C++ for High-Frequency Trading Systems

Complete Guide: Theory, Implementation, and Low-Latency Optimization

Chapter 1: Modern C++ Fundamentals for HFT

1.1 Why C++ for High-Frequency Trading?

Performance Requirements:

- **Latency:** Microsecond (μs) to nanosecond (ns) time scales
- **Throughput:** Millions of operations per second
- **Determinism:** Predictable, consistent execution times
- **Hardware control:** Direct memory management, CPU cache optimization

C++ Advantages:

1. **Zero-cost abstractions:** High-level features without runtime overhead
2. **Manual memory management:** No garbage collection pauses
3. **Inline assembly:** Direct CPU instruction access when needed
4. **Compile-time optimization:** Templates, constexpr eliminate runtime work

1.2 Modern C++ Standards (C++11/14/17/20)

C++11 Key Features:

```
// Move semantics - avoid unnecessary copies
class OrderBook {
    std::vector&lt;Order&gt; orders;
public:
    // Move constructor
    OrderBook(OrderBook&amp;&amp; other) noexcept
        : orders(std::move(other.orders)) {}

    // Move assignment
    OrderBook&amp; operator=(OrderBook&amp;&amp; other) noexcept {
        orders = std::move(other.orders);
        return *this;
    }
};

// Lambda expressions
auto price_filter = [threshold](const Order&amp; o) {
    return o.price &gt; threshold;
};
```

```
// Smart pointers - automatic memory management
std::unique_ptr&lt;Strategy&gt; strategy = std::make_unique&lt;MomentumStrategy&gt;();
std::shared_ptr&lt;MarketData&gt; data = std::make_shared&lt;MarketData&gt;();

// Range-based for loops
for (const auto&amp; order : orderbook) {
    process(order);
}
```

C++14 Enhancements:

```
// Generic lambdas
auto generic_comparator = [](auto a, auto b) { return a &lt; b; };

// Return type deduction
auto calculate_vwap(const std::vector&lt;Trade&gt;&amp; trades) {
    double sum_price_volume = 0.0;
    double total_volume = 0.0;
    for (const auto&amp; t : trades) {
        sum_price_volume += t.price * t.volume;
        total_volume += t.volume;
    }
    return sum_price_volume / total_volume;
}
```

C++17 Features:

```
// Structured bindings
std::map&lt;std::string, double&gt; positions;
for (const auto&amp; [symbol, quantity] : positions) {
    std::cout &lt;&lt; symbol &lt;&lt; ": " &lt;&lt; quantity &lt;&lt; "\n";
}

// std::optional - handle missing values without exceptions
std::optional&lt;Order&gt; find_order(uint64_t order_id) {
    if (auto it = orders.find(order_id); it != orders.end()) {
        return it-&gt;second;
    }
    return std::nullopt;
}

// if with initializer
if (auto result = try_execute_order(); result.has_value()) {
    process(result.value());
}
```

C++20 Concepts:

```
// Concepts - compile-time type constraints
template&lt;typename T&gt;
concept Priceable = requires(T t) {
```

```
    { t.get_price() } -&gt; std::convertible_to&lt;double&gt;;
};

template&lt;Priceable T&gt;
double calculate_mid(const T&amp; bid, const T&amp; ask) {
    return (bid.get_price() + ask.get_price()) / 2.0;
}

// Ranges
#include &lt;ranges&gt;
auto expensive_orders = orders
    | std::views::filter([](const auto&amp; o) { return o.price &gt; 1000; })
    | std::views::take(10);
```

Chapter 2: STL Containers for Trading Systems

2.1 Container Selection Guide

Performance Characteristics:

Container	Insert	Lookup	Delete	Use Case
std::vector	O(1) amortized end	O(n)	O(n)	Time-series data, tick storage
std::deque	O(1) both ends	O(n)	O(1) ends	Ring buffers, sliding windows
std::list	O(1) anywhere	O(n)	O(1)	Order lists (price levels)
std::map	O(log n)	O(log n)	O(log n)	Price levels (Red-Black tree)
std::unordered_map	O(1) average	O(1) average	O(1) average	Symbol lookups, order IDs
std::set	O(log n)	O(log n)	O(log n)	Unique sorted elements

2.2 Order Book Implementation

Limit Order Book Structure:

```
#include &lt;map&gt;
#include &lt;list&gt;
#include &lt;unordered_map&gt;

struct Order {
    uint64_t order_id;
    uint64_t timestamp;
    char side;  // 'B' for buy, 'S' for sell
    double price;
    uint32_t quantity;

    Order(uint64_t id, uint64_t ts, char s, double p, uint32_t q)
        : order_id(id), timestamp(ts), side(s), price(p), quantity(q) {}
};
```

```
class OrderBook {
private:
    // Price level -&gt; list of orders (FIFO queue)
    using OrderList = std::list&lt;Order&gt;;
    using PriceLevel = std::pair&lt;uint32_t, OrderList&gt;;  // (total_quantity, orders)

    // Buy side: highest price first (descending)
    std::map&lt;double, PriceLevel, std::greater&lt;double&gt;&gt; bids;

    // Sell side: lowest price first (ascending)
    std::map&lt;double, PriceLevel, std::less&lt;double&gt;&gt; asks;

    // Fast order lookup by ID
    std::unordered_map&lt;uint64_t, std::pair&lt;double, OrderList::iterator&gt;&gt; orde

public:
    // Add order: O(log n) for price level + O(1) for order insertion
    void add_order(const Order&amp; order) {
        auto&amp; levels = (order.side == 'B') ? bids : asks;
        auto&amp; [total_qty, order_list] = levels[order.price];

        order_list.push_back(order);
        total_qty += order.quantity;

        // Store iterator for O(1) deletion
        order_map[order.order_id] = {order.price, std::prev(order_list.end())};
    }

    // Cancel order: O(log n) + O(1)
    bool cancel_order(uint64_t order_id) {
        auto it = order_map.find(order_id);
        if (it == order_map.end()) return false;

        auto [price, order_it] = it-&gt;second;
        char side = order_it-&gt;side;

        auto&amp; levels = (side == 'B') ? bids : asks;
        auto level_it = levels.find(price);

        auto&amp; [total_qty, order_list] = level_it-&gt;second;
        total_qty -= order_it-&gt;quantity;
        order_list.erase(order_it);

        // Remove empty price level
        if (order_list.empty()) {
            levels.erase(level_it);
        }

        order_map.erase(it);
        return true;
    }

    // Get best bid/ask: O(1)
    std::optional&lt;double&gt; best_bid() const {
        if (bids.empty()) return std::nullopt;
        return bids.begin()-&gt;first;
```

```
    }

    std::optional&lt;double&gt; best_ask() const {
        if (asks.empty()) return std::nullopt;
        return asks.begin()-&gt;first;
    }

    // Match incoming order
    std::vector&lt;std::pair&lt;uint64_t, uint32_t&gt;&gt; match_order(Order&amp; incomir
        std::vector&lt;std::pair&lt;uint64_t, uint32_t&gt;&gt; fills;

        auto&amp; levels = (incoming.side == 'B') ? asks : bids;

        while (!levels.empty() &amp;&amp; incoming.quantity &gt; 0) {
            auto&amp; [price, level] = *levels.begin();
            auto&amp; [total_qty, order_list] = level;

            // Check price crossing
            bool crosses = (incoming.side == 'B' &amp;&amp; incoming.price &gt;= price) |
                           (incoming.side == 'S' &amp;&amp; incoming.price &lt;= price);
            if (!crosses) break;

            // Match with orders at this price level (FIFO)
            while (!order_list.empty() &amp;&amp; incoming.quantity &gt; 0) {
                Order&amp; resting = order_list.front();
                uint32_t fill_qty = std::min(incoming.quantity, resting.quantity);

                fills.push_back({resting.order_id, fill_qty});

                incoming.quantity -= fill_qty;
                resting.quantity -= fill_qty;
                total_qty -= fill_qty;

                if (resting.quantity == 0) {
                    order_map.erase(resting.order_id);
                    order_list.pop_front();
                }
            }

            // Remove empty price level
            if (order_list.empty()) {
                levels.erase(levels.begin());
            }
        }

        return fills;
    }
};
```

Chapter 3: Multithreading and Concurrency

3.1 Thread Basics

Creating Threads:

```
#include &lt;thread&gt;
#include &lt;iostream&gt;

void market_data_handler() {
    while (running) {
        // Process incoming market data
        process_tick();
    }
}

void order_execution_handler() {
    while (running) {
        // Execute pending orders
        execute_orders();
    }
}

int main() {
    std::thread md_thread(market_data_handler);
    std::thread exec_thread(order_execution_handler);

    // Wait for threads to complete
    md_thread.join();
    exec_thread.join();
}
```

3.2 Thread Synchronization

Mutex and Lock Guards:

```
#include &lt;mutex&gt;

class ThreadSafeOrderBook {
private:
    OrderBook orderbook;
    mutable std::mutex mtx;

public:
    void add_order(const Order&amp; order) {
        std::lock_guard&lt;std::mutex&gt; lock(mtx);  // RAII-style locking
        orderbook.add_order(order);
    }  // Automatically unlocks when lock_guard destructs

    std::optional&lt;double&gt; best_bid() const {
        std::lock_guard&lt;std::mutex&gt; lock(mtx);
        return orderbook.best_bid();
```

```
    }
};
```

Condition Variables (Producer-Consumer):

```
#include &lt;condition_variable&gt;
#include &lt;queue&gt;

class ThreadSafeQueue {
private:
    std::queue&lt;Order&gt; queue;
    mutable std::mutex mtx;
    std::condition_variable cv;

public:
    void push(Order order) {
        {
            std::lock_guard&lt;std::mutex&gt; lock(mtx);
            queue.push(std::move(order));
        }
        cv.notify_one();  // Wake up waiting consumer
    }

    Order pop() {
        std::unique_lock&lt;std::mutex&gt; lock(mtx);
        cv.wait(lock, [this] { return !queue.empty(); });  // Wait for data

        Order order = std::move(queue.front());
        queue.pop();
        return order;
    }
};
```

3.3 Lock-Free Programming with std::atomic

Atomic Operations:

```
#include &lt;atomic&gt;

class LockFreeCounter {
private:
    std::atomic&lt;uint64_t&gt; counter{0};

public:
    uint64_t increment() {
        return counter.fetch_add(1, std::memory_order_relaxed);
    }

    uint64_t get() const {
        return counter.load(std::memory_order_acquire);
    }
};
```

Lock-Free SPSC (Single Producer Single Consumer) Queue:

```
template&lt;typename T, size_t Size&gt;
class SPSCQueue {
private:
    std::array&lt;T, Size&gt; buffer;
    alignas(64) std::atomic&lt;size_t&gt; write_idx{0};  // Separate cache lines
    alignas(64) std::atomic&lt;size_t&gt; read_idx{0};

public:
    bool push(const T&amp; item) {
        size_t current_write = write_idx.load(std::memory_order_relaxed);
        size_t next_write = (current_write + 1) % Size;

        // Check if queue is full
        if (next_write == read_idx.load(std::memory_order_acquire)) {
            return false;  // Queue full
        }

        buffer[current_write] = item;
        write_idx.store(next_write, std::memory_order_release);
        return true;
    }

    bool pop(T&amp; item) {
        size_t current_read = read_idx.load(std::memory_order_relaxed);

        // Check if queue is empty
        if (current_read == write_idx.load(std::memory_order_acquire)) {
            return false;  // Queue empty
        }

        item = buffer[current_read];
        read_idx.store((current_read + 1) % Size, std::memory_order_release);
        return true;
    }
};
```

Memory Ordering Explanation:

- `memory_order_relaxed`: No synchronization, only atomicity
- `memory_order_acquire`: Reads happen-before subsequent operations
- `memory_order_release`: Writes happen-after previous operations
- `memory_order_acq_rel`: Both acquire and release
- `memory_order_seq_cst`: Sequential consistency (strongest, default)

Chapter 4: Memory Management and Optimization

4.1 Custom Allocators

Object Pool Allocator:

```
template&lt;typename T, size_t PoolSize = 1024&gt;
class ObjectPool {
private:
    union Slot {
        T object;
        Slot* next;
    };

    std::array&lt;Slot, PoolSize&gt; pool;
    Slot* free_list;

public:
    ObjectPool() {
        // Initialize free list
        free_list = &amp;pool[0];
        for (size_t i = 0; i &lt; PoolSize - 1; ++i) {
            pool[i].next = &amp;pool[i + 1];
        }
        pool[PoolSize - 1].next = nullptr;
    }

    template&lt;typename... Args&gt;
    T* allocate(Args&amp;&amp;... args) {
        if (!free_list) return nullptr;  // Pool exhausted

        Slot* slot = free_list;
        free_list = free_list-&gt;next;

        // Placement new
        return new (&amp;slot-&gt;object) T(std::forward&lt;Args&gt;(args)...);
    }

    void deallocate(T* ptr) {
        ptr-&gt;~T();  // Explicit destructor call

        Slot* slot = reinterpret_cast&lt;Slot*&gt;(ptr);
        slot-&gt;next = free_list;
        free_list = slot;
    }
};

// Usage
ObjectPool&lt;Order&gt; order_pool;
Order* order = order_pool.allocate(12345, timestamp, 'B', 100.5, 1000);
// ... use order ...
order_pool.deallocate(order);
```

4.2 Cache Optimization

False Sharing Prevention:

```
// BAD: False sharing (both on same cache line)
struct BadCounter {
    std::atomic&amp;lt;uint64_t&amp;gt; producer_count;
    std::atomic&amp;lt;uint64_t&amp;gt; consumer_count;
};

// GOOD: Separate cache lines (64 bytes)
struct alignas(64) GoodCounter {
    std::atomic&amp;lt;uint64_t&amp;gt; producer_count;
    char padding[64 - sizeof(std::atomic&amp;lt;uint64_t&amp;gt;)];
};

struct alignas(64) ConsumerCounter {
    std::atomic&amp;lt;uint64_t&amp;gt; consumer_count;
};
```

Cache-Friendly Data Layout (SoA vs AoS):

```
// Array of Structures (AoS) - poor cache utilization
struct Trade {
    double price;
    uint32_t volume;
    uint64_t timestamp;
};
std::vector&amp;lt;Trade&amp;gt; trades;  // Bad for iterating over just prices

// Structure of Arrays (SoA) - better cache utilization
struct TradeData {
    std::vector&amp;lt;double&amp;gt; prices;
    std::vector&amp;lt;uint32_t&amp;gt; volumes;
    std::vector&amp;lt;uint64_t&amp;gt; timestamps;
};
TradeData trades;  // Good: prices contiguous in memory
```

4.3 Memory Alignment

```
// Align data for SIMD operations
alignas(32) float prices[8];  // Aligned for AVX (256-bit)

// Check alignment
static_assert(alignof(prices) == 32);

// Aligned allocation
void* aligned_alloc_wrapper(size_t size, size_t alignment) {
    void* ptr = nullptr;
    if (posix_memalign(&amp;amp;ptr, alignment, size) != 0) {
        throw std::bad_alloc();
    }
```

```
    return ptr;
}
```

Chapter 5: SIMD and Vectorization

5.1 Introduction to SIMD

SIMD = Single Instruction Multiple Data

- Process multiple data elements with one CPU instruction
- Modern CPUs: SSE (128-bit), AVX/AVX2 (256-bit), AVX-512 (512-bit)

5.2 AVX2 Intrinsics

Vectorized Dot Product:

```
#include &lt;immintrin.h&gt;  // AVX/AVX2 intrinsics

// Dot product of two arrays (AVX2: 8 floats at once)
float dot_product_avx2(const float* a, const float* b, size_t n) {
    __m256 sum = _mm256_setzero_ps();  // Zero vector

    size_t i = 0;
    for (; i + 7 &lt; n; i += 8) {
        __m256 va = _mm256_loadu_ps(&amp;a[i]);  // Load 8 floats
        __m256 vb = _mm256_loadu_ps(&amp;b[i]);
        __m256 prod = _mm256_mul_ps(va, vb);  // Element-wise multiply
        sum = _mm256_add_ps(sum, prod);       // Accumulate
    }

    // Horizontal sum: sum all 8 elements in sum vector
    __m128 sum_low = _mm256_castps256_ps128(sum);
    __m128 sum_high = _mm256_extractf128_ps(sum, 1);
    __m128 sum128 = _mm_add_ps(sum_low, sum_high);

    sum128 = _mm_hadd_ps(sum128, sum128);
    sum128 = _mm_hadd_ps(sum128, sum128);

    float result = _mm_cvtss_f32(sum128);

    // Handle remaining elements (scalar)
    for (; i &lt; n; ++i) {
        result += a[i] * b[i];
    }

    return result;
}
```

VWAP Calculation with AVX2:

```
double calculate_vwap_avx2(const double* prices, const double* volumes, size_t n) {
    __m256d sum_pv = _mm256_setzero_pd();  // 4 doubles
    __m256d sum_vol = _mm256_setzero_pd();

    size_t i = 0;
    for (; i + 3 &lt; n; i += 4) {
        __m256d p = _mm256_loadu_pd(&amp;prices[i]);
        __m256d v = _mm256_loadu_pd(&amp;volumes[i]);

        __m256d pv = _mm256_mul_pd(p, v);
        sum_pv = _mm256_add_pd(sum_pv, pv);
        sum_vol = _mm256_add_pd(sum_vol, v);
    }

    // Horizontal sum
    double pv_array[4], vol_array[4];
    _mm256_storeu_pd(pv_array, sum_pv);
    _mm256_storeu_pd(vol_array, sum_vol);

    double total_pv = pv_array[0] + pv_array[1] + pv_array[2] + pv_array[3];
    double total_vol = vol_array[0] + vol_array[1] + vol_array[2] + vol_array[3];

    // Scalar remainder
    for (; i &lt; n; ++i) {
        total_pv += prices[i] * volumes[i];
        total_vol += volumes[i];
    }

    return total_pv / total_vol;
}
```

Chapter 6: Template Metaprogramming

6.1 Templates for Generic Code

Function Templates:

```
template&lt;typename T&gt;
T max_price(const T&amp; a, const T&amp; b) {
    return (a &gt; b) ? a : b;
}

// Usage
double max_d = max_price(100.5, 101.2);
int max_i = max_price(100, 200);
```

Class Templates:

```
template&lt;typename T, size_t Capacity&gt;
class CircularBuffer {
private:
```

```
    std::array&lt;T, Capacity&gt; buffer;
    size_t head = 0;
    size_t tail = 0;
    size_t count = 0;

public:
    bool push(const T&amp; item) {
        if (count == Capacity) return false;
        buffer[tail] = item;
        tail = (tail + 1) % Capacity;
        ++count;
        return true;
    }

    bool pop(T&amp; item) {
        if (count == 0) return false;
        item = buffer[head];
        head = (head + 1) % Capacity;
        --count;
        return true;
    }

    constexpr size_t capacity() const { return Capacity; }
};
```

6.2 CRTP (Curiously Recurring Template Pattern)

Static Polymorphism (Zero Runtime Overhead):

```
template&lt;typename Derived&gt;
class Strategy {
public:
    void execute() {
        static_cast&lt;Derived*&gt;(this)-&gt;execute_impl();
    }

    double calculate_signal() {
        return static_cast&lt;Derived*&gt;(this)-&gt;calculate_signal_impl();
    }
};

class MomentumStrategy : public Strategy&lt;MomentumStrategy&gt; {
public:
    void execute_impl() {
        // Momentum-specific logic
    }

    double calculate_signal_impl() {
        return momentum_indicator;
    }
};

class MeanReversionStrategy : public Strategy&lt;MeanReversionStrategy&gt; {
public:
    void execute_impl() {
```

```
        // Mean reversion logic
    }

    double calculate_signal_impl() {
        return z_score;
    }
};

// Generic function works with any Strategy
template&lt;typename T&gt;
void run_strategy(Strategy&lt;T&gt;&amp; strategy) {
    double signal = strategy.calculate_signal();
    if (signal &gt; threshold) {
        strategy.execute();
    }
}
```

6.3 Compile-Time Computation

constexpr Functions:

```
constexpr uint64_t factorial(int n) {
    return (n &lt;= 1) ? 1 : n * factorial(n - 1);
}

constexpr uint64_t combinations(int n, int k) {
    return factorial(n) / (factorial(k) * factorial(n - k));
}

// Computed at compile time!
constexpr uint64_t result = combinations(10, 3);  // No runtime cost
```

Chapter 7: Low-Latency Techniques

7.1 Latency Sources

1. **Algorithmic complexity:** O(n) vs O(1)
2. **Cache misses:** L1 (~1ns) vs L2 (~3ns) vs L3 (~12ns) vs RAM (~100ns)
3. **Branch misprediction:** ~15-20 cycles penalty
4. **Context switches:** Microseconds
5. **System calls:** Avoid kernel transitions
6. **Network:** Physical distance, kernel stack

7.2 Branch Prediction Hints

```
// Likely/unlikely macros (GCC/Clang)
#define LIKELY(x)   __builtin_expect(!!(x), 1)
#define UNLIKELY(x) __builtin_expect(!!(x), 0)

if (LIKELY(order.quantity &gt; 0)) {
    // Hot path - processor prefetches this branch
    execute_order(order);
} else {
    // Cold path
    handle_error();
}
```

7.3 CPU Affinity (Pin Thread to Core)

```
#include &lt;pthread.h&gt;
#include &lt;sched.h&gt;

void set_thread_affinity(int core_id) {
    cpu_set_t cpuset;
    CPU_ZERO(&amp;cpuset);
    CPU_SET(core_id, &amp;cpuset);

    pthread_t current_thread = pthread_self();
    pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &amp;cpuset);
}

// Pin critical thread to core 0
set_thread_affinity(0);
```

7.4 Timestamping with RDTSC

```
#include &lt;x86intrin.h&gt;

inline uint64_t rdtsc() {
    return __rdtsc();  // Read CPU Time Stamp Counter
}

// Measure latency
uint64_t start = rdtsc();
process_order(order);
uint64_t end = rdtsc();
uint64_t cycles = end - start;

// Convert to nanoseconds (assuming 3 GHz CPU)
double ns = cycles / 3.0;
```

Chapter 8: Networking for HFT

8.1 TCP Socket Programming

Basic TCP Server:

```
#include &lt;sys/socket.h&gt;
#include &lt;netinet/in.h&gt;
#include &lt;arpa/inet.h&gt;
#include &lt;unistd.h&gt;

int create_tcp_server(int port) {
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);

    int opt = 1;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &amp;opt, sizeof(opt));

    sockaddr_in address;
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(port);

    bind(server_fd, (sockaddr*)&amp;address, sizeof(address));
    listen(server_fd, 10);

    return server_fd;
}
```

8.2 UDP Multicast (Market Data Feeds)

Multicast Receiver:

```
int create_multicast_receiver(const char* group, int port) {
    int sock = socket(AF_INET, SOCK_DGRAM, 0);

    // Allow multiple sockets to bind to same port
    int reuse = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &amp;reuse, sizeof(reuse));

    sockaddr_in local_addr;
    memset(&amp;local_addr, 0, sizeof(local_addr));
    local_addr.sin_family = AF_INET;
    local_addr.sin_addr.s_addr = INADDR_ANY;
    local_addr.sin_port = htons(port);

    bind(sock, (sockaddr*)&amp;local_addr, sizeof(local_addr));

    // Join multicast group
    ip_mreq mreq;
    mreq.imr_multicast.s_addr = inet_addr(group);
    mreq.imr_interface.s_addr = INADDR_ANY;
    setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &amp;mreq, sizeof(mreq));
```

```
    return sock;
}
```

8.3 Kernel Bypass with DPDK (Conceptual)

DPDK = Data Plane Development Kit

- Bypass kernel network stack
- Poll-mode drivers (no interrupts)
- User-space packet processing
- Achieve <1 μs latency

Key Concepts:

- **Huge pages:** Reduce TLB misses
- **CPU isolation:** Dedicated cores for packet processing
- **Zero-copy:** DMA directly to user space
- **Batch processing:** Process packets in bursts

Chapter 9: Project Implementations

Project 1: Lock-Free Order Book

Difficulty: Advanced
Technologies: std::atomic, Red-Black tree, SPSC queue
Metrics: 450K+ operations/second, <100ns latency

Project 2: Market Data Handler

Difficulty: Advanced
Technologies: UDP multicast, FIX protocol, multithreading
Metrics: 10M+ messages/second, <5μs tick-to-trade

Project 3: Matching Engine Simulator

Difficulty: Intermediate-Advanced
Features: FIFO matching, price-time priority, trade reporting

Project 4: Custom Memory Pool

Difficulty: Intermediate
Features: Object pooling, NUMA-aware allocation, zero fragmentation

Project 5: SIMD Matrix Operations Library

Difficulty: Advanced
Features: AVX2 vectorization, cache blocking, 10x+ speedup

Chapter 10: Interview Preparation

10.1 Common C++ Interview Questions

Question 1: Implement std::unique_ptr

```
template&lt;typename T&gt;
class UniquePtr {
private:
    T* ptr;

public:
    explicit UniquePtr(T* p = nullptr) : ptr(p) {}

    ~UniquePtr() { delete ptr; }

    // Delete copy constructor and assignment
    UniquePtr(const UniquePtr&amp;) = delete;
    UniquePtr&amp; operator=(const UniquePtr&amp;) = delete;

    // Move constructor
    UniquePtr(UniquePtr&amp;&amp; other) noexcept : ptr(other.ptr) {
        other.ptr = nullptr;
    }

    // Move assignment
    UniquePtr&amp; operator=(UniquePtr&amp;&amp; other) noexcept {
        if (this != &amp;other) {
            delete ptr;
            ptr = other.ptr;
            other.ptr = nullptr;
        }
        return *this;
    }

    T&amp; operator*() const { return *ptr; }
    T* operator-&gt;() const { return ptr; }
    T* get() const { return ptr; }

    T* release() {
        T* old_ptr = ptr;
        ptr = nullptr;
        return old_ptr;
    }

    void reset(T* p = nullptr) {
        delete ptr;
        ptr = p;
```

```
    }
};
```

Question 2: Explain Virtual Function Mechanism

- Virtual table (vtable): Array of function pointers
- Virtual pointer (vptr): Hidden pointer in each object
- Runtime polymorphism cost: One extra indirection (~3-5 ns)
- Can prevent inlining and optimization

Question 3: What is False Sharing?

- Multiple threads modify different variables on same cache line
- Cache line (64 bytes) invalidated on each write
- Solution: Align variables to separate cache lines (`alignas(64)`)

10.2 System Design Questions

Question: Design a High-Performance Order Management System

Components:

1. **Order Ingestion:** Multithreaded receivers, lock-free queues
2. **Risk Checks:** Pre-trade validation (limits, exposure)
3. **Order Routing:** Smart order routing to exchanges
4. **Execution:** FIX protocol, TCP/UDP connections
5. **Position Tracking:** Real-time P&L calculation
6. **Logging:** Async logging to avoid blocking critical path

Performance Targets:

- Order-to-wire: <10 µs
- Throughput: 100K+ orders/second
- Uptime: 99.999% (5 nines)

Further Reading

Books:

- Scott Meyers: *Effective Modern C++*
- Anthony Williams: *C++ Concurrency in Action*
- Fedor Pikus: *The Art of Writing Efficient Programs*
- Agner Fog: *Optimizing Software in C++* (free PDF)

Online Resources:

- CPPReference.com - C++ standard library reference
- Compiler Explorer (godbolt.org) - Assembly output analysis
- Intel Intrinsics Guide - SIMD instruction reference
- CppCon YouTube - Conference talks on performance

Open Source Projects:

- Folly (Facebook) - High-performance C++ library
- Abseil (Google) - C++ common libraries
- DPDK - Data Plane Development Kit
- libfixpp - FIX protocol parser

This textbook provides comprehensive coverage of C++ for HFT systems, from fundamentals to advanced low-latency techniques, with complete working implementations and interview preparation materials.