2);
* &lt; (Less than): bool isLess = (4 &lt; 9);
* &gt;= (Greater than or equal to): bool isGreaterOrEqual = (7 &gt;= 7);
* &lt;= (Less than or equal to): bool isLessOrEqual = (1 &lt;= 6);
* **Logical Operators:**

Combine or modify boolean expressions.

* && (Logical AND): bool andResult = (true && false);
* || (Logical OR): bool orResult = (true || false);
* ! (Logical NOT): bool notResult = !true;
* **Bitwise Operators:**

Perform operations on individual bits of data.

* & (Bitwise AND): int bitwiseAnd = 5 & 3;
* | (Bitwise OR): int bitwiseOr = 5 | 3;
* ^ (Bitwise XOR): int bitwiseXor = 5 ^ 3;
* ~ (Bitwise NOT): int bitwiseNot = ~5;
* &lt;&lt; (Left shift): int leftShift = 5 &lt;&lt; 2;
* &gt;&gt; (Right shift): int rightShift = 5 &gt;&gt; 1;
* **Assignment Operators:**

Assign values to variables.

* = (Assignment): int a = 10;
* += (Add and assign): a += 5;
* -= (Subtract and assign): a -= 3;
* \*= (Multiply and assign): a \*= 2;
* /= (Divide and assign): a /= 4;
* %= (Modulus and assign): a %= 3;
* **Conditional (Ternary) Operator:**

A shorthand for if-else statements.

* condition ? expression1 : expression2: int max = (a &gt; b) ? a : b;
* **Other operators**
* sizeof: Returns the size of a variable or data type in bytes.
* . (member access): Accesses members of a class or struct.
* -&gt; (pointer member access): Accesses members of a class or struct through a pointer.
* [] (subscript): Accesses elements of an array.
* () (function call): Calls a function.
* \* (dereference): Accesses the value pointed to by a pointer.
* & (address-of): Returns the memory address of a variable.
* new and delete: Allocate and deallocate memory dynamically.
* , (comma operator): Evaluates multiple expressions sequentially.
* :: (scope resolution operator): Accesses static members or disambiguates identifiers.
* this pointer: Points to the current object instance.

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

**Constants:-**

* Constants are used to define values that remain fixed and cannot be changed during the execution of a program.
* In C++, they are typically declared using the **const** keyword. When you declare a constant, you specify its data type and give it a name.
* Constants are often used to make code more readable, and self-documenting, and to prevent accidental changes to important values.
* You can define constants of various data types, including integers, floating-point numbers, characters, and more.
* Constants are evaluated at compile-time, and their values are replaced directly in the code.

In the below code, maxCount, pi, and newline are constants, and their values cannot be modified after declaration.

const int maxCount = 100;  
const double pi = 3.14159;  
const char newline = '\n';

**Literals:-**

Literals are the actual values that are directly written into your code to represent specific data. They are used to provide initial values for variables, as operands in expressions, or as direct values in statements.

C++ supports different types of literals:

1) **Integer Literals:** These represent whole numbers, and they can be written in decimal, octal, or hexadecimal formats.

int decimal = 42; // Decimal integer literal  
int octal = 052; // Octal integer literal (0 prefix)  
int hex = 0x2A; // Hexadecimal integer literal (0x prefix)

2) **Floating-Point Literals:**These represent real numbers and can be written in decimal or exponential notation.

double decimalNum = 3.14159; // Decimal floating-point literal  
double exponentNum = 6.02e23; // Exponential floating-point literal

3) **Character Literals:**These represent single characters and are enclosed in single quotes.

char ch = 'A'; // Character literal 'A'  
char newline = '\n'; // Character literal for newline

4) **String Literals:** These represent sequences of characters and are enclosed in double quotes.

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

5) **Boolean Literals:** C++ has two boolean literals, true and false, which represent the Boolean values.

bool isTrue = true;  
bool isFalse = false;

* **Control Flow Statements:-**

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

In C, programs can choose which part of the code to execute based on some condition. This ability is called **decision making** and the statements used for it are called **conditional statements.**

* **switch Statement:-**

A control flow statement called a [switch statement](https://www.geeksforgeeks.org/c-switch-statement/) enables a program to compare an expression to a set of potential constant values by managing many scenarios according to the expression's value. When handling several potential scenarios, the switch statement makes the code easier to comprehend and maintain.

**Syntax of switch Statement**

**switch** (*expression*) {  
 **case** value1:  
 break;  
 **case** value2:  
 break;  
 default:  
}

* **if-else Statement:-**

Conditional control structure called [if-else statements](https://www.geeksforgeeks.org/c-if-else-statement/) are used to allow the execution of a particular code blocks on the basis that the given condition results in true of false. By running code in this way i.e. selectively according to whether a certain condition is true or false, we can easily make judgements.

**Syntax of if-else Statement**

**if** (condition) {  
 *//code*  
}   
**else** {   
 *//code*  
}

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

| **Feature** | **for Loop** | **while Loop** | **do-while Loop** |
| --- | --- | --- | --- |
| **Syntax** | for (initialization; condition; increment/decrement) {} | while (condition) { } | do { } while (condition); |
| **Initialization** | Declared within the loop structure and executed once at the beginning. | Declared outside the loop; should be done explicitly before the loop. | Declared outside the loop structure |
| **Condition** | Checked before each iteration. | Checked before each iteration. | Checked after each iteration. |
| **Update** | Executed after each iteration. | Executed inside the loop; needs to be handled explicitly. | Executed inside the loop; needs to be handled explicitly. |
| **Use Cases** | Suitable for a known number of iterations or when looping over ranges. | Useful when the number of iterations is not known in advance or based on a condition. | Useful when the loop block must be executed at least once, regardless of the initial condition. |
| **Initialization and Update Scope** | Limited to the loop body. | Scope extends beyond the loop; needs to be handled explicitly. | Scope extends beyond the loop; needs to be handled explicitly. |

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

**Break:-**

You have already seen the break statement used in an earlier chapter of this tutorial. It was used to "jump out" of a [switch](https://www.w3schools.com/CPP/cpp_switch.asp) statement.

The break statement can also be used to jump out of a **loop**.

This example jumps out of the loop when i is equal to 4:

Example

for (int i = 0; i < 10; i++) {  
  if (i == 4) {  
    break;  
  }  
  cout << i << "\n";  
}

**Continue:-**

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

Example

for (int i = 0; i < 10; i++) {  
  if (i == 4) {  
    continue;  
  }  
  cout << i << "\n";  
}

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

**Nested Structure in C with Examples**

A **nested structure** in C is a structure within structure. One structure can be declared inside another structure in the same way structure members are declared inside a structure.

**Syntax:**

*struct name\_1  
{  
    member1;  
   member2;  
   .  
   .  
   membern;*

*struct name\_2  
   {  
       member\_1;  
       member\_2;  
       .  
       .  
      member\_n;  
   }, var1  
} var2;*

The member of a nested structure can be accessed using the following syntax:

*Variable name of Outer\_Structure.Variable name of Nested\_Structure.data member to access*

**Example:**

* Consider there are two structures **Employee (depended structure)** and another structure called **Organisation(Outer structure)**.
* The structure Organisation has the data members like organisation\_name,organisation\_number.
* The Employee structure is nested inside the structure Organisation and it has the data members like employee\_id, name, salary.

For accessing the members of Organisation and Employee following syntax will be used:

*org.emp.employee\_id;  
org.emp.name;  
org.emp.salary;*

*org.organisation\_name;  
org.organisation\_number;*

*Here, org is the structure variable of the outer structure Organisation and emp is the structure variable of the inner structure Employee.*

* **Functions and Scope:-**

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

In C++, a function is a block of organized, reusable code that performs a specific task. Functions enable modularity and code reuse, making programs more efficient and easier to manage.

**Function Declaration**

A function declaration, also known as a function prototype, specifies the function's name, return type, and parameters without defining its implementation. It informs the compiler about the function's existence and signature before it is used.

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

**Function Definition**

A function definition provides the actual implementation or body of the function. It includes the code that will be executed when the function is called. [[2](https://www.geeksforgeeks.org/function-declaration-vs-function-definition/?ref=oin_asr2)]

|  |
| --- |
| return\_type function\_name(parameter\_list) {  // Code to be executed  return value; // Optional, if the function returns a value } |

**Function Calling**

A function call is the act of executing a function. It involves using the function's name followed by parentheses, optionally passing arguments if the function expects them.

|  |
| --- |
| function\_name(argument\_list); |

When a function is called, the program control jumps to the function's definition, executes the code within the function body, and then returns to the point where the function was called.

|  |
| --- |
| #include <iostream>  // Function declaration int add(int a, int b);  int main() {  int num1 = 5;  int num2 = 3;    // Function calling  int sum = add(num1, num2);    std::cout << "The sum is: " << sum << std::endl;    return 0; }  // Function definition int add(int a, int b) {  return a + b; } |

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

In C++, the **scope**of a variable is the extent in the code upto which the variable can be accessed or worked with. It is the region of the program where the variable is accessible using the name it was declared with.

Let’s take a look at an example:

#include *<iostream>*

**using** **namespace** **std**;

*// Declaring first variable*

int a = 10;

int main() {

*// Declaring second variable*

int b = 9;

*// Accessing a and b variable in their scope*

cout << a << " " << b;

**return** 0;

}

**Difference between Local Variable and Global variables:**

| **Aspect** | **Local Variables** | **Global Variables** |
| --- | --- | --- |
| **Scope** | **Limited to the block of code** | **Accessible throughout the program** |
| **Declaration** | **Typically within functions or specific blocks** | **Outside of any function or block** |
| **Access** | **Accessible only within the block where they are declared** | **Accessible from any part of the program** |
| **Lifetime** | **Created when the block is entered and destroyed when it exits** | **Retain their value throughout the lifetime of the program** |
| **Name conflicts** | **Can have the same name as variables in other blocks** | **Should be used carefully to avoid unintended side effects** |
| **Usage** | **Temporary storage, specific to a block of code** | **Values that need to be accessed and modified by multiple parts of the program** |

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

In C++, recursion is a technique in which a function calls itself repeatedly until a given condition is satisfied. It is used for solving a problem by breaking it down into smaller, simpler sub-problems. Then finding the solution of it and combining this solution to find the global solution.

A function that calls itself is called a **recursive function**. When a recursive function is called, it executes a set of instructions and then calls itself to execute the same set of instructions with a smaller input. A recursive function should contain,

* **Recursive Case**: Recursive case is the way in which the recursive call is present in the function.
* **Base Condition:** The base condition is the condition that is used to terminate the recursion.

**Basic Example:**

#include <iostream>

using namespace std;

void printHello(int n) {

if (n == 0) return;

cout << "Hello" << endl;

printHello(n - 1);

}

int main() {

printHello(5);

return 0;

}

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

In C++, a function prototype is a declaration of a function's name, return type, and parameters before its actual definition. It serves as a blueprint, informing the compiler about the function's existence and structure. The syntax for a function prototype is as follows:

|  |
| --- |
| return\_type function\_name(parameter\_type1 parameter\_name1, parameter\_type2 parameter\_name2, ...); |

**Function prototypes are used for several key reasons:**

* **Enabling function calls before definition:** Prototypes allow functions to be called before their actual implementation in the code. This provides flexibility in code organization and readability.
* **Facilitating modular programming:** Prototypes support the creation of reusable code modules, as functions can be declared in header files and defined in separate source files.
* **Ensuring type checking:** The compiler uses prototypes to verify that function calls match the expected arguments and return types, preventing errors during compilation.
* **Supporting separate compilation:** Prototypes enable the compilation of individual source files independently, which is crucial for large projects.
* **Improving code organization:** Prototypes contribute to cleaner and more structured code by separating function declarations from their implementations.
* **Avoiding ambiguity:** Prototypes help resolve potential ambiguities that may arise when multiple functions with the same name but different signatures exist.
* **Arrays and Strings:-**

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

An array in C++ is a data structure that stores a fixed-size, sequential collection of elements of the same data type. Arrays are used to organize and manage data efficiently.

|  |  |  |
| --- | --- | --- |
| **Basis** | **One Dimension Array** | **Two Dimension Array** |
| **Definition** | Store a single list of the element of a similar data type. | Store a ‘list of lists’ of the element of a similar data type. |
| **Representation** | Represent multiple data items as a list. | Represent multiple data items as a table consisting of rows and columns. |
| **Declaration** | The declaration varies for different programming language:   1. For C++,   ***datatype variable\_name[row]*** 2. For Java,   ***datatype [] variable\_name= new datatype[row]*** | The declaration varies for different programming language:   1. For C++,  ***datatype variable\_name[row][column]*** 2. For Java,   ***datatype [][] variable\_name= new datatype[row][column]*** |
| **Dimension** | One | Two |
| **Size(bytes)** | size of(datatype of the variable of the array) \* size of the array | size of(datatype of the variable of the array)\* the number of rows\* the number of columns. |
| **Address calculation.** | Address of a[index] is equal to (base Address+ Size of each element of array \* index). | Address of a[i][j] can be calculated in two ways row-major and column-major   1. **Column Major:**Base Address + Size of each element (number of rows(j-lower bound of the column)+(i-lower bound of the rows)) 2. **Row Major:**Base Address + Size of each element (number of columns(i-lower bound of the row)+(j-lower bound of the column)) |
| **Example** | int arr[5];  //an array with one row and five columns will be created.  {a , b , c , d , e} | int arr[2][5];  //an array with two rows and five columns will be created.                 a  b  c  d  e                 f  g   h  i   j |

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

In C++, **strings**are sequences of characters that are used to store words and text. They are also used to store data, such as numbers and other types of information in the form of text. Strings are provided by <string> header file in the form of std::string class.

**Creating a String**

Creating a string means creating an instance of std::string class as shown:

string str\_name;

where str\_name is the name of the string.

Initializing a String

Initializing means assigning some initial value to the string. This can be done by using assignment operator and the text enclosed inside “” double quotes.

string str = "Some Text here";

The text inside “” is called [string literal](https://www.geeksforgeeks.org/cpp-literals/) and it is the value that is assigned to the string variables. It doesn’t need to have any meaning. It can be any text that is the sequence of characters from the [ASCII charset](https://www.geeksforgeeks.org/what-is-ascii-a-complete-guide-to-generating-ascii-code/).

**Accessing String**

A string can be referred using its name anywhere in the scope once it is declared. For example, the below example prints string using [cout](https://www.geeksforgeeks.org/cout-in-c/" \t "_blank):

{...}

*// Creating a string*

string greeting = "Welcome to GfG!";

*// Accessing string*

cout << greeting;

{...}

Output

Welcome to GfG!

The individual characters of the strings can also be accessed using their position (or index) like arrays with [] square brackets. The index in C++ starts from 0 and goes till size – 1, so be careful not go outside this limit.

{...}

string str = "Sonu";

*// Accessing 3rd character*

cout << str[2] << endl;

*// Accessing first character*

cout << str[0];

{...}

Output

n

S

**Updating String**

The string variable can be updated store a new string literal in a similar way it is initialized.

{...}

string str = "Tara";

cout << str << endl;

*// Updating string*

str = "Singh";

cout << str;

{...}

A single character can also be changed by first accessing the character and then using assignment operator to assign value.

{...}

string str = "Tara";

*// Updating second character*

str[1] = 'o';

{...}

**Pass Strings to Functions**

The string can be passed to a function in the same was as any other type of variable.

{...}

*// Taking string as argument*

void print(string s) {

cout << s;

return;

}

int main() {

string s = "GeeksforGeeks";

*// Passing string*

print(s);

{...}

Output

GeeksforGeeks

C Style Strings

C++ is a superset of C language, so it also inherits the way in which we used to create [strings in C](https://www.geeksforgeeks.org/strings-in-c/). In C, strings were nothing, but an array of characters terminated by a NULL character ‘\0’. They were created as:

char str[] = "Hello";

Due to being array, there were limitations on C strings:

* Fixed Size: Once declared, the size of the C string cannot be changed.
* Lack of Easy String Operations: No high-level operations like concatenation or substring extraction. Moreover, updating was also complex.

C++ strings resolve this issue by providing a lot of operations that are easy to perform. Internally, these strings are still implemented as [dynamic array](https://www.geeksforgeeks.org/how-do-dynamic-arrays-work/) of characters (or more precisely [vectors](https://www.geeksforgeeks.org/vector-in-cpp-stl/)) Thats why we can access a single character by its index. But the [std::string class](https://www.geeksforgeeks.org/stdstring-class-in-c/) act as a wrapper and provides lot of built-in functionality for easier and more efficient handling of strings.

C++ String vs C Strings

The main difference between a string and a character array is that strings are immutable, while character arrays are not.

| String | Character Array |
| --- | --- |
| Strings define objects that can be represented as string streams. | The null character terminates a character array of characters. |
| No Array decay occurs in strings as strings are represented as objects. | The threat of [array decay](https://www.geeksforgeeks.org/what-is-array-decay-in-c-how-can-it-be-prevented/) is present in the case of the character array. |
| A string class provides numerous functions for manipulating strings. | Character arrays do not offer inbuilt functions to manipulate strings. |
| Memory is allocated dynamically. | The size of the character array has to be allocated statically. |

Know more about the [difference between strings and character arrays in C++](https://www.geeksforgeeks.org/stdstring-class-in-c/)

C++ String Functions

C++ provides some inbuilt functions which are used for string manipulation, such as the strcpy() and strcat() functions for copying and concatenating strings. Some of them are:

| Function | Description |
| --- | --- |
| length() | This function returns the length of the string. |
| [swap()](https://www.geeksforgeeks.org/swap-in-cpp/) | This function is used to swap the values of 2 strings. |
| size() | Used to find the size of string |
| [resize()](https://www.geeksforgeeks.org/stdstringresize-in-c/) | This function is used to resize the length of the string up to the given number of characters. |
| [find()](https://www.geeksforgeeks.org/string-find-in-cpp/) | Used to find the string which is passed in parameters |
| [push\_back()](https://www.geeksforgeeks.org/stdstringpush_back-in-cpp/) | This function is used to push the passed character at the end of the string |
| pop\_back() | This function is used to pop the last character from the string |
| clear() | This function is used to remove all the elements of the string. |
| [strncmp()](https://www.geeksforgeeks.org/stdstrncmp-in-c/) | This function compares at most the first num bytes of both passed strings. |
| [strncpy()](https://www.geeksforgeeks.org/why-strcpy-and-strncpy-are-not-safe-to-use/) | This function is similar to strcpy() function, except that at most n bytes of src are copied |
| [strrchr()](https://www.geeksforgeeks.org/strrchr-function-in-c-c/) | This function locates the last occurrence of a character in the string. |
| [strcat()](https://www.geeksforgeeks.org/strcat-vs-strncat-c/) | This function appends a copy of the source string to the end of the destination string |
| find() | This function is used to search for a certain substring inside a string and returns the position of the first character of the substring. |
| [replace()](https://www.geeksforgeeks.org/stdstringreplace-stdstringreplace_if-c) | This function is used to replace each element in the range [first, last) that is equal to old value with new value. |
| substr() | This function is used to create a substring from a given string. |
| compare() | This function is used to compare two strings and returns the result in the form of an integer. |
| erase() | This function is used to remove a certain part of a string. |
| [rfind()](https://www.geeksforgeeks.org/stdstringrfind-in-c-with-examples/) | This function is used to find the string’s last occurrence. |

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

**Array Initialization**

Initialization in C is the process to assign some initial value to the variable. When the array is declared or allocated memory, the elements of the array contain some garbage value. So, we need to initialize the array to some meaningful values by using initializer list is the list of values enclosed within braces **{ }** separated by a comma.

int arr[5] = {2, 4, 8, 12, 16};

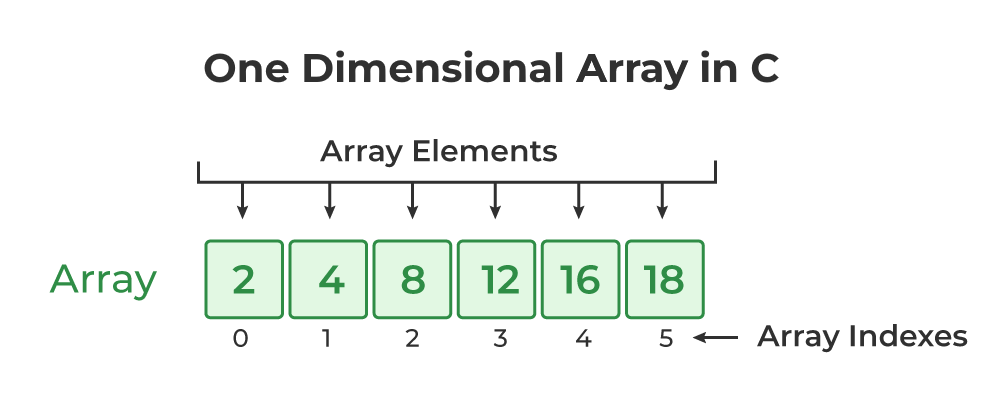
* **One Dimensional Arrays in C**

In C, an array is a collection of elements of the same type stored in contiguous memory locations. This organization allows efficient access to elements using their index. Arrays can also be of different types depending upon the direction/dimension they can store the elements. It can be 1D, 2D, 3D, and more. We generally use only one-dimensional, two-dimensional, and three-dimensional arrays.

In this article, we will learn all about one-dimensional (1D) arrays in C, and see how to use them in our C program.

**One-Dimensional Arrays in C**

A one-dimensional array can be viewed as a linear sequence of elements. We can only increase or decrease its size in a single direction.



Only a single row exists in the one-dimensional array and every element within the array is accessible by the index. In C, array indexing starts zero-indexing i.e. the first element is at index 0, the second at index 1, and so on up to n-1 for an array of size n.

**Syntax of One-Dimensional Array in C**

The following code snippets shows the syntax of how to declare an one dimensional array and how to initialize it in C.

**1D Array Declaration Syntax**

In declaration, we specify then name and the size of the 1d array.

elements\_type array\_name[array\_size];

In this step, the compiler reserved the given amount of memory for the array but this step does not define the value of the elements. They may contain some random values. So we initialize the array to give its elements some initial value.

* **Two-Dimensional Array (2D Array)**

A two-dimensional array is like a table with rows and columns. It allows us to store data in a grid-like format, making it useful for representing matrices, tables, or game boards. It is used in applications where data is arranged in a structured way, such as storing student marks in subjects or representing a chessboard in a game.

Consider the array arr[05][20]:

* The array int arr[05][20] can store total of (05\*20) = 100 elements.
* To find the size in bytes, multiply the size of each element (in bytes) by the total number of elements in the array.
* The size of array int arr[05][20] = 05 \* 20 \* 4  = 400 bytes, where the size of int is 4 bytes.

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

In C++, strings are objects that represent sequences of characters. The &lt;string&gt; header file provides the std::string class, which offers various operations and functions for manipulating strings.

**String Operations**

* Concatenation: Combining two or more strings into a single string. It can be done using the + operator or the append() function.
* Comparison: Comparing two strings lexicographically. It can be done using comparison operators (==, !=, &lt;, &gt;, &lt;=, &gt;=) or the compare() function.
* Accessing characters: Accessing individual characters in a string using the index operator [] or the at() function.
* Substring extraction: Extracting a portion of a string as a new string using the substr() function.
* String length: Determining the number of characters in a string using the length() or size() function.
* Searching: Finding the position of a character or substring within a string using the find() function.
* Replacing: Replacing a portion of a string with another string using the replace() function.
* Insertion: Inserting a string into another string at a specified position using the insert() function.
* Deletion: Removing characters from a string using the erase() function.

String Functions

* getline(): Reads a line of text from an input stream, including spaces, and stores it in a string.
* push\_back(): Appends a single character to the end of a string.
* pop\_back(): Removes the last character from a string.
* swap(): Exchanges the contents of two strings.
* resize(): Changes the length of a string.
* clear(): Removes all characters from a string, making it empty.
* to\_string(): Converts a numerical value to its string representation.
* **Introduction to Object-Oriented Programming:-**

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

As the name suggests, Object-Oriented Programming or OOPs refers to languages that use objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

**OOPs Concepts:**

* Class
* Objects
* Data Abstraction
* Encapsulation
* Inheritance
* Polymorphism
* Dynamic Binding
* Message Passing

**1. Class:**

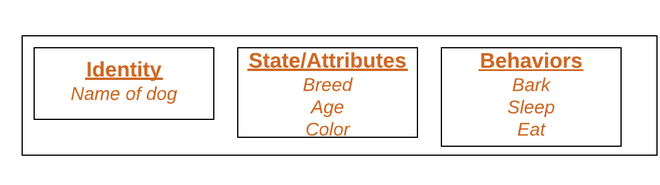
A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class. It represents the set of properties or methods that are common to all objects of one type. A class is like a blueprint for an object.

***For Example:***Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, Car is the class, and wheels, speed limits, mileage are their properties.

**2. Object:**

It is a basic unit of Object-Oriented Programming and represents the real-life entities. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. An object has an identity, state, and behavior. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other’s data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

For example “Dog” is a real-life Object, which has some characteristics like color, Breed, Bark, Sleep, and Eats.



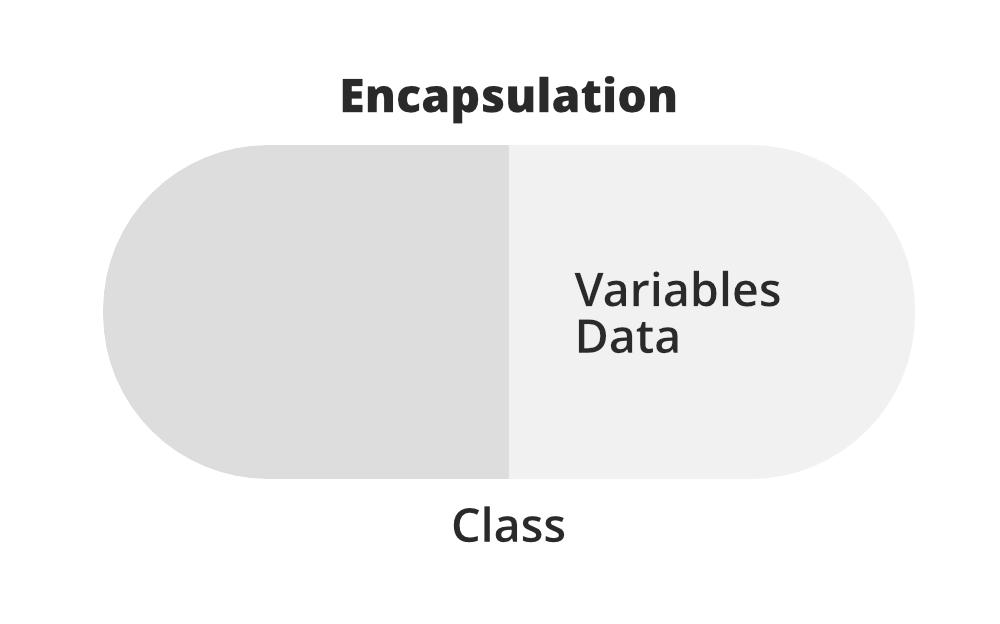
*Object*

**3. Data Abstraction:**

Data abstraction is one of the most essential and important features of object-oriented programming. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

**4. Encapsulation:**

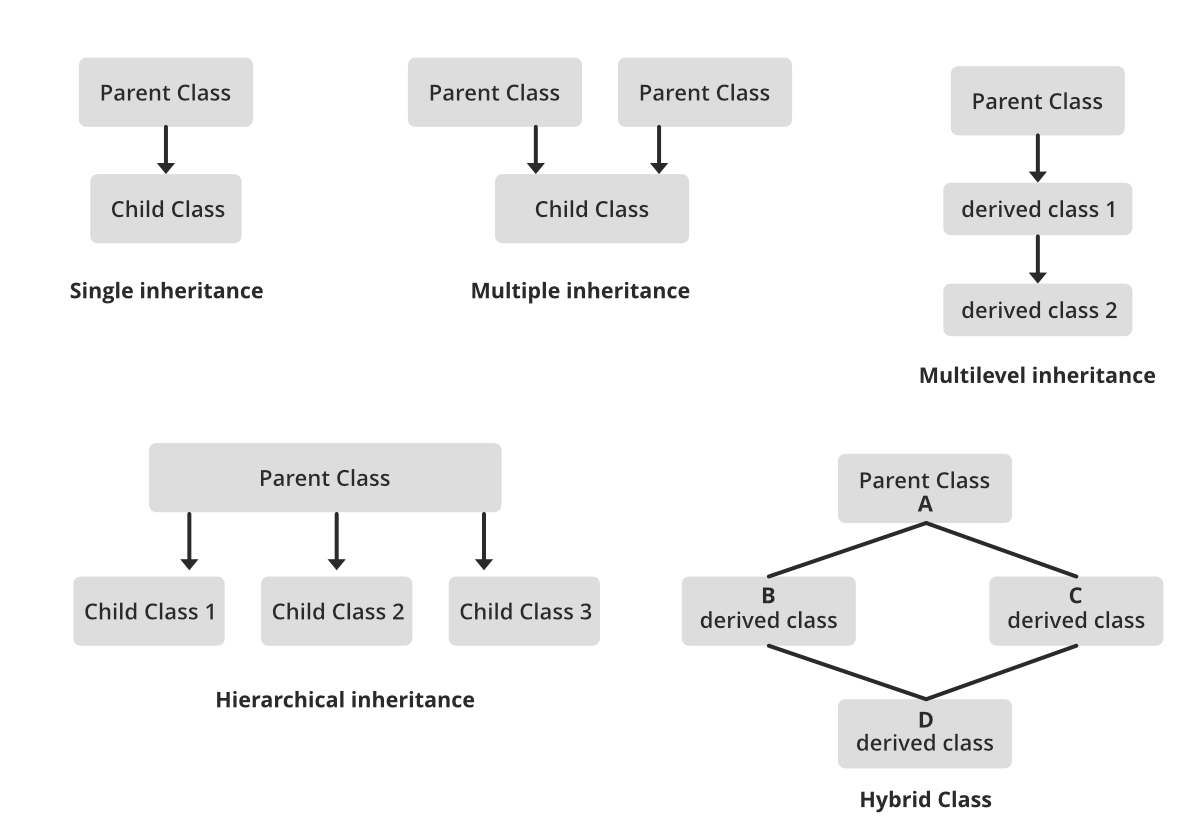
Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. In Encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of their class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.



Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

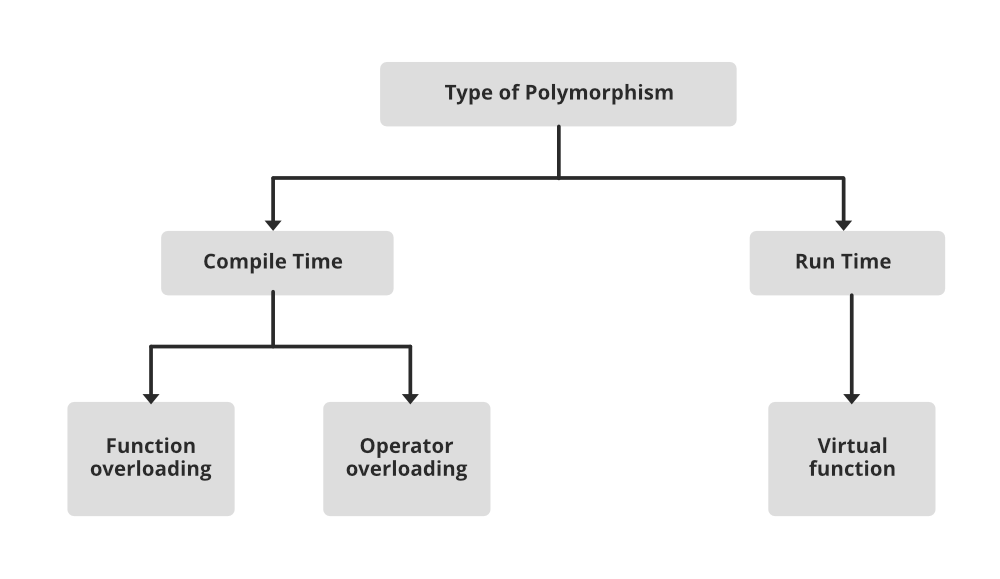
**5. Inheritance:**

Inheritance is an important pillar of OOP(Object-Oriented Programming). The capability of a class to derive properties and characteristics from another class is called Inheritance. When we write a class, we inherit properties from other classes. So when we create a class, we do not need to write all the properties and functions again and again, as these can be inherited from another class that possesses it. Inheritance allows the user to reuse the code whenever possible and reduce its redundancy.



**6. Polymorphism:**

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. For example, A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.



**7. Dynamic Binding:**

In dynamic binding, the code to be executed in response to the function call is decided at runtime. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. Dynamic Method Binding One of the main advantages of inheritance is that some derived class D has all the members of its base class B. Once D is not hiding any of the public members of B, then an object of D can represent B in any context where a B could be used. This feature is known as subtype polymorphism.

**8. Message Passing:**

It is a form of communication used in object-oriented programming as well as parallel programming. Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function, and the information to be sent.

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

In C++, a class is a blueprint or template for creating objects, acting as a user-defined data type that combines data and functions into a single unit. An object is an instance of a class, a concrete realization of that blueprint. Think of it like a cookie cutter (class) and the cookies it makes (objects).

Example:

|  |
| --- |
| #include <iostream> #include <string>  // Define a class named "Person" class Person { public: // Access specifier, making members accessible from outside the class  // Data members (variables)  std::string name;  int age;   // Member function (method)  void introduce() {  std::cout << "My name is " << name << " and I am " << age << " years old." << std::endl;  } };  int main() {  // Create objects (instances) of the "Person" class  Person person1; // Creates an object named person1  Person person2; // Creates another object named person2   // Access and modify data members of the objects  person1.name = "Alice";  person1.age = 30;   person2.name = "Bob";  person2.age = 25;   // Call member functions on the objects  person1.introduce(); // Output: My name is Alice and I am 30 years old.  person2.introduce(); // Output: My name is Bob and I am 25 years old.   return 0; } |

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

Inheritance in C++ is a mechanism that allows a new class (derived class) to inherit properties and behaviors (methods) from an existing class (base class). It promotes code reuse and establishes a hierarchical relationship between classes. The derived class can access and extend the functionality of the base class.

|  |
| --- |
| #include <iostream> #include <string>  class Animal { public:  std::string name;  Animal(std::string name) : name(name) {}  void eat() {  std::cout << name << " is eating." << std::endl;  } };  class Dog : public Animal { public:  Dog(std::string name) : Animal(name) {}  void bark() {  std::cout << name << " is barking: Woof!" << std::endl;  } };  int main() {  Dog myDog("Buddy");  myDog.eat();   myDog.bark();  return 0; } |

In this example, Animal is the base class, and Dog is the derived class. Dog inherits the name attribute and eat() method from Animal and adds its own method, bark().

**4. What is encapsulation in C++? How isit achieved in classes?**

Encapsulation in C++ is **a core object-oriented programming principle that bundles data (attributes) and the methods (functions) that operate on that data into a single unit called a class**. It achieves data hiding by making class members (data and methods) private or protected, controlling their access and preventing direct manipulation from outside the class. [[1](https://www.w3schools.com/cpp/cpp_encapsulation.asp#:~:text=The%20meaning%20of%20Encapsulation%2C%20is%20to%20make,can%20provide%20public%20get%20and%20set%20methods.), [1](https://www.w3schools.com/cpp/cpp_encapsulation.asp#:~:text=The%20meaning%20of%20Encapsulation%2C%20is%20to%20make,can%20provide%20public%20get%20and%20set%20methods.), [2](https://unstop.com/blog/encapsulation-in-cpp#:~:text=Encapsulation%20in%20C++%20is%20achieved%20by%20using,integrity%2C%20and%20controls%20access%20to%20class%20members.), [2](https://unstop.com/blog/encapsulation-in-cpp#:~:text=Encapsulation%20in%20C++%20is%20achieved%20by%20using,integrity%2C%20and%20controls%20access%20to%20class%20members.), [3](https://www.wscubetech.com/resources/cpp/encapsulation), [4](https://www.programiz.com/cpp-programming/encapsulation), [5](https://www.geeksforgeeks.org/encapsulation-in-cpp/)]

**Here's how encapsulation is achieved in C++: [**[**1**](https://www.w3schools.com/cpp/cpp_encapsulation.asp#:~:text=The%20meaning%20of%20Encapsulation%2C%20is%20to%20make,can%20provide%20public%20get%20and%20set%20methods.)**,** [**1**](https://www.w3schools.com/cpp/cpp_encapsulation.asp#:~:text=The%20meaning%20of%20Encapsulation%2C%20is%20to%20make,can%20provide%20public%20get%20and%20set%20methods.)**,** [**2**](https://unstop.com/blog/encapsulation-in-cpp#:~:text=Encapsulation%20in%20C++%20is%20achieved%20by%20using,integrity%2C%20and%20controls%20access%20to%20class%20members.)**,** [**2**](https://unstop.com/blog/encapsulation-in-cpp#:~:text=Encapsulation%20in%20C++%20is%20achieved%20by%20using,integrity%2C%20and%20controls%20access%20to%20class%20members.)**,** [**6**](https://www.boardinfinity.com/blog/a-quick-guide-to-encapsulation-in-c/#:~:text=encapsulation%20in%20C++%20can%20be%20implemented%20using,function%20that%20manipulates%20data%20members%20as%20public.)**,** [**7**](https://logicmojo.com/encapsulation-in-cpp#:~:text=Encapsulation%20manages%20the%20visibility%20and%20accessibility%20of,that%20the%20data%20remains%20concealed%20and%20encapsulated.)**,** [**8**](https://medium.com/@the_infinity/friends-in-c-919653ecffd0#:~:text=A%20class%20in%20C++%20can%20mark%20other,is%20how%20we%20selectively%20loosens%20the%20encapsulation.)**,** [**9**](https://www.tutorialspoint.com/cplusplus/cpp_data_encapsulation.htm)**,** [**10**](https://trainings.internshala.com/blog/encapsulation-in-cpp/#:~:text=The%20following%20stage%20in%20encapsulation%20is%20to,functions%20to%20interact%20with%20the%20class's%20data.)**,** [**11**](https://logicmojo.com/encapsulation-in-cpp)**]**

1. **Defining Classes:** Create a class to encapsulate related data and methods.
2. **Using Access Specifiers:**
   * private: Makes members accessible only within the class itself and its friend functions.
   * protected: Makes members accessible within the class, its derived classes, and its friend functions.
   * public: Makes members accessible from anywhere, including outside the class.
3. **Declaring Data Members:** Declare the data members (attributes) of the class as private to restrict direct access from outside.
4. **Providing Public Interface:** Define public member functions (methods) to interact with and manipulate the private data. These methods can include:
   * **Getters:** Functions that retrieve the values of private data members.
   * **Setters:** Functions that modify the values of private data members.
5. **Example:**

|  |
| --- |
| #include <iostream>  class MyClass { private:  int data; // Private data member public:  void setData(int value) {  data = value; // Setter  }  int getData() {  return data; // Getter  } };  int main() {  MyClass obj;  obj.setData(10);  std::cout << "Data: " << obj.getData() << std::endl;  return 0; } |

In this example, the data member is private, and its value can only be set or retrieved using the public setData and getData methods, respectively.