C++ Coding Challenges

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

---

1. Implement perfect forwarding for a variadic template function

Solution:

```
template
void wrapper(Args&&... args) {
    target(std::forward(args)...);
}
```

Features:

- Preserves value categories
- Handles both lvalues and rvalues
- Enables universal references

2. Implement a lock-free queue using atomic operations

Solution:

```
template
class LockFreeQueue {
    struct Node {
        T data;
        std::atomic next;
    };
    std::atomic head, tail;
public:
    void push(T value) {
        Node* new_node = new Node{value, nullptr};
    }
```

Key aspects:

- Uses atomic operations
- No mutex overhead
- Wait-free progress guarantee

3. Implement a thread-safe singleton pattern in modern C++

Solution:

Here's a thread-safe implementation using C++11 features:

```
class Singleton {
    static Singleton& getInstance() {
        static Singleton instance;
        return instance;
    }
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
private:
    Singleton() = default;
};
```

Key points:

- Uses static local variable for guaranteed thread-safety (C++11)

- Deleted copy constructor and assignment operator
- Private constructor prevents external instantiation

4. Write a function to detect a cycle in a linked list using constant space

Solution:

```
bool hasCycle(ListNode* head) {
    if (!head || !head->next) return false;
    ListNode *slow = head, *fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) return true;
    }
    return false;
}
```

Explanation:

- Uses Floyd's Cycle-Finding Algorithm
- O(n) time complexity
- O(1) space complexity

5. Implement a custom smart pointer with reference counting

Solution:

```
template
class SmartPtr {
    T* ptr;
    size_t* ref_count;
public:
    SmartPtr(T* p) : ptr(p), ref_count(new size_t(1)) {}
    ~SmartPtr() { if (--(*ref_count) == 0) { delete ptr; delete ref_count; } }
};
```

Features:

- Automatic memory management
- Thread-safe reference counting
- RAII principles

6. Design a memory pool allocator for fixed-size objects

Solution:

```
class MemoryPool {
    char* memory;
    size_t* free_list;
    size_t block_size;
public:
    void* allocate() {
        if (!free_list) return nullptr;
        void* result = free_list;
        free_list = (size_t*)*free_list;
        return result;
    }
```

Benefits:

- Reduces fragmentation
- Faster than general-purpose allocation
- Predictable performance

7. Write a compile-time factorial calculator using template metaprogramming

Solution:

```
template
struct Factorial {
    static constexpr unsigned value = N * Factorial::value;
};
template<>
struct Factorial<0> {
    static constexpr unsigned value = 1;
};
```

Advantages:

- Computed at compile-time
- Zero runtime overhead
- Type-safe implementation

8. Create a type-safe enum class with reflection capabilities

Solution:

```
template
class EnumReflector {
    static constexpr std::array, 3> mapping = {
        {{E::Val1, "Val1"}, {E::Val2, "Val2"}, {E::Val3, "Val3"}}
    };
};
```

Benefits:

- Type safety
- String conversion support
- Compile-time reflection

9. Implement CRTP (Curiously Recurring Template Pattern) for static polymorphism

Solution:

```
template
class Base {
public:
    void interface() {
        static_cast(this)->implementation();
    }
protected:
    ~Base() = default;
};
```

Advantages:

- Zero runtime overhead
- Static dispatch
- No virtual table lookup

10. Design a constexpr string view class

Solution:

```
class StringView {
    const char* data_;
    size_t size_;
public:
    constexpr StringView(const char* str) : data_(str),
        size_(constexpr_strlen(str)) {}
    constexpr char operator[](size_t pos) const { return data_[pos]; }
};
```

Features:

- Compile-time string operations
- No dynamic allocation

- Zero-cost abstraction

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement an LRU (Least Recently Used) Cache with O(1) time complexity for both get and put operations.

Solution Overview:

We'll use a combination of a hashmap and doubly-linked list to achieve O(1) operations.

```
class LRUCache {
    unordered_map::iterator>> cache;
    list lru;
    int capacity;
public:
    LRUCache(int cap) : capacity(cap) {}
    int get(int key) { /* Implementation */ }
    void put(int key, int value) { /* Implementation */ }
};
```

Time Complexity: O(1) for both operations

2. Design and implement a thread-safe ring buffer in C++.

Implementation:

```
template
class RingBuffer {
    vector buffer;
    atomic head{0}, tail{0};
    mutex mtx;
public:
    void push(T item);
    bool pop(T& item);
};
```

Key Points:

- Use atomic operations for thread safety
- Mutex for synchronization
- Circular array implementation
- Size must be power of 2 for efficient wrapping

3. Implement a sliding window maximum algorithm that returns the maximum element in each window of size k.

Solution using Deque:

```
vector maxSlidingWindow(vector& nums, int k) {
    deque dq;
    vector result;
    for(int i = 0; i < nums.size(); i++) {
        while(!dq.empty() && nums[dq.back()] < nums[i]) dq.pop_back();
        dq.push_back(i);
        if(dq.front() <= i - k) dq.pop_front();
        if(i >= k-1) result.push_back(nums[dq.front()]);
    }
```

Time Complexity: O(n)

4. Implement a lock-free stack using atomic operations in C++.

Implementation:

```
template
class LockFreeStack {
    struct Node {
        T data;
        atomic next;
    };
    atomic head;
public:
    void push(T value);
    bool pop(T& result);
};
```

Key Considerations:

- Use compare_exchange_weak for atomic updates
- Handle ABA problem
- Memory ordering semantics

5. Design a concurrent hash map with fine-grained locking.

Implementation Approach:

```
template
class ConcurrentHashMap {
    struct Bucket {
        mutex mtx;
        unordered_map data;
    };
    vector buckets;
public:
    void insert(const K& key, const V& value);
```

Features:

- Bucket-level locking for better concurrency
- Dynamic resizing capability
- Thread-safe operations

6. Implement a memory pool allocator for fixed-size objects.

Implementation:

```
class MemoryPool {
    struct Block { Block* next; };
    Block* freeList = nullptr;
    size_t blockSize;
    char* memory = nullptr;
public:
    void* allocate();
    void deallocate(void* ptr);
```

Benefits:

- Reduces fragmentation
- Faster allocation than general-purpose allocator
- Constant-time operations

7. Design a task scheduler with priority queue and thread pool.

Implementation:

```
class TaskScheduler {
    priority_queue tasks;
    vector workers;
```

```
    mutex queueMutex;
    condition_variable cv;
public:
    void schedule(Task task, int priority);
```

Features:

- Priority-based execution
- Thread pool management
- Task cancellation support
- Load balancing

8. Implement a trie (prefix tree) with autocomplete functionality.

Implementation:

```
class TrieNode {
    unordered_map> children;
    bool isEnd = false;
public:
    void insert(string_view word);
    vector autocomplete(string_view prefix);
```

Time Complexity:

- Insert: O(m) where m is word length
- Search: O(m) for exact match
- Autocomplete: O(p + n) where p is prefix length

9. Design a thread-safe object pool with timeout-based resource reclamation.

Implementation:

```
template
class ObjectPool {
    queue> objects;
    mutex mtx;
    condition_variable cv;
public:
    unique_ptr acquire(chrono::milliseconds timeout);
```

Features:

- Automatic resource cleanup
- Timeout-based acquisition
- Thread-safe operations

10. Implement a wait-free queue using atomic operations.

Implementation:

```
template
class WaitFreeQueue {
    struct Node {
        T data;
        atomic next{nullptr};
    };
    atomic head{nullptr}, tail{nullptr};
```

Characteristics:

- No locks or waiting
- Progress guarantee for all threads
- Uses CAS operations
- Handles ABA problem

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly

Key Components

- **API Gateway**: Handle incoming requests, rate limiting
- **URL Generator Service**: Create unique short URLs using base62 encoding
- **Database**: Store URL mappings (NoSQL for scale)
- **Cache Layer**: Redis for frequently accessed URLs

URL Generation Algorithm

```
string generateShortUrl(string longUrl) {
   string hash = md5(longUrl + timestamp);
   return base62Encode(hash.substr(0, 7));
}
```

Scale Considerations

- Distributed counter for ID generation
- Eventual consistency model
- Geographic DNS routing

2. Design a real-time chat system supporting millions of concurrent users

Architecture Components

- **WebSocket Server**: Handle persistent connections
- **Message Queue**: Kafka/RabbitMQ for async processing
- **Presence Service**: Track online status
- **Storage**: Cassandra for messages, Redis for sessions

Key Features

- Message delivery guarantees
- Last seen/read receipts
- Push notifications
- Message encryption

Scaling Strategy

- Shard by conversation ID
- Connection pooling
- Message batching

3. Design a distributed cache system like Redis

Core Components

- **Cache Nodes**: Store key-value pairs in memory
- **Consistent Hashing**: Distribute data across nodes
- **Replication Manager**: Handle data redundancy
- **Eviction Policy**: LRU/LFU implementation

Sample LRU Implementation

```
class LRUCache {
    unordered_map>::iterator> cache;
    list> items;
    size_t capacity;
};
```

Consistency Model

- Write-through vs Write-back
- Cache invalidation strategies
- CAP theorem trade-offs

4. Design a social media feed system like Twitter

System Components

- **Post Service**: Handle tweet creation/deletion
- **Timeline Service**: Generate user feeds
- **Fan-out Service**: Push updates to followers
- **Media Service**: Handle images/videos

Feed Generation

- Push model for active users
- Pull model for inactive users
- Hybrid approach for balanced load

Data Storage

- Posts: Cassandra
- User Graph: Neo4j
- Cache: Redis
- Media: CDN + Object Storage

5. Design a distributed task scheduler system

Core Components

- **Task Queue**: Priority-based job storage
- **Worker Pool**: Execute tasks
- **Scheduler**: Manage task timing
- **Monitor**: Track execution status

Task Definition

```
struct Task {
    string id;
    int priority;
    time_t scheduled_time;
    function callback;
    RetryPolicy retry_policy;
};
```

Reliability Features

- At-least-once delivery
- Dead letter queues
- Task timeout handling
- Retry mechanisms

6. Design a distributed rate limiter

Implementation Approaches

- **Token Bucket**: Flexible rate control
- **Leaky Bucket**: Constant rate output
- **Fixed Window**: Simple time-based

- **Sliding Window**: More accurate control

Token Bucket Implementation

```
bool allowRequest(string user_id, int tokens) {
    double now = getCurrentTime();
    refillTokens(user_id, now);
    return consumeTokens(user_id, tokens);
}
```

Distributed Considerations

- Redis for centralized counting
- Local + Global rate limiting
- Clock synchronization

7. Design a distributed logging and monitoring system

System Components

- **Log Collector**: Gather application logs
- **Stream Processor**: Real-time analysis
- **Storage Engine**: Time-series database
- **Alert Manager**: Notification system

Log Format

```
struct LogEntry {
    string timestamp;
    string service_id;
    LogLevel level;
    string message;
    map metadata;
};
```

Scale Considerations

- Log aggregation
- Sampling strategies
- Data retention policies
- Search optimization

8. Design a distributed key-value store

Architecture Components

- **Data Partitioner**: Shard management
- **Replication Manager**: Data redundancy
- **Consistency Manager**: Quorum handling
- **Failure Detector**: Node health checks

Basic Operations

```
class KVStore {
    Result put(string key, string value);
    Optional get(string key);
    bool delete(string key);
};
```

Consistency Features

- Vector clocks
- Conflict resolution
- Read repair
- Hinted handoff

9. Design a notification service supporting multiple channels

System Components

- **Channel Manager**: Handle delivery methods
- **Template Engine**: Message formatting
- **Rate Controller**: Throttle notifications
- **Analytics Engine**: Track delivery/opens

Notification Model

```
struct Notification {
    string user_id;
    vector channels;
    Template template;
    map params;
};
```

Delivery Features

- Priority queuing
- Fallback channels
- Delivery scheduling
- User preferences

10. Design a distributed configuration management system

Core Components

- **Config Store**: Versioned storage
- **Change Notifier**: Push updates
- **Access Control**: Permission management
- **Audit Logger**: Track changes

Config Structure

```
class ConfigNode {
    string key;
    json value;
    version_t version;
    vector dependencies;
};
```

Key Features

- Hierarchical configs
- Environment isolation
- Hot reloading
- Rollback support

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a function to check if a string is a palindrome in C++, handling both alphanumeric and special characters.

Solution:

Here's an efficient two-pointer approach that ignores non-alphanumeric characters:

```
bool isPalindrome(string s) {
    int left = 0, right = s.length() - 1;
    while (left < right) {
        while (left < right && !isalnum(s[left])) left++;
        while (left < right && !isalnum(s[right])) right--;
        if (tolower(s[left++]) != tolower(s[right--]))
            return false;
    }
    return true;
}
```

Time Complexity: O(n), where n is string length

2. Implement a function to flatten a nested vector in C++ without using recursion.

Solution:

```
vector flatten(vector>& nested) {
    vector result;
    for (const auto& inner : nested) {
        result.insert(result.end(), inner.begin(), inner.end());
    }
    return result;
}
```

Key points:

- Uses vector's insert method with iterators
- Avoids multiple memory allocations
- Time complexity: O(n) where n is total elements

3. Create a thread-safe singleton class in modern C++.

Solution:

```
class Singleton {
    static std::unique_ptr instance;
    static std::mutex mutex;
    Singleton() = default;
public:
    static Singleton* getInstance() {
        std::lock_guard lock(mutex);
        if (!instance) instance.reset(new Singleton());
        return instance.get();
    }
};
```

Features:

- Thread-safe with std::mutex
- Memory-safe using unique_ptr

- Follows modern C++ practices

4. Write a function to find the first non-repeating character in a string using C++.

Solution:

```
char firstNonRepeating(string s) {
    array count = {};
    for (char c : s) count[c]++;
    for (char c : s)
        if (count[c] == 1) return c;
    return '0';
}
```

Advantages:

- Single pass solution with O(n) complexity
- Uses fixed-size array for constant space
- Handles ASCII characters efficiently

5. Implement a circular buffer class with thread-safe operations in C++.

Solution:

```
template
class CircularBuffer {
    std::vector buffer;
    size_t head = 0, tail = 0;
    std::mutex mtx;
    bool push(T item) {
        std::lock_guard lock(mtx);
        if ((tail + 1) % buffer.size() == head) return false;
        buffer[tail] = item;
        tail = (tail + 1) % buffer.size();
        return true;
    }
};
```

Features:

- Thread-safe operations
- Fixed-size circular implementation
- O(1) push/pop operations

6. Write a memory leak detector class that can track allocations and deallocations.

Solution:

```
class MemoryLeakDetector {
    static std::map allocations;
    static std::mutex mtx;
public:
    static void* allocate(size_t size) {
        std::lock_guard lock(mtx);
        void* ptr = malloc(size);
        allocations[ptr] = size;
        return ptr;
    }
};
```

Usage:

- Tracks memory allocations
- Thread-safe implementation
- Useful for debugging memory leaks

7. Implement a smart pointer with reference counting in C++.

Solution:

```
template
class SmartPtr {
    T* ptr;
    size_t* ref_count;
public:
    SmartPtr(T* p) : ptr(p), ref_count(new size_t(1)) {}
    ~SmartPtr() {
        if (--(*ref_count) == 0) {
            delete ptr; delete ref_count;
        }
    }
};
```

Features:

- Automatic memory management
- Reference counting
- RAII compliant

8. Create a function to detect and break cycles in a linked list.

Solution:

```
bool hasCycle(ListNode* head) {
    if (!head) return false;
    ListNode *slow = head, *fast = head;
    while (fast->next && fast->next->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) return true;
    }
    return false;
}
```

Algorithm:

- Floyd's cycle detection
- O(n) time complexity
- O(1) space complexity

9. Implement a thread pool with a job queue in C++.

Solution:

```
class ThreadPool {
    vector workers;
    queue> tasks;
    mutex queue_mutex;
    condition_variable condition;
    bool stop;
public:
    ThreadPool(size_t threads) : stop(false) {
        for(size_t i = 0; i < threads; ++i)
            workers.emplace_back([this] { worker_thread(); });
    }
};
```

Features:

- Fixed thread pool size
- Task queue management
- Thread synchronization

10. Write a template metafunction to calculate factorial at compile-time.

Solution:

```
template
struct Factorial {
    static constexpr unsigned value = N * Factorial::value;
};

template<>
struct Factorial<0> {
    static constexpr unsigned value = 1;
};
```

Advantages:

- Compile-time computation
- Zero runtime overhead
- Template metaprogramming demonstration