**INTRODUCTION:**

**Goof Morning sir,**

First of all thank you for giving me this opportunity to introduce myself.

My name is K.Devaraju, and I am a software Developer specializing in c++ and qt, with 10 months of experience in the automotive domain, specifically in infotainment HMI development for General Motors.

I completed my B.Tech in Computer Science and Engineering from MVGR college.During my academic journey I had developed a strong foundation in c++, DSA, STL and multithreading.

Currently I am working as a software engineer at People Tech Enterprises Limited. My responsibilities include implementing features, debugging and maintaining software. I have hands on experience with c++ and qt. And I am proficient in version controls like Git, Bitbucket and source tree.

**ABOUT PROJECT:**

**PROJECT-1**

**Project:** INFO4 HMI Development for Instrument Panel Cluster(IPC).

**Company:** People Tech Enterprises(client General Motors).

**Technologies:** C++14,Qt 5.15, ODI Simulator Tool, Wecan.

Developed HMI for an in-vehicle cluster system from scratch based on General Motors requirements.

Designed and implemented various HMI modules, including Driver Assistance, Alerts, and Telletales and sizes(8 to 54 inches).

Developed VCS HMI Designer, an automated tool that generates data bindings between UI widgets and the data pool, reducing manual errors in HMI development.

Implemented multiple widgets, including Label Widget, image widget, progressbar widget , Group widget and List widget.

Provided support for VCS HMI Designer and GMXML renderer, IPC, FCC, RCC applications.

**PROJECT-2**

**Project:** Food Wastage Management Website.

**Technologies**:HTML, CSS, Javascript and SQL.

Designed and developed a centralized platform to streamline food donation coordination within specific locations.

Ensured efficient surplus food redistribution from donors to recipients, reducing food wastage.

Implemented database management using MYSQL to track donations and beneficiaries.

**Similarities between Java and C++:**

**In java compilation:** source code(.java file)->compile->byte code(.class file)->interpreter->machine code.

**In c++ compilation:** source code->compile->machine code. That’s why it is faster execution.

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**1.What is c++?**

**C++ is considered a** **middle-level programming language** because it has features of both **high-level** and **low-level** programming languages.

**Low-Level Features (Close to Hardware):**

\*Can interact directly with **CPU registers and hardware** (e.g., embedded systems, OS development).

**High-Level Features (User-Friendly & Abstraction):**

**\***Supports Object-Oriented Programming (OOP) (Classes, Inheritance, Polymorphism).

**\***Has Standard Template Library (STL) for data structures and algorithms.

**Features:**

1.**Simple**

2. **Machine Independent but Platform Dependent**: Run on different machines but compilation have to be done on same machine.

3. **Rich library support**: Has rich library support (STL - built-in data structures, algorithms etc.) as well as 3rd party well libraries (e.g. QT) for widgets.

4.**Speed of execution:** C++ programs excel in execution speed. Since, it is a compiled language, and hugely procedural.

5. **Object-Oriented**: support helps C++ to make maintainable and extensible programs. i.e. Large-scale applications can be built

**Applications:**

* Operating Systems & Systems Programming. e.g. *Linux-based OS (Ubuntu etc.)*
* Browsers *(Chrome & Firefox)*
* Graphics & Game engines *(Photoshop, Blender, Unreal-Engine)*
* Database Engines *(MySQL, MongoDB, Redis etc.)*
* Cloud/Distributed Systems

**🡪#include<iostream>**

#include=preprocessor directive used to include header files.

iostream=header file used to store definitions of standard input/output streams.

**Namespace:** Advantage of Namespace to avoid name collision.

The best example of namespace scope is the C++ standard library (std) where all the classes, methods and templates are declared. Hence while writing a C++ program we usually include the directive using namespace std;

**Example:**

#include <iostream>

using namespace std;

// first name space

namespace first\_space

{

void func()

{

cout << "Inside first\_space" << endl;

}

}

// second name space

namespace second\_space

{

void func()

{

cout << "Inside second\_space" << endl;

}

}

using namespace first\_space;

int main ()

{

// This calls function from first name space.

func();

return 0;

}

You can access members of nested namespace by using resolution operators as follows:  
// to access members of namespace\_name2  
using namespace namespace\_name1::namespace\_name2;

**Identifiers:** We use [identifiers](https://www.geeksforgeeks.org/c-identifiers/)for the naming of variables, functions, and other user- defined data types. An identifier may consist of uppercase and lowercase alphabetical characters, underscore, and digits. The first letter must be an underscore or an alphabet.

**Keywords:** In the C++ programming language, there are some reserved words that are used for some special meaning in the C++ program. It can’t be used for identifiers. For example, the words **int, return, and using** are some [keywords](https://www.geeksforgeeks.org/keywords-in-c/) used in our program.

**Comments:** Comments in C++ are meant to explain the code as well as to make it more readable. Their purpose is to provide information about code lines.

//single line comments

/\*this is for the multiple line comments   
in the program\*/

**Tokens in c:** In C programming, tokens are the smallest units in a program that have meaningful representations. Tokens are the building blocks of a C program, and they are recognized by the C compiler to form valid expressions and statements.

1.Punctuators: (), {}, [], #, \*;

2.Keywords:int, float, return, while …

3.Strings:” hello deva”

4.Identifiers: nothing but variables

5.Operators:<=,>=, ...

6.Constants: const.

**Data types**: specify the type of data that a variable can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data type with which it is declared as every data type requires a different amount of memory.

**primitive data types:** int, float, double, char, bool, void

**derived data types:** array, pointer, reference, function

**user defined data types:** class, struct, union, using.

**Type conversions:**

**Int to (long, float, bool):** (long)int or static\_cast<long>(int).

**String to (long, float, int):** stoi, stol, stof.

**Char to int:** ch- ‘0’

**static\_cast:** The[**static\_cast**](https://www.geeksforgeeks.org/static_cast-in-cpp/)operator is the most commonly used casting operator in C++. It performs compile-time type conversion and is mainly used for explicit conversions that are considered safe by the compiler.

Ex: static\_cast<new>(exp).

**dynamic\_cast:** The [dynamic\_cast](https://www.geeksforgeeks.org/dynamic-_cast-in-cpp/)operator is mainly used to perform down casting (converting a pointer/reference of a base class to a derived class) in polymorphisms and inheritance. It ensures type safety by performing a runtime check to verify the validity of the conversion.

Ex: Animal\* animalptr = new Dog ();

Dog\* dogptr = dynamic\_cast<Dog\*>(Animal);//converting base class to derived class.

**Jump Statements:**

Jump statements are used to manipulate the flow of the program if some conditions are met. It is used to terminate or continue the loop inside a program or to stop the execution of a function.

1.continue: it continues to execute the next iteration of the same loop

2.break: is used to terminate the whole loop if the condition is met.

3.return: It is used to terminate the entire function after the execution of the function or after some condition

4.goto: is used to jump directly to that part of the program to which it is being called.

Ex:

#include <iostream>

using namespace std;

int main() {

int n = 4;

if (n % 2 == 0) {

// Skipping to label1

goto label1;

}

else {

// Skipping to label2

goto label2;

}

label1:

cout << "Even" << endl;

return 0;

label2:

cout << "Odd" << endl;

return 0;

label3:

cout << "Unspecified";

return 0;

}

**Function:** a function is a block of code that runs only when it is called**.**

**\***Functions help us in *reducing code redundancy.*

***Pass by Value:***In this parameter passing method, values of actual parameters are copied to the function’s formal parameters. The actual and formal parameters are stored in different memory locations, so any changes made in the functions are not reflected in the actual parameters of the caller.

**Ex:** void fun(int a){}

**Pass by Reference*:*** Both actual and formal parameters refer to the same locations, so any changes made inside the function are reflected in the actual parameters of the caller.

**Ex:** void fun(int &a){}   
**Functions using Pointers:**

// C++ Program to demonstrate working of function using pointers

#include <iostream>

using namespace std;

void fun(int\* ptr)//storing address of x passed by the fun(&x);

{

\*ptr = 30;

}

int main()

{

int x = 20;

fun(&x);

cout << "x = " << x;

return 0;

}

**Default arguments**:

#include <iostream>

using namespace std;

// Function with an argument with default value

void f(int a = 10) {

cout << a << endl;

}

int main() {

// Uses default argument

f();

// Uses passed value

f(221);

return 0;

}

**Inline function:**

a function can be specified as **inline** to reduce the function call overhead. The whole code of the inline function is inserted or substituted at the point of its call during the compilation instead of using normal function call mechanism.

A screenshot of a computer

AI-generated content may be incorrect.

Ex:

#include <iostream>

using namespace std;

// Inline function

inline int square(int x) {

return x \* x;

}

int main() {

int num = 5;

// Calling inline function

int res = square(num);

cout << res;

return 0;

}

**Normal function:**

A **normal function** is compiled separately, and its address is stored in memory. At runtime, the function call jumps to this address.

**Lambda function:**

introduced **lambda expressions** to allow inline functions which can be used for short snippets of code that are not going to be reused. Therefore, they do not require a name. They are mostly used in STL algorithms as callback functions.

**Syntax:** [capture-clause] (parameters) -> return-type {

// definition

}

* **&]:** capture all external variables by reference.
* **[=]:** capture all external variables by value.
* **[a, &b]:** capture ‘a’ by value and ‘b’ by reference.

Ex:

#include <bits/stdc++.h>

using namespace std;

int main()

{

// Defining a lambda

auto res = [](int x) {

return x + x;

};

cout << res(5);

return 0;

}

**Pointers and References:**

In C++ pointers and references both are mechanisms used to deal with memory, memory address, and data in a program. Pointers are used to store the memory address of another variable whereas references are used to create an alias for an already existing variable.

// C++ program to demonstrate use of pointers in C++;

#include <iostream>

using namespace std;

int main()

{

int x = 10; // variable declared

int\* myptr; // pointer variable

// storing address of x in pointer myptr

myptr = &x;

cout << "Value of x is: ";

cout << x << endl;

// print the address stored in myptr pointer variable

cout << "Address stored in myptr is: ";

cout << myptr << endl;

// printing value of x using pointer myptr

cout << "Value of x using \*myptr is: ";

cout << \*myptr << endl;

return 0;

}

* **Applications of Pointers in c++:**
* Dynamic memory allocation: We can use pointers to dynamically allocate memory. The advantage of dynamically allocated memory is, that it is not deleted until we explicitly delete it.
* To implement data structures.
* To do system-level programming where memory addresses are useful.

**this pointer:**

\*It helps in accessing members of the current object when there is ambiguity due to name conflicts.

**\***The this pointer holds the memory address of the calling object**.**

**\*** It is available only in non-static member functions.

Ex:

#include <iostream>

using namespace std;

class Example {

public:

int x;

void setX(int x) {

this->x = x; // Resolves ambiguity between parameter 'x' and class member 'x'

}

void show() {

cout << "x = " << this->x << endl;

}

};

int main() {

Example obj;

obj.setX(10);

obj.show(); // Output: x = 10

return 0;

}

**Array:** an **array**is a derived data type that is used to store multiple values of similar data types in a contiguous memory location. Each element can be accessed using its index (position starting from 0).

**Structure:** [**structure**](https://www.geeksforgeeks.org/structures-in-cpp/)is a user-defined data type that is used to combine data of different types. It is similar to an array but unlike an array, which stores elements of the same type, a structure can store elements of different data types. C++ structures can also have member functions to manipulate its data.

**Ex:**

#include <bits/stdc++.h>

using namespace std;

// Define structure

struct GFG {

int G1;

char G2;

float G3;

};

int main() {

// Create object of structure

GFG Geek = {85, 'G', 989.45};

// Accessing structure members values

cout << Geek.G1 << endl;

cout << Geek.G2 << endl;

cout << Geek.G3;

return 0;

}

**Union:** [union](https://www.geeksforgeeks.org/cpp-unions/) is a user-defined datatype in which we can define members of different types of data types just like structures but unlike a structure, where each member has its own memory, a union member shares the same memory location.

Ex:

#include <iostream>

using namespace std;

// Defining a Union

union GFG {

int G1;

char G2;

float G3;

};

int main() {

// Create an object of GFG union

GFG Geek;

// Assign union's member variables

Geek.G1 = 85;

// Accessing union members values

cout << Geek.G1 << endl;

Geek.G2 = 'G';

cout << Geek.G2 << endl;

Geek.G3 = 989.45;

cout << Geek.G3;

return 0;

}

**Enum:**  [**enumeration**](https://www.geeksforgeeks.org/enumeration-in-cpp/)(enum) is a user-defined type that consists of a set of named integral constants.

Ex:

#include <bits/stdc++.h>

using namespace std;

int main() {

// Defining enum Gender

enum GFG { Male, Female, };

// Creating GFG type variable and assigning

// value

GFG Geek = Male;

switch (Geek) {

case Male:

cout << "Who is he?";

break;

case Female:

cout<< "Who is she?";

break;

default:

cout << "Who is they?";

}

return 0;

}

**New and delete operators:**

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by a programmer. Dynamically allocated memory is allocated on Heap, and non-static and local variables get memory allocated on Stack .

**Ex:**

int\* arr = new int[size];// to create an array dynamically.

int\* ptr=new int(19);//to create an integer dynamically.

**delete:**

delete ptr;//deleting the integer and avoiding memory leak;

delete[] arr;//deleting the array and avoiding memory leak;

C uses the [malloc() and calloc()](https://www.geeksforgeeks.org/calloc-versus-malloc/) function to allocate memory dynamically at run time and uses a free() function to free dynamically allocated memory. C++ supports these functions and also has two operators **new** and **delete,** that perform the task of allocating and freeing the memory in a better and easier way.

**Pure Virtual function:**

A pure virtual function is a function without a definition in the base class, forcing derived classes to implement it.

Ex: virtual void functionName() = 0; // "= 0" makes it pure virtual

**Abstract class:**

\*A class that contains at least one pure virtual function is called an abstract class.

\* Cannot be instantiated (i.e., you cannot create objects of an abstract class).

\*Can be used as a base class to enforce method implementation in derived classes.

Ex:

#include <iostream>

using namespace std;

// Abstract class (has at least one pure virtual function)

class Animal {

public:

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

void eat() { // Normal function

cout << "This animal eats food." << endl;

}

};

// Derived class implementing the pure virtual function

class Dog : public Animal {

public:

void makeSound() override {

cout << "Dog barks: Woof woof!" << endl;

}

};

int main() {

// Animal obj; // ❌ ERROR: Cannot instantiate an abstract class

Dog dog;

dog.makeSound(); // Calls overridden function

dog.eat(); // Calls normal function from base class

return 0;

}

**Virtual Destructor:**

Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.

**Ex:**

// A program with virtual destructor

#include <iostream>

using namespace std;

class base {

public:

base()

{ cout << "Constructing base\n"; }

virtual ~base()

{ cout << "Destructing base\n"; }

};

class derived : public base {

public:

derived()

{cout << "Constructing derived\n";}

~derived()

{cout << "Destructing derived\n";}

};

int main()

{

derived \*d = new derived();

base \*b = d;

delete b;

getchar();

return 0;

}

**Output:**

Constructing base

Constructing derived

Destructing derived

Destructing base

**For pure virtual destructor:**

#include <iostream>

// Initialization of base class

class Base {

public:

virtual ~Base() = 0; // Pure virtual destructor

};

Base::~Base() // Explicit destructor call

{

std::cout << "Pure virtual destructor is called";

}

// Initialization of derived class

class Derived : public Base {

public:

~Derived() { std::cout << "~Derived() is executed\n";}

};

int main()

{

// Calling of derived member function

Base\* b = new Derived();

delete b;

return 0;

}

**Output:**

~Derived() is executed

Pure virtual destructor is called

**Static :** Static functions or variables aren’t tied to the instance of a class but they are tied to the class.

**File Handling :**

For achieving file handling we need to follow the following steps:-  
 STEP 1-Naming a file  
 STEP 2-Opening a file  
 STEP 3-Writing data into the file  
 STEP 4-Reading data from the file  
 STEP 5-Closing a file.

\* ifstream – for reading files (input)

\* ofstream – for writing files (output)

\* fstream – for both reading and writing

**Writing to a file(ofstream):**

#include <iostream>

#include <fstream> // Include fstream for file handling

using namespace std;

int main() {

ofstream file("example.txt"); // Open file in write mode

if (file.is\_open()) {

file << "Hello, this is C++ file handling!\n"; // Write to file

file.close(); // Close the file

cout << "File written successfully.\n";

} else {

cout << "Error opening file!\n";

}

return 0;

}

**Reading from a file(ifstream):**

#include <iostream>

#include <fstream>

using namespace std;

int main() {

ifstream file("example.txt"); // Open file in read mode

string line;

if (file.is\_open()) {

while (getline(file, line)) { // Read file line by line

cout << line << endl;

}

file.close(); // Close file

} else {

cout << "Error opening file!\n";

}

return 0;

}

**Template:**

Templates in C++ allow writing generic code that works with any data type. They provide two main types:

1. Function Template → Generic functions
2. Class Template → Generic classes

**Function Template:**

A function template is a blueprint for creating functions that can operate on different data types without code duplication.

**Ex:**

#include <iostream>

using namespace std;

template <typename T> // Template definition

T add(T a, T b) {

return a + b;

}

int main() {

cout << add(5, 3) << endl; // Works with int

cout << add(5.5, 2.3) << endl; // Works with double

cout << add('A', 1) << endl; // Works with char (ASCII addition)

return 0;

}

**Class Template:** A class template allows defining a generic class that works with multiple data types.

**Ex:**

#include<iostream>

using namespace std;

template <typename T>

class Box

{

private:

T a;

public:

Box(T val)

{

a=val;

}

void show()

{

cout<<"the value of a is : "<<a<<endl;

}

};

int main()

{

Box<int> box1(10);

Box<float> box2(2.4);

box1.show();

box2.show();

}

**Friend Function:**

A friend function in C++ is a function that is not a member of a class but can access private and protected members of the class.

🔹 Why use a Friend Function?

* To access private members of a class from outside the class.
* To allow two or more classes to share private data.
* Used in operator overloading when the left operand is not an object of the class.

**Ex:**

#include<iostream>

using namespace std;

class Solution

{

private:

int num;

public:

Solution(int a)

{

num=a;

}

friend void showfunction(Solution s);

};

void showfunction(Solution s)

{

cout<<"this is a friend function"<<s.num<<endl;

}

int main()

{

Solution obj(12);

showfunction(obj);

}

**Friend Class:**

A friend class allows one class to access the private and protected members of another class.

**Ex:**

#include <iostream>

using namespace std;

class A {

private:

int data;

public:

A(int val) {data = val;}

// Declaring class B as a friend

friend class B;

};

class B {

public:

void show (A obj) {

cout << "Private Data from A: " << obj.data << endl;

}

};

int main() {

A objA(10);

B objB;

objB.show(objA); // Accessing private data of A

return 0;

}

**Multiple Inheritance in c++:**

Multiple inheritance is a feature in C++ where a class inherits from more than one base class.

**Ex:**

#include <iostream>

using namespace std;

class A {

public:

void show() { cout << "Class A\n";}

};

class B {

public:

void show() { cout << "Class B\n";}

};

// Multiple Inheritance

class C : public A, public B {};

int main() {

C obj;

// obj.show(); // ❌ ERROR: Ambiguous call

obj.A::show(); // ✅ Resolving ambiguity

obj.B::show();

return 0;

}

**Diamond Problem:**

#include <iostream>

using namespace std;

class A {

public:

int value;

};

// Virtual Inheritance to avoid duplication

class B : virtual public A {};

class C : virtual public A {};

class D : public B, public C {}; // ✅ Only ONE copy of A

int main() {

D obj;

obj.value = 10; // ✅ No ambiguity

cout << "Value: " << obj.value << endl;

return 0;

}

\*B and C share only one instance of A.

\*D inherits only one copy of A, avoiding ambiguity.

**Operator Overloading:**

process of using an operation instead of function to do a particular task.

+,-,\*,% these are the symbols which are used to do operation on two predefined data types.

**Ex:**

3+4, 3-1.

Here 3 and 4 are predfined datatypes like integer and float.

But obj1+obj2 are the user defined data types for the class to do operation we use operator overloading in c++.

**Ex:**

// C++ Program to Demonstrate

// Operator Overloading

#include <iostream>

using namespace std;

class Complex {

private:

int real, imag;

public:

Complex(int r = 0, int i = 0)

{

real = r;

imag = i;

}

// This is automatically called when '+' is used with

// between two Complex objects

Complex operator+(Complex const& obj)

{

Complex res;

res.real = real + obj.real;

res.imag = imag + obj.imag;

return res;

}

void print() { cout << real << " + i" << imag << '\n'; }

};

int main()

{

Complex c1(10, 5), c2(2, 4);

Complex c3 = c1 + c2;

c3.print();

}

**JAVA:**

**Ex:**

import java.util.\*; // Import everything from java.util

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in); // Taking input

System.out.print("Enter your name: ");

String name = sc.nextLine();

ArrayList<Integer> list = new ArrayList<>(); // Using ArrayList

list.add(10);

list.add(20);

System.out.println("Hello, " + name);

System.out.println("ArrayList: " + list);

}

}

Ex: **ARRAY**

import java.util.Scanner;

class ArrayInput {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the size of the array: ");

int size = sc.nextInt(); // Taking array size input

int[] arr = new int[size]; // Declaring array

System.out.println("Enter " + size + " elements: ");

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

arr[i] = sc.nextInt(); // Taking array elements input

}

System.out.println("Array elements are: ");

for (int num : arr) {

System.out.print(num + " ");

}

sc.close();

}

}

**Conversions in java:**

A screenshot of a computer

AI-generated content may be incorrect.

**FUNCTIONS IN JAVA:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Function Pointer:**

A screenshot of a computer code

AI-generated content may be incorrect.

**Ex:**

#include <iostream>

using namespace std;

void add(int a, int b) { cout << "Add: " << a + b << endl; }

void multiply(int a, int b) { cout << "Multiply: " << a \* b << endl; }

int main() {

void (\*operation)(int, int);

operation = add;

operation(3, 4);

operation = multiply;

operation(3, 4);

return 0;

}

**STRINGS FUNCTIONS**

String str= “hello world”;

str.append(string newstring);

str.push\_back(char c);

str.pop\_back();

str.insert(size\_t pos, string newstring);

str.erase(size\_t pos, size\_t len);

str.replace(size\_t pos, size\_t len, string newstring);

str.clear();

str.resize(size\_t len, char c);//until the size is reached the extra elements are added as c;

str.at[size\_t pos);//returns the character at position.

str[size\_t pos);

str.front();//returns the first character.

str.back();//returns the last character.

str.find( “hello”);//returns the index pos;

str.rfind(“hello”);//returns the index pos of last occurrence of a string;

str.substr(size\_t pos, size\_t len);

str.compare(str2);// if str.compare(str2)==0 then same or <0 str is less or >0 str2 is less;

isupper(char ch);//check the condition the ch is lower or not;

toupper(char ch);//converts into the uppercase;

ex:

for(char &ch : str)

{  
 if(islower(ch)

{

ch=toupper(ch);

}

}

str.length();//length of the string;

str.empty();//checking the string is empty or not;

str.capacity();//check the storage capacity;

str.shrink\_to\_fit();//reduce capacity to matchsize;

int num=stoi(“1234”);//converting into a integer;

string str3=to\_string(123);//converting into a string;