❖ Introduction to C++

(1) Differences between pop vs oop:

POP	OOP
Full-from of pop is procedural oriented programming.	Full from of oop is object oriented programming.
It deals with algorithms.	It deals with data.
Programs are divided into function.	Programs are divided into objects.
Data move from function to function.	Functions that operate on data are bind to form classes.
It is top-down approach.	It is bottom-up approach.
It needs very less memory	It needs more memory that pop
Example: C, Fortran	Example: C++, java, .Net
It is less secure	It is more secure

(2) Advantages of POP over OOP.

1. Modularity (Encapsulation)

OOP: In OOP, everything is organized into objects. This makes it easier to break down a complex program into smaller, manageable sections. Each object can be worked on separately.

POP: In POP, the code is written in functions that operate on data. It can become harder to manage as the program grows, because the data and functions are separate, and any change in data may require changing multiple functions.

2. Reusability

OOP: oop allows inheritance, which means you can create a new class by reusing code from an existing class. This saves time and reduces errors because you don't have to rewrite the same code.

POP: In pop, reusability is harder because functions are often independent, and reusing the same functionality may involve copying and pasting code.

3. Maintainability

OOP: Because of encapsulation and modularity, oop makes it easier to update and maintain code. If you need to make changes to a specific object, you can do so without affecting other parts of the program.

POP: In pop, if you need to change a function, it can affect other parts of the program, making maintenance more complex.

4. Abstraction

OOP: Data and the functions that operate on that data are bundled together within objects. This hides the internal details and only exposes what is necessary.

POP: Data and functions are separate, so there's no built-in protection to control how data is accessed.

5. Abstraction

OOP: You can use objects without worrying about their internal workings. For example, when you use a "Car" object, you don't need to know how the engine works just how to drive it.

POP: Everything is visible, and you have to manage all the details yourself.

6. Inheritance

OOP: Objects can inherit properties and behaviours from other objects. This allows you to create new classes based on existing ones, saving time.

POP: There's no concept of inheritance. You would have to repeat code for similar structures.

(3) Involved in setting up a C++ development environment

→ C++ development environment can be done easily in a few steps.

1. Install a C++ Compiler

→ To compile and run C++ code, you need a C++ compiler.

2. Install an Integrated Development Environment (IDE)

→ An IDE provides a friendly interface to write, compile, and debug your code.

→ Visual Studio (Windows): A powerful IDE with C++ support.

3. Configure the Compiler

→ Make sure your IDE is set up to use the C++ compiler:

For Code:: Blocks: It usually detects the compiler automatically. If not, go to Settings > Compiler and select the installed compiler.

For Visual Studio: It configures MSVC automatically during installation. You can check and adjust under the "Tools" menu if needed.

For VS Code: After installing the C++ extension, configure the tasks.json and launch.json to point to the compiler. You might need to install g++ on Linux or MinGW on Windows.

4. Debugging (Optional)

→If you need to debug, set breakpoints and use the built-in debugger.

→**In Visual Studio**: Press F5 to run with debugging.

→**In VS Code**: You can add a launch.json configuration to run the debugger.

(4) Main input/output operations in C++

→In C++, the main input/output (I/O) operations are performed using the **iostream** library.

1. Input (Taking user input)

→To take input from the user, C++ uses the cin object (from the <iostream> library).

Syntax: cin >> variable;

→The >> is called the **extraction operator**, which extracts the value from the input stream and stores it in the variable.

Example:

```
#include <iostream>

using namespace std;

main()

{

   int num;
```

```
    cout << "Enter a number: ";
    cin >> num;
    cout << "You entered: " << num << endl;
}
```

Output:

Enter a number:567

You entered:567

1. **Output (Displaying data)**

→ To display output, C++ uses the cout object (also from the <iostream> library).

Syntax: cout << expression;

→ The << is called the insertion operator, which sends the value to the output stream (usually the screen).

Example:

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

Output:

Hello, World!

2. Reading Multiple Inputs

→ You can use cin to read multiple values in a single line.

Example:

```
#include <iostream>
using namespace std;
main()
{
   int a, b;
   cout << "Enter two numbers: ";
   cin >> a >> b;
   cout << "Sum: " << a + b << endl;
}
```

Output:

Enter two numbers: 45

67

Sum:112

3. Reading Strings

To read a string (with spaces), use getline() instead of cin.

Syntax: getline(cin, string_variable);

Example:

```
#include <iostream>
using namespace std;
main()
{
    string name;
    cout << "Enter your name: ";
    getline(cin, name);
    cout << "Hello, " << name << "!" << endl;
}
```

Output:

Enter your name: Krisha

Hello, Krisha!

❖ Variables, Data Types, and Operators

1.Different data types available in C++

A. Basic Data Types:

→These are the fundamental types in C++:

Integer Types:

→Used for whole numbers (both positive and negative)

int: integer type.

Example:

int age = 25;

Character Types

→Used for storing individual characters or letters.

Char: Stores a single character.

Example:

char grade = 'A';

Floating-Point Types

→Used for storing decimal numbers.

float: Stores a single-precision floating-point number.

Example:

float price = 19.99f;

B. Derived Data Types:

→These types are based on the basic data types and allow more complex structures.

Arrays:

→Stores multiple values of the same type in a sequence.

Example:

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

Pointers

→Stores the memory address of another variable.

Example:

int value = 10;

int* ptr = &value;

C. User-defined Data Types

→You can define your own data types using structures, classes, etc.

Structures (struct)

→Groups different data types together.

Example:

```
struct Person {string name; int age;};
Person person1 = {"Alice", 30};
```

Classes

→Similar to structures but with added features like functions and access control.

Example:

```
class Rectangle
 {
       public: int length, width; int area()
             {
                   return length * width;
             }
};
Rectangle rect;
rect.length = 10;
rect.width = 5;
```

Unions (union)

→Stores different data types in the same memory location.

Example:

```
union Data
{
    int i;
    float f;
    char c;
};
Data data;
data. i = 10;
```

D. Void Type:

→The void type is used when no value is returned by a function or when you don't need to specify a type.

void: Represents no data or return value.

Example:

```
void greet()
{
    cout << "Hello, World!";
}
```

void*: A pointer that can point to any data type.

Example:

```
void* ptr;
int num = 100;
ptr = &num;
```

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

→Key Differences:

Aspect	Implicit Type Conversion	Explicit Type Conversion
Initiation	Done automatically by the compiler.	Done manually by the programmer.
Control	No control over the conversion.	Programmer has full control over the conversion.
Example	double y = 5; (int to double)	int x = (int)5.67; (double to int)
Data Loss	No data loss typically (e.g., int to double).	Can cause data loss (e.g., double to int may lose decimals).

→In C++, type conversion refers to converting one data type into another. There are two main types of type conversion: implicit and explicit. Here's the difference between them:

1.Implicit Type Conversion:

Automatic: The compiler automatically converts a data type to another type when needed.

No explicit instruction: The programmer does not need to provide any explicit instructions for this conversion.

Example:

```
int x = 5;

double y = x;
```

2. Explicit Type Conversion:

Manual: The programmer explicitly tells the compiler how to convert a data type to another.

Requires instructions: The programmer uses type casting to indicate the conversion.

Example:

```
double a = 5.67;

int b = (int)a;
```

3.Different types of operators in C++

→In C++, operators are symbols that perform operations on variables and values.

→They are used to manipulate data and variables.

→C++ has a rich set of operators, categorized as follows:

1. Arithmetic Operators:

→These operators are used to perform basic arithmetic operations.

+: Addition

-: Subtraction

*: Multiplication

/: Division

%: Modul

Example:

```
int a = 10, b = 5;
int sum = a + b;
int difference = a - b;
int product = a * b;
int quotient = a / b;
int remainder = a % b;
```

2. Relational Operators:

→These operators compare two values and return a boolean result

==: Equal to

!=: Not equal to

<: Less than

>: Greater than

<=: Less than or equal to

>=: Greater than or equal to

Example:

```
int a = 10, b = 5;
bool Equal = (a == b);
bool Not Equal = (a != b);
bool Greater = (a > b);
bool Less = (a < b);
bool Greater Equal = (a >= b);
bool Less Equal = (a <= b);
```

3. Logical Operators

→These operators are used to combine conditional statements and return a boolean value.

&&: Logical AND

||: Logical OR

!: Logical NOT

Example:

```
Bool x = true, y = false;
bool and Result = (x && y);
bool or Result = (x || y);
bool not Result = !x;
```

4. Bitwise Operators

→These operators perform bit-level operations on data.

& : Bitwise AND

| : Bitwise OR

^ : Bitwise XOR (exclusive OR)

~ : Bitwise NOT

<< : Left shift

>> : Right shift

Example:

```
int a = 5, b = 3;
int and Result = a & b;
int or Result = a | b;
int x or Result = a ^ b;
int not Result = ~a;
int left Shift = a << 1;
int right Shift = a >> 1;
```

5. Assignment Operators

→These operators are used to assign values to variables.

= : Simple assignment

+= : Add and assign

-= : Subtract and assign

*= : Multiply and assign

/= : Divide and assign

%= : Modulo and assign

&= : Bitwise AND and assign

|= : Bitwise OR and assign

^= : Bitwise XOR and assign

<<= : Left shift and assign

>>= : Right shift and assign

Example:

```
int a = 5;
a += 3;
a -= 2;
a *= 4;
a /= 6;
a %= 3;
```

6. Increment and Decrement Operators

→These operators are used to increase or decrease the value of a variable by 1.

++ : Increment (can be pre-increment ++a or post-increment a++)

-- : Decrement (can be pre-decrement --a or post-decrement a--)

Example:

```
int a = 5;
a++;
++a;
a--;
--a;
```

7. Conditional Operator

→This operator provides a shorthand for an if-else statement.

? :: Conditional expression

Example:

```
int a = 10, b = 5;
int max = (a > b) ? a : b;
```

8. Type-Casting Operators

→These operators are used to convert one data type to another.

C-style Cast ((type))

static_cast

dynamic_cast

const_cast

reinterpret_cast

Example:

```
float a = 5.7;
int b = (int)a;
int c = static_cast<int>(a);
```

9. Pointer Operators

→ These operators are used for working with pointers.

Address-of (&)

Dereference (*)

Example:

```
int a = 10;
int* ptr = &a;
int value = *ptr;
```

10. Sizeof Operator

→ This operator is used to determine the size of a data type or object in bytes**.**

Sizeof

Example:

```
int a = 10;
std::cout << sizeof(a);
```

11. Scope Resolution Operator

→ This operator is used to define the scope of a function or variable.

Scope Resolution (::)

Example:

```
int a = 10;
namespace My Namespace
{
    int a = 20;
   void display()
    {
        std::cout << ::a;
    }}
```

4.The purpose and use of constants and literals in C++.

1. Constants

→A constant is a value in your program that cannot be changed after it is set.

→ Think of it like a "fixed" number or value that stays the same throughout the program.

How to use Constants:

→ To create a constant, you use the const keyword:

Syntex:

```
const int MAX_AGE = 100;
```

Example:

```
#include <iostream>
using namespace std;
int main()
{
   const int MAX_AGE = 100;
   cout << "Max age is: " << MAX_AGE << endl;
   return 0;
}
```

2. Literals

→ A literal is a fixed value directly used in the program.

→ These are the actual numbers, characters, or text you use in your code.

Types of Literals in C++:

Integer literals: Whole numbers, like 10, -5, 0

Floating-point literals: Numbers with decimals, like 3.14, -2.5

Character literals: Single characters, like 'A', 'z'

String literals: Text inside double quotes, like "Hello", "World"

Boolean literals: true or false

Null pointer literal: null ptr

Example:

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10;
    double pi = 3.14;
    char letter = 'A';
    string word = "Hello";

    cout << "a: " << a << ", pi: " << pi << ", letter: " << letter << ", word: " << word << endl;
    return 0;
}
```

❖ Control Flow Statements

1. Conditional statements in C++:

→ Conditional statements in C++ are used to execute certain blocks of code based on whether a condition is true or false.

→Conditional statements in C++ allow you to make decisions in your code.

1. If-Else Statement

→An if-else statement checks a condition and executes different blocks of code based on whether that condition is true or false**.**

Example:

```
#include <iostream>
using namespace std;
int main()
{
   int number;
   cout << "Enter a number: ";
   cin >> number;
   if (number > 0)
   {
      cout << "The number is positive!" << endl;
   }
   else
   {
```

```
        cout << "The number is negative!" << endl;
    }
}
```

Output:

Enter a number: 5

The number is positive!

Or

Enter a number: -3

The number is negative!

2. Switch Statement

→A switch statement is useful when you have several conditions based on the value of a single variable**.**

→You don't need to write multiple if statements.

Example 2

```
#include <iostream>
using namespace std;
int main()
{
    int day;
    cout << "Enter a number (1-7) to choose a day: ";
    cin >> day;
```

```
switch(day)
{
    case 1:
        cout << "Monday" << endl;
        break;
    case 2:
        cout << "Tuesday" << endl;
        break;
    case 3:
        cout << "Wednesday" << endl;
        break;
    case 4:
        cout << "Thursday" << endl;
        break;
    case 5:
        cout << "Friday" << endl;
        break;
    case 6:
        cout << "Saturday" << endl;
        break;
    case 7:
        cout << "Sunday" << endl;
        break;
```

```
        default:
            cout << "Invalid day! Please enter a number between 1 and 7." << endl;
            break;
    }
}
```

Output:

Enter a number (1-7) to choose a day: 3

Wednesday

Or

Enter a number (1-7) to choose a day: 8

Invalid day! Please enter a number between 1 and 7.

2. Difference between for, while, and do-while loops in C++

1. for loop:

→**use**: When you know the number of iterations in advance.

→**Condition check**: It checks the condition **before** running the loop.

Syntax:

```
for (initialization; condition; increment)
{
    // loop body
}
```

Example:

```
for (int i = 0; i < 5; i++)
{
        cout << i <<" ";
}
```

2. while loop:

→**use**: When the number of iterations is not known in advance, and you want to keep looping as long as a condition is true.

→**Condition check**: It checks the condition before running the loop.

Syntax:

```
while (condition)
{
        // loop body
}
```

Example:

```
int i = 0;
while (i < 5)
{
        cout << i << " ";
        i++;
}
```

3. do-while loop:

→**use**: When you want the loop to run at least once before checking the condition.

→**Condition check**: It checks the condition after running the loop.

Syntax:

```
do
{
    // loop body
} while (condition);
```

Example:

```
int i = 0;
do
{
    cout << i << " ";
    i++;
} while (i < 5);
```

3.Break and continue statements used in loops

1. break Statement:

→The break statement is used to exit the loop entirely, regardless of the loop's condition.

→It terminates the loop immediately and the program continues with the next line of code after the loop.

Example:

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

int main()
{
    int i;
   for(i=1; i<10; i++)
  {
        if (i==5)
        {
                break;
          }
          cout<<i<<endl;
    }
}
```

2. continue Statement:

→The continue statement is used to skip the current iteration and jump to the next iteration of the loop.

→It does not terminate the loop, but simply skips the remaining code in the loop for that particular iteration.

Example:

```
#include <iostream>
using namespace std;
int main()
{
   int i;
   for(i=1; i<10; i++)
  {
      if(i==5)
      {
         continue;
      }
      cout<<i<<endl;
   }
}
```

4. nested control structures.

→A nested control structure is when one control statement is placed inside another control statement.

→This allows you to check multiple conditions in a layered way.

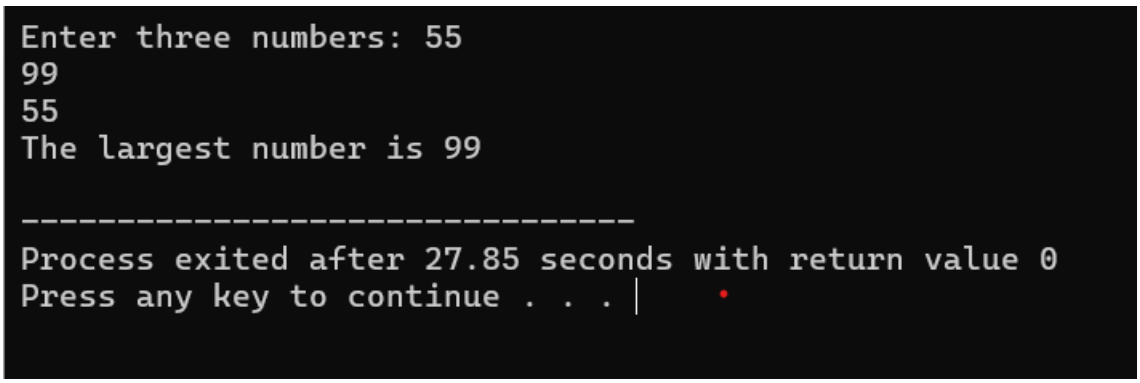
Example:

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

```
{
  int a, b, c;
  cout<<"Enter three numbers: ";
  cin>>a>>b>>c;
  if(a>=b)
  {
     if(a>=c)
     {
          cout <<"The largest number is "<<a<<endl;
      }
     else
     {
           cout<<"The largest number is "<<c<<endl;
     }
  }
  else
  {
      if(b>=c)
     {
          cout<<"The largest number is "<<b<<endl;
     }
     else
     {
```

```
            cout<<"The largest number is "<<c<<endl;
        }
    }
}
```

Output:



❖ Functions and Scope

1. function in C++

→A function is a block of code that performs a specific task.

→Functions help break down complex problems into smaller, manageable parts.

A. Function Declaration

→A function declaration is a statement that tells the C++ compiler about the function's name, return type, and parameters.

→It does not contain the actual code that performs the task.

→it just gives the compiler enough information to know how to call the function.

Syntax:

return_type function_name(parameter_list);

2. Function Definition

→The function definition provides the actual code that defines what the function does.

→This is where the logic of the function is written.

Syntax:

```
return_type function_name(parameter_list)
{
    // function body (code to perform the task)
}
```

3. Function Calling

→ A function call is the way you invoke or use a function in your program.

→ When you call a function, you provide the arguments for the parameters, and the function performs the task and possibly returns a value.

Syntax:

```
function_name(argument_list);
```

Example:

```
#include <iostream>
using namespace std;
int add (int a, int b);
int main ()
{
   int result = add (3, 4);
```

```
    cout << "The result is: " << result << endl;
}
int add (int a, int b)
{
    return a + b;
}
```

2. Scope of variables in C++.

→ In C++, the **scope** of a variable refers to the region or part of the program where the variable can be accessed or modified.

→ The **scope** defines the visibility and lifetime of a variable.

→ There are two main types of variable scope:

1.Local Scope

2.Global Scope

1. Local Scope:

→ A local variable is a variable that is declared within a function, block, or any curly braces {}.

→ The **scope** of a local variable is limited to the block or function where it is defined.

→ This means the variable is only accessible inside the function or block where it was declared and cannot be accessed outside of it.

Example:

```
#include <iostream>
using namespace std;
void myFunction()
```

```
{
   int localVar=10;
   cout<<"Inside myFunction, localVar: "<<localVar<<endl;
}
int main()
{
   myFunction();
}
```

2. Global Scope

→ A **global variable** is declared outside of all functions, usually at the top of the program.

→ The **scope** of a global variable extends throughout the entire program, meaning it can be accessed by any function or block of code within the program.

Example:

```
#include <iostream>
using namespace std;
int globalVar = 20;
void myFunction()
{
   cout<<"Inside myFunction, globalVar: "<<globalVar<<endl;
}
int main()
{
```

```
    cout<<"In main, globalVar: "<<globalVar<<endl;
    myFunction();
}
```

Difference Between Local and Global Scope:

Aspect	Local Scope	Global Scope
Declaration Location	Declared inside a function or block {}	Declared outside any function, typically at the top of the program
Accessibility	Accessible only within the function/block where it is declared	Accessible anywhere in the program after it is declared
Lifetime	Exists only during the execution of the block/function	Exists throughout the program, until it finishes executing
Default Value	Uninitialized	Automatically initialized to zero if not explicitly initialized
Memory	Memory is allocated when the function/block is executed and freed when it exits	Memory is allocated when the program starts and freed when the program ends

3.Recursion in C++ with an example.

→ Recursion in C++ is when a function calls itself in order to solve a problem.

→ It allows you to break a complex problem into smaller, more manageable parts.

→It continues calling itself with modified parameters until a base case is reached.

Steps in Recursion:

1. **Base case**: The condition under which the recursion stops. Without a base case, the recursion would continue indefinitely and cause a stack overflow.
2. **Recursive case**: The part of the function where the function calls itself to reduce the problem into a smaller, easier sub-problem.

Example:

```
#include <iostream>

using namespace std;

int factorial(int n)

{

   if(n==0)

   {

      return 1;

   }
```

```
    return n*factorial(n-1);
}
int main()
{
    int num;
    cout<<"Enter a number: ";
    cin>>num;
    cout<<"Factorial of "<<num<<"is"<<factorial(num)<<endl;
}
```

3. function prototypes in C++.

→In C++, a **function prototype** is a declaration of a function that specifies the function's name, return type, and parameters, but without providing the function's body.

→It serves as a forward declaration, telling the compiler about the function's signature before the function is actually defined or called in the code.

Syntax:

return_type function_name(parameter_list);

Why Function Prototypes are Used:

1.Enable Function Calls Before Definition:

→Function prototypes allow you to declare and call functions before they are defined in the code.

→This is especially useful when you want to organize your code in a structured manner, such as splitting it into multiple files.

2.Type Checking:

→Prototypes help the compiler check the types of function arguments and return values before calling the function.

→This prevents errors that could arise from mismatched data types or incorrect number of arguments.

3.Better Code Organization:

→Prototypes make the code more modular and maintainable.

→You can declare functions at the top of your file or in a separate header file, which allows you to focus on the function implementation later, making the code easier to manage and debug.

4.Support for Multiple Source Files:

→In larger programs with multiple files, function prototypes are crucial for enabling communication between different parts of the code.

Example:

```
#include <iostream>

using namespace std;

int add(int a, int b);

int subtract(int a, int b);

int multiply(int a, int b);

int main()
```

```
{
    int num1=10, num2=5;
    int sum=add(num1, num2);
    int difference=subtract(num1, num2);
    int product=multiply(num1, num2);
    cout<<"Sum: "<<sum<<endl;
    cout<<"Difference: "<<difference<<endl;
    cout<<"Product: "<<product<<endl;
}
int add(int a, int b)
{
    return a+b;
}
int subtract(int a, int b)
{
    return a-b;
}
int multiply(int a, int b)
{
    return a*b; }
```

Arrays and Strings:

1.Arrays in C++:

→An **array** in C++ is a collection of elements, all of the same type, stored in contiguous memory locations.

→The elements in an array are accessed using an index, with the first element starting at index 0.

→Arrays allow you to store and manage multiple values in a single variable, which is especially useful when you need to work with large sets of data.

How to declare an array:

type arrayName[size];

A.Single-Dimensional Arrays

→A **single-dimensional array** (1D array) is a simple array that stores a list of values in a linear sequence. Each element is accessed using a single index.

Example:

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

B.Multi-Dimensional Arrays

→A **multi-dimensional array** is an array of arrays, meaning you have more than one index to access elements.

→These arrays are used to represent more complex data structures, like matrices.

Example:

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

Difference Between Single-Dimensional and Multi-Dimensional Arrays:

Feature	**Single-Dimensional Array (1D)**	**Multi-Dimensional Array (2D, 3D, etc.)**
Structure	A simple list of elements	A list of lists (or tables, grids, etc.)
Indexing	Uses a single index (e.g., arr[0])	Uses multiple indices (e.g., arr[0][1])
Use Case	For a sequence of data, like a list	For structured data, like matrices or grids
Accessing Elements	arr[index]	arr[index1][index2] (or more for higher dimensions)
Example	int arr[3] = {1, 2, 3};	int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};

2. string handling in C++.

→String handling in C++ involves manipulating sequences of characters using either C-style strings or C++'s more powerful std::string class.

1. C-style Strings in C++

→A C-style string is an array of characters, ending with a special null character ('\0') to indicate the end of the string.

Declaring and Initializing C-style Strings:

```
#include <iostream>

using namespace std;

int main()

{

   char str[] = "Hello, World!";

   cout << str << endl;

}
```

2. C++ std::string Class

→The std::string class from the C++ Standard Library is a more flexible and safer way to handle strings.

→It provides many useful functions and automatically manages memory.

Declaring and Initializing a std::string:

```
#include <iostream>

class using namespace std;

int main()
```

```
{
    string str = "Hello, World!";
    cout << str << endl; }
```

3. Basic String Operations in C++

→You can concatenate strings using the + operator or append() function.

Example:

```
#include <iostream>
using namespace std;
 int main()
{
    string str1 = "Hello, ";
    string str2 = "World!";
    string result = str1 + str2;
    cout << result << endl;
    str1.append(str2);
    cout << str1 << endl;
}
```

4. String Comparison in C++

→You can compare strings using relational operators (==, !=, <, >, etc.).

Relational Operators:

- == checks if two strings are equal.

- != checks if two strings are not equal.
- < and > check lexicographical (dictionary) order.

5. Finding Substrings in C++

→You can search for substrings using the find() method.

→It returns the index of the first occurrence or std::string::npos if the substring is not found.

Example:

```
#include <iostream>
using namespace std;
int main()
{
    string str = "Hello, World!";
    size_t pos = str.find("World");
    if (pos != string::npos)
    {
        cout << "'World' found at position: " << pos << endl;
        // Output: 'World' found at position: 7
    }
    Else
    {
        cout << "'World' not found." << endl;
    }
}
```

6. String Iteration

→You can iterate over the characters of a string using a loop.

Example:

```
#include <iostream>
using namespace std;
int main()
{
	string str = "Hello";
	for (char c : str)
	{
		cout << c << " ";
	}
 }
```

3.Arrays initialized in C++

→In C++, arrays are initialized in several ways.

→Let's break down the methods for initializing both 1D and 2D arrays.

1D Array Initialization:

→A 1D array is a simple linear structure with a single dimension.

→The initialization can be done in different ways:

Example:

```
#include<iostream>
main()
```

```
{
   int arr[]={1, 2, 3, 4, 5};
   for(int i=0; i<5; ++i)
   {
      std::cout<<arr[i]<<" ";
   }
}
```

Output:

1 2 3 4 5

2D Array Initialization

→A 2D array is like an array of arrays, having both rows and columns.

→You can initialize it similarly, but with nested braces for each row.

Example:

```
#include<iostream>
main()
{
   int arr[3][3]=
   {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
      };
```

```
        for(int i=0; i<3; ++i)
       {
              for(int j=0; j<3; ++j)
             {
                    std::cout<<arr[i][j]<<" ";
                    }
                    std::cout<<std::endl;
      }
}
```

Output:

1 2 3

4 5 6

7 8 9

4.string operations and functions in C++.

1. C-Style Strings:

→In C++, a C-style string is just an array of characters that ends with a special character called the **null terminator** ('\0').

Basic Operations with C-Style Strings:

A. Initialization:

```
char str[] = "Hello";
```

B. Length (strlen):

```
size_t len = strlen(str);
```

C. Copying Strings (strcpy):

```
char str2[20];
strcpy(str2, str);
```

D. Concatenation(strcat):

```
char str3[30] = "Hello";
strcat(str3, " World!");
```

E. Comparing Strings (strcmp):

```
int result = strcmp("Hello", "World");
```

2. C++ String Class (std::string)

→The std::string class in C++ is much easier to use than C-style strings.

→It automatically manages the size and memory of the string for you.

Basic Operations with std::string:

A. Initialization:

```
std::string str = "Hello";
```

B. Length (size() or length()):

```
size_t len = str.size();
```

C. Concatenation:

```
std::string str2 = "World";
std::string str3 = str + " " + str2;
```

D. Accessing Characters ([] or at()):

```
char ch = str[0];
char ch2 = str.at(1);
```

E. Substring (substr):

```
std::string subStr = str.substr(1, 3);
```

F. Finding Substring (find):

```
 size_t pos = str.find("llo");
```

G. Replacing Part of the String (replace):

```
str.replace(0, 5, "Hi");
```

H. Convert to Number and Back (std::to_string and stoi):

```
int num = 123;
std::string numStr = std::to_string(num);
 std::string strNum = "456";
int num2 = std::stoi(strNum);
```

Example:

```
#include<iostream>
main()
{
    std::string str = "Hello";
    std::cout << "Length: " << str.size() << std::endl;
    std::string str2 = " World";
    std::string result = str + str2;
    std::cout << "Concatenated: " << result << std::endl;
    std::cout << "First character: " << str[0] << std::endl;
    std::string subStr = result.substr(6, 5);
    std::cout << "Substring: " << subStr << std::endl;
```

```
    size_t pos = result.find("World");
    if (pos != std::string::npos)
{
        std::cout << "Found 'World' at position: " << pos << std::endl;
    }
    result.replace(0, 5, "Hi");
    std::cout << "Replaced: " << result << std::endl;
}
```

Output:

Length: 5

Concatenated: Hello World

First character: H

Substring: World

Found 'World' at position: 6

Replaced: Hi World

❖ Introduction to Object-Oriented Programming

1. Key concepts of Object-Oriented Programming (OOP)

→ Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around **objects**, which are instances of **classes**.

→ Here are the key concepts of OOP explained in an easy way:

1. Class

→ Think of a **class** as a blueprint or a template for creating objects.

→ It defines the properties and behaviours that the objects created from it will have.

2. Object

→ An **object** is an instance of a class.

→ Once you have a class, you can create many objects from it.

→ Each object can have different values for its properties, but it shares the same behaviours defined in the class.

3. Encapsulation

→ **Encapsulation** is the concept of keeping an object's internal state hidden from the outside world.

→It allows you to control how data is accessed or modified through methods.

4. Abstraction

→ **Abstraction** is about hiding complex details and showing only the essential features of an object.

→ It simplifies the interface for the user.

5. Inheritance

→ **Inheritance** allows one class to inherit properties and behaviours from another class.

→ This helps to reuse code and create a hierarchy of classes.

6. Polymorphism

→ **Polymorphism** means "many forms".

→ It allows objects of different classes to be treated as objects of a common superclass.

→ The same method or operation can behave differently based on the object that calls it.

7. Methods

→ **Methods** are functions that are defined within a class.

→ They define behaviours or actions that an object can perform.

2. classes and objects in C++

1. Class

→A **class** is a blueprint or template for creating objects.

→It defines the properties and behaviors that the objects created from the class will have.

→It encapsulates data and functions that manipulate that data.

2. Object

→An **object** is an instance of a class.

→When you create an object, you are creating a specific example of the class, with its own unique data.

Example:

```
#include <iostream>
using namespace std;
class Car
{
    public:
```

```
    string brand;
    int year;
    string color;
     void drive()
    {
    cout<<"The"<<color<<" "<<brand<<" is driving!"<<endl;
    }
};
main()
{
   Car myCar;
   myCar.brand="Toyota";
   myCar.year=2020;
   myCar.color="red";
   myCar.drive();
}
```

Output:

Thered Toyota is driving!

3. Inheritance in C++?

→Inheritance is a fundamental concept in **OOP** that allows a class to inherit properties and behaviors from another class.

→This helps in reusing code, enhancing code organization, and creating hierarchical relationships between classes.

→In C++, inheritance is implemented using a special relationship where one class inherits the features of another class.

Types of Inheritance:

1. **Single Inheritance**: A derived class inherits from one base class.
2. **Multiple Inheritance**: A derived class inherits from more than one base class.
3. **Multilevel Inheritance**: A class is derived from another derived class.
4. **Hierarchical Inheritance**: Multiple derived classes inherit from a single base class.
5. **Hybrid Inheritance**: A combination of different types of inheritance.

Example:

```
#include <iostream>
using namespace std;
class Animal
{
public:
    void eat()
        {
        cout<<"This animal eats food."<<endl;
    }
```

```
};
class Dog:public Animal
{
public:
    void bark()
        {
        cout<<"The dog barks."<<endl;
    }
};
int main()
{
    Dog dog;
    dog.eat();
    dog.bark();
}
```

Output:

This animal eats food.

The dog barks.

4.Encapsulation in C++

→Encapsulation is one of the four fundamental principles of Object-Oriented Programming (OOP).

→. It is the process of bundling data and methods that operate on the data into a single unit or class.

→Additionally, it restricts direct access to some of an object's components, which is usually done through access control mechanisms.

→This allows you to hide the internal implementation details of a class and only expose a controlled interface to the outside world.

In C++, encapsulation is achieved using:

1. Private/Protected Members:

→Data and methods are kept private or protected to prevent unauthorized access and modification from outside the class.

2. Public Methods:

→These methods allow controlled access to private data members.

Example:

```
#include <iostream>
using namespace std;
class Person
{
   private:
        string name;
        int age;
   public:
        void setName(string n)
        {
             name=n;
```

```
    }
    void setAge(int a)
    {
        if(a>0)
        {
            age=a;
        }
        else
        {
            cout<<"Age must be positive!"<<endl;
        }
    }
    string getName()
    {
        return name;
    }
     int getAge()
    {
        return age;
    }
};
int main()
{
```

```
    Person p;
    p.setName("krisha");
    p.setAge(21);
    cout<<"Name: "<<p.getName()<<endl;
    cout<<"Age: "<<p.getAge()<<endl;
}
```

Output:

Name: krisha

Age: 21

****