**C++ Interview Questions and Answers (2024)**

Last Updated : 18 Jul, 2024

**C++** –  **Evernote, LinkedIn, Microsoft, Opera, NASA**, and **Meta**because of its dependability, performance, and wide range of settings

**C++ Interview Questions For Freshers**

**1. What is C++? What are the advantages of C++?**

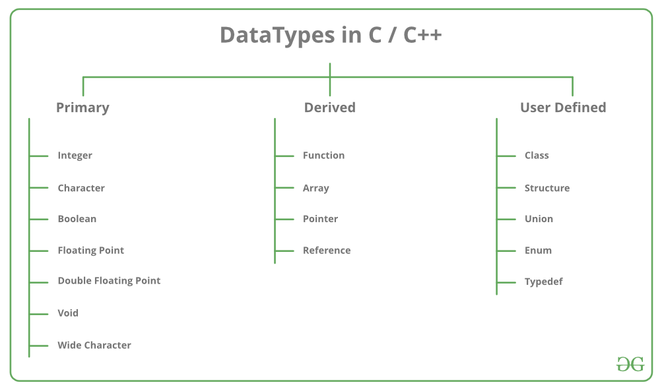
C++ is an object-oriented programming language that was introduced to overcome the jurisdictions where C was lacking. By object-oriented we mean that it works with the concept of [*polymorphism*](https://www.geeksforgeeks.org/polymorphism-in-c/)*,*[*inheritance*](https://www.geeksforgeeks.org/inheritance-in-c/)*,*[*abstraction*](https://www.geeksforgeeks.org/abstraction-in-c/)*,*[*encapsulation*](https://www.geeksforgeeks.org/encapsulation-in-c/)*,*[*object, and class*](https://www.geeksforgeeks.org/c-classes-and-objects/)*.*

***Advantages of C++****:*

1. C++ is an OOPs language that means the data is considered as objects.
2. C++ is a multi-paradigm language; In simple terms, it means that we can program the logic, structure, and procedure of the program.
3. Memory management is a key feature in C++ as it enables dynamic memory allocation
4. It is a Mid-Level programming language which means it can develop games, desktop applications, drivers, and kernels

To read more, refer to the article – [What are the advantages of C++?](https://www.geeksforgeeks.org/advantages-and-disadvantages-of-cpp/)

**2. What are the different data types present in C++?**



*Different types of data types in C++*

For more information, refer to [C++ data types](https://www.geeksforgeeks.org/cpp-data-types/)

**3. Define ‘std’?**

‘**std’**is also known as Standard or it can be interpreted as a namespace. The command “*using namespace std”*informs the compiler to add everything under the *std namespace* and inculcate them in the *global namespace*. This all inculcation of global namespace benefits us to use “**cout**” and “**cin”**without using “**std::\_operator\_”.**

For more information, refer to [namespace and std](https://www.geeksforgeeks.org/why-it-is-important-to-write-using-namespace-std-in-cpp-program/).

**4. What are references in C++?**

When a variable is described as a reference it becomes an alias of the already existing variable. In simple terms, a referenced variable is another named variable of an existing variable keeping in mind that changes made in the reference variable will be reflected in the already existing variable. A reference variable is preceded with a **‘&’** symbol.

**Syntax:**

int GFG = 10;  
  
// reference variable  
int& ref = GFG;

For more information, refer to [references in C++](https://www.geeksforgeeks.org/references-in-c/)

**5. What do you mean by Call by Value and Call by Reference?**

In this programming language to call a function we have 2 methods: **Call by Value**and **Call by Reference**

| **Call by Value** | **Call by Reference** |
| --- | --- |
| A copy of a variable is passed. | A variable itself is passed fundamentally. |
| Calling a function by sending the values by copying variables. | Calling a function by sending the address of the passed variable. |
| The changes made in the function are never reflected outside the function on the variable. In short, the original value is never altered in Call by Value. | The changes made in the functions can be seen outside the function on the passed function. In short, the original value is altered in Call by reference. |
| Passed actual and formal parameters are stored in different memory locations. Therefore, making Call by Value a little memory insufficient | Passed actual and formal parameters are stored in the same memory location. Therefore, making Call by Reference a little more memory efficient. |

For information, refer to [the difference between call by value and call by reference](https://www.geeksforgeeks.org/difference-between-call-by-value-and-call-by-reference/)

**6. Define token in C++**

A token is the smallest individual element of a program that is understood by a compiler. A token comprises the following:

**KISSCO**

1. **Keywords** – That contain a special meaning to the compiler
2. **Identifiers** – That hold a unique value/identity
3. **Constants**– That never change their value throughout the program
4. **Strings**– That contains the homogenous sequence of data
5. **Special Symbols**– They have some special meaning and cannot be used for another purpose; eg: **[] () {}, ; \* = #**
6. **Operators**– Who perform operations on the operand

For more information, refer to [Tokens in C++](https://www.geeksforgeeks.org/cc-tokens/)

**7. What is the difference between C and C++?**

| **C** | **C++** |
| --- | --- |
| It is a procedural programming language. In simple words, it does not support classes and objects | It is a mixture of both procedural and object-oriented programming languages. In simple words, it supports classes and objects. Multi-Paradigm Language, Standard Template Library (STL) |
| It does not support any OOPs concepts like polymorphism, data abstraction, encapsulation, classes, and objects. | It supports all concepts of data |
| It does not support Function and Operator Overloading | It supports Function and Operator Overloading respectively |
| It is a function-driven language | It is an object-driven language |

For more information, refer to [Difference between C and C++](https://www.geeksforgeeks.org/difference-between-c-and-c/)

1. **What is the difference between struct and class?**
2.  **struct:** Members (both data and functions) are public by default.

 **class:** Members are private by default.

**struct DerivedStruct : BaseStruct {**

**// Inherits BaseStruct's members publicly**

**};**

**class DerivedClass : BaseClass {**

**// Inherits BaseClass's members privately**

**};**

**struct: Traditionally used for simple data structures**

**class: Typically used for more complex structures where encapsulation**

**In modern C++, struct and class can be used interchangeably with the main difference being the default access control.**

|  |  |
| --- | --- |
| **Struct** | **Class** |
| Members of the struct are always by default public mode | Members of the class can be in private, protected, and public modes. |
| Structures are of the value type. They only hold value in memory. | Classes are of reference type. It holds a reference of an object in memory. |
| The memory in structures is stored as stacks | The memory in classes is stored as heaps. |

For more information, refer to the [Difference between struct and class.](https://www.geeksforgeeks.org/structure-vs-class-in-cpp/)

**9. What is the difference between reference and pointer?**

|  |  |
| --- | --- |
| **Reference** | **Pointer** |
| The value of a reference cannot be reassigned | The value of a pointer can be reassigned |
| It can never hold a *null*value as it needs an existing value to become an alias of | It can hold or point at a *null* value and be termed as a *nullptr* or *null pointer* |
| It cannot work with arrays | It can work with arrays |
| To access the members of class/struct it uses a ‘ **.**‘ | To access the members of class/struct it uses a ‘ **->**‘ |
| The memory location of reference can be accessed easily or it can be used directly | The memory location of a pointer cannot be accessed easily as we have to use a dereference ‘ **\***‘ |

For more information, refer to the [Difference between reference and pointer](https://www.geeksforgeeks.org/pointers-vs-references-cpp/)

**10. What is the difference between function overloading and operator overloading?**

| **Function Overloading** | **Operator Overloading** |
| --- | --- |
| It is basically defining a function in numerous ways such that there are many ways to call it or in simple terms you have multiple versions of the same function | It is basically giving practice of giving a special meaning to the existing meaning of an operator or in simple terms redefining the pre-redefined meaning |
| Parameterized Functions are a good example of Function Overloading as just by changing the argument or parameter of a function you make it useful for different purposes | Polymorphism is a good example of an operator overloading as an object of allocations class can be used and called by different classes for different purposes |
| Example of Function Overloading:   1. int GFG(int X, int Y); 2. int GFG(char X, char Y); | Example of Operator Overloading:   1. int GFG() = X() + Y(); 2. int GFG() = X() – Y(); |

For more information, refer to [Operator Overloading](https://www.geeksforgeeks.org/operator-overloading-c/)and [Function Overloading](https://www.geeksforgeeks.org/function-overloading-c/)

**11. What is the difference between an array and a list?**

| **Arrays** | **Lists** |
| --- | --- |
| Array are contiguous memory locations of homogenous data types stored in a fixed location or size. | Lists are classic individual elements that are linked or connected to each other with the help of pointers and do not have a fixed size. |
| Arrays are static in nature. | Lists are dynamic in nature |
| Uses less memory than linked lists. | Uses more memory as it has to store the value and the pointer memory location |

For more information, refer to [Arrays Vs List](https://www.geeksforgeeks.org/linked-list-vs-array/)

**12. What is the difference between a while loop and a do-while loop?**

| **While Loop** | **do-while Loop** |
| --- | --- |
| While loop is also termed an entry-controlled loop | The do-while loop is termed an exit control loop |
| If the condition is not satisfied the statements inside the loop will not execute | Even if the  condition is not satisfied the statements inside the loop will execute for at least one time |
| **Example**of a while loop:  while(condition)  {statements to be executed;}; | **Example**of a do-while loop:  do {  statements to be executed;  } while(condition or expression); |

For more information, refer to the [Difference between while and do-while loop](https://www.geeksforgeeks.org/difference-between-while-and-do-while-loop-in-c-c-java/)

**13. Discuss the difference between prefix and postfix?**

| **prefix** | **postfix** |
| --- | --- |
| It simply means putting the operator before the operand | It simply means putting the operator after the operand |
| It executes itself before **‘; ‘** | It executes itself after **‘; ‘** |
| Associativity of prefix ++ is right to left | Associativity of postfix ++ is left to right |

For more information, refer to the [Difference between prefix and postfix](https://www.geeksforgeeks.org/increment-and-decrement-operator-overloading-in-c/)

**14. What is the difference between new and malloc()?**

| **new** | **malloc()** |
| --- | --- |
| new is an operator which performs an operation | malloc is a function that returns and accepts values |
| new calls the constructors | malloc cannot call a constructor |
| new is faster than malloc as it is an operator | malloc is slower than new as it is a function |
| new returns the exact data type | malloc returns void\* |

For more information, refer to [Difference between new and malloc()](https://www.geeksforgeeks.org/malloc-vs-new/)

**Language Context:**

* **new**: It is a C++ operator, specifically designed for object-oriented programming in C++.
* **malloc()**: It is a C function, used for dynamic memory allocation in both C and C++.

**2. Memory Allocation:**

* **new**: Allocates memory for an object or array of objects and automatically calls the constructor of the object. It returns a pointer of the appropriate type.
* **malloc()**: Allocates a block of memory of a specified size (in bytes) but does not call any constructors. It returns a void\* pointer, which must be typecast to the appropriate type.

**3. Memory Deallocation:**

* **new**: Memory allocated with new should be deallocated using the delete operator (or delete[] for arrays), which also calls the destructor of the object.
* **malloc()**: Memory allocated with malloc() should be deallocated using free(), which does not call destructors.

**4. Initialization:**

* **new**: Automatically initializes objects (using constructors). For example, int\* p = new int(5); initializes p with the value 5.
* **malloc()**: Does not initialize the allocated memory. The memory block contains garbage values until explicitly initialized.

**5. Operator vs Function:**

* **new**: Is an operator, which means it can be overloaded and has different behaviors for different data types or objects.
* **malloc()**: Is a function and cannot be overloaded. It is more generic and less flexible compared to new.

**6. Type Safety:**

* **new**: Provides type safety by returning a pointer of the correct type, reducing the chances of type errors.
* **malloc()**: Returns a void\*, requiring explicit casting to the desired pointer type, which can lead to errors if the wrong type is used.

**7. Overloading:**

* **new**: Can be overloaded globally or within a class to customize memory allocation behavior.
* **malloc()**: Cannot be overloaded.

**Example:**

**Using new:**

cpp

Copy code

int\* p = new int(10); // Allocates memory and initializes it to 10

// Use the memory

delete p; // Deallocates memory and calls the destructor (if applicable)

**Using malloc():**

cpp

Copy code

int\* p = (int\*)malloc(sizeof(int)); // Allocates memory but does not initialize it

\*p = 10; // Manually initialize the value

// Use the memory

free(p); // Deallocates memory

In summary, new is more suitable for C++ due to its object-oriented features, such as calling constructors and destructors, while malloc() is more basic and is used in C or when you need more control over memory allocation.

4o

**You said:**

give me the above in tabular form

**ChatGPT said:**

ChatGPT

Here's the comparison between new and malloc() in tabular form:

| **Feature** | **new** | **malloc()** |
| --- | --- | --- |
| **Language Context** | C++ operator | C function (also used in C++) |
| **Memory Allocation** | Allocates memory and calls constructors | Allocates memory without calling constructors |
| **Memory Deallocation** | Deallocate using delete (or delete[]) and calls destructors | Deallocate using free() with no destructor calls |
| **Initialization** | Automatically initializes memory | Memory is not initialized, contains garbage values |
| **Type Safety** | Type-safe, returns a pointer of the correct type | Returns void\*, requiring explicit casting |
| **Operator vs Function** | Operator (can be overloaded) | Function (cannot be overloaded) |
| **Overloading** | Can be overloaded globally or within a class | Cannot be overloaded |

4o

**15. What is the difference between virtual functions and pure virtual functions?**

| **Virtual Function** | **Pure Virtual Function** |
| --- | --- |
| A Virtual Function is a member function of a base class that can be redefined in another derived class. | A Pure Virtual Function is a member function of a base class that is only declared in a base class and defined in a derived class to prevent it from becoming an abstract class. |
| A virtual Function has its definition in its respective base class. | There is no definition in Pure Virtual Function and is initialized with a pure specifier **(= 0).** |
| The base class has a virtual function that can be represented or instanced; In simple words, its object can be made. | A base class having pure virtual function becomes abstract that cannot be represented or instanced; In simple words, it means its object cannot be made. |

 For more information, refer to the [Difference between virtual functions and pure virtual functions](https://www.geeksforgeeks.org/difference-between-virtual-function-and-pure-virtual-function-in-c/)

**16. What are classes and objects in C++?**

A class is a user-defined data type where all the member functions and data members are tailor-made according to demands and requirements in addition to which these all can be accessed with the help of an **object**. To declare a user-defined data type we use a keyword ***class.***

An object is an instance of a class and an entity with value and state; In simple terms, it is used as a catalyst or to represent a class member. It may contain different parameters or none.

***Note:****A class is a blueprint that defines functions which are used by an object.*

For more information, refer to this [What are classes and objects](https://www.geeksforgeeks.org/c-classes-and-objects/)

**17. What is Function Overriding?**

When a function of the same name, same arguments or parameters, and same return type already present/declared in the base class is used in a derived class is known as Function Overriding. It is an example of Runtime Polymorphism or Late Binding which means the overridden function will be executed at the run time of the execution.

For more information, refer to [Function Overriding in C++](https://www.geeksforgeeks.org/function-overloading-vs-function-overriding-in-cpp/)

**18. What are the various OOPs concepts in C++?**

* ***Classes*:** It is a user-defined datatype
* ***Objects*:**It is an instance of a class
* **Abstraction:** It is a technique of showing only necessary details
* **Encapsulation:** Wrapping of data in a single unit: typically a class, and restricting access to some of the object's components.
* **Inheritance:** The capability of a class to derive properties and characteristics from another class (derived class) to inherit attributes and methods from another class (base class),
* **Polymorphism:**Polymorphism is known as many forms of the same thing

For more information, refer to [Various OOPs concepts in C++](https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/)

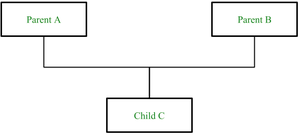
**19. Explain inheritance**

The capability or ability of a class to derive properties and characteristics from another class is known as inheritance. In simple terms, it is a system or technique of reusing and extending existing classes without modifying them.

For more information, refer to [Inheritance](https://www.geeksforgeeks.org/inheritance-in-c/)

**20. When should we use multiple inheritance?**

Multiple inheritances mean that a derived class can inherit two or more base/parent classes. It is useful when a derived class needs to combine numerous attributes/contracts and inherit some, or all, of the implementation from these attributes/contracts. To take a real-life example consider your Parents where Parent A is your DAD Parent B is your MOM and Chid C is you.



*Multiple Inheritances*

For more information, refer to [Multiple Inheritance](https://www.geeksforgeeks.org/multiple-inheritance-in-c/).

**21. What is virtual inheritance?**

Virtual inheritance is a technique that ensures only one copy of a base class’s member variables is inherited by grandchild-derived classes. Or in simple terms, virtual inheritance is used when we are dealing with a situation of multiple inheritances but want to prevent multiple instances of the same class from appearing in the inheritance hierarchy.

**22. What is polymorphism in C++?**

Polymorphism is known as many forms of the same thing. In simple terms, we can say that Polymorphism is the ability to display a member function in multiple forms depending on the type of object that calls them.

In other words, we can also say that a man can be an employee to someone, a son of someone, a father of someone, and a husband of someone; this is how polymorphism can be projected in real life.

There is 2 type of polymorphism:

1. ***Compile Time Polymorphism***
   * Function Overloading
   * Operator Overloading
2. ***Run Time Polymorphism***
   * Function Overriding
   * Virtual Function

To know more about it, refer to [Polymorphism](https://www.geeksforgeeks.org/polymorphism-in-c/)

**23. What are the different types of polymorphism in C++?**

There is 2 type of polymorphism

**Compile Time Polymorphism or Static Binding**

This type of polymorphism is achieved during the compile time of the program which results in it making a bit faster than Run time. Also, Inheritance is not involved in it. It is comprised of ***2 further techniques***:

**Function Overloading:**When there are multiple functions with the same name but different parameters then this is known as function overloading.

C++

Explain

*// same name different arguments*

int GFG() {}

int GFG(int a) {}

float GFG(double a) {}

int GFG(int a, double b) {}

**Operator Overloading:**It is basically giving practice of giving a special meaning to the existing meaning of an operator or in simple terms redefining the pre-redefined meaning

C++

Explain

**class** **GFG** {

*// private and other modes*

statements **public** returnType

**operator** symbol(arguments){ statements } statements

};

**Run-Time Polymorphism or Late Binding**

Run-time polymorphism takes place when functions are invoked during run time.   
   
**Function Overriding:**Function overriding occurs when a base class member function is redefined in a derived class with the same arguments and return type.

C++

Explain

*// C++ program to demonstrate*

*// Function overriding*

#include

*<iostream> using namespace std;*

**class** **GFG** {

**public**:

**virtual** void display()

{

cout << "Function of base class" << endl;

}

};

**class** **derived\_GFG** : **public** GFG {

**public**:

void display()

{

cout << "Function of derived class" << endl;

}

};

int main()

{

derived\_GFG dg;

dg.display();

**return** 0;

}

**Output:**

Function of derived class

For more information, refer to [Different types of Polymorphism](https://www.geeksforgeeks.org/polymorphism-in-c/)

**24. Compare compile-time polymorphism and Runtime polymorphism**

| **Compile-Time Polymorphism** | **Runtime Polymorphism** |
| --- | --- |
| It is also termed static binding and early binding. | It is also termed Dynamic binding and Late binding. |
| It is fast because execution is known early at compile time. | It is slow as compared to compile-time because execution is known at runtime. |
| It is achieved by function overloading and operator overloading. | It is achieved by virtual functions and function overriding. |

For more information, refer to [Compile-time polymorphism and Runtime polymorphism](https://www.geeksforgeeks.org/difference-between-compile-time-and-run-time-polymorphism-in-java/)

**25. Explain the constructor in C++.**

A constructor is a special type of member function of a class, whose name is the same as that of the class by whom it is invoked and initializes value to the object of a class.

There are 3 types of constructors:

**A. Default constructor:**It is the most basic type of constructor which accepts no arguments or parameters. Even if it is not called the compiler calls it automatically when an object is created.

**Example:**

C++

Explain

**class** **Class\_name** {

**public**:

Class\_name() { cout << "I am a default constructor"; }

};

**B. Parameterized constructor:**It is a type of constructor which accepts arguments or parameters. It has to be called explicitly by passing values in the arguments as these arguments help initialize an object when it is created. It also has the same name as that of the class.

Also, It is used to overload constructors.

**Example:**

C++

Explain

*// CPP program to demonstrate*

*// parameterized constructors*

#include

*<iostream> using namespace std;*

**class** **GFG** {

**private**:

int x, y;

**public**:

*// Parameterized Constructor*

GFG(int x1, int y1)

{

x = x1;

y = y1;

}

int getX() { **return** x; }

int getY() { **return** y; }

};

int main()

{

*// Constructor called*

GFG G(10, 15);

*// Access values assigned by constructor*

cout << "G.x = " << G.getX() << ", G.y = " << G.getY();

**return** 0;

}

**Output**

G.x = 10, G.y = 15

**C. Copy Constructor:**A copy constructor is a member function that initializes an object using another object of the same class. Also, the Copy constructor takes a reference to an object of the same class as an argument.

**Example:**

C++

Sample(Sample& t) { id = t.id; }

For more information, refer to [Constructors](https://www.geeksforgeeks.org/constructors-c/)

**26. What are destructors in C++?**

Destructors are members of functions in a class that delete an object when an object of the class goes out of scope. Destructors have the same name as the class preceded by a tilde (~) sign. Also, destructors follow a **down-to-top**approach, unlike constructors which follow a top-to-down.

**Syntax:**

**~**constructor\_name(); // tilde sign signifies that it is a destructor

For more information, refer to [Destructor](https://www.geeksforgeeks.org/destructors-c/).

**27. What is a virtual destructor?**

When destroying instances or objects of a derived class using a base class pointer object, a virtual destructor is invoked to free up memory space allocated by the derived class object or instance.

Virtual destructor guarantees that first the derived class’ destructor is called. Then the base class’s destructor is called to release the space occupied by both destructors in the inheritance class which saves us from the memory leak. It is advised to make your destructor virtual whenever your class is polymorphic.( class is considered **polymorphic** if it contains at least one **virtual function**. This means the class supports **polymorphism**, which is the ability to call derived class methods through a base class pointer or reference, allowing for dynamic (runtime) method binding.)

For more information, refer to [Virtual Destructor](https://www.geeksforgeeks.org/virtual-destructor/)



**28. Is destructor overloading possible? If yes then explain and if no then why?**

The simple answer is **NO** we cannot overload a destructor. It is mandatory to only destructor per class in C++. Also to mention, Destructor neither take arguments nor they have a parameter that might help to overload.



**C++ Interview Questions – Intermediate Level**

**29. Which operations are permitted on pointers?**

Pointers are the variables that are used to store the address location of another variable. Operations that are permitted to a pointer are:

1. Increment/Decrement of a Pointer
2. Addition and Subtraction of integer to a pointer
3. Comparison of pointers of the same type

**30. What is the purpose of the “*delete*” operator?**

The delete operator is used to delete/remove all the characteristics/properties from an object by deallocating its memory; furthermore, it returns true or false in the end. In simple terms, it destroys or deallocates array and non-array(pointer) objects which are created by new expressions.

C++

int GFG = **new** int[100];

*// uses GFG for deletion*

**delete**[] GFG;

For more information, refer to [Delete operator](https://www.geeksforgeeks.org/delete-in-c/)

**31. How delete [] is different from delete?**

| **delete[]** | **delete** |
| --- | --- |
| It is used for deleting a whole array | It is used to delete only one single pointer |
| It is used for deleting the objects of **new[];**By this, we can say that **delete[]** is used to delete an array of objects | It is used for deleting the objects of **new;**By this, we can say that **delete**is used to delete a single object |
| It can call as many destructors it wants | It can only call the destructor of a class once |

**32. What do you know about friend class and friend function?**

A friend class is a class that can access both the protected and private variables of the classes where it is declared as a friend.

**Example of friend class:**

C++

Explain

**class** **Class\_1st** {

*// ClassB is a friend class of ClassA*

**friend** **class** **Class\_2nd**;

statements;

} **class** **Class\_2nd** {

statements;

}

A friend function is a function used to access the private, protected, and public data members or member functions of other classes. It is declared with a friend keyword. The advantage of a friend function is that it is not bound to the scope of the class and once it is declared in a class, furthermore to that, it cannot be called by an object of the class; therefore it can be called by other functions. Considering all the mentioned points we can say that a friend function is a global function**.**

**Example of friend function:**

C++

Explain

**class** **GFG** {

statements;

**friend** dataype function\_Name(arguments);

statements;

} OR **class** **GFG** {

statements' **friend** int divide(10, 5);

statements;

}

For more information, refer to the [friend function and friend class](https://www.geeksforgeeks.org/friend-class-function-cpp/)

**33. What is an Overflow Error?**

Overflow Error occurs when the number is too large for the data type to handle. In simple terms, it is a type of error that is valid for the defined but exceeds used the defined range where it should coincide/lie.

For example, the range of int data type is **–2,147,483,648**to **2,147,483,647** and if we declare a variable of size**2,247,483,648** it will generate a overflow error.

**34. What does the Scope Resolution operator do?**

A scope resolution operator is denoted by a ‘**::**‘ symbol. Just like its name this operator resolves the barrier of scope in a program. A scope resolution operator is used to reference a member function or a global variable out of their scope furthermore to which it can also access the concealed variable or function in a program.

Scope Resolution is used for numerous amounts of tasks:

1. To access a global variable when there is a local variable with the same name
2. To define the function outside the class
3. In case of multiple inheritances
4. For namespace

For more information, refer to [Scope resolution operator](https://www.geeksforgeeks.org/scope-resolution-operator-in-c/)

**35. What are the C++ access modifiers?**

The access restriction specified to the class members( whether it is member function or data member) is known as access modifiers/specifiers.

**Access Modifiers are of 3 types:**

1. **Private –**It can neither be accessed nor be viewed from outside the class
2. **Protected –**It can be accessed if and only if the accessor is the derived class
3. **Public –**It can be accessed or be viewed from outside the class

For more information, refer to[Access Modifiers](https://www.geeksforgeeks.org/access-modifiers-in-c/)

**36. Can you compile a program without the main function?**

Yes, it is absolutely possible to compile a program without a main(). For example Use Macros that defines the main

C++

Explain

*// C++ program to demonstrate the*

*// a program without main()*

#include

*<stdio.h>*

#define fun main

int fun(void)

{

printf("Geeksforgeeks");

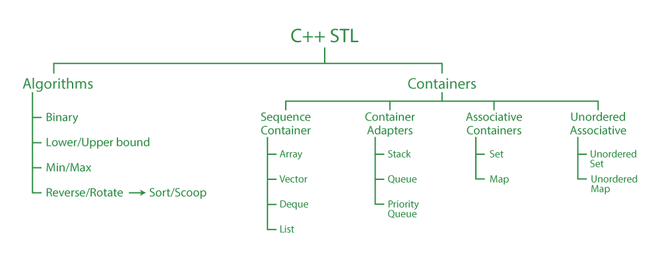
**return** 0;

}

For more information, refer to [Can you compile a program without the main function](https://www.geeksforgeeks.org/write-running-c-code-without-main/)

**37. What is STL?**

STL is known as Standard Template Library, it is a library that provides 4 components like container, algorithms, and iterators.



*C++ STL*

For more information, refer to [STL in C++](https://www.geeksforgeeks.org/the-c-standard-template-library-stl/)

**Key Categories of STL Algorithms**

1. **Non-Modifying Sequence Operations**:
   * These algorithms do not change the elements in the container but may perform operations like searching, counting, or comparing.
   * **Examples**:
     + std::find: Searches for a specific value in a container.
     + std::count: Counts the occurrences of a specific value in a container.
     + std::equal: Compares elements in two containers to check if they are equal.
     + std::mismatch: Finds the first position where two containers differ.
2. **Modifying Sequence Operations**:
   * These algorithms change the elements in the container by performing operations like replacing, copying, or swapping elements.
   * **Examples**:
     + std::fill: Fills a container with a specific value.
     + std::copy: Copies elements from one container to another.
     + std::replace: Replaces all instances of a specific value in a container with another value.
     + std::remove: Removes elements with a specific value from a container (though it doesn't change the container's size).
3. **Sorting and Searching**:
   * These algorithms are used to order elements in a container or search for elements.
   * **Examples**:
     + std::sort: Sorts the elements in a container in ascending order by default.
     + std::stable\_sort: Sorts while maintaining the relative order of equivalent elements.
     + std::binary\_search: Checks if a value exists in a sorted container using binary search.
     + std::lower\_bound and std::upper\_bound: Find the first or last position in a sorted range where a value could be inserted.
4. **Partitioning**:
   * These algorithms reorder elements in a container based on a predicate (a condition).
   * **Examples**:
     + std::partition: Reorders elements so that elements satisfying the predicate come before those that do not.
     + std::stable\_partition: Like std::partition, but maintains the relative order of elements.
5. **Merge and Set Operations**:
   * These algorithms combine elements from multiple containers or perform set-based operations.
   * **Examples**:
     + std::merge: Merges two sorted ranges into a single sorted range.
     + std::set\_union: Computes the union of two sorted ranges.
     + std::set\_intersection: Computes the intersection of two sorted ranges.
     + std::set\_difference: Computes the difference between two sorted ranges.
6. **Heap Operations**:
   * These algorithms work with heaps, a specialized tree-based data structure.
   * **Examples**:
     + std::make\_heap: Converts a range into a heap.
     + std::push\_heap: Adds a new element to the heap while maintaining the heap property.
     + std::pop\_heap: Removes the largest element from the heap.
     + std::sort\_heap: Sorts a heap into ascending order.
7. **Minimum and Maximum Operations**:
   * These algorithms are used to find the smallest or largest elements in a container.
   * **Examples**:
     + std::min\_element and std::max\_element: Find the smallest or largest element in a range.
     + std::min and std::max: Return the smaller or larger of two values.
     + std::minmax: Returns both the smallest and largest of two values.
8. **Permutations**:
   * These algorithms generate different orderings or permutations of elements in a container.
   * **Examples**:
     + std::next\_permutation: Generates the next lexicographical permutation of a range.
     + std::prev\_permutation: Generates the previous lexicographical permutation.
9. **Numeric Operations**:
   * These algorithms perform arithmetic operations on elements in containers.
   * **Examples**:
     + std::accumulate: Sums up all the elements in a range.
     + std::inner\_product: Computes the inner product of two ranges.
     + std::partial\_sum: Computes the partial sums of a range.
     + std::adjacent\_difference: Computes the differences between adjacent elements in a range.

**How STL Algorithms Work**

* **Iterators**: STL algorithms are designed to work with iterators, which provide a way to traverse containers. The flexibility of iterators allows these algorithms to work with any container that supports the required iterator type (e.g., random access iterators, bidirectional iterators).
* **Generic Programming**: STL algorithms are highly generic and can operate on different types of data. For example, you can use std::sort to sort a vector<int>, list<double>, or even a custom container if it provides the necessary iterator support.
* **Function Objects and Lambda Expressions**: Many STL algorithms accept function objects (functors) or lambda expressions to define custom operations. For example, you can use a lambda function to define a custom comparison for std::sort.

**Example of Using STL Algorithms**

cpp

Copy code

#include <iostream>

#include <vector>

#include <algorithm>

int main() {

std::vector<int> vec = {3, 1, 4, 1, 5, 9, 2, 6, 5};

// Sort the vector

std::sort(vec.begin(), vec.end());

// Remove duplicates (must be sorted first)

auto last = std::unique(vec.begin(), vec.end());

vec.erase(last, vec.end());

// Print the sorted and unique vector

for (int n : vec) {

std::cout << n << " ";

}

return 0;

}

**Output:**

Copy code

1 2 3 4 5 6 9

In this example:

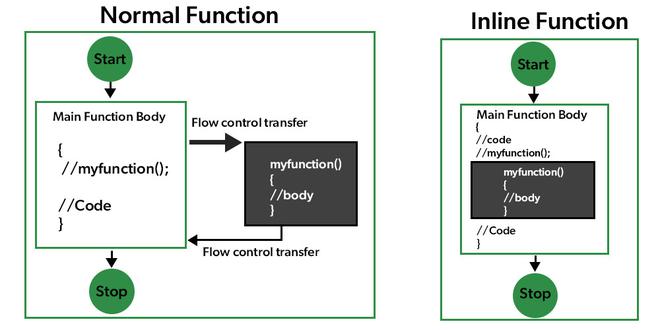
* **std::sort** sorts the elements in the vector.
* **std::unique** removes duplicate elements (though it requires that the container is sorted first).
* **std::erase** is used to remove the redundant elements after calling std::unique.

STL algorithms are essential for writing efficient and clean C++ code, enabling developers to perform complex operations with minimal code.

4o

**38. Define inline function. Can we have a recursive inline function in C++?**

An inline function is a form of request not an order to a compiler which results in the inlining of our function to the main function body. An inline function can become overhead if the execution time of the function is less than the switching time from the caller function to called function. To make a function inline use the keyword **inline** before and define the function before any calls are made to the function.



*Inline Function Explanation*

**Syntax:**

inline data\_type function\_name()  
{  
Body;  
}

The answer is **No;**It cannot be recursive.

An inline function cannot be recursive because in the case of an inline function the code is merely placed into the position from where it is called and does not maintain a piece of information on the stack which is necessary for recursion.

Plus, if you write an inline keyword in front of a recursive function, the compiler will automatically ignore it because the inline is only taken as a suggestion by the compiler.

For more information, refer to [Inline Function](https://www.geeksforgeeks.org/inline-functions-cpp/)

**What is an Inline Function?**

An **inline function** is a request to the compiler to substitute the function call with the actual code of the function at the place where it is invoked. This can potentially reduce the overhead of function calls, especially for small functions, by avoiding the usual function call mechanism (which involves pushing arguments to the stack, jumping to the function code, executing it, and then returning).

**Syntax:**

cpp

Copy code

inline data\_type function\_name() {

// Function body

}

**When to Use Inline Functions?**

Inline functions are generally used for small, frequently called functions where the overhead of a function call would be significant compared to the execution time of the function itself. Common examples include simple accessor functions, small mathematical calculations, or other short operations.

**Key Points about Inline Functions:**

1. **Inline is a Request, Not a Command**: The inline keyword is merely a suggestion to the compiler. The compiler may choose to ignore this request, especially if inlining the function doesn't make sense (e.g., if the function is too large, or contains loops).
2. **Inline Overhead**: Inlining a function can actually be counterproductive if the function is large or complex because the code will be duplicated each time the function is called, potentially increasing the binary size (also known as code bloat).
3. **Declaration and Definition**: To inline a function, its definition must be available to the compiler before any calls are made. This typically means that the function must be defined in the header file if it's used in multiple source files.

**Recursive Inline Functions**

Now, regarding the statement:

**"The answer is No; It cannot be recursive."**

This is a bit misleading. Technically, you **can** define a recursive function as inline, but there are important considerations:

1. **Compiler Behavior**: If you mark a recursive function as inline, the compiler is likely to ignore the inline request. This is because inlining a recursive function is generally counterproductive. If a function were truly inlined in a recursive manner, it could lead to infinite code expansion at compile time.
2. **Stack and Recursion**: Inline functions do not maintain a call stack frame as typical functions do, because their code is substituted directly at the call site. However, recursion inherently requires maintaining a stack to track function calls and returns. For this reason, recursive functions do not lend themselves well to inlining.
3. **Real-World Use**: In practice, if you write an inline recursive function, the compiler will most likely compile it as a normal recursive function and ignore the inline suggestion.

**Example:**

cpp

Copy code

inline int factorial(int n) {

return (n <= 1) ? 1 : n \* factorial(n - 1);

}

In this example, factorial is defined as an inline function. However, the compiler will generally ignore the inline request because recursion is involved.

**Conclusion:**

* **Yes**, you can write a recursive inline function, but the compiler will typically not inline it due to the nature of recursion.
* **Inline functions** are beneficial for small, non-recursive functions where function call overhead is significant.
* The decision to inline a function is ultimately made by the compiler, based on various factors like function complexity and potential code bloat.

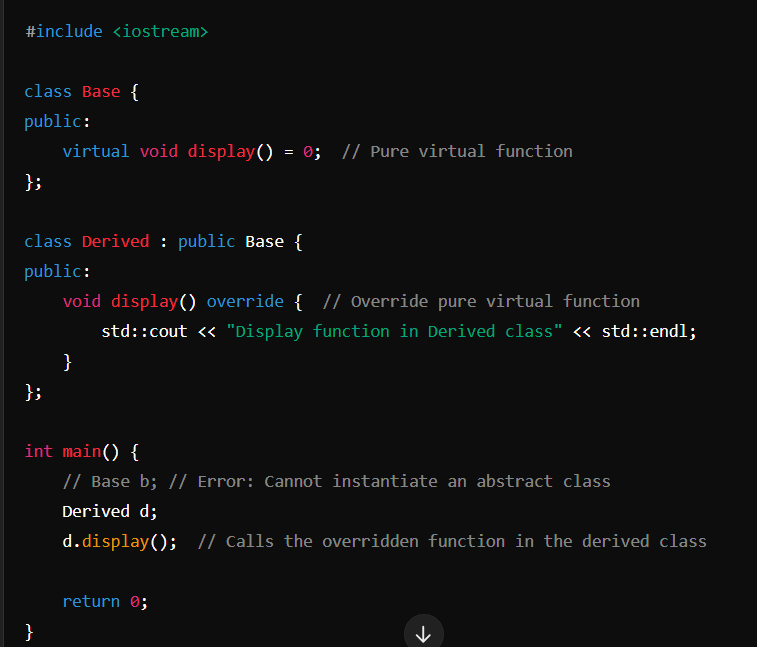
By clarifying these points, your understanding of inline functions and their relationship with recursion should be more accurate and complete.

**39. What is an abstract class and when do you use it?**

An abstract class is a class that is specifically designed to be used as a base class. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a ***pure specifier(= 0)*** in the declaration of a virtual member function in the class declaration

You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class. However, it can be used to declare pointers and references to an abstract class.

An abstract class is used if you want to provide a common, implemented functionality among all the implementations of the component. Abstract classes will allow you to partially implement your class, whereas interfaces would have no implementation for any members whatsoever. In simple words, Abstract Classes are a good fit if you want to provide implementation details to your children but don’t want to allow an instance of your class to be directly instantiated.



**40. What are the static data members and static member functions?**

The static data member of a class is a normal data member but preceded with a static keyword. It executes before main() in a program and is initialized to 0 when the first object of the class is created. It is only visible to a defined class but its scope is of a lifetime.

**Syntax:**

**static Data\_Type Data\_Member;**

The static member function is the member function that is used to access other static data members or other static member functions. It is also defined with a static keyword. We can access the static member function using the class name or class objects.

**Syntax:**

classname::function name(parameter);

**C++ Interview Questions – Expert Level**

**41. What is the main use of the keyword “Volatile”?**

Just like its name, things can change suddenly and unexpectantly; So it is used to inform the compiler that the value may change anytime. Also, the volatile keyword prevents the compiler from performing optimization on the code. It was intended to be used when interfacing with memory-mapped hardware, signal handlers, and machine code instruction.

For more information, refer to this [Volatile](https://www.geeksforgeeks.org/understanding-volatile-qualifier-c-set-1-introduction/)

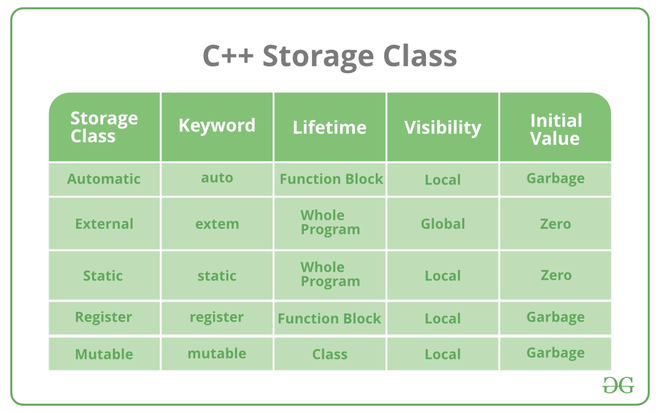
**42. Define storage class in C++ and name some**

Storage class is used to define the features(lifetime and visibility) of a variable or function. These features usually help in tracing the existence of a variable during the runtime of a program.

**Syntax:**

storage\_class var\_data\_type var\_name;

**Some types of storage classes:**



*Examples of storage class*

For more information, refer to [Storage Class](https://www.geeksforgeeks.org/storage-classes-in-c-with-examples/)

**43. What is a mutable storage class specifier? How can they be used?**

Just like its name, the mutable storage class specifier is used only on a class data member to make it modifiable even though the member is part of an object declared as const. Static or const, or reference members cannot use the mutable specifier. When we declare a function as const, this pointer passed to the function becomes const.

**44. Define the Block scope variable.**

So the scope of a variable is a region where a variable is accessible. There are two scope regions, A global and block or local.

A block scope variable is also known as a local scope variable. A variable that is defined inside a function (like main) or inside a block (like loops and if blocks) is a local variable. It can be used ONLY inside that particular function/block in which it is declared. a block-scoped variable will not be available outside the block even if the block is inside a function.

For more information, refer to [Scope of a variable](https://www.geeksforgeeks.org/scope-of-variables-in-c/)

**45. What is the function of the keyword “Auto”?**

The auto keyword may be used to declare a variable with a complex type in a straightforward fashion. You can use auto to declare a variable if the initialization phrase contains templates, pointers to functions, references to members, etc. With type inference capabilities, we can spend less time having to write out things the compiler already knows. As all the types are deduced in the compiler phase only, the time for compilation increases slightly but it does not affect the runtime of the program.

For more information, refer to[Auto in C++](https://www.geeksforgeeks.org/type-inference-in-c-auto-and-decltype/)

**46.  Define namespace in C++.**

Namespaces enable us to organize named items that would otherwise have global scope into smaller scopes, allowing us to give them namespace scope. This permits program parts to be organized into distinct logical scopes with names. The namespace provides a place to define or declare identifiers such as variables, methods, and classes.

Or we could say that A namespace is a declarative zone that gives the identifiers (names of types, functions, variables, and so on) within it a scope. Namespaces are used to arrange code into logical categories and to avoid name clashes, which might happen when you have many libraries in your code base.

For more information, refer to [Namespace in C++](https://www.geeksforgeeks.org/namespace-in-c/)

**47. When is void() return type used?**

The void keyword, when used as a function return type, indicates that the function does not return a value. When used as a parameter list for a function, void indicates that the function takes no parameters. Non-Value Returning functions are also known as void functions. They’re called “void” since they’re not designed to return anything. True, but only partially. We can’t return values from void functions, but we can certainly return something. Although void functions have no return type, they can return values.

For more information, refer to [Void return type](https://www.geeksforgeeks.org/return-from-void-functions-in-cpp/).

**48. What is the difference between shallow copy and deep copy?**

| **Shallow Copy** | **Deep Copy** |
| --- | --- |
| In Shallow copy, a copy of the original object is stored and only the reference address is finally copied. In simple terms, Shallow copy duplicates as little as possible | In Deep copy, the copy of the original object and the repetitive copies both are stored. In simple terms, Deep copy duplicates everything |
| A shallow copy of a collection is a copy of the collection structure, not the elements. With a shallow copy, two collections now share individual elements. | A deep copy of a collection is two collections with all of the elements in the original collection duplicated. |
| A shallow copy is faster | Deep copy is comparatively slower. |

For more information, refer to [Shallow copy VS Deep Copy](https://www.geeksforgeeks.org/difference-between-shallow-and-deep-copy-of-a-class/)

**49. Can we call a virtual function from a constructor?**

Yes, we can call a virtual function from a constructor. But it can throw an exception of overriding.

**50. What are void pointers?**

Just like its name a void pointer is a pointer that is not associated with anything or with any data type. Nevertheless, a void pointer can hold the address value of any type and can be converted from one data type to another.

For more, information refers to [Void Pointer in C++](https://www.geeksforgeeks.org/void-pointer-c-cpp/)

**Bonus Question:**

**What is ‘*this*‘ pointer in C++?**

***this***pointer enables every object to have access to its own address through an essential pointer. All member functions take ***this*** pointer as an implicit argument. ***this***pointermay be used to refer to the calling object within a member function.

* *this*pointer is used to pass an object as a parameter to another method.
* Each object gets its own copy of the data member.
* *this*pointer is used to declare indexers.

For more information, refer to [*this*pointer in C++](https://www.geeksforgeeks.org/this-pointer-in-c/)

[Three 90 Challenge is back on popular demand! After processing refunds worth INR 1CR+, we are back with the offer if you missed it the first time. Get 90% course fee refund in 90 days. Avail now!](https://www.geeksforgeeks.org/courses?utm_source=geeksforgeeks&utm_medium=bottomtext_ad&utm_campaign=three90)

Summer-time is here and so is the time to skill-up! More than 5,000 learners have now completed their journey from **basics of DSA to advanced level development programs** such as Full-Stack, Backend Development, Data Science.   
  
And why go anywhere else when our [DSA to Development: Coding Guide](https://gfgcdn.com/tu/Q8V/) will help you master all this in a few months! Apply now to our [DSA to Development Program](https://gfgcdn.com/tu/Q8V/) and our counsellors will connect with you for further guidance & support.