C interview questions and answers

1. **What is C?**
    - C is a general-purpose programming language developed by Dennis Ritchie at Bell Labs in 1972. It is widely used for system programming and developing operating systems.
2. **What are the key features of C?**
    - Fast execution, structured programming, rich library support, portability, and flexibility.
3. **What is the difference between C and C++?**
    - C is a procedural language, while C++ supports both procedural and object-oriented programming.
4. **What is a compiler?**
    - A compiler is a program that translates C source code into machine code.
5. **What is the difference between a compiler and an interpreter?**
    - A compiler translates the entire code at once, while an interpreter translates code line by line.

Data Types and Variables

6. **What are the basic data types in C?**
    - `int`, `float`, `char`, `double`, and `void`.
7. **What is the size of an `int`?**
    - Typically 4 bytes, but it depends on the system.
8. **What is the difference between signed and unsigned integers?**
    - Signed integers can store negative values, whereas unsigned integers can only store positive values.
9. **What is a pointer?**
    - A pointer is a variable that stores the memory address of another variable.
10. **What is the difference between `float` and `double`?**
    - `float` has 6-7 decimal precision, while `double` has 15-16 decimal precision.

Operators and Expressions

11. **What are the different types of operators in C?**
    - Arithmetic, relational, logical, bitwise, assignment, and special operators.
12. **What is the modulus operator (%) used for?**
    - It returns the remainder of a division operation.
13. **What is the difference between `=` and `==`?**
    - `=` is an assignment operator, while `==` is a comparison operator.
14. **What is a ternary operator?**
    - `condition ? expression1 : expression2` is a shorthand for `if-else`.
15. **What is the difference between pre-increment and post-increment?**
    - `++i` increments before use, while `i++` increments after use.

Control Flow

16. **What are conditional statements in C?**
    - `if`, `if-else`, `else-if`, and `switch`.

17.What is the syntax of a `switch` statement?

```
switch(expression)
  { case value1:
        //
      code
     break;
  case value2:
        //
      code
     break;
  default:
     // code
}
```

17. **What is the difference between `for`, `while`, and `do-while` loops?**
    - `for`: Initialization, condition, and increment in one line.

- `while`: Checks condition before execution.
- `do-while`: Executes at least once before checking the condition.

20.Write a C program to print numbers from 1 to 10 using a `for` loop.

```
#include
<stdio.h> int
main() {
   for(int i = 1; i <= 10; i++) {
      printf("%d\n", i);
   }
   return 0;
}
```

Functions

18. **What is a function in C?**
    - A function is a block of code designed to perform a specific task.
19. **What are function prototypes?**
    - A function declaration before its definition.
20. **What is recursion?**
    - A function that calls itself.

24.Write a recursive function for factorial.

```
int factorial(int n) {
   if(n == 0) return 1;
   return n * factorial(n - 1);
}
```

21. **What is the difference between `call by value` and `call by reference`?**
    - Call by value passes a copy of the argument, while call by reference passes the memory address.

Pointers and Arrays

22. **What is a NULL pointer?**

- A pointer that does not point to any memory location.

23. **What is a pointer to a pointer?**
    - A pointer that stores the address of another pointer.
24. **What is a dangling pointer?**
    - A pointer that points to a deallocated memory location.

Structures and Unions

25. **What is a structure in C?**
    - A user-defined data type that groups variables.
26. **What is the difference between `struct` and `union`?**
    - `struct` allocates memory for all members, `union` shares memory among members.

Memory Management

27. **What is `malloc()`?**
    - Allocates memory dynamically.
28. **What is `free()`?**
    - Deallocates memory allocated by `malloc()`.

File Handling

29. **What are the file handling functions in C?**
    - `fopen()`, `fclose()`, `fscanf()`, `fprintf()`, `fgets()`, `fputs()`.

Advanced C Concepts

30. **What are macros in C?**
    - Preprocessor directives using `#define`.
31. **What is the use of `typedef`?**
    - Used to define new names for data types.
32. **What is an `enum` in C?**
    - A user-defined data type for constants.
33. **What are header files?**
    - Files that contain function declarations and macros.
34. **What is `volatile` in C?**
    - It prevents compiler optimization for a variable.

Here are the remaining 60 C interview questions along with their answers:

Memory Management (Continued)

35. **What is `calloc()`?**
    - `calloc()` allocates memory for multiple blocks and initializes them to zero.
36. **What is `realloc()`?**
    - `realloc()` changes the size of an allocated memory block.
37. **What happens if `free()` is called twice on the same pointer?**
    - It may cause undefined behavior or program crash.
38. **What is memory leak?**
    - When dynamically allocated memory is not freed, causing memory wastage.
39. **How can you avoid memory leaks in C?**
    - Always free allocated memory using `free()` before losing its reference.

Strings in C

46.How do you declare a string in C?
char str[] = "Hello";

47. **What is `strlen()` used for?**
    - Returns the length of a string.
48. **What is `strcpy()` used for?**
    - Copies one string into another.
49. **What is the difference between `strcat()` and `strncat()`?**
    - `strcat()` appends the full string, `strncat()` appends a limited number of characters.
50. **How do you compare two strings in C?**
    - Using `strcmp(str1, str2)`.

Preprocessor Directives

51. **What is `#define` in C?**
    - Used to define macros.
52. **What is `#include`?**
    - Used to include header files.
53. **What is the difference between `#include <filename>` and `#include "filename"`?**
    - `<filename>` searches in standard directories, `"filename"` searches in the current directory first.
54. **What is `#ifdef` used for?**
    - Checks if a macro is defined.
55. **What is the purpose of `#pragma`?**
    - Used for compiler-specific instructions.

Bitwise Operators

56. **What are bitwise operators in C?**
    - `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), `<<` (left shift), `>>` (right shift).
57. **What does `x << 1` do?**
    - Multiplies `x` by 2.
58. **What does `x >> 1` do?**
    - Divides `x` by 2.
59. **What is bit masking?**
    - Using bitwise operations to set, clear, or toggle specific bits.

60.Write a program to check if a number is even or odd using bitwise operators.

```
#include
<stdio.h> int
main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    if (num & 1)
        printf("Odd\n");
    else
        printf("Even\n");
    return 0;
}
```

Structures and Unions (Continued)

61.How do you declare a structure?

```
struct Student {
```

```
    char
    name[50]; int
    age;
};
```

62.How do you access structure members?

- Using the dot operator (`student.age`).

63.How do you pass a structure to a function?

```
void display(struct Student s) { printf("%s %d", s.name, s.age); }
```

64.How do you allocate memory dynamically for a structure?

```
struct Student *s = (struct Student*)malloc(sizeof(struct Student));
```

65. **What is the difference between structure and class (in C++)?**
    - Structures have public access by default, whereas classes have private access by default.

File Handling (Continued)

What is the syntax of `fopen()`?

```
FILE *fp = fopen("file.txt", "r");
```

66. **What are different file modes in C?**
    - `"r"`, `"w"`, `"a"`, `"r+"`, `"w+"`, `"a+"`.
67. **How do you read a file in C?**
    - Using `fscanf()`, `fgets()`, `fgetc()`.
68. **How do you write to a file in C?**
    - Using `fprintf()`, `fputs()`, `fputc()`.

70.How do you close a file?

```
fclose(fp)
```

Advanced C Concepts

71. **What is an `enum`?**

```
enum Days { MON, TUE, WED };
```

72. **What is a function pointer?**

```
void
    (*func_ptr)(int);
```

73. **What is a volatile variable?**
    - Prevents compiler optimizations.
74. **What is a static variable?**
    - Retains its value across function calls.
75. **What is the `const` keyword?**
    - Declares a variable as read-only.

Common Mistakes in C

76. **What happens if a pointer is not initialized?**
    - It may cause undefined behavior.
77. **What is an off-by-one error?**
    - A common error in loops or array indexing.
78. **What happens if you access an array out of bounds?**
    - May cause segmentation fault.
79. **What happens if a function is declared but not defined?**
    - Linker error.
80. **What happens if `return` is missing in a non-void function?**
    - Undefined behavior.

Multithreading and Concurrency

81. **Does C support multithreading natively?**
    - No, but we can use `pthread` library.

82.How do you create a thread in C?
pthread_create(&thread, NULL, function, NULL);

83. **What is a mutex?**
    - A lock mechanism to prevent race conditions.
84. **What is a race condition?**
    - When multiple threads access shared data incorrectly.
85. **What is deadlock?**
    - When two or more threads are stuck waiting for each other.

Debugging and Optimization

86. **What is `gdb`?**
    - A debugger for C programs.
87. **How do you use `printf()` for debugging?**
    - Print variable values during execution.
88. **What is `valgrind`?**
    - A tool for memory leak detection.
89. **How do you optimize C code?**
    - Using efficient algorithms, compiler optimizations, and reducing memory usage.
90. **What is the purpose of `inline` functions?**
    - Reduces function call overhead.

Miscellaneous Questions

91. **What is the output of `printf("%d", sizeof(int));`?**
    - 4 (on most systems).
92. **What is an lvalue and an rvalue?**
    - lvalue: Can be assigned to.
    - rvalue: Cannot be assigned to.
93. **How do you implement a stack in C?**
    - Using arrays or linked lists.
94. **What is a segmentation fault?**
    - Accessing invalid memory.
95. **How do you reverse a string in C?**
    - Using a loop or recursion.

96. Factorial of a Number

```
#include <stdio.h>


long long factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}


int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printf("Factorial of %d is %lld\n", num, factorial(num));
    return 0;
}
```

✅ **Explanation:**

- Uses recursion to calculate factorial (`n! = n * (n-1)!`).
- If `n == 0`, it returns 1 (base case).
- Otherwise, it keeps calling itself with `n-1` until reaching 0.

97. Fibonacci Series

```
#include <stdio.h>


void fibonacci(int n) {
    int a = 0, b = 1,
    next;
    printf("Fibonacci Series: %d %d ", a, b);
    for (int i = 2; i < n; i++) {
        next = a + b;
        printf("%d ", next);
        a = b;
        b = next;
    }
}


int main() {
    int n;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    fibonacci(n);
    return 0;
}
```

✅ **Explanation:**

- Uses iteration to generate `n` Fibonacci numbers.
- Starts with `0` and `1`, then calculates `next = a + b`.
- Updates `a` and `b` in each iteration.

---

98. Palindrome Number Check

```
#include <stdio.h>


int isPalindrome(int num) {
    int rev = 0, original = num, remainder;
    while (num > 0) {
        remainder = num % 10;
        rev = rev * 10 +
        remainder; num /= 10;
    }
    return original == rev;
}


int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    if (isPalindrome(num))
        printf("%d is a Palindrome\n", num);
    else
        printf("%d is not a Palindrome\n", num);
    return 0;
```

```
}
```

✅ **Explanation:**

- Reverses the number and checks if the original and reversed numbers are the same.
- Uses a loop to extract digits and build the reversed number.

---

99. Swapping Two Numbers (Using a Third Variable)

```
#include <stdio.h>


int main() {
    int a, b, temp;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);

    temp = a;
    a = b;
    b = temp;

    printf("After swap: a = %d, b = %d\n", a, b);
    return 0;
}
```


✅ **Explanation:**

- Uses a temporary variable to swap values.
- `temp` stores `a`, then `a = b` and `b = temp`.

---

100. Swapping Two Numbers (Without Using a Third Variable)

```
#include <stdio.h>

int main() {
    int a, b;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);

    a = a + b;
    b = a - b;
    a = a - b;

    printf("After swap: a = %d, b = %d\n", a, b);
    return 0;
}
```


✅ **Explanation:**

- Uses arithmetic operations to swap values without extra memory.
- Addition and subtraction are used to swap values logically.

C++ interview questions and answers

1. **What is C++?**
    - C++ is a general-purpose, object-oriented programming language that extends C with additional features such as classes, objects, and polymorphism.
2. **What are the key features of C++?**
    - Object-oriented, platform-independent, rich standard library, memory management, and strong type checking.

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

    - C is procedural, while C++ supports object-oriented programming. C++ has features like classes, polymorphism, and exception handling.

4. **What is a class in C++?**

    - A class is a blueprint for creating objects. It defines data members and member functions.

5. **What is an object in C++?**

    - An object is an instance of a class.

6. **What are access specifiers in C++?**

    - Public, private, and protected.

7. **What is a constructor?**

    - A constructor is a special function that initializes an object when it is created.

8. **What is a destructor?**

    - A destructor is a special function that is automatically called when an object goes out of scope.

9. **What is function overloading?**

    - Function overloading allows multiple functions with the same name but different parameters.

10. **What is operator overloading?**

    - Operator overloading allows defining new meanings for existing operators.

11. **What is inheritance in C++?**

    - Inheritance allows a class (child) to derive properties from another class (parent).

12. **What is polymorphism?**

    - Polymorphism allows functions to behave differently based on the object calling them.

13. **What is encapsulation?**

    - Encapsulation binds data and functions into a single unit.

14. **What is abstraction?**

    - Abstraction hides implementation details from the user.

15. **What is a virtual function?**

    - A virtual function is a function in a base class that can be overridden in a derived class.

16. **What is pure virtual function?**

    - A pure virtual function has `= 0` in its declaration and forces derived classes to implement it.

17. **What is an abstract class?**

    - A class with at least one pure virtual function.

18. **What is multiple inheritance?**

    - Multiple inheritance allows a class to inherit from more than one base class.

19. **What is the difference between struct and class in C++?**

    - `struct` members are public by default, while `class` members are private by default.

20. **What is a reference variable in C++?**

    - A reference variable is an alias for another variable.

21. **What is the ‘this’ pointer?**

- The `this` pointer refers to the calling object.

22. **What is the difference between `new` and `malloc`?**

- `new` initializes objects, while `malloc` does not.

23. **What is a static member function?**

- A static function belongs to the class, not an object.

24. **What is a namespace in C++?**

- A namespace prevents naming conflicts.

25. **What is the difference between `endl` and `\n`?**

- `endl` flushes the output buffer, while `\n` does not.

Intermediate Level (26-50)

26. **What is a friend function?**

- A function that can access private members of a class.

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

- Deep copy duplicates dynamically allocated memory; shallow copy only copies pointers.

28. **What is an inline function?**

- An inline function is expanded in place to reduce function call overhead.

29. **What is a copy constructor?**

- A copy constructor initializes an object using another object of the same class.

30. **What is the difference between `const int *ptr` and `int *const ptr`?**

    - `const int *ptr` means the value is constant; `int *const ptr` means the pointer is constant.

31. **What are function pointers?**

    - Function pointers store addresses of functions.

32. **What is exception handling?**

    -

    Handling runtime errors using `try`, `catch`, and `throw`.

33. **What is RAII (Resource Acquisition Is Initialization)?**

    - A technique where resources are allocated in constructors and released in destructors.

34. **What is a template in C++?**

    - A template allows writing generic code for multiple data types.

35. **What are smart pointers?**

    - Smart pointers manage dynamic memory automatically.

36. **What is the Standard Template Library (STL)?**

    - A collection of classes and functions for data structures and algorithms.

37. **What are iterators in C++?**

    - Iterators provide a way to traverse STL containers.

38. **What is `std::vector`?**

    - A dynamic array implementation in STL.

39. **What is the difference between `map` and `unordered_map`?**

- `map` is ordered (RB Tree), while `unordered_map` is unordered (Hash Table).

40. **What is `std::pair` in C++?**

    - A pair stores two values of different types.

41. **What is lambda expression?**

    - A lambda is an anonymous function.

42. **What is `std::unique_ptr`?**

    - A smart pointer for unique ownership.

43. **What is `std::shared_ptr`?**

    - A smart pointer for shared ownership.

44. **What is `std::weak_ptr`?**

    - A weak reference to avoid circular dependencies.

45. **What is `std::move`?**

    - `std::move` transfers ownership of resources.

46. **What is move semantics?**

    - Move semantics allow efficient resource transfer.

47. **What is the difference between `++i` and `i++`?**

    - `++i` increments before use; `i++` increments after use.

48. **What is `volatile` keyword?**

    - `volatile` tells the compiler not to optimize variable access.

49. **What is memory leak?**

- A memory leak occurs when allocated memory is not deallocated.

50. **What is `delete` operator in C++?**

- It deallocates memory allocated using `new`.

51. Smart Pointers

❓ What is `std::unique_ptr`? How do you use it?

✅ Answer: `std::unique_ptr` ensures exclusive ownership of dynamically allocated

objects.

```
#include <iostream>
#include <memory>

class Example {
public:
    Example() { std::cout << "Constructor\n"; }
    ~Example() { std::cout << "Destructor\n"; }
};

int main() {
    std::unique_ptr<Example> ptr = std::make_unique<Example>(); return 0;
}
```

52. `std::shared_ptr` Usage

❓ What is `std::shared_ptr`? Demonstrate usage.

✅ Answer: `std::shared_ptr` allows multiple owners of a single object. #include

<iostream>

#include <memory>

class Example {
public:

Example() { std::cout << "Constructor\n"; }

~Example() { std::cout << "Destructor\n"; }
};

int main() {

std::shared_ptr<Example> ptr1 = std::make_shared<Example>();
std::shared_ptr<Example> ptr2 = ptr1;

return 0;

}

53. `std::weak_ptr` and Circular References

❓ Why use `std::weak_ptr`?

✅ Answer: Prevents circular references in `std::shared_ptr`.

#include <iostream>
#include <memory>

class A { public:

std::shared_ptr<A> self;

```
    ~A() { std::cout << "Destructor called\n"; }
};


int main() {


    std::shared_ptr<A> obj = std::make_shared<A>();
    obj->self = obj; // Circular reference leads to memory leak
}
```

👉 Fix: Use `std::weak_ptr`.

54. Custom Deleter in Smart Pointer

```
#include <iostream>
#include <memory>


struct Free {

    void operator()(int* ptr) {
        std::cout << "Custom Deleter called\n";
        delete ptr;

    }

};


int main() {
    std::unique_ptr<int, Free> ptr(new int(42));
    return 0;

}
```

55. Implement a Singleton Pattern

```
#include <iostream>


class Singleton {
public:
        static Singleton& getInstance() {
            static Singleton instance;
            return instance;

    }

    void show() { std::cout << "Singleton Instance\n"; }
private:

    Singleton() {}

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


    int main() {
        Singleton::getInstance().show();

}
```

56. Using `std::future` and `std::async`

```
#include <iostream>
#include <future>


    int compute()
        { return 42;

}


int main() {

    std::future<int> f = std::async(std::launch::async, compute); std::cout <<
    "Result: " << f.get() << "\n";

}
```

57. Implementing a Thread-safe Singleton

```
#include <iostream>
#include <mutex>


class ThreadSafeSingleton { public:
        static ThreadSafeSingleton& getInstance() {
            static ThreadSafeSingleton instance;
            return instance;

    }

private:
    ThreadSafeSingleton() = default;
    ThreadSafeSingleton(const ThreadSafeSingleton&) = delete; ThreadSafeSingleton&
    operator=(const ThreadSafeSingleton&) = delete;

};



int main() {
    ThreadSafeSingleton& obj = ThreadSafeSingleton::getInstance();
}
```

58. Implementing RAII for File Handling

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


    class FileHandler {
        std::ofstream
        file;
```

```
public:

    FileHandler(const std::string& filename) {
        file.open(filename);

    }

    ~FileHandler(
        ) {
        file.close();

    }

};


int main() {

    FileHandler fh("test.txt");
}
```

59. Using `std::variant` for Type Safety

```
#include <iostream>
#include <variant>


int main() {

    std::variant<int, double, std::string> var; var =
    "Hello";

    std::cout << std::get<std::string>(var) << "\n";

}
```

60. Implementing Producer-Consumer using `std::condition_variable`

```
#include <iostream>
#include <thread>
#include <queue>

#include <condition_variable>



std::queue<int> q;
std::mutex mtx;
std::condition_variable cv;
bool done = false;


void producer() {

        for (int i = 0; i < 5; ++i) {
            std::unique_lock<std::mutex> lock(mtx);
            q.push(i);

        cv.notify_one();

    }
    done = true;
    cv.notify_all();


}


    void consumer() {
        while (true) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [] { return !q.empty() || done; }); if
        (!q.empty()) {
            std::cout << "Consumed: " << q.front() << "\n";
            q.pop();
            } else if (done)
                { break;

        }

    }
}


int main() {

    std::thread t1(producer);
    std::thread t2(consumer);
```

```
    t1.join();

    t2.join();

}
```

61.
Using `std::transform` for Function Application

```
#include <iostream>
#include <vector>
#include <algorithm>


int main() {

    std::vector<int> v = {1, 2, 3, 4};
    std::transform(v.begin(), v.end(), v.begin(), [](int x) { return x * x; }); for (int
    n : v) std::cout << n << " ";

}
```

62. Implementing a Custom Allocator

```
#include <iostream>
#include <memory>


template <typename T>
struct CustomAllocator {

    T* allocate(size_t n) {

        return static_cast<T*>(::operator new(n * sizeof(T)));
    }
```

```
    void deallocate(T* p, size_t) {

        ::operator delete(p);
    }
};


int main() {

    CustomAllocator<int> allocator;
    int* ptr = allocator.allocate(5);
    allocator.deallocate(ptr, 5);

}
```

63. Using `std::optional` to Handle Nullable Values

```
#include <iostream>
#include <optional>


    std::optional<int> findValue(bool found) { if
        (found) return 42;

    return std::nullopt;

}


int main() {

    auto val = findValue(true);
    if (val) std::cout << "Value: " << val.value() << "\n";
}
```

Here are 64-100 advanced C++ interview questions with answers, covering STL, memory management, multithreading, smart pointers, design patterns, templates, and more.

📌 Memory Management & Pointers

64. What is memory alignment in C++?

✅ Answer: Memory alignment ensures that variables are stored in memory at addresses that are multiples of their size, improving CPU efficiency.
#include <iostream>

struct Aligned {

    char a; // 1 byte

    int b; // 4 bytes
    double c; // 8 bytes

}; // Struct size will be 16 due to padding.

int main() {
    std::cout << "Size of Aligned: " << sizeof(Aligned) << std::endl;
}

65. What is placement new?

✅ Answer: Placement `new` allows constructing an object in pre-allocated memory. #include

<iostream>

int main() {
    char buffer[sizeof(int)];
    int* p = new (buffer) int(42); // Placement new
    std::cout << *p << std::endl;

}

66.

What is the difference between `new` and `malloc`?

✅ **Answer:**

- `new` **calls the constructor, while** `malloc` **does not.**
- `new` **returns the correct type,** `malloc` **returns** `void*`**.**

📌 Advanced Object-Oriented Programming (OOP)

67. What is slicing in C++?

✅ Answer: Object slicing happens when a derived class object is assigned to a base class, losing derived-specific data.

```
#include <iostream>


class Base { public: int x = 10; };

class Derived : public Base { public: int y = 20; };


   int main() {
      Derived
      d;

   Base b = d; // Object slicing: `b` loses `y`

}
```

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

✅ **Answer:**

- **Static Polymorphism: Function overloading, operator overloading, templates.**
- **Dynamic Polymorphism: Virtual functions, runtime method overriding.**

📌 Multithreading & Concurrency

69. How does `std::mutex` prevent race conditions?

✅ Answer: `std::mutex` ensures only one thread accesses shared data at a time.

```
#include <iostream>

#include <thread>
#include <mutex>


std::mutex mtx;
int counter = 0;


    void increment() {
        std::lock_guard<std::mutex> lock(mtx);
        counter++;

}


int main() {

    std::thread     t1(increment);
    std::thread     t2(increment);
    t1.join();

    t2.join();

    std::cout << "Counter: " << counter << std::endl;
}
```

70. What is `std::atomic` and how does it work?

✅ Answer: `std::atomic` ensures atomic operations without using locks. #include

<iostream>

#include <atomic>

std::atomic<int> count(0);

void increment() {
    count.fetch_add(1, std::memory_order_relaxed);
}

    int main() {
        increment();

    std::cout << "Count: " << count.load() << std::endl;

}

71. What is `std::condition_variable`?

✅ Answer: `std::condition_variable` allows a thread to wait for a condition. #include

<iostream>

#include <thread>
#include <mutex>

#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

    void waitForEvent() {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [] { return ready; });

    std::cout << "Event received!" << std::endl;

}

```
    void signalEvent() { std::this_thread::sleep_for(std::chrono::seconds(1));
        std::lock_guard<std::mutex> lock(mtx);
    ready = true;
    cv.notify_one();

}


int main() {

    std::thread t1(waitForEvent);
    std::thread t2(signalEvent);
    t1.join();

    t2.join();

}
```

📌 STL & Advanced Data Structures

72. What is `std::map` and how is it implemented?

✅ Answer: `std::map` is implemented as a Red-Black Tree (self-balancing BST). #include

<iostream>

#include <map>

int main() {

    std::map<int, std::string> m; m[1]
    = "One";

    m[2] = "Two";

    for (auto& p : m)
        std::cout << p.first << ": " << p.second << std::endl;
}

73. What is `std::unordered_map` and how is it implemented?

✅ Answer: `std::unordered_map` is implemented using a Hash Table, giving O(1)

average time complexity for lookups.

```
#include <iostream> #include
<unordered_map>

int main() {
    std::unordered_map<int, std::string> um; um[1]
    = "One";
    um[2] = "Two"; for
    (auto& p : um)
        std::cout << p.first << ": " << p.second << std::endl;
}
```

📌 Design Patterns

74. Implement a Factory Pattern in C++

```
#include <iostream>

class Animal { public:
    virtual void speak() = 0;
};

class Dog : public Animal {
public:
    void speak() override { std::cout << "Bark!" << std::endl; }
};

class AnimalFactory { public:
    static Animal* createAnimal() { return new Dog(); }
};

int main() {
    Animal* animal = AnimalFactory::createAnimal();
```

```
    animal->speak();

    delete animal;

}
```

75. Implement the Observer Pattern in C++

```
#include <iostream>
#include <vector>


class Observer {
public:

    virtual void notify() = 0;

};


    class Subject {
        std::vector<Observer*> observers;

public:

    void addObserver(Observer* obs) { observers.push_back(obs); } void
    notifyAll() {

        for (auto obs : observers) obs->notify();

    }
};


class ConcreteObserver : public Observer { public:

    void notify() override { std::cout << "Notified!" << std::endl; }

};


    int main() {
        Subject
        subject;
    ConcreteObserver obs;
    subject.addObserver(&obs); subject.notifyAll();


}
```

📌 Advanced C++ Features

76. What is `std::any` and how is it used?

✅ Answer: `std::any` can hold any data type.

```
#include <iostream>

#include <any>


int main() {

    std::any data = 42;
    std::cout << std::any_cast<int>(data) << std::endl;
}
```

77.

What is `std::variant` and why use it?

✅ Answer: `std::variant` is a type-safe union. #include

```
<iostream>

#include <variant>


int main() {
    std::variant<int, double, std::string> var = "Hello";
    std::cout << std::get<std::string>(var) << std::endl;

}
```

Here are 23 advanced C++ interview questions and answers (78-100) covering multithreading, memory management, STL, design patterns, and modern C++ features.

🔹 78. What is the difference between `std::function` and function pointers?

✅ **Answer:**

- `std::function` **is a wrapper for callable objects, including lambda expressions, function pointers, and functors.**
- **Function pointers only store addresses of functions.**

```
#include <iostream>
#include <functional>
void func(int x) { std::cout << "Function: " << x << "\n"; } int

main() {

    std::function<void(int)> f = func; // std::function

    void (*ptr)(int) = func;          // Function pointer

    f(10);

    ptr(20);
}
```

🔹 79. What is `std::bind`, and how does it work?

✅ **Answer:**

- `std::bind` **binds function arguments and creates a callable object.**

```
#include <iostream>

#include <functional>

void multiply(int a, int b) { std::cout << "Result: " << a * b << "\n"; } int

main() {
```

```
    auto boundFunc = std::bind(multiply, 10, std::placeholders::_1);
    boundFunc(5); // Equivalent to multiply(10, 5);

}
```

🔹 80. How does `std::visit` work with `std::variant`?

✅ **Answer:**

- `std::visit` **is used to apply a visitor function to**

`std::variant`**. #include <iostream>**

```
#include <variant>


int main() {

    std::variant<int, double, std::string> var = 10;


    std::visit([](auto&& val) { std::cout << "Value: " << val << "\n"; }, var);

}
```

🔹 81. What is the CRTP (Curiously Recurring Template Pattern)?

✅ **Answer:**

- **CRTP is used to achieve static**

polymorphism. #include <iostream>

```
template <typename T>
class Base {

public:

    void interface() { static_cast<T*>(this)->implementation(); }
};


class Derived : public Base<Derived> {
public:
```

```
    void implementation() { std::cout << "Derived implementation\n"; }
};


    int main() {
        Derived d;
        d.interface();
}
```

🔹 82. What is `std::invoke` in C++?

✅ **Answer:**

- `std::invoke` **calls functions, function objects, or member**

functions. #include <iostream>

```
#include <functional>


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


    int main() {
        Foo obj;
    auto result = std::invoke(&Foo::add, obj, 5, 3); std::cout
    << "Result: " << result << "\n";
}
```

🔹 83. How do you implement a thread pool in C++?

✅ **Answer:**

- **Use** `std::thread`, `std::mutex`,

`std::condition_variable`. **#include <iostream>**

```
#include <thread>

#include <vector>
#include <queue>
```

```
#include <functional>

#include <condition_variable>



   class ThreadPool { std::vector<std::thread>
      workers;
      std::queue<std::function<void()>> tasks;
      std::mutex queue_mutex;
      std::condition_variable condition;

   bool stop = false;



public:
   ThreadPool(size_t threads);
   void enqueue(std::function<void()> task);
   ~ThreadPool();
};


   ThreadPool::ThreadPool(size_t threads) {
      for (size_t i = 0; i < threads; ++i) {
          workers.emplace_back([this]
             { while (true) {

             std::function<void()> task;

             {
                std::unique_lock<std::mutex> lock(queue_mutex);
                condition.wait(lock, [this] { return stop || !tasks.empty(); }); if
                (stop && tasks.empty()) return;
                task = std::move(tasks.front());
                tasks.pop();

             }

             task();
          }
      });
   }
}


void ThreadPool::enqueue(std::function<void()> task) {

   {
      std::unique_lock<std::mutex> lock(queue_mutex);
      tasks.emplace(std::move(task));


   }
   condition.notify_one();
}
```

```
ThreadPool::~ThreadPool() {
    {
        std::unique_lock<std::mutex> lock(queue_mutex);
        stop = true;
    }
    condition.notify_all();
        for (std::thread &worker : workers)
            worker.join();
}


    int main() {
        ThreadPool
        pool(4);
    pool.enqueue([] { std::cout << "Task executed\n"; });
}
```

🔹 84. What is the difference between `std::mutex`, `std::recursive_mutex`, and `std::shared_mutex`?

✅ **Answer:**

- `std::mutex`: **Basic lock mechanism.**
- `std::recursive_mutex`: **Allows reentrant locking by the same thread.**
- `std::shared_mutex`: **Allows multiple readers but only one writer.**

🔹 85. How does `std::atomic` ensure thread safety?

✅ **Answer:**

- `std::atomic` **provides atomic operations to prevent data**

races. #include <iostream>

```
#include <atomic>

#include <thread>

std::atomic<int> counter(0);


void increment() {

    for (int i = 0; i < 1000; ++i) counter++;
}

int main() {
    std::thread t1(increment), t2(increment);
    t1.join();

    t2.join();

    std::cout << "Counter: " << counter << "\n";
}
```

🔹 86. What is `std::scoped_lock` in C++17?

✅ **Answer:**

- **A safer alternative to `std::lock_guard` for handling multiple**

mutexes. #include <iostream>

```
#include <mutex>

std::mutex m1, m2;

void safe_function() {

    std::scoped_lock lock(m1, m2); std::cout
    << "Thread safe execution\n";

}
```

🔹 87. Explain how `std::forward` works with perfect

forwarding.

✅ **Answer:**

- `std::forward` **preserves value category when forwarding function**

arguments. #include <iostream>

```
#include <utility>


template <typename T> void
wrapper(T&& arg) {

   process(std::forward<T>(arg));

}
```

🔹 88. Explain Copy Elision in C++.

✅ **Answer:**

- **Optimizes object copying by eliminating unnecessary constructor calls.**

```
struct Test {
   Test() { std::cout << "Constructor\n"; }
   Test(const Test&) { std::cout << "Copy Constructor\n"; }
};


   Test create() {
      return Test();

}


int main() {

   Test obj = create();
}
```

🔹 89. What is `std::optional` and when to use it?

✅ **Answer:**

- **Use `std::optional` to represent missing values instead of using `nullptr`.**

Here are 10 advanced C++ interview questions and answers (90-100), covering metaprogramming, STL, multithreading, design patterns, memory management, and modern C++ features.

90. What is Expression Templates in C++?

✅ Answer: Expression templates enable lazy evaluation and eliminate unnecessary temporary objects in operations like matrix manipulation or vector arithmetic.

👉 **Example:**

```
#include <iostream>
#include <vector>


template <typename L, typename R>
class Add {
   const L& lhs; const
   R& rhs;

public:

   Add(const L& l, const R& r) : lhs(l), rhs(r) {}
   auto operator[](size_t i) const { return lhs[i] + rhs[i]; }
};

template <typename L, typename R>
   auto operator+(const L& lhs, const R& rhs) {
      return Add<L, R>(lhs, rhs);

}



int main() {
   std::vector<int> a = {1, 2, 3}, b = {4, 5, 6};
   auto result = a + b; // No temporary vector!
   std::cout << result[0] << " " << result[1] << " " << result[2] << "\n";
}
```

91. What are Compile-time and Runtime Polymorphism?

✅ **Answer:**

- **Compile-time: Achieved using function overloading, operator overloading, and templates.**
- **Runtime: Achieved using virtual functions and dynamic dispatch.**

👉 **Example:**

```
#include <iostream>

class Base {

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

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

int main() {
    Base* obj = new Derived();
    obj->show(); // Runtime Polymorphism
    delete obj;
}
```

92. What is Type Erasure in C++?

✅ Answer: Type erasure removes type-specific details, allowing polymorphic behavior without inheritance.

👉 Example Using `std::function`

```
#include <iostream>
#include <functional>
void hello() { std::cout << "Hello World\n"; } int

main() {


    std::function<void()> func = hello; // Type erased function

    func();
}
```

Key Concept: `std::function<void()>` can hold any callable entity.

93. How Does `std::any` Work?

✅ Answer: `std::any` stores any type but requires explicit casting.

👉 **Example:**

```
#include <iostream>
#include <any>


int main() {

    std::any data = 42;
    std::cout << std::any_cast<int>(data) << "\n";
}
```

Key Concept: `std::any_cast<T>` retrieves stored data.

94. Explain `std::variant` and How It Differs from `std::any`

✅ Answer: `std::variant` holds one type at a time (like a type-safe union).

👉 **Example:**

```
#include <iostream>
#include <variant>

int main() {
    std::variant<int, std::string> v = "Hello"; std::cout
    << std::get<std::string>(v) << "\n";

}
```

Key Concept: Use `std::get<T>()` to retrieve the active type.

95. What is `std::monostate` in `std::variant`?

✅ Answer: `std::monostate` is a default type when `std::variant` may be empty.

👉 **Example:**

```
#include <iostream>
#include <variant>
std::variant<std::monostate, int, std::string> v; int
```

```
main() {


    if (std::holds_alternative<std::monostate>(v))

        std::cout << "Variant is empty\n";
}
```

Key Concept: Helps when default-initializing a `std::variant`.

96. What is the Curiously Recurring Template Pattern (CRTP)?

✅ Answer: CRTP allows static polymorphism, avoiding virtual function overhead.

👉 **Example:**

```
#include <iostream>


template <typename Derived> class
Base {

public:

    void interface() {
        static_cast<Derived*>(this)->implementation();
    }
};


class Derived : public Base<Derived> {
public:

    void implementation() { std::cout << "Derived class method\n"; }

};

    int main() {
        Derived
```

```
    d;

    d.interface(); // Calls Derived::implementation()
}
```

Key Concept: Simulates polymorphism at compile-time.

97. What is Placement `new`? Why Use It?

✅ Answer: Placement `new` constructs an object at a specific memory location.

👉 **Example:**

```
#include <iostream>

int main() {

    char buffer[sizeof(int)];

    int* p = new (buffer) int(42); // Placement new std::cout
    << *p << "\n";

}
```

🚀 Key Concept: Avoids dynamic memory allocation.

98. What is `std::launder` in C++17?

✅ Answer: `std::launder` helps access memory safely after `placement new`.

👉 **Example:**

```
#include <iostream>
#include <new>
struct A { int x; };

int main() {

    alignas(A) char buffer[sizeof(A)];

    A* ptr = new (buffer) A{10};
    A* safe_ptr = std::launder(ptr); // Safe access std::cout <<
    safe_ptr->x << "\n";
}
```

🚀 Key Concept: Prevents undefined behavior in memory management.

99. What is `std::span`? How is It Better Than Raw Arrays?

✅ Answer: `std::span` is a lightweight view over contiguous data.

👉 **Example:**

```
#include <iostream>
#include <span>

void print(std::span<int> arr) {

    for (int i : arr) std::cout << i << " ";
}

int main() {

    int data[] = {1, 2, 3, 4};
    print(data); // No need to pass size
}
```

🚀 Key Concept: `std::span` avoids pointer decay issues.

100. What is `std::forward_list`? How is It Different from `std::list`?

✅ Answer: `std::forward_list` is a singly linked list, using less memory than `std::list`.

👉 **Example:**

```
#include <iostream>
#include <forward_list>


int main() {

    std::forward_list<int> fl = {1, 2, 3};
    fl.push_front(0); // Efficient insertion for
    (int n : fl) std::cout << n << " ";

}
```