Date
Page

COMPLETE NOTES ON 'C++'

Instagram : @ Curios_ . programer

Telegram : @ Curious_Coder

Written by :

Sai Sumanth (IT student)

Index

INTRODUCTION TO "C++"

* What is C++ programming Language?

→ C++ is a programming language that allows you to create software and programs for computers.

→ It's a popular language because it is versatile and powerful, giving you the ability to build a wide range of applications.

→ C++ is known for its efficiency and performance, making it a great choice for tasks that require speed and resource management.

→ It is also an object-oriented language, which means you can organize your code into reusable pieces called classes, making it easier to develop and maintain large projects.

→ Overall, C++ is a valuble language for beginners to learn as it provides a solid foundation in programming concepts and offers many possibilities for future projects.

1. Introduction to C++

1.1 Overview of C++

→ C++ is a powerful and popular programming language that allows developers to create efficient and robust software applications.

→ It is an extension of the C programming language, adding features that support object-oriented programming (OOP) and other advanced programming paradigms.

→ C++ is widely used in various domains, including system programming, game development, embedded systems, and scientific computing.

→ C++ provides a rich set of features that make it suitable for a wide range of applications.

→ It offers low-level control over hardware resources, high-level abstractions for complex problem-solving, and a balance between performance and productivity.

→ With its versatility and extensive libraries, C++ is an excellent choice for both beginners and experienced programmers.

→ This is the wide overview about C++ programming language.

1.2 History and Evolution of C++

→ C++ has a fascinating history that dates back to early 1980s when it was developed by Bjarne Stroustrup at Bell Laboratories.

→ Stroustrup wanted to enhance the C language to support object-oriented programming, inspired by languages like simula and smalltalk.

→ The first version of C++ was named "C with Classes" and introduced the concept of classes and objects.

→ Over the years, C++ evolved through various standardized versions, each introducing new features and improvements.

→ The ANSI/ISO standardization of C++ in 1998 (known as C++98) played a significant role in establishing C++ as a mature and widely adopted programming language.

→ Subsequent versions, such as C++11, C++14, C++17, and C++20, brought additional features and enhancements to the language.

1.3 Features of C++

→ C++ incorporates a wide range of features that enable developers to write efficient, maintainable, and reusable code.

→ Some notable features of C++ include:

a) Object-Oriented programming (OOP):

→ C++ supports the principles of OOP, such as encapsulation, inheritance, and polymorphism.

→ It allows you to create classes, objects and use them to model real-word entities and interactions.

b) Strong Typing:

→ C++ enforces strict type checking, which helps catch errors at compile-time and ensures safer code execution.

c) Templates:

→ C++ provides template support, allowing you to create generic classes and functions that can work with different data types.

→ Templates enable code reusability and provide a powerful mechanism for generic programming.

d.) Standard Library:

→ C++ comes with a comprehensive standard library that provides a wide range of ready-to-use functions and data structures.

→ The standard library includes containers (such as vectors, lists and maps), algorithms, input/output operations and more.

1.4 Setting up a C++ Development Environment

→ To start programming in C++, you need to set up a development environment on your computer.

→ Here are the general steps to get started:

a.) Choose a Text Editor or Integrated Development Environment (IDE):

→ You can use or use a simple text editor like Notepad++ or a specialized IDE like Visual Studio code, Eclipse, or Code::Blocks.

→ IDEs provide additional features such as code highlighting, debugging tools, and project

management capabilities.

6) Install a C++ Compiler:

→ A C++ compiler translates your source code into machine-readable instructions.

→ Popular C++ compilers include GCC (GNU Compiler Collection), Clang, and Microsoft Visual C++.

→ You can choose the one that suits your operating system.

c) Write and Compile Your 1st C++ program:

→ Once your development environment is set up, you can write a simple C++ program, often referred to as a "Hello, World!" program, to test your setup.

→ Compile the program using the C++ compiler, and if everything is set up correctly, you should be able to run the executable and see the output.

→ By following these steps, you will have a basic C++ program.

2. Basic Concepts.

2.1 Structure of a C++ program.

→ A C++ program is composed of various elements that work together to accomplish a specific task.

→ Example for structure of simple C++ program:-

```
#include <iostream>

int main()
{
    // Program Statements
    return 0;
}
```

→ The '#include <iostream>' directive tells the compiler to include the input/ouput stream library, which provides functions for input and output operations.

→ The 'int main()' function is the entry point of a C++ program. It is where the program execution begins.

→ The 'int' indicates that the function should return an integer value. The 'main()' function is enclosed within curly braces '{}'.

→ The program statements are written inside

the curly braces. These statements define the actions and logic of the program.

→ The "return 0;" statement is used to indicate the successful termination of the program. The '0' represents the exit status of the program.

2.2 Variables, Data Types, and Constants.

→ Variables are used to store and manipulate data in a program.

→ In C++, you need to declare variables before using them.

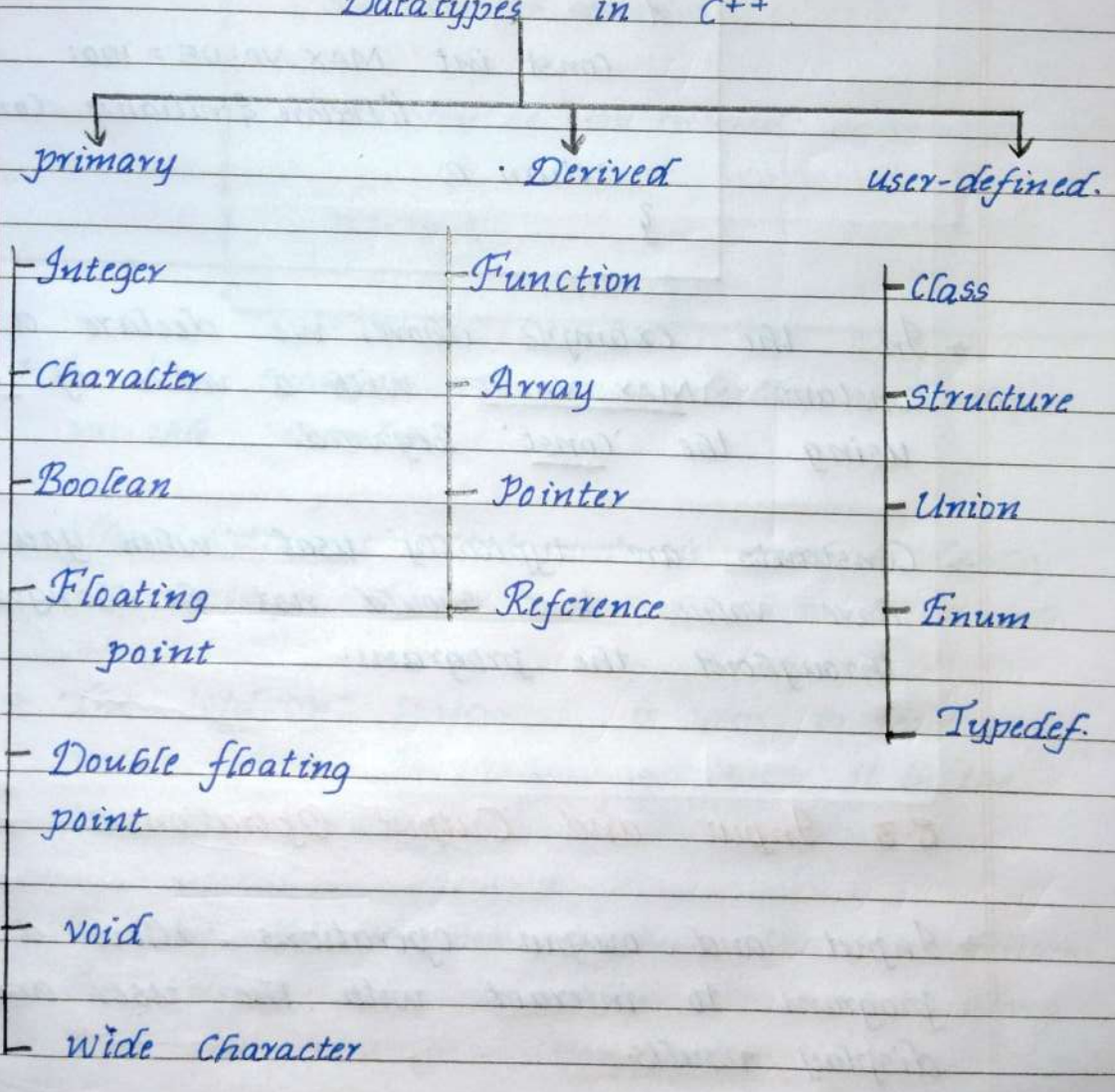
→ Example for Variable declaration and initialization:

```
#include <iostream>

int main()
{
  int age = 25;          // Declaration & Initialization
  double weight = 68.5;  //      "
  char grade = 'A';      //      "

  return 0;
}
```

→ In the example above, we declare variables 'age', 'weight', and 'grade' of types 'int',

'double', and 'char', respectively.

→ we also initialize these variables with specific values. for example, 'age' is assigned the value '25', 'weight' is assigned '68.5', and 'grade' is assigned 'A'.



Constants:

Constants are fixed values that cannot be modified during program execution. They

provide a way to represent fixed values or literals in the program.

Ex:-

```
#include <iostream>

int main()
{
    const int MAX_VALUE = 100;
                // Declare & initialize Constant.
    return 0;
}
```

→ In the example above, we declare a constant 'MAX_VALUE' with a value of '100' using the 'const' keyword.

→ Constants are typically used when you have values that should not be changed throughout the program.

2.3 Input and Output Operations.

→ Input and output operations allow a program to interact with the user and display results.

→ The 'iostream' library provides functions like 'cin' and 'cout' for input and output, respectively.

Ex:

```
#include <iostream>

int main()
{
    int number;
    std::cout<< "Enter a number:";
    std::cin >> number;

    std::cout<< "you entered: "<< number<<endl;

    return 0;
}
```

→ In the example above, we declare an integer variable "number".

→ The 'std::cout' statement is used to display the message "Enter a number:" to the console.

→ The 'std::cin' statement is used to read a value from the user and store it in the 'number' variable.

→ Finally, the program displays the entered number using the 'std::cout' statement along with the message 'You entered:'.

→ This is the breif introduction about basic input / output operations using the library '<iostream>'.

2.4 Operators and Expressions

→ Operators in C++ are symbols or keywords that perform specific operations on operands.

→ Some examples of operators:-

1. Arithmetic Operators:-

These operators perform basic arithmetic operations like Addition, Subtraction, division, Multiplication etc.

Operators :- '+', '-', '*', '/'

2. Relational operators:-

These Operators will determine the relationship between the operands.

Operators:- '<', '>', '==', '!='

3. Assignment Operators:-

These operators will increament or decrement or assign the values from one operand to another.

Operators:- '=', '+=', '-='

Expressions:-

Expressions are combination of variables, constants, and operators that evaluate to a single value.

Ex:-

```
int x=5;
int y=3;
int result = x+y; // Expression : x+y
```

2.5 Control Structures.

→ Control structures enable you to control the flow of execution in a program.

→ Two common control statements (if-else statements) and looping statements (for loop, while loop, do-while loop).

2.5.1 Decision-Making Statements.

→ Decision-making statements allow the program to make decisions based on certain conditions.

→ The most common decision-making statement is the 'if-else' statement and also we can also use switch statements.

Ex:-

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Enter a number: ";
    cin >> num;
```

```
if (num > 0)
  {
    cout << "number is positive" << endl;
  }
else if (num < 0)
  {
    cout << " number is negative" << endl;
  }
  else
  {
    Cout << "number is zero" << endl;
  }
return 0;
}
```

Output:

```
Enter a number : 5
number is positive.

program executed successfully with
return status 0 .....
```

Switch:

We can make use of different cases with in a single switch statement along with the default case.

Syntax:

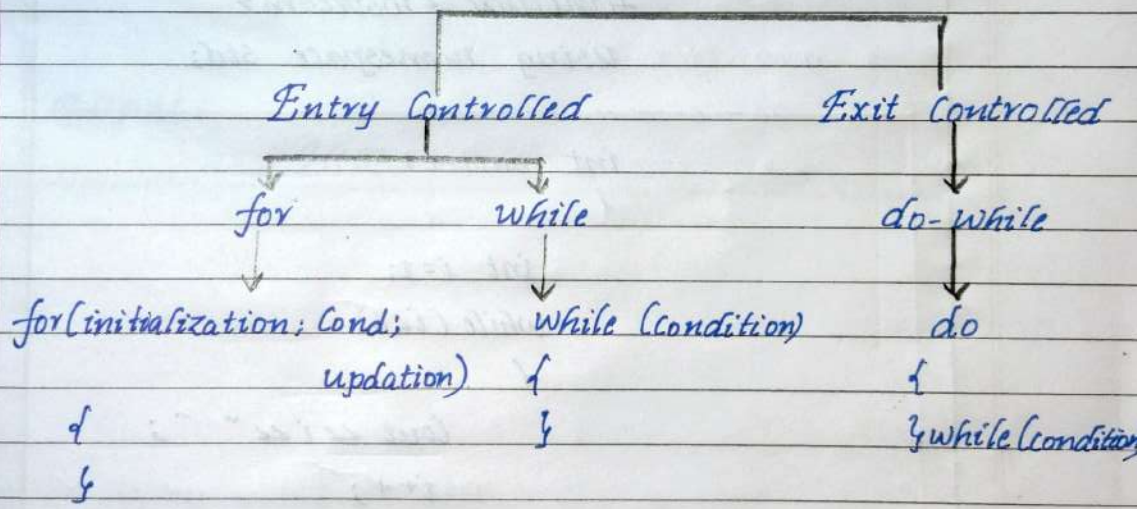
```
switch (n)
  {
    case 1: //code
            break;
```

```
case 2 : //code
         break;
  :
  :
default : //code
          break;
};
```

2.5.2 Looping Statements.

→ Looping statements allow you to repeat a block of code multiple times.

→ There are mainly 3 types of loops:



Ex:

```
for-loop.
#include <iostream>
using namespace std;
int main()
{
    int i=99;
```

```
for (i=0; i<5; i++)
 {
     cout<<i<<" ";
 }
 cout<<"\n"<<i;
 return 0;
}
```

Output:

```
0 1 2 3 4    // inside loop values.
5            // i-value after completing loop.
```

While - loop.

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

int main()
{
    int i=1;
    while (i<6)
    {
        cout<<i<<" ";
        i++;
    }
    return 0;
}
```

Output:

```
1 2 3 4 5.
```

do - while loop:

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

int main()
{
    int i=8;
    do
    {
        cout<< "Hello world \n";
        i++;
    } while(i<1);
return 0;
}
```

Output:

Hello world.

3. Functions and Scope.

3.1 Introduction to Functions.

→ Functions play a crucial role in C++ programming as they allow us to break down the complex tasks into smaller, manageable piece of code.

→ A function is a block of code that performs a specific task and can be called from different parts of program.

→ By using functions, you can acheive code reusability, modularity and improve readability of your code.

Ex:

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

void sayHello()
{
    cout << "Hello world" << endl;
}
int main()
{
    sayHello();
    return 0;
}
```

Output:

```
Hello world.
```

→ In this example, 'sayHello()' function is declared before the 'main()' function and defined later in code.

→ The function prints "Hello world" to console. The 'main()' function calls the 'sayHello()' function, and when executed, it will output the message.

3.2 Function Declaration and Definition.

→ A function declaration tells the compiler about the name of the function, the parameters it takes (if any), and the return type.

→ The Function definition provides the actual implementation of the function.

Ex:-

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

// function declaration.

int add(int a, int b);
                    // Function protype
int main()
{
   int x = 5;
   int y = 3;
```

```
// Function Call
  int result = add (x, y)

  cout << "Sum : " << result << endl;
 return 0;
}
  // Function definition.

  int add (int a, int b)
   {
       return a+b;
   }
```

done in background

output: 5+3 = 8.

→ In this example, the 'add()' function is declared before 'main()' funtion using a function prototype.

→ The function prototype specifies the function name, the types of its parameters, and the return type.

→ The actual implementation of the 'add()' function is defined later in the code.

3.3 Function Parameters and Return types.

→ Functions can have parameters, which are variables used to pass values into the function.

→ Parameters allow functions to receive input

and perform operations on them.

→ Functions can also have a return type, which specifies the type of data the function will return after its execution.

→ Example of "Function with parameters & a return type.

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

// Function declaration.
int multiply (int a, int b);

int main()
{
    int x = 5;
    int y = 3;

    // Function call
    int result = multiply (x, y);
cout << "product : " << result << endl;

    return 0;
}

// Function definition

int multiply (int a, int b)
{
    return a*b;
}
```

Output:

product : 15

→ In this example, the 'multiply()' function takes two parameters ('a' and 'b') of type 'int'.

→ The function multiplies the two values and returns the result as an 'int'.

→ The 'main()' function calls the 'multiply()' function, passing the values 'x' and 'y', and stores the returned result in the 'result' variable.

3.4. Function Overloading

→ Function overloading allows you to define multiple functions with the same name but different parameter lists.

→ The compiler determines which function to call based on the number, types, and order of the arguments passed during the function call.

→ Function Overloading provides flexibility and simplifies code maintainance.

→ Example for function Overloading:-

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

//Function declaration
int add (int a, int b);
double add (double a, double b);

int main()
{
   int x = 5;
   int y = 3;
   double a = 2.5;
   double b = 1.5;

    //Function call
    int result1 = add (x,y);
    double result2 = add (a,b);

   cout << "Sum (int): " << result1 << endl;
   cout << "Sum (double): " << result2 << endl;

   return 0;
}
//Function definition.
int add (int a, int b)
{
    return a+b;
}

double add (double a, double b)
{
    return a+b;
}
```

Output:

Sum(int) : 8
Sum(double) : 4.0

→ In this example, the 'add()' function is overloaded with 2 versions,
  i) one that takes two 'int' parameters and returns an 'int' result.
  ii) another that takes two 'double' result.

→ The appropriate version of the function is called based on the argument types during the function call.

3.5 Recursion.

→ Recursion is a powerful technique in which a function calls itself directly or indirectly.

→ It allows you to solve complex problems by breaking them down into simpler subproblems.

→ Recursions or recursive functions have two essential components:
  i) a base case that defines termination condition.
  ii) recursive calls that solve

Smaller instances of the problem.

Example:

factorial of a number.

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

// Function declaration
int factorial(int n);

int main()
{
    int num;
    cout << "Enter a positive number: ";
    cin >> num;

    // Function call
    int result = factorial(num);
    cout << "Factorial: " << result << endl;
    return 0;
}
// Function definition
int factorial(int n)
{
    if (n == 0)
    {
        return 1; // Base case
    }
    else
    {
        return n * factorial(n-1); // Recursive
    }
}
```

Output:

Enter a positive number : 5
120.

→ In this example, the 'factorial()' function calculates the factorial of a given number using recursion.

→ The function checks for the base case when 'n' equals '0' and returns 1.

→ Otherwise, it recursively calls itself with 'n-1' until it reaches the base case.

3.6 Scope of Variables

→ The scope of a variable refers to the region of a program where the variable is accessible.

→ In C++, variables can have local scope or global scope.

→ Local variables are defined within a specific block or function and can only be accessed within that block or function.

→ Global variables are declared outside any function and can be accessed from anywhere in the program.

Ex:-

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

//Global variable
int globalVar = 10;

//Function declaration
void testFunction();

int main()
{
    //Local variable
    int localVar = 5;

    cout << "Local variable: " << localVar << endl;
    cout << "Global variable: " << globalVar << endl;

    testFunction();
    return 0;
}

//Function definition
void testFunction()
{
    //Accessing global variable.
    cout << "global variable inside function: " << globalVar;
}
```

Output:-

```
local variable :5
global variable :10
global variable inside function :10
```

→ In this example, 'localVar' is a local variable that is accessible only within the 'main()' function.

→ The 'globalVar' is a global variable that can be accessed from both the 'main()' function and the 'testFunction()' function.

→ Understanding functions and the scope of variables is essential for creating modular and well-structured programs.

→ It enables you to break down complex tasks, reuse code, and control the accessibility of variables within your program.

→ Therefore, Global variables are declared ouside the all functions including main() after the header files.

→ Local variables are declared only inside the functions where it is needed.

4. Arrays and Strings.

4.1 Introduction to Arrays.

→ Arrays are a fundamental concept in programming that allow you to store multiple values of the same data type in a continous block of memory.

→ They provide a convenient way to organize and access related data.

→ In C++, arrays have a fixed size and can store elements of any data type, such as integers, characters, or even user-defined types.

→ Ex:-

Declaring and accessing array.

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

int main()
{   // Array initialization and declaration.
    int num[5] = { 1, 2, 3, 4, 5 }

    // Accessing array elements.
    cout << "First element : " << num[0] << endl;
    cout << "Second element : " << num[1] << endl;

    return 0;
}
```

Output:

First element : 1
Second element : 2.

-> In this example, we declare an integer array named 'numbers' with a size of 5.

-> The array is initialized with values 1, 2, 3, 4 and 5.

-> We can access individual elements of the array using index notation, where the index starts from 0.

-> For example, num[0] refers to the first element of the array.

4.2 Array Declaration, Initialization, and Access.

-> To declare an array, you specify the data type of its elements, followed by the array name and the size of the array in square brackets.

-> Array initialization can be done at the time of declaration using curly braces '{}'.

Ex:

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

```
int main()
{
    // Array declaration.
    int array[3];

    // Array initialization.
    array[0] = 10;
    array[1] = 20;
    array[2] = 30;

    // Accessing array elements.

    cout<<"First element :"<<array[0]<<endl;
    cout<<"Second element :"<<array[1]<<endl;
    cout<<"Third element :"<<array[2]<<endl;

    return 0;
}
```

Output :

```
First element :10
Second element :20
Third element :30
```

→ In this example, we declare an integer array "array" with a size of 3.

→ Then, we initialize its elements individually in the 'main()' function.

→ Finally, we access and print the values of the array elements using index notation.

4.3 Multi dimensional Arrays.

→ In C++, a multidimensional array is a way to store data in a table-like structure with multiple dimensions.

→ It allows you to organize information in rows and columns, similar to a spreadsheet.

→ Each element in a multidimensional array is accessed using multiple indices that represent the dimensions.

→ For example, a two-dimensional array can be used to represent a grid, where the first index represents the row and the second index represents the columns.

→ Ex:-

```
int matrix[3][3];
matrix[3][3] = {{1,2,3},
                {4,5,6},
                {7,8,9}};
```

→ To access elements in a multidimensional array, you specify the indices for each dimension.

→ for example, to access element in 1st row and 3rd coloumn,

```
int element = matrix[1][3]; //assigns '6'
```

4.4 Introduction to Strings

→ Strings are sequence of characters.

→ In C++, strings are represented by using the 'string' class from the standard library.

→ The 'string' class provides a convenient way to create, manipulate, and perform operations on strings.

→ To use the 'string' class, you need to include the '<string>' header at the beginning of your program.

→ For example, to declare and initialize a string, you can use the following syntax,

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

int main()
{
    String greeting = "Hello, world!";

    cout << greeting <<endl;

    return 0;
}
```

Output:

Hello, world!

4.5 String Operations.

→ The 'String' class provides various operations to manipulate and work with strings.

→ Some common String operations include:

1. Concatenation :-

→ You can concatenate two strings using the '+' operator or the "+=" operator.

→ Ex:-

```
String firstname = "John";
String lastname = "Doe";
String fullname;
fullname = firstname + " " + lastname;

// Result = John Doe.
```

2. Length :-

→ you can determine the length of a string using the 'length()' or 'size()' member function.

Ex:-

```
String message = "Hello";
int length = message.length();
```

→ The length of the string assigned to the variable 'message' will be given to 'length' i.e; 5.

3. Sub String :

→ You can extract a substring from a string using the 'substr()' member function.

→ Ex:-

```
string text = "Hello, world!";
string sub = text.substr(7,5);
```

→ In the above example "7" denotes from which position in the main string it should start.
→ And "5" denotes the no. of characters it should consider in a string from "7".

Output / Result in "sub" variable is:-
world.

4.6 Character Arrays and C-style Strings.

→ In addition to the 'string' class, C++ also supports character arrays, which are also known as C-style strings.

→ A C-style string is an array of the characters terminated by a null character ('\0').

→ To declare and initialize a character

array, you need to use the following syntax:-

```
Char name[] = "John";
```

-> C-style strings can be manipulated using various string functions from the C library, such as 'strcpy()', 'strlen()' and 'strcat()'.

-> understanding arrays and strings is fundamental to many programming tasks.

-> In the next chapter, we will explore the concepts of object-oriented programming in C++ and learn about classes, objects, and more.

5. Object - Oriented Programming.

5.1 Introduction to Object-Oriented programming (OOP).

→ Object - Oriented programming (OOP) is a programming paradigm that organizes code into objects, which are instances of the classes.

→ OOP focuses on modelling real-word entities, their attributes, and their behaviours in code.

→ It provides a way to structure programs, enhance code reusability, and improve maintainability.

→ OOP is based on four main principles:-

1. encapsulation,
2. Inheritance.
3. Polymorphism.
4. Abstraction.

→ 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.

5.2 Classes and Objects

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

→ It defines the properties (data members) and behaviours (member functions) that objects of the class will have.

→ To create an object, you instantiate a class by using `class_name object_name' syntax.

→ Objects are instances of a class and represent specific entities.

→ Ex:-

Consider a 'car' class:-

```
Class Car
{
    public:
        string make;
        string model;
        int year;
};
```

To create objects of the 'Car' class:-

```
Car myCar;
myCar.make = "Honda";
myCar.model = "Civic";
myCar.year = 2021;
```

→ When a class is defined, no memory is allocated but when it is instantiated memory is allocated (i.e; an object is created).

→ Objects take up space in memory and have an associated address like record in pascal, or structure or union.

→ When a program is executed the objects interact by sending message to one another.

→ 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 the type of response returned by the objects.

5.3 Constructors and Destructors.

Constructors:

Constructors are special member functions of a class that are used to initialize objects.

→ They have the same name as the class and are called automatically when an object is created.

Ex:-

adding a constructor to 'Car' class.

```
Class Car
{
    public :
        String make;
        String model;
        int year;

    Car (String Carmake, String CarModel, int Caryear)
    {
            make = Carmake;
            model = Carmodel;
            year = Caryear;
    }
};
```

→ Now, you can create a 'Car' object using the constructor:

```
Car myCar("Honda", "Civic", 2021);
```

Destructors :-

→ Destructors are special member functions that are called automatically when an object is destroyed or goes out of slope.

→ They are used to perform any necessary cleanup tasks.

→ Destructor has the same name as their class name preceded by a tilde (~) symbol.

→ It is not possible to define more than one destructor.

→ It is automatically called when an object goes out of scope.

Syntax:

```
~<class-name>()
{
    // some instructions.
}
```

Syntax for defining the destructor outside the class:

```
<class-name> ::~<class-name>()
{
    // some instructions.
}.
```

5.4 Member Functions and Data Members.

→ Member functions are functions defined inside a class and are used to perform operations on objects of that class.

→ They can access the class's data members

and other member functions.

→ Mostly, member functions are declared in the public scope of the class.

→ We can also define member functions outside the class using the scope resolution operator ("::").

→ Data members are variables defined within a class and represent the properties or attributes of objects.

→ They hold the state or information of objects.

→ They will be mostly declared in the private scope of the class.

Ex:-

```
Class Car
{
    public:
        String make;
        String model;
        int year;

    Car(string Carmake, string Carmodel, int Caryear)
    {
        make = Carmake;
        model = Carmodel;
        year = Caryear;
    }
```

```
void startEngine()
{
    cout << "The " << make << " " << model << " engine
                        has started." << endl;
}
};
```

→ Now, you can call the member function on 'Car' object:

```
Car myCar("Honda", "Civic", 2021);
myCar.startEngine();

// Output: The Honda Civic engine has started.
```

5.5 Inheritance and Polymorphism

Inheritance:

→ Inheritance is a key feature of OOP that allows you to create new classes based on existing classes.

→ The new classes, called derived classes or subclasses, inherit the properties and behaviours of the existing class, known as the base class or super class.

→ • Sub Class: The class that inherits properties from another class is called sub class or Derived class.

- Super Class : The class whose properties are inherited by a sub-class is called Base class or Super class.

- Reusability : Inheritance supports the concept of "reusability", i.e; when we want to create a new class and there is already a class that includes some of the code that we want.

  → we can derive our new class from the existing class.

  → By doing this, we are reusing the fields and methods of the existing class.

polymorphism:

→ It is the ability of objects of different classes related through inheritance to respond to the same message or function call differently.

→ It allows you to write code ~~but~~ that can work with objects of different types but perform appropriate actions based on the actual object's type.

→ An operation may exhibit different behaviours in different instances.

→ The behaviour depends upon the types of data used in the operation.

→ C++ supports operator overloading and function overloading.

- Operator Overloading :
  → The process of making an operator exhibit different behaviours in different instances is known as operator overloading.

- Function Overloading :
  → Function overloading is using a single function name to perform different type of tasks.

  → Polymorphism is extensively used in implementing inheritance.

Ex: Consider a 'SportsCar' class that inherits from the 'Car' class:

```
Class SportsCar : public Car
{
    public :
        void accelrate()
        {
            cout << "The" << make << " " << model
                 << "is accelrating!" << endl;
        }
};
```

→ Now you can create objects of the 'SportsCar' class and access both the inherited and new member functions:

```
SportsCar mySportsCar("Ferrari", "488 GTB", 2022);
mySportsCar.startEngine();

//output: The Ferrari 488 GTB engine has started.
mySportsCar.accelrate();

//output: The Ferrari 488 GTB is accelrating!
```

5.6 Encapsulation and Data Hiding.

Encapsulation:

→ Encapsulation is the practice of bundling data and the methods that operate on that data within a single Unit (class).

→ It helps in organizing and managing code by hiding the internal details of an object and providing public interfaces to interact with it.

→ Two important properties of Encapsulation,

1. Data protection:

Encapsulation protects the internal

state of an object by keeping its data members private.

→ Access to and modification of these data members is restricted to the class's public methods, ensuring controlled and secure data manipulation.

2. Information Hiding:

→ Encapsulation hides the internal implementation details of a class from external code.

→ Only the public interface of the class is accessible, providing abstraction and simplifying the usage of the class while allowing the internal implementation to be modified without impacting external code.

Data Hiding:

→ It is the concept of protecting the internal state of an object by making certain data members private.

→ private data members can only be accessed or modified by member functions of the class.

→ This ensures data integrity and helps prevent unwanted changes or direct access to sensitive information.

Ex:

modifying the 'Car' class to use private data members:

```
Class Car
  {
    private:
        String make;
        String model;
        int year;

    public:
     Car(string carmake, String carmodel,
                          int Caryear)
     {
          make = carmake;
          model = Carmodel;
          year = Caryear;
     }
   void StartEngine()
    {
      Cout << "The" << make << " " << model
          << "engine has started" << endl;
    }
  };
```

5.7 Object - Oriented Design principles

→ Object - Oriented Design (OOD) principles provide guidelines for designing and organizing code in an object - oriented manner.

→ They promote code maintainability, reusability and flexibility.

→ Some common OOD principles include:

- Single Responsibility principle (SRP):
  → A class should have only one reason to change.

- Open/Closed principle (OCP):
  → Classes should be open for extension but closed for modification.

- Liskov Substitution Principle (LSP):
  → Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

- Interface Segregation principle (ISP):
  → Clients should not be forced to depend on interfaces they do not use.

- Dependency Inversion principle (DIP):
  → High-level modules should not depend on low-level modules.

  → Both should depend on abstractions.

6. Pointer and Dynamic memory Allocation.

6.1 Introduction to pointers.

→ Pointers are variables that store memory addresses.

→ They allow you to directly manipulate memory and access data indirectly.

→ Pointers are a powerful feature of C++ and are commonly used in various programming tasks.

→ To declare a pointer, you use the 'type *name; syntax.

→ The '*' indicates that the variable is a pointer.

Ex: int *ptr;

6.2 Pointer Variables and Dereferencing

→ Pointer variables hold memory addresses as their values.

→ To access the value at the memory address stored in a pointer variable, you use the dereference operator '*':

Ex:

given an integer pointer 'ptr':

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

ptr = &value. // assigns the memory address
                of 'value' to 'ptr':
```

→ To access the value using the pointer, you use dereference operator:

```
Cout << *ptr;   //output : 10.
```

6.3 Pointer Arithmetic.

→ Pointer arithmetic allows you to perform arithmetic operations on pointers.

→ When you add an integer value to a pointer, it moves to the next memory location based on the size of the type it points to.

→ Ex:

given an integer pointer 'ptr':

```
int *ptr;
int values[] = {10, 20, 30, 40, 50};

ptr = values; // Assign the memory address
                of the first element 'values'
                to 'ptr':
```

```
cout << *ptr << endl;
cout << *(ptr+1);
```

Output:

10
20.

→ Here, 'ptr' initially points to the first element of the 'values' array.

→ By adding '1' to the pointer ('ptr+1'), we move the pointer to the next element.

6.4 Pointers and Arrays.

→ Pointers and arrays are closely related in C++.

→ An array name can be treated as a pointer to its first element.

→ You can use pointers to access array elements and perform operations on them.

→ Ex:

```
int numbers[] = {1, 2, 3, 4, 5};
int *ptr = numbers;

cout << *ptr;
cout << *(ptr+2);
```

Output:

1

3.

→ Here, 'ptr' points to the first element of the 'numbers' array.

→ By using pointer arithmetic, we can access different elements of the array.

6.5 Dynamic memory Allocation and Deallocation

→ Dynamic memory allocation allows you to allocate memory during runtime using the 'new' operator.

→ It is useful when you don't know the size of the data in advance or when you need to allocate memory for objects dynamically.

→ Ex:

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

int main()
{
    int* numPtr = new int;
    *numPtr = 5;

    cout << *numPtr; //output :5
```

```
delete numPtr; //Deallocate the memory.
```

→ Here, we dynamically allocate memory for an integer using the keyword 'new'.

→ We assign a value to the memory location and later deallocate the memory using 'delete'.

6.6 Memory Management and Resource Handling.

→ Proper memory management is important to avoid memory leaks and ensure the efficient resource handling.

→ In addition to 'new' and 'delete', C++ provides other constructs like Constructors, destructors, and smart pointers to automate memory management.

→ Constructors and Destructors are special member functions that are called automatically when objects are created and destroyed, respectively.

→ They are used to initialize and release resources associated with an object.

→ Smart pointers are objects that manage the lifetime of dynamically allocated memory.

→ They automatically deallocate memory when it is no longer needed, preventing the memory leaks.

→ Ex:

'unique_ptr',
'shared_ptr' and
'weak_ptr'.

→ Understanding pointers and dynamic memory allocation is crucial for advanced memory management and data manipulation in C++.

→ In the next chapter, we will explore exception handling, which helps in dealing with errors and unexpected situations in a program.

7. Exception Handling.

7.1 Introduction to Exception Handling.

→ Exception handling is a mechanism in C++ that allows you to handle errors and exceptional situations in a structured and controlled manner.

→ It provides a way to seperate error handling code from normal code execution, making programs more robust and maintainable.

→ In C++, exceptions are used to represent and handle errors or exceptional conditions that may occur during program execution.

→ These exceptions can be generated explicitly by the programmer or thrown automatically by the system when an error occurs.

→ Exception Handling involves 3 key components:

1. Throwing :
   → when an error or exceptional condition occurs, you can throw an exception to indicate that something unexpected has happened.

2. Catching :

→ You can catch and handle the thrown exception handlers.

→ This allows you to take appropriate actions to recover from the error or terminate the program gracefully.

3. Handling :

→ Inside the exception handlers, you can write code to handle the exception, such as displaying an error message, logging the error, or taking corrective actions.

7.2 Exception Handling Mechanism.

→ In C++, the exception handling mechanism revolves around 3 key words: 'try', 'catch', and 'throw'.

→ The 'try' block is used to enclose the code that may potentially throw an exception.

→ If an exception is thrown within the 'try' block, the program control is transferred to the corresponding 'catch' block.

→ The 'catch' block is used to catch and handle exceptions.

→ It specifies the type of exception it can handle and contains code to handle the exception appropriately.

→ You can have multiple 'catch' blocks to handle different types of exceptions.

→ The 'throw' keyword is used to explicitly throw an exception.

→ It is followed by an expression, which can be of any type.

→ This expression is usually an object that represents the exception being thrown.

→ Syntax for basic exception handling mechanism:

```
try
{
    // code that may throw exception.
}
catch (ExceptionType1 ex1)
{
    // Exception handling code for
    // exceptionType1.
}
catch (ExceptionType2 ex2)
{
    // Exception handling code for
    // exception Type2
}
catch (.....) {
}
```

→ The '...' in the last 'catch' block is used to catch any other types of exceptions that are not explicitly handled by previous 'catch' blocks.

7.3 Throwing and Catching Exceptions

→ To throw an exception, you can use the 'throw' statement followed by an expression that represents the exception.

→ This expression can be of any type.

→ Ex:

```
void divide(int a, int b)
{
    if (b==0)
    {
        throw "Division by zero is not
              allowed!";
    }
    int result= a/b;

    cout<< "Result: " << result <<endl;
}
```

→ In this example, the 'divide' function checks if the divisor ('b') is zero.

→ If it is, it throws a string literal as an exception.

→ To catch and handle exceptions, you use the 'catch' block.

→ The 'catch' block specifies the type of exception it can handle and contains code to handle the exception.

→ Ex:

```
try
 {
    divide (10,0);
 }
 catch (const char * ex)
 {
    cout<< "Exception caught: "<<ex<<endl;
 }
```

→ In this example, the 'catch' block catches the exception thrown by the 'divide' function.

→ It specifies the type 'const char*' to match the type of the thrown exception.

→ They caught exception is then printed as an error message.

7.4 Exception Classes and Hierarchies.

→ In C++, you can create your own exception classes to represent different types of exceptions.

→ This allows you to have more specific and meaningful exception types for different situations.

→ To create an exception class, you typically derive it from the base class.
`Std:: exception ' or its derived classes like
`Std:: runtime_error' or
`Std:: logic_error'.

→ You can add additional member variables and functions to the exception class to provide more information about the exception.

Ex:

```
#include <stdexcept>

Class MyException : public std:: exception
{
    public:
        MyException (const std:: String &
                message) : msg (message) {}

        Const char* what() Const noexcept override
        {
            return msg.c_str();
        }
    private:
        Std:: String msg;
};
```

→ In this example, the `MyException' class is derived from 'std:: exception'.

→ It has a constructor that takes a message as a parameter and stores it in the member variable 'msg'.

→ The `what()' function is overridden to return the exception message.

→ You can then throw and catch instances of your custom exception class, similar to how we handled string exceptions earlier.

7.5 Error Handling Strategies.

→ When it comes to error handling, different strategies can be employed based on the requirements of your program and the nature of the exceptions.

1. Catch and Handle:
   → Catch specific exceptions and handle them appropriately within the 'catch' block.
   → This allow you to handle different exceptions differently based on their types.

2. Rethrow :

→ Catch an exception, perform some operations, and then rethrow the same exception or a different one.

→ This can be useful when you want to add additional information or perform cleanup actions before propagating the exception further.

3. Terminate:

→ If an exception occurs that cannot be handled within the current context, you may choose to terminate the program.

→ This can be done by not catching the exception or rethrowing it without a corresponding 'catch' block.

→ It is important to choose appropriate error handling strategy based on the specific requirements and constraints of your program.

8. File Input / Output.

8.1 Introduction to File Input / Output.

→ File Input / Output (I/O) operations are essential for handling external data in your C++ programs.

→ With file I/O, you can read data from files into your program or write data from your program to files.

→ This chapter will introduce you to the basics of file I/O in C++, including file handling modes, opening and closing files, reading and writing data, file pointers, and error handling.

→ Files can be of various types, such as text files, binary files, or special file formats.

→ In C++, you can work with both text and binary files using different file I/O operations.

8.2 File Handling Modes.

→ Before performing any file I/O operations, you need to specify the file handling mode.

→ The file handling mode determines the purpose and behaviour of file Operations.

→ Here are 3 common file handling modes:

→ Input Mode (std::ifstream) :

→ This mode is used to read data from a file.

→ The file must already exist, and you can only read data from it.

→ Output Mode (std::ofstream) :

→ This mode is used to write data to the file.

→ If the file does not exist, it will be created.

→ If it already exists, the previous content will be overwritten by default.

→ Append Mode (std::ofstream with std::ios::app flag) :

→ This mode is used to write data to a file while preserving its existing content.

→ If the file does not exist, it will be created.

→ If it already exists, the new data will be appended to the end of the file.

8.3 Opening and Closing Files.

→ To perform file I/O operations, you need to open the file first.

→ C++ provides the 'std::ifstream' and 'std::ofstream' classes to work with input and output files, respectively.

→ Here's how youm can open a file for reading or writing:-

```
#include <fstream>

// Opening a file for reading.
std::ifstream inputFile;
inputFile.open("input.txt");

// Opening a file for writing.
std::ofstream outputFile;
outputFile.open("output.txt");

// Opening a file in append mode
std::ofstream appendFile;
appendFile.open("data.txt", std::ios::app);
```

→ After you have finished working with a file, it's important to close it to release system resources.

→ To close a file, simply call the 'close()' function on the file stream object:

```
inputFile.close();
outputFile.close();
appendfile.close();
```

8.4 Reading and writing to Files

→ Once you have opened a file, you can perform read and write operations on it.

→ Let's start with reading data from a file using the input stream ('std::ifstream') class.

→ To read data from a file, you can use various input functions, such as '>>' (extraction operator) and 'getline()'.

→ Here's an example:

```
#include <fstream>
#include <iostream>
#include <string>

Using namespace Std;
```

```
int main()
{
    ifstream inputFile;
    inputFile.open("input.txt");

    String name;
    int age;

    //Reading data from file using
    //extraction operator.

    inputFile >>name >>age;
    Cout << "Name:" << name << ", Age:"
        << age << endl;

    // Reading a line of text from file
    // using getLine()

    String city;
    getLine(inputFile, city);
    Cout<< "City:" << city << endl;

    inputFile.close();
    return 0;
}
```

→ In this example, the program reads the name and age from the file using the extraction operator (`>>`).

→ It then reads a line of text using the `getLine()` function.

→ Similarly, you can write data to a file using the output stream ('std:: ofstream') class.

→ You can use the insertion operator ('<<') to write data to a file.

→ Here's an example:

```
#include <fstream>
#include <iostream>
#include <ofstream>

using namespace std;

int main()
{
    ofstream outputFile;
    outputFile.open("output.txt");

    string name = "John Doe";
    int age = 85;

    //writing data to file using insertion
    //operator.

    outputFile << "Name:" << name << ", Age:"
               << age << endl;
    string city = "New York";

    // Appending data to file.
    outputFile << "City:" << City << endl;
```

```
    outputFile.close();

    return 0;
}
```

→ In this example, the program writes the name and age to the file using the insertion operator ('<<').

→ It then appends the city to the file.

8.5 File pointers and positioning.

→ File pointers are used to keep track of the current position within a file.

→ They are essential when performing operations like reading or writing data at specific positions in the file.

→ C++ provides file pointer manipulation functions to control and query the position of the file pointer.

Here are some commonly used file pointer manipulation functions:

- 'seekg()' and 'seekp()':
  → These functions are used to set the position of the file pointer for input & output operations.

- `tellg()` and `tellp()`:

→ These functions return the current position of the file pointer for input and output operations.

Ex:

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    fstream file;
    file.open("data.txt", ios::in | ios::out);

    if (file)
    {
        // writing data at beginning of file.
        file.seekp(0);
        file << "start of the file \n";

        // Reading data from middle of file.
        file.seekg(5);
        string content;
        getline (file, content);
        cout << "content:" << content << endl;

        // Appending data at end of file.
        file.seekp(0, ios::end);
        file << "end of the file \n";
```

```
        file.close();
    }
    else
    {
        cout<< "Failed to open file!" <<endl;
    }

    return 0;
}
```

→ In this example, the program opens a file in both input & output mode.

→ It then uses 'seekp()' function to set the file pointer to the beginning of the file and writes data.

→ Next, it uses 'seekg()' function to set the file pointer to a specific position and reads data from that position.

→ Finally, it uses the 'seekp()' function with the 'std::ios::end' flag to append data at the end of the file.

8.6 Error Handling in File Operations

→ Errors can occur due to various reasons, such as file not being found, insufficient permissions, or disk full.

→ One way to handle errors in file operations is to check the status of the file stream object after performing an operation.

→ You can use the 'good()' function to check if the stream is in a good state or the 'fail()' function to check if the previous operation failed.

→ Additionally, you can use the 'bad()' function to check if a non-recoverable error occurred.

Ex:-

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    ifstream inputFile;
    inputFile.open("nonexistent.txt");

    if(!inputFile)
    {
        cout<< "Error opening the file!"<<endl;
        return 1;
    }

    // Perform read operations.
```

```
    inputFile.close();

    return 0;
}
```

→ In this example, the program attempts to open a file that doesn't exist.

→ If the file fails to open, the program displays an error message and returns from the main function with a non-zero value.