Q 1. concept of a virtual function

=>A virtual function in C++ is a member function in a base class that can be overridden in a derived class. It enables **runtime polymorphism**, allowing the correct function to be called **based on the object's actual type, not the pointer/reference type.**

For_each	Syntax :
	for_each(begin, end ,function_name/lambda)
	Ex.
	For array
	for_each(array,array+size [](int x) { cout<<x});
	for Vector
	for_each(vec.begin(),v.end(),[](int x) { cout<<x});

Time Complexity :

- If your code **divides** the problem size **by 2**, the time complexity is O(log n) with an implied base of 2.
- If it divides by 3, 4, 5, or any constant, it's still O(log n) because the base only affects the **constant** factor, which Big O ignores.

Common Time Complexities

- **O(1)**: Constant (e.g., accessing an array index).
- **O(log n)**: Logarithmic (e.g., binary search).
- **O(n)**: Linear (e.g., searching an unsorted list).
- **O(n log n)**: Linearithmic (e.g., efficient sorting like mergesort).
- **O(n^2)**: Quadratic (e.g., bubble sort, nested loops).
- **O(2^n)**: Exponential (e.g., recursive solutions to some problems like the Tower of Hanoi, Fibonacci series).

Complexity	Example	Explanation
O(1)	Accessing an array element	Constant time (no loops)
O(log n)	Binary search	Halving input each iteration
O(n)	Linear search	Single loop
O(n log n)	Merge sort	Divide-and-conquer with merging
$O(n^2)$	Bubble sort	Two nested loops
$O(2^n)$	Generating all subsets	Recursive tree (double choices)

•

Data Structures	Algorithms	Concepts
Linked Lists	Breadth-First Search	Bit Manipulation
Trees, Tries, & Graphs	Depth-First Search	Memory (Stack vs. Heap)
Stacks & Queues	Binary Search	Recursion
Heaps	Merge Sort	Dynamic Programming
Vectors / ArrayLists	Quick Sort	Big O Time & Space
Hash Tables		

Walk through your brute force with BUD(Bottle neck,Unused, Duplicate)

optimization or try some of these ideas:

• Look for any unused info. You usually need all the information in a problem.

• Solve it manually on an example, then reverse engineer your thought process.

• How did you solve it? � Solve it"incorrectly" and then think about why the algorithm fails.

• Can you fix those issues? � Make a time vs. space tradeoff. Hash tables are especially useful!

shared_ptr, weak_ptr, unique_ptr

unique_ptr

=> Only one unique_ptr can own the object at a time.

Feature : 1. Automatically delete when goes out of scope.
2. Non-Copyble.
3. Move support

Shared_ptr

=>multiple reference count for same object.

Features: 1. Shared ownership, tracks reference count.
2. Deletes the object when the last shared_ptr is destroyed.

weak_ptr

=>std::weak_ptr: Non-owning reference to a shared_ptr-managed object, used to break circular references.

By Grok : C++ interview question answers for experience

1. What are the key differences between C and C++?

Answer:

- **Paradigm**: C is procedural, while C++ supports both procedural and object-oriented programming (OOP) with features like classes, objects, and inheritance.
- **Memory Management**: C uses manual memory management (e.g., malloc/free), while C++ provides new/delete and smart pointers for safer memory management.
- **Features**: C++ supports polymorphism, encapsulation, templates, and the Standard Template Library (STL), which are absent in C.
- **Function Overloading**: C++ allows function and operator overloading, while C does not.
- **Exception Handling**: C++ has try-catch blocks for exception handling, whereas C relies on error codes.

2. Explain the concept of a virtual function and why it's used.

Answer: A virtual function in C++ is a member function in a base class that can be overridden in a derived class. It enables **runtime polymorphism**, allowing the correct function to be called based on the object's actual type, not the pointer/reference type.

- **Usage**: Declared with the virtual keyword in the base class. When a derived class overrides it, the call is resolved via a **vtable** (virtual table) at runtime.
- **Example**:
- cpp
- CollapseWrapRun
- Copy

```cpp
class Base {

public:

    virtual void show() { std::cout << "Base\n"; }

};

class Derived : public Base {

public:

    void show() override { std::cout << "Derived\n"; }

};

int main() {

    Base* ptr = new Derived();

    ptr->show(); // Outputs "Derived" due to virtual function

    delete ptr;

    return 0;
```

-
- }
- **Why Used**: To achieve dynamic dispatch, enabling flexible and extensible code in inheritance hierarchies.

3. What is the difference between delete and delete[]?

Answer:

- **delete**: Used to deallocate memory for a single object allocated with new. It calls the destructor of the object.
- **delete[]**: Used to deallocate memory for an array of objects allocated with new[]. It calls the destructor for each object in the array.
- **Key Difference**: Using delete on an array (new[]) or delete[] on a single object (new) leads to undefined behavior. Always match new with delete and new[] with delete[].
- **Example**:
- cpp
- CollapseWrap
- Copy
- `int* single = new int;`
-
- `int* array = new int[10];`
-
- `delete single;   // Correct`
-
- `delete[] array;  // Correct`

4. What are smart pointers, and how do they differ from raw pointers?

Answer: Smart pointers are C++ objects that manage the lifetime of dynamically allocated memory, preventing memory leaks. They are part of the STL (<memory>).

- **Types**:

- **std::unique_ptr**: Exclusive ownership, non-copyable, movable. Deletes the object when the pointer goes out of scope.
- **std::shared_ptr**: Shared ownership, tracks reference count. Deletes the object when the last shared_ptr is destroyed.
- **std::weak_ptr**: Non-owning reference to a shared_ptr-managed object, used to break circular references.
- **Differences from Raw Pointers**:
- **Ownership**: Smart pointers automatically manage memory; raw pointers require manual delete.
- **Safety**: Smart pointers prevent dangling pointers and memory leaks.
- **Cost**: Smart pointers have overhead (e.g., reference counting in shared_ptr).
- **Example**:
- cpp
- CollapseWrap
- Copy
- `std::unique_ptr<int> uptr = std::make_unique<int>(10); // Auto-deleted`
-
- `std::shared_ptr<int> sptr = std::make_shared<int>(20); // Shared ownership`

5. What is RAII, and how is it implemented in C++?

Answer:RAII (Resource Acquisition Is Initialization) is a C++ idiom where resource management (e.g., memory, file handles) is tied to the lifetime of objects. Resources are acquired in the constructor and released in the destructor, ensuring automatic cleanup.

- **Implementation**:
- Use classes to encapsulate resources.
- Allocate resources in the constructor.
- Release resources in the destructor.
- **Example**:
- cpp

- CollapseWrapRun
- Copy
- class FileHandler {
-
-     FILE* file;
-
- public:
-
-     FileHandler(const char* name) : file(fopen(name, "r")) {}
-
-     ~FileHandler() { if (file) fclose(file); }
-
- };
-
- int main() {
-
-     FileHandler fh("example.txt"); // File opened in constructor, closed in destructor
-
-     return 0; // File automatically closed when fh goes out of scope
-
- }
- **Benefits**: Prevents resource leaks, simplifies error handling, and ensures exception safety.

6. Explain the difference between const and constexpr in C++.

Answer:

- **const**:
- Indicates a variable's value cannot be modified after initialization.
- Can be evaluated at runtime or compile-time, depending on context.
- Example: const int x = computeValue(); (runtime initialization).
- **constexpr**:
- Indicates a variable or function must be evaluated at compile-time.
- Used for performance optimization and compile-time computations.

- Example: constexpr int square(int n) { return n * n; } constexpr int x = square(5); (x = 25 at compile-time).
- **Key Difference**: const ensures immutability, while constexpr ensures compile-time evaluation.

7. What is the role of the volatile keyword in C++?

Answer: The volatile keyword tells the compiler that a variable's value may change unexpectedly (e.g., by hardware or another thread), preventing aggressive optimizations.

- **Usage**:
- Common in embedded systems for memory-mapped I/O.
- Prevents the compiler from caching the variable's value in registers.
- **Example**:
- cpp
- CollapseWrap
- Copy

```cpp
volatile int* statusRegister = (int*)0x1234; // Hardware register

while (*statusRegister) {} // Compiler won't optimize away the read
```

- **Note**: volatile is not a substitute for thread synchronization (use std::atomic for that).

8. What is a vtable, and how does it work in C++?

Answer: A **vtable** (virtual table) is a mechanism used to implement runtime polymorphism for virtual functions.

- **How it Works**:
- For each class with virtual functions, the compiler creates a vtable containing pointers to the virtual functions.
- Each object of the class contains a hidden pointer (vptr) to its class's vtable.
- When a virtual function is called via a base-class pointer/reference, the vptr is used to look up the correct function in the vtable.

- **Overhead**: Adds memory (vptr per object, vtable per class) and runtime cost (indirect function calls).
- **Example**: See the virtual function example above (question 2).

9. What are templates in C++? Explain with an example.

Answer: Templates enable generic programming by allowing functions or classes to work with any data type.

- **Types**:
- **Function Templates**: Generic functions.
- **Class Templates**: Generic classes.
- **Example** (Function Template):
- cpp
- CollapseWrapRun
- Copy

```cpp
template <typename T>

T max(T a, T b) {

    return (a > b) ? a : b;

}

int main() {

    std::cout << max(5, 10) << "\n";        // int: 10

    std::cout << max(3.14, 2.71) << "\n";   // double: 3.14

    return 0;

}
```

- **Benefits**: Code reusability, type safety, and performance (compile-time resolution).
- **Drawbacks**: Complex error messages, increased compile time.

10. What is the difference between std::vector and std::array?

Answer:

- **std::vector**:
- Dynamic array, resizable at runtime.
- Allocates memory on the heap.
- Provides bounds checking in debug mode (e.g., at()).
- Example: std::vector<int> vec = {1, 2, 3}; vec.push_back(4);
- **std::array**:
- Fixed-size array, size defined at compile-time.
- Allocates memory on the stack.
- Provides bounds checking with at().
- Example: std::array<int, 3> arr = {1, 2, 3};
- **Key Difference**: vector is dynamic and flexible, while array is fixed-size and more efficient for small, fixed collections.

11. How does exception handling work in C++?

Answer: Exception handling in C++ uses try, catch, and throw to manage errors.

- **Mechanism**:
- Code that might throw an exception is placed in a try block.
- If an exception is thrown (via throw), control transfers to the matching catch block.
- Uncaught exceptions propagate up the call stack, potentially terminating the program.
- **Example**:
- cpp
- CollapseWrap
- Copy
- `try {`
-
- `    throw std::runtime_error("Error occurred!");`

-
- } catch (const std::runtime_error& e) {
-
-     std::cerr << e.what() << "\n"; // Outputs: Error occurred!
-
- }
- **Best Practices**:
- Use specific exception types.
- Ensure exception safety (e.g., RAII for resource management).
- Avoid throwing in destructors.

12. What is the Rule of Three/Five/Zero in C++?

Answer:

- **Rule of Three** (C++98/03): If a class defines any of the following, it should define all three:
- Destructor
- Copy constructor
- Copy assignment operator
- Reason: To manage resources (e.g., dynamic memory) correctly.
- **Rule of Five** (C++11 onwards): Adds two more for move semantics:
- Move constructor
- Move assignment operator
- Reason: To support efficient resource transfer with move semantics.
- **Rule of Zero**: Design classes to avoid manual resource management by using RAII (e.g., smart pointers, STL containers), letting the compiler generate default implementations.
- **Example** (Rule of Three):
- cpp
- CollapseWrap
- Copy
- class Resource {

```
    int* data;

public:

    Resource() : data(new int) {}

    ~Resource() { delete data; } // Destructor

    Resource(const Resource& other) : data(new int(*other.data)) {} // Copy constructor

    Resource& operator=(const Resource& other) { // Copy assignment

        if (this != &other) {

            delete data;

            data = new int(*other.data);

        }

        return *this;

    }

};
```

13. What is the difference between static and dynamic polymorphism?

Answer:

- **Static Polymorphism** (Compile-time):
- Resolved at compile-time using templates or function overloading.
- Example: Template functions, overloaded functions.
- Benefits: Faster (no runtime overhead), inlineable.
- **Dynamic Polymorphism** (Runtime):

- Resolved at runtime using virtual functions and inheritance.
- Example: Virtual function calls via base-class pointers.
- Benefits: Flexible for runtime object types, but has vtable overhead.
- **Example**:
- cpp
- CollapseWrap
- Copy
- `// Static polymorphism (template)`
-
- `template <typename T>`
-
- `void print(T val) { std::cout << val << "\n"; }`
-
- `// Dynamic polymorphism`
-
- `class Base { virtual void show() { std::cout << "Base\n"; } };`
-
- `class Derived : public Base { void show() override { std::cout << "Derived\n"; } };`

14. Explain move semantics and std::move.

Answer: Move semantics (introduced in C++11) allow transferring resources from one object to another without copying, improving performance.

- **How it Works**:
- A move constructor/assignment operator transfers ownership of resources (e.g., pointers) instead of copying.
- std::move casts an object to an rvalue reference, enabling move semantics.
- **Example**:
- cpp
- CollapseWrapRun
- Copy
- `class MyString {`
-

```
    char* data;

public:

    MyString(const char* str) : data(strdup(str)) {}

    MyString(MyString&& other) noexcept : data(other.data) { other.data = nullptr; } // Move constructor

    ~MyString() { free(data); }

};

int main() {

    MyString s1("Hello");

    MyString s2 = std::move(s1); // Move s1's resources to s2

    // s1 is now in a valid but unspecified state

    return 0;

}
```

- **Benefits**: Reduces unnecessary copying, especially for large objects.

15. What is the difference between std::mutex and std::atomic for thread safety?

Answer:

- **std::mutex**:
- Used for protecting shared resources in multi-threaded code.
- Provides mutual exclusion, ensuring only one thread accesses a critical section.
- Example: std::lock_guard<std::mutex> lock(mtx);
- Use Case: Protecting complex operations or shared data structures.
- **std::atomic**:

- Provides atomic operations (indivisible) for basic types (e.g., int, bool).
- Avoids locks, reducing contention.
- Example: std::atomic<int> counter(0); counter++;
- Use Case: Simple counters or flags in concurrent code.
- **Key Difference**: mutex is for coarse-grained locking, while atomic is for lock-free, fine-grained operations.

16. How would you optimize a C++ program?

Answer: Optimization strategies depend on the context, but common approaches include:

- **Algorithmic Improvements**: Use efficient algorithms/data structures (e.g., replace linear search with binary search).
- **Memory Management**:
- Minimize dynamic allocations using stack-based storage or std::array.
- Use smart pointers to avoid leaks.
- Reserve capacity in std::vector to avoid reallocations.
- **Code-Level Optimizations**:
- Use const and constexpr for compile-time computations.
- Avoid unnecessary copies with move semantics or pass-by-reference.
- Inline small, frequently called functions.
- **Profiling**: Use tools like gprof, Valgrind, or Intel VTune to identify bottlenecks.
- **Multithreading**: Parallelize tasks using std::thread or std::async for CPU-bound work.
- **Compiler Optimizations**: Enable -O2 or -O3 flags, use profile-guided optimization (PGO).

17. What are the differences between public, protected, and private inheritance?

Answer:

- **Public Inheritance**:

- Public members of the base class remain public; protected members remain protected.
- Models an "is-a" relationship (e.g., Dog is-a Animal).
- Example: class Dog : public Animal { ... };
- **Protected Inheritance**:
- Public and protected members of the base class become protected in the derived class.
- Used rarely, typically for implementation inheritance.
- Example: class Dog : protected Animal { ... };
- **Private Inheritance**:
- Public and protected members of the base class become private in the derived class.
- Models a "has-a" relationship (implementation detail, not exposed).
- Example: class Dog : private Animal { ... };
- **Key Difference**: Affects accessibility of base-class members in the derived class and its clients.

18. What is undefined behavior in C++? Give examples.

Answer: Undefined behavior (UB) occurs when a program's behavior is unpredictable due to violating C++ standard rules. The program may crash, produce incorrect results, or appear to work.

- **Examples**:
- Dereferencing a null or dangling pointer: int* p = nullptr; *p = 5;
- Accessing an array out of bounds: int arr[5]; arr[10] = 0;
- Using a variable after its lifetime ends: int& ref = *new int; delete &ref; ref = 5;
- Modifying a const object: const int x = 10; *(int*)&x = 20;
- **Avoiding UB**: Use modern C++ features (smart pointers, bounds-checked containers) and tools like UBSan or static analyzers.

19. Explain the STL and its main components.

Answer: The **Standard Template Library (STL)** is a collection of generic algorithms, containers, and iterators in C++.

- **Main Components**:
- **Containers**: Store data (e.g., std::vector, std::map, std::set, std::list).
- **Algorithms**: Operate on containers (e.g., std::sort, std::find, std::transform).
- **Iterators**: Provide a way to traverse containers (e.g., begin(), end()).
- **Function Objects (Functors)**: Callable objects for algorithms (e.g., std::less).
- **Example**:
- cpp
- CollapseWrap
- Copy

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

std::sort(vec.begin(), vec.end()); // Sort using STL algorithm

for (auto x : vec) std::cout << x << " "; // Outputs: 1 2 5 9
```

- **Benefits**: Reusability, type safety, and performance.

20. How would you implement a thread-safe singleton in C++?

Answer: A thread-safe singleton ensures a single instance of a class across threads. C++11 and later simplifies this with static initialization.

- **Implementation** (Meyers' Singleton, thread-safe in C++11+):
- cpp
- CollapseWrap
- Copy

```
class Singleton {

public:

    static Singleton& getInstance() {
```

```cpp
        static Singleton instance; // Static initialization is thread-safe

        return instance;

    }

    // Delete copy/move operations

    Singleton(const Singleton&) = delete;

    Singleton& operator=(const Singleton&) = delete;

private:

    Singleton() {} // Private constructor

};
```

- **Why Thread-Safe**: C++11 guarantees that static local variables are initialized in a thread-safe manner.
- **Alternative**: Use std::call_once or double-checked locking (with std::atomic) for older C++ versions.

Tips for C++ Interview Success

- **Demonstrate Experience**: Highlight projects where you used C++ (e.g., performance-critical systems, embedded software).
- **Know Modern C++**: Be familiar with C++11/14/17/20 features (e.g., smart pointers, lambdas, auto, ranges).
- **Practice Coding**: Solve problems on platforms like LeetCode or HackerRank using C++.
- **Understand Low-Level Details**: Be ready to discuss memory management, pointers, and vtables.
- **Ask Questions**: Clarify requirements in system design or coding questions to show problem-solving skills.

If you'd like me to expand on any question, provide more examples, or focus on specific C++ topics (e.g., multithreading, memory management, or C++20 features), let me know! I can also generate code snippets or analyze specific scenarios if you provide details.

Observer pattern , singleton pattern

AWS Services :

- services such as Lambda, API Gateway, DynamoDB, Step Functions, ECS/Fargate, and S3.
- RESTful APIs for internal and external service integration.
  microservices multi-tenant SaaS platform.

DynamoDB

1. key-value and document data models.
2. NO SQL – no relational DB
3. Serverless - don't have to manage the underlying infrastructure.

Lambda:

1. serverless computing service.
2. It allows you to run code without servers.
3. You write functions , then Lambda execute them as HTTP request

API Gateway:

1. It handles API requests
2.