TRADING SYSTEMS DEVELOPER INTERVIEW GUIDE (C++ EDITION)

Insider's Guide to Top Tech Jobs in Finance

JEFF VOGELS

Jeff Vogels

Trading Systems Developer Interview Guide (C++ Edition)

Insider's Guide to Top Tech Jobs in Finance

First edition

This book was professionally typeset on Reedsy
Find out more at reedsy.com

Contents

Introduction to Trading Systems Developer Jobs

Trading System Developers are software engineers in a firm that is in the business of capital markets such as hedge funds, HFTs, prop/algorithmic trading firms, market makers, exchanges, trading platform providers, etc. This book is catered to the interviews of the software engineer roles in capital markets firms especially in teams that are responsible for electronic trading systems.

Trading Systems Developers build systems that enable electronic trading in financial markets. The market participants are broadly 1) exchanges i.e. venues which facilitate the trading of various financial instruments and act as middlemen for accepting orders, execution of orders, clearing, etc. between buyers and sellers 2) the trading systems which react to the market events and either uses the algorithms/models(majorly) or use human traders(small percentage) to send buy/sell order events (New/Cancel/Replace) to the exchange. Example of exchanges: NYSE, Nasdaq, CBOE, etc. Examples of companies running trading systems: hedge funds such as Citadel, market makers such as Getco, prop trading firms such as Hudson River Trading, trading desks within banks such as Morgan Stanley, brokerages supporting institutional and retail investors such as Fidelity Investments, Schwab, etc.

In an electronic trading environment, typical roles we see are the trading systems developers(software engineers typically with STEM background), quants(math geniuses), and traders(various backgrounds). Quants do research on historical market

data and build models/strategies for the trading system that are expected to outperform the market. Software Engineers(Trading Systems Developers) are responsible for developing systems that enable trading and implementing the algorithms/models which include software development(mostly) and hardware optimization(where ultra-low latency is a must). Traders have the final say on which strategies/models to use and are responsible for the P&L. In some tech/quant driven firms, these roles can vary multiple hats.

Trading Systems developers are programmers who have are good with algorithms and system design like any software engineer at a good tech company but they also need a good understanding of the internals of OS/Compilers/Network etc. so they can build low latency, high-performance systems. Trading systems developers work on various software components like market data handling, building order book, matching engine, implementing strategies/models developed by quants, order management systems, order routing systems, smart order routers, etc. They also optimize the systems by choosing the right hardware such as the number of cores, memory, network cards, kernel bypass network cards, FPGAs, etc.

Trading systems software is expected to be low latency in general due to the competitive nature of various firms involved and opportunities are available only for a short amount of time due to the dynamic nature of the markets. Typically systems are built in C++ language on the Linux OS. C++ gives control of the machine to the programmer to extract maximum performance. Since money and reputation of firms are at stake, trading systems need to be reliable and fault-tolerant.

Interview Planning and Tips

Tech/Quant Finance jobs are mostly located near the financial hubs - New York, Chicago in the US; London, Amsterdam in Europe; Shanghai, Hong Kong, Singapore, Tokyo in Asia; Sydney in Australia, etc.

These jobs are typically hired by firms via head hunters. The best way to get noticed is with a good profile on LinkedIn. Usually, the recruiters reach you out if you hit certain keywords related to the job such as C++, Linux, Multithreading, previous work experience at other tech/quant finance roles, etc. You may also reach them via LinkedIn InMails. Another way to get interviews is by applying to jobs on efinancialcareers.com, LinkedIn Jobs, dice.com, etc.

Most firms are fine with developers with no previous work experience in tech/quant finance roles as long as they have strong technical skills although some senior roles are typically for people with the background in developing similar systems.

There are a variety of firms operating in this space such as - Hedge Funds, Algorithmic Trading firms, Exchanges, Market Makers, Investment Banks, Brokerages, Trading Software, and Analytics providers, etc. While the work varies based on what the firm is doing, the underlying technical skills are very similar.

The interview process consists of a basic introductory call with the recruiter/head hunter followed by a 30-45 minute phone interview with the hiring manager which is a

mix of a tech interview, going through your background, and fit for the position. Some firms are requiring a “Hacker Rank” test to solve a few coding problems online before the call with the hiring manager as part of screening. If all goes well so far, you will be invited to onsite interviews and meet about 5-7 different people. You will be tested on various topics that are the chapters of this book.

Compensation is very competitive for the market with a big base and bonuses. It is not uncommon for strong developers to make similar salaries or more than in FAANG companies such as Google, Facebook, etc.

Book Organization

This book is organized into chapters based on the topics asked in the trading systems developer interviews at top capital markets firms.

This book is written with a purpose to give you an idea of the kind of topics that will be asked in the interview. This is not a complete reference book. There is a Resources section in the end with a list of books if you need to improve your understanding and knowledge in a certain topic.

At the beginning of each chapter, there is a small paragraph that mentions the kind of topics that are asked in the interviews related to that chapter. After that, there will be questions with full solutions/answers that will cover those topics in breadth and depth in the context of interviews.

This book can serve two groups of people. If you are a software developer but new to the tech/quant finance jobs, this book can let you assess where you lack, so you can read on those topics more and then practice the questions from this book. For experienced developers already in this industry, it is a great way to quickly ramp up your preparation if you are changing your job.

The goal of the book is not to cover every question that can be possibly asked because that is impossible in my lifetime. The goal of this book is to cover most topics that will

be asked in the interviews with questions that are similar and are of comparable difficulty to those asked in interviews.

1

C++

- Many C++ questions will be asked
- You are expected to understand how different language features work and how core features work under the hood
- It is good to be current with the new C++ language updates
- Be comfortable with STL, Modern C++, boost(good to know)
- Esoteric questions are typically not asked (don’t worry about them)

Q1

```
class Test {
};
Test t;
```

What is sizeof(t)?

A1

at least 1. Because if size is 0, situations will arise where two distinct objects of class Test will have same address, which is illegal as per C++ standard.

```
//Example:
int main() {
```

```
    // 2 stack objects of Test,if size is 0, addresses of t1,t2
    // are not unique
    Test t1, t2;
}
```

Q2

```
const char* p = "Test";
```

What is output of expression *p++?

A2

T. Due to operator precedence rules in C++, *p++ evaluates as *(p++) . As postfix increment returns old value, output is *(p) which is T.

Q3

```
std::map<const char*, int> strMap;
```

What do you think of above code?

A3

The above code has char* as key for the map, which means the default comparison function compares two char pointers instead of the null terminated strings. So, if the intent is that the exact same pointer is used for insert/lookup/delete on map, it works. If the intent is that the C-strings are to be keys, then a string compare function needs to be defined. (Two distinct char pointers can point to same string content)

```
struct StrCmp {
   bool operator()(char const *a, char const *b) const
```

```
    {
       return std::strcmp(a, b) < 0;
    }
};

//StrCmp specified in map declaration
map<const char *, int, StrCmp> strMap;
```

Q4

```
class Test {
public:
  virtual ~Test() {}
};
Test t;
```

What is sizeof(t)?

A4

size of pointer on the machine. (Word size for 32 bit machine is 4, for 64 bit machine is 8.) Note, in the class Test, the destructor is declared virtual which will make the compiler declare a virtual table pointer variable in all the objects of class Test, that points to the class Test's virtual table.

Q5

What is the time complexity of accessing element by index in std::deque?

A5

Constant time. std::deque is typically implemented as dynamic array of pointers to fixed size chunks where the elements are stored. Since each of the fixed size chunks have same number of elements except the first and last chunks, it is same number of steps to find any element by index.

Q6

```
int *p = new int[10];
p++;
delete[] p;
```

Is above code safe?

A6

No, this code has undefined behavior. p++ increments p by 1, the pointer at address p+1 was not created by new[], so the statement delete[] p causes undefined behavior during runtime.

Q7

```
int main() {
  int x = 5;
  delete &x;
}
```

Is above code safe?

A7

No, code has undefined behavior. x is a stack variable not created with new, so calling delete on its pointer is undefined behavior typically in the form of a crash.

Q8

```
class Test {
public:
  static int x;
  char c;
```

```
};
int Test::x = 3;
Test t;
```

What is sizeof(t)?

A8

1. static data members don’t contribute to size of object.

Q9

```
class Test {
public:
Test(const Test& obj) {}
};
Test t;
```

Does above code work?

A9

No. Code will not compile. Because of user defined copy constructor, the compiler will not define the default constructor required for the Test t; statement.

Q10

What happens if exception is not caught in a C++ program?

```
void f() {
   throw 5;
}

int main() {
   f();
}
```

A10

Program will abort during runtime if exception is not caught anywhere in the call stack.

Q11

```
void f(string & s) {}
f("test");
```

Comment on above code

A11

The above code doesn't compile. In C++ a non-const reference cannot bind to a temporary object.

Q12

```
class Test {
public:
    Test(const Test& t) {}
};
```

Why the argument to copy constructor is const reference?

A12

const because you don't want to change the object from which copy is made. Alternative to pass by reference would be pass by value which is to make a copy, so that would invoke copy constructor again and again causing infinite recursion.

Q13

Can Destructor throw exception?

A13

Technically yes. But it is dangerous to do so. Because if stack unwinding is in progress for a previous exception and then an object's destructor throws another exception, its ambiguous which exception the C++ runtime should handle from a safety perspective, so the standard says to abort the program instead.

Q14

Can Constructor throw exception?

A14

Yes. In fact, it is good to signal the failure of constructor by throwing exception because of 2 reasons. 1) constructors don't have a return code unlike other functions 2) there will be no memory leak if exception is thrown by constructor (caveat being if constructor itself allocated memory, then use smart pointer to not cause memory leak)

Q15

```
class Test {};
Test t; // case 1
Test * tp = new Test(); // case 2
```

How to ensure Test objects are only created with new operator (case 2) but not on stack (case 1)?

A15

Make the destructor in class Test private. Now code in case 1 won't compile.

Q16

```
class Test{};
Test t; // case 1
Test * tp = new Test(); or new Test[10];// case 2
```

How to ensure Test objects cannot be created with new operators (case 2) but only on stack (case 1)?

A16

Make the operators new and new [] private in class Test. Now code in case 2 won't compile.

Q17

How are virtual functions implemented in C++?

A17

Typically implemented by most compilers with a virtual table aka vtable. Every class that has a virtual function will have a vtable constructed by the compiler which has function pointers to all virtual functions of the class. Each object of the class with a virtual function will have a hidden data member called virtual pointer aka vptr which points to the vtable of the class. If a virtual function is called, the C++ compiler won't resolve the function address during compile time, instead it puts the code to load the function following vptr into the vtable.

Example:

```
class B {
public:
    virtual void f() {}
    virtual void g() {}
};

class D : public B {
public:
    virtual void h() {}
    void f() {}
};
```

virtual table of B has:

```
&B::f
&B::g
```

virtual table of D has:

```
&B::g
&D::f
&D::h
```

Q18

```
class Test {};
Test * tp = new Test();
free(tp);
```

Is the above code safe?

A18

No, the above code can cause undefined behavior, potentially corrupting the heap. C++ standard expects to match the new/new []/malloc calls with delete/ delete [] / free calls

respectively.

Q19

```
class Base {
public:
    virtual void f() { cout << "Base::f()" << endl; }
};

class Derived : public Base {
private:
    void f() { cout << "Derived::f()" << endl; }
};

int main() {
   Base * p = new Derived();
   p->f();
}
```

Will code above compile? What will be output?

A19

Yes. Output is “Derived::f()“. Here the Derived::f() is private but access control is only checked during compile time. Since p is pointer of type Base and Base has public function f, the code will compile. Since function f() is a virtual function, dynamic dispatch during runtime is used to find the actual function f() that will be called.

Q20

Implement Singleton class that is safe to use in a multi-threading environment.

A20

// **Option 1** using std::call_once function to create Singleton object. Any threads trying to simultaneously execute getInstance method below will block until the first thread that started to execute the call_once() function will finish.

```
class Singleton {
public:
    static Singleton& getInstance();
    static int getCount() {
        return count;
    }
private:
    Singleton() {
        cout << "Constructor called" << endl;
        ++count;
    }
    Singleton(const Singleton& that) = delete;
    Singleton& operator=(const Singleton& that) = delete;
    int x;
    static std::once_flag m_onceFlag;
    static std::unique_ptr<Singleton> m_Instance;
    static int count;
};

int Singleton::count=0;
std::once_flag Singleton::m_onceFlag;
std::unique_ptr<Singleton> Singleton::m_Instance;

Singleton& Singleton::getInstance() {
    std::call_once(m_onceFlag, []{m_Instance.reset(new Singleton());});
    return *(m_Instance.get());
}
```

// **Option 2** - from C++ 11 onwards, other threads will block until the first thread that starts to execute the code to initialize the local static object finishes.

```
static Singleton* getSingletonInstance()
{
    static Singleton instance;
    return &instance;
}
```

Q21

Implement smart pointer class

A21

```
class RefCounter {
    RefCounter(const RefCounter&)=delete;
    RefCounter& operator=(const RefCounter&)=delete;
    int m_counter;
public:
    RefCounter() {
        m_counter=0;
    }

    int get() {
        return m_counter;
    }

    void operator++() {
        m_counter++;
    }

    void operator++(int) {
        m_counter++;
    }

    void operator--() {
        m_counter--;
    }

    void operator--(int) {
        m_counter--;
    }

};

template <typename T>
class SmartPtr {
    RefCounter* counter;
    T* obj;
public:
    SmartPtr(T* raw) {
        counter = new RefCounter();
        obj = raw;
        if (obj) (*counter)++;
```

```
    }

    SmartPtr(const SmartPtr& that) {
        counter = that.counter;
        (*counter)++;
        obj = that.obj;
    }

    SmartPtr& operator=(const Smartptr& that) {
        if (this != &that) {
            (*counter)--;
            if (counter->get() == 0) {
               delete counter;
               delete obj;
            }
            counter = that.counter;
            (*counter)++;
            obj = that.obj;
        }

        return *this;
    }

    int use_count() {
        return counter->get();
    }

    ~SmartPtr() {
        (*counter)--;
        if (counter->get() == 0) {
            delete counter;
            delete obj;
        }
    }
};
```

Q22

When is dynamic_cast useful?

A22

dynamic_cast is used on polymorphic classes. If we need to do downcast safely i.e. from a base class pointer to derived class pointer, dynamic_cast will do run time type checking and will signal error where the cast fails. If pointers are used in dynamic_cast and type cast fails, then null pointer is returned. In case of references, an exception is thrown.

Q23

What is CRTP? When is it useful?

A23

CRTP stands for Curiously Recurring Template Pattern. In CRTP, a class X is derived from a base class which is template specialization of class X itself.

```
template <class T>
class Base {...};

class X : public Base<X> {...};
```

There are many cases where we can take advantage of CRTP. One is to cut down runtime by avoiding runtime costs of virtual function with static polymorphism as in example below.

//**With Virtual Function**

```
class Interface {
public:
  virtual void process() = 0;
};
```

```
class Impl : public Interface {
public:
  virtual void process() {
    ...
  }
};

void do_work(Interface* obj) {
   obj->process();
}
Interface* iobj = new Impl();
do_work(iobj);
```

// above code implemented with CRTP, we avoid virtual function

```
template <typename Impl>
class Interface {
public:
  void process() {
    impl().process();
  }

private:
  Impl& impl() {
    return *static_cast<Impl*>(this);
  }
};

class Impl : public Interface<Impl> {
public:
  void process() {
    ...
  }
};

void do_work(Interface<Impl>* obj) {
   obj->process();
}

Interface<Impl>* iobj = new Impl();
do_work(iobj);
```

Q24

What is Proxy class? Give an example.

A24

Proxy class is used to implement Proxy design pattern in which an object serves as interface or mediator for another object.

Example: design an integer array class which only accepts 0 or 1 to be the values in the array. Exception should be thrown if we do arr[i] = 5 for example.

```
class ArrayProxy {
public:
    ArrayProxy(int& val) : elemPtr(&val) {}
    void operator = (int val) {
        if (val > 1 || val < 0) {
            throw "not 0 or 1";
        }
        *elemPtr = val;
    }
 private:
    int * elemPtr;
};

class Array {
private:
    int elemArray[10];
public:
    ArrayProxy operator[](int i) {
        return ArrayProxy(elemArray[i]);
    }
};

int main() {
    try {
        Array arr;
        arr[0] = 1;   // ok
        arr[0] = 5;  // throws exception
    }
    catch (const char * errmsg) {
        cout << errmsg << endl;
```

```
    }
}
```

Q25

How is placement new useful? Explain with an example.

A25

If we need to create multiple objects of a type during program execution but don't want to dynamically allocate memory each time, memory can be allocated at once initially based on max number of objects needed, then objects can be created directly in the pre-allocated memory slots. This is very helpful in critical systems if we don't want to have allocation failures. Memory pools can be implemented with help of placement new. Also once objects are destroyed, we can use the memory vacated to construct other objects.

```
class Test {};

// pre-allocated buffer for 10 Test objects
char *buf  = new char[10*sizeof(Test)];

// using placement new to place object directly in pre-allocated memory
Test *tp = new (buf) Test();
```

Q26

```
class Test {};

void foo(Test& t); // func1
void foo(Test&& t); // func2

Test t;
```

```
Test getTest() {
    Test t;
    return t;
}

int main() {
    foo(t); // case 1
    foo(getTest()); // case 2
}
```

Which version of foo function is called in case 1 and case 2 above?

A26

In case 1, func 1 is called because the argument t is a lvalue.

In case 2, func 2 is called because the argument is a temporary object returned by compiler as return value of the function getTest()

Q27

How is std::move(..) function useful?

A27

std::move(..) returns rvalue reference of an object which can be used with a move constructor or move assignment operator to improve performance when moving objects. The move constructor/assignment operator can be used to assign the moved object's internals to the new object since the moved object will be unusable once it is moved.

Q28

Explain copy elision with example

A28

Copy Elision is a compiler optimization technique that eliminates unnecessary copying or moving of objects.

```
class Test {
public:
  Test() {}
  ~Test() {}
  Test(const Test&) {}
};

Test foo() {
  return Test();
}

int main() {
   Test t = foo(); // line X
}
```

In above example in line X, if compiler does copy elision, the returned object is directly constructed in t. No copy constructor is invoked. From C++17, copy elision is guaranteed by the standard in cases like above.

Q29

Is std::shared_ptr thread safe?

A29

Depends on how it is used between threads. Multiple instances of shared_ptr used in different threads sharing ownership of the same object are thread safe. But single

instance of shared_ptr used among multiple threads, is not thread safe. Also thread safety for the managed object is not guaranteed.

Q30

```
vector<int> vec;
vec[0] = 1;
```

Why does above program crash?

A30

vector has no memory allocated on the heap yet.

```
vector<int> vec(10); // memory for 10 elements
vec[0] = 1; // fine now
```

Q31

Explain what is policy class with example

A31

Policy classes are used to inject behavior into the parent class.

For example,

```
template<class T, class Allocator = std::allocator<T>> class vector;
```

In above, the template parameter Allocator is a policy class that injects memory allocation/deallocation policy into class vector. If no Allocator template parameter is given, the default std::allocator<T> is used.

Q32

Compare performance of std::list and std::vector when iterating over all elements

A32

vector has better performance because elements in vector reside contiguously in memory giving better data locality. Data is fetched in size of the cache line and also the hardware cache pre-fetcher can recognize linear access patterns in vector and prefetch data in advance in direction of the iteration, which helps with better data locality. The list is not cache friendly for iterating over all elements because they reside in non-contiguous areas of memory causing a lot of data cache misses.

Q33

Are STL containers thread safe?

A33

No. STL containers are thread safe only when concurrent threads call the const member functions of the same container.

Q34

```
class Test {
    int y, x;
public:
    Test(int i) : x(i), y(x + 1) {
    }
};
```

Is above code safe?

A34

No. In class Test, data members will initialize in order they are declared in the class which is first y and then x. In the constructor when we first try to initialize y, x is not initialized yet which may have garbage value causing undefined behavior when the member y is used later in the program execution.

Q35

```
namespace X
{
    class A {};
    A operator+( const A& a, const A& b) {
        return a;
    }
}

int main() {
    X::A a, b;
    X::A c = a+b; // line N - will this line compile
    return 0;
}
```

In code above, will line N, compile? How will the compiler find operator+?

A35

Yes. operator+ that matches the arguments a and b is not in local scope, so compiler performs koenig lookup to look in namespaces where the argument types are defined, in this case X::A is defined in namespace X.

Q36

```
class Out {
    int x;
public:
    class In {
        int y;
    public:
        In() {y=0;}
        int getxy(Out* op) { return op->x+y; } // line a
    };

    Out() {x=0;}
    int getxy(In* ip) { return x + ip->y; } // line b
};

int main() {
    Out o;
    Out::In i;
    i.getxy(&o);
    o.getxy(&i);
    return 0;
}
```

Will above code compile?

A36

No. In line b, access to class In's private data member y is forbidden. Outer classes have to respect Inner classes privacy. However, Inner classes can access private data members of outer classes like in line a.

Q37

```
int x;

template<class T> void foo() {
    T::f * x;
}

struct X {
    typedef int f;
};
```

```
int main() {
    foo<X>();
    return 0;
}
```

Above code doesn't compile, how to fix it?

A37

In above template function foo, "T::f" is dependent name, which depends on the template parameter T. In above code we want to declare x as type int pointer.

If we pass a template parameter Y with definition below,

```
struct Y { static  int const f = 23; };
```

we are declaring a multiplication expression.

To avoid this ambiguity, the C++ standard mandates to explicitly state the intent when the name is to be treated as type in above code.

```
int x;

template<class T> void foo() {
    typename T::f * x; //added typename keyword to treat T::f as type
}

struct X {
    typedef int f;
};

int main() {
    foo<X>();
    return 0;
}
```

Q38

```
template<typename T, typename U>
void foo() {} // case 1

template<typename T>
void foo<char, T>() {}   // case 2
```

In above code, is case 2 a valid specialization of function template in case 1?

A38

No. Function templates cannot be partially specialized in C++. You will see a compiler error.

Q39

Can memory leak with shared_ptr?

A39

Yes, if there are circular references.

```
class X;
class Y;

class X { shared_ptr<Y> y; };

class Y { shared_ptr<X> x; };

shared_ptr<X> xo(new X);
xo->y = new Y;
xo->y->x = xo;
```

In above code, reference count of managed object X is 2. When xo goes out of scope, reference count becomes 1, not 0. therefore the managed object X doesn’t get destroyed

and memory leak happens.

The above can be fixed by using weak_ptr for one of the managed object like below.

```
class X { shared_ptr<Y> y; };
class Y { weak_ptr<X> x; };
shared_ptr<X> xo(new X);
xo->y = new Y;
xo->y->x = xo; // ref count won't increase here
```

In above code, reference count of managed object X is only 1, as weak_ptr doesn't increase ref count. So when xo goes out of scope, ref count becomes 0 and both X and Y are destroyed.

Q40

```
class Base {
public:
   Base() { memset(this, 0, sizeof(*this)); }
   virtual void foo() {...}
private:
   int i;
   float f;
};

class Derived : public Base {
public:
   Derived() { memset(this, 0, sizeof(*this)); }
};
```

Is it safe to use memset in constructor to initialize?

A40

No, it is potentially dangerous. The format of floats, doubles could be different messing up the internal data representation. Also, in above example, the vptr, i.e. the pointer to

virtual table could be corrupted. Essentially, using memset like this can corrupt the internals of different objects.

Q41

What is time complexity(big O) of vector push_back method?

A41

Pushing element to the end of vector push_back method naively seems like a O(1) operation. Big O time complexity is for worst case scenario. If the vector needs to grow because of this push operation, it needs to create new block of bigger memory and copy the elements there, making the push_back method worst case time complexity O(n).

Q42

```
vector<int> vec = {1,1,1,1,1,1,1,1,1,1};
vec.size(); //10
vec[0] = 2;
remove(vec.begin(), vec.end(), 2);
vec.size(); // ?? what is vec.size() here
```

what is vec.size() output after remove call?

A42

10 i.e didn't change. remove doesn't erase the elements from the vector. They will be pushed to the end.

Call erase on the iterator returned by remove call to actually delete the elements from vector.

```
vec.erase(remove(vec.begin(), vec.end(), 2), vec.end());
```

Now size of vec is 9.

Q43

Explain performance of virtual functions

A43

Virtual functions can be slower because:

1. First a load of vtable pointer is needed // potentially data cache miss
2. Then a load of virtual function pointer is needed // potentially data cache miss
3. Then code of function is called // potential instruction cache miss

In case of non virtual function, only impact comes from case 3.

Q44

Can virtual function be called in constructor?

A44

Yes, it can be called but virtual call mechanism is disabled inside the constructor. Because the base portion is constructed first and the derived portion is not constructed yet, there is nothing to be overridden.

Q45

Is virtual inline function legal in C++? Will it ever be inlined?

```
class Test {
public:
    virtual inline void foo() {...}
};
```

A45

Yes it is valid C++ code. However, virtual inline function code can be inlined only when the function is called with the object instead of pointer or reference. Because to inline, compiler needs to know the exact class during compile time, so it can find the function definition.

Q46

Difference between creating shared_ptr with make_shared and new

```
class Test {};
shared_ptr<Test> tp1 = make_shared<Test>(); // case1
shared_ptr<Test> tp2(new Test()); //case2
```

A46

make_shared performs just one heap allocation for both the managed object and control block (for reference counting etc.)

calling shared_ptr constructor in case2 does two heap allocations, one for managed object and another for control block

make_shared can prevent memory leak if exception thrown, for example in creating managing object

example:

```
void foo(shared_ptr<Test>& test1, shared_ptr<Test>& test2);
//call foo
foo(shared_ptr<Test>(new Test()), shared_ptr<Test>(new Test()));
```

In above, the arguments to foo can be evaluated as below four expressions in order as C++ standard doesn't define the order:

1. new Test() // for arg1
2. new Test() // for arg2
3. shared_ptr<Test> // for arg1
4. shared_ptr<Test> // for arg2

If exception is thrown in step 2, then there is memory leak of memory in step 1. Safe way to do this would be,

```
foo(make_shared<Test>(), make_shared<Test>());
```

Q47

```
template <class T>
class Test {
    template <class U>
    void foo(T&& t, U&& u);
};
```

Which of the T&& and U&& is forwarding reference?

A47

only U needs to be deduced, so U&& is the forwarding reference. T is already known while instantiating the template.

Q48

How to call inner lambda function below?

```
int myInt = 5;
auto lambda_outer = [&] { // outer lambda
  return [&myInt] { // inner lambda
    myInt = 10;
  };
};
```

A48

```
lambda_outer()();
```

Q49

when is raw pointer better to use over shared_ptr?

A49

In multi-threaded application, shared_ptr used in multiple threads will have performance penalty because of synchronization of common control block for reference counting. In case of raw pointer, there is no such penalty to be paid.

Q50

Which boost libraries did you use?

A50

Strong C++ developers are expected to be familiar with boost libraries at least, if not already used extensively.

boost has many advanced c++ libraries in multiple domains, some of which are now accepted into the C++ standard.

Some important boost libraries to know:

- Boost.Asio - library for network and low-level I/O
- Boost.Signals2 - event driven programming
- Boost.Thread - multithreading
- Boost.Test - unit testing

For more boost libraries check www.boost.org

2

Multithreading

Multi-threading is a very important topic for the trading systems developer interviews as trading systems related software are typically running multiple threads. Be comfortable with various types of locks, single producer single consumer queues(spsc), multiple producer multiple consumer queues(mpmc), deadlock, and thread-safety.

Q1

Difference between mutex and binary semaphore

A1

Mutex has exclusive ownership, only the owner can release the mutex. Semaphore can be signaled by any thread or process. Mutex is a locking mechanism for critical section while semaphore is a signaling mechanism.

Q2

Difference between thread safe function and re-entrant function

A2

A thread-safe function is one when executed by multiple threads concurrently should work correctly. Any shared data access(involving reads and writes) should be only one thread at a time to prevent race conditions.

A re-entrant function is typically used in the context of a single thread if the function can be interrupted in the middle of execution and its safe to call the same function again in the same thread. Example: the first invocation of function is only executed partially, then the second execution of the function is invoked and completed, and then the first invocation resumes and completed. Example: recursive function

Q3

Explain difference between deadlock and livelock

A3

Deadlock occurs when two tasks are waiting on resources from each other causing a circular dependency. Example: Task T1 has lock L1 and waits on lock L2. Task T2 has lock L2 and waits on lock L1, both of them get blocked and no work is done by them.

Livelock occurs when two tasks are trying to cooperate due to the need for the same shared resources but yet cannot make any progress. Example: In above example, to recover from deadlock, let's say task T1 releases L1 and task T2 releases L2 but then acquire in the same order as before i.e. T1 acquires L1 and T2 acquires L2, then they may end up in the same situation and keep repeating the same process without any real progress.

Q4

When to use spin lock

A4

If a thread tries to lock a mutex and the mutex is owned by another thread, the first thread will block and the OS will put the thread to sleep until the mutex is available to lock. This requires context switch i.e saving the thread's state and then waking the thread again when the mutex is available. In cases where high performance is required, the overhead from sleep/wake is not desirable, especially when the lock is expected to be available shortly. Spin lock will keep trying until the lock is available thereby avoiding this context switch. Particularly good in situations where the lock will eventually be available in a very short time.

Q5

When is thread pool useful

A5

If there are a large number of asynchronous tasks to execute, then with a thread pool, we can create a bounded number of threads and execute the tasks without incurring cost of creating and destroying a thread for each task OR simply just using a single thread with a longer response time.

Q6

Implement lockguard class i.e. uses a scoped mutex and locks/unlocks the mutex automatically

Assume mutex on your OS has the below interface.

```
struct mutex {
    void Lock() {}
    void Unlock() {}
};
```

A6

```
struct LockGuard {
    LockGuard(mutex& _mutex) : _mref(_mutex){
         _mref.Lock();
    }

    ~LockGuard() {
          _mref.Unlock();
     }

     private:
     LockGuard(const LockGuard&);
     LockGuard& operator=(const LockGuard&);

     mutex& _mref;
};
```

The above code uses RAII (resource acquisition is initialization) principle to acquire the mutex in the constructor and in the destructor unlocks the mutex when lockguard goes out of scope.

Q7

Implement thread safe queue with mutex and condition variable

A7

```
template <typename T>
class Queue
{
 public:

  void pop(T& item)
  {
    std::unique_lock<std::mutex> mlock(_m);
    while (_q.empty())
    {
      _c.wait(mlock); // wait until notified if queue is empty
    }
    item = _q.front();
    _q.pop();
  }

  void push(const T& item)
  {
    std::unique_lock<std::mutex> mlock(_m);
    _q.push(item);
    mlock.unlock();
    _c.notify_one(); // notify one of the waiting threads
  }

 private:
  std::queue<T> _q;
  std::mutex _m; // for locking critical section
  std::condition_variable _c; // for notifying and waiting
};
```

Q8

Implement read write lock with priority for waiting writers

A8

```
class RWLock {
  std::mutex mutex;
  std::condition_variable waitingReaders;
```

```
  std::condition_variable waitingWriters;
  int readers = 0;
  int queuedWriters = 0;
  bool writer = false;

public:
  void readLock() {
    std::unique_lock<std::mutex> lock(mutex);
    waitingReaders.wait(lock, [this]{!writer && queuedWriters == 0});
    ++readers;
  }

  void readUnlock() {
    std::unique_lock<std::mutex> lock(mutex);
    --readers;
    if (readers == 0 && queuedWriters > 0) {
      lock.unlock();
      waitingWriters.notify_one();
    }
  }

  void writeLock() {
    std::unique_lock<std::mutex> lock(mutex);
    ++queuedWriters;
    waitingWriters.wait(lock, [this]{!writer && readers == 0});
    --queuedWriters;
    writer = true;
  }

  void writeUnlock() {
    std::unique_lock<std::mutex> lock(mutex);

    if(queuedWriters > 0) {
        lock.unlock();
        waitingWriters.notify_one();
    } else {
        writer = false;
        lock.unlock();
        waitingReaders.notify_all();
    }
  }
};
```

Q9

How to prevent deadlock - Thread1 and Thread2 both need Locks L1 and L2

A9

Make the threads acquire locks in same order, for example, acquire L1 first and then L2

Q10

Difference between std::this_thread::yield and std::this_thread::sleep_for

A10

yield is a hint to the implementation to reschedule the execution of calling thread allowing other threads to run. sleep_for will block the calling thread atleast until the duration mentioned

Additional Questions to Think About:

1. How will you test a multiple producers multiple consumers threadsafe queue?

3

Lock-free Programming

Trading systems requiring low latency use lockless queues to communicate between threads to avoid the blocking nature of locks. Developers are expected to understand in depth about atomics, memory model, and memory ordering. For an interview, knowing the single producer single consumer bounded lockless queue implementation is a must while mpmc (multiple producer multiple consumer) lockless queue could be discussed for senior engineers especially if they claim to have worked on them.

Q1

What does it mean to say a variable is atomic?

```
std::atomic<T> x;
```

A1

An atomic variable is one which can be read or modified by different threads as a whole. For example, a change to variable involves steps - load, update and store. Other threads won't see the change in the variable in middle of those steps.

Q2

What are the uses of data alignment using alignas keyword?

A2

With alignment using alignas keyword, we can make sure data used by different threads fall on different cachelines to avoid false sharing.

Q3

Explain C++ memory orders - relaxed ordering, release-acquire ordering, sequentially consistent ordering

A3

relaxed ordering - Atomic operations tagged memory_order_relaxed are not synchronization operations; they do not impose an order among concurrent memory accesses. They only guarantee atomicity and modification order consistency.

release-acquire ordering - If an atomic store in thread A is tagged memory_order_release and an atomic load in thread B from the same variable is tagged memory_order_acquire, all memory writes (non-atomic and relaxed atomic) that *happened-before* the atomic store from the point of view of thread A, become *visible side-effects* in thread B. That is, once the atomic load is completed, thread B is guaranteed to see everything thread A wrote to memory. The synchronization is established only between the threads *releasing* and *acquiring* the same atomic variable. Other threads can see different order of memory accesses than either or both of the synchronized threads.

sequentially consistent ordering - For atomic operations tagged memory_order_seq_cst, a load operation with this memory order performs an *acquire operation*, a store performs a *release operation*, and read-modify-write performs both an *acquire operation* and a *release operation*, plus a single total order exists in which all threads observe all modifications in the same order

Q4

Difference between atomic_compare_exchange_weak and atomic_compare_exchange_strong

A4

In atomic_compare_exchange_weak spurious failures is allowed i.e. even if contained value and expected value match still the exchange won't happen. In strong version, spurious failure is not allowed.

Q5

Implement single producer single consumer bounded lockless queue. Focus on the push and pop methods.

A5

```
template <typename T>
class SPSCQueue {
public:
    // give queue capacity+1 size so we can distinguish
    // queue empty from queue full
    // queue empty: mReadPos == mWritePos
    // queue full: mReadPos == (mWritePos+1) % queue_size
    SPSCQueue(int capacity)
```

```
        : mBuffer(capacity + 1), mReadPos(0),mWritePos(0) {
        assert(capacity > 0);
        assert(capacity + 1 > 0); // Protect from integer overflow
    }

    // Queue Full - mReadPos == increment(mWritePos)
    bool push_back(const T &item) {
        const int r = mReadPos.load(std::memory_order_acquire);
        const int w = mWritePos.load(std::memory_order_relaxed);

        //check next write position to see if queue is full
        const int next_w = increment(w);
        if (r == next_w) return false;

        // store item if not full
        mBuffer[w] = item;
        mWritePos.store(next_w, std::memory_order_release);
        return true;
    }

   // queue empty: mReadPos == mWritePos
    bool pop_front(T &item) {
        int r = mReadPos.load(std::memory_order_relaxed);
        int w = mWritePos.load(std::memory_order_acquire);

        // Queue is empty (or was empty when we checked)
        if (r == w) return false;

        item = mBuffer[r];
        mReadPos.store(increment(r), std::memory_order_release);
        return true;
    }
private:
    inline int increment(int pos) const {
        return (pos + 1) % int(mBuffer.size());
    }

    alignas(64) std::atomic<int> mReadPos; //cacheline size 64 bytes
    alignas(64) std::atomic<int> mWritePos;
    char _padding[64 - sizeof(mWritePos)];
    std::vector<T> mBuffer;
};
```

Recommended Reading:

Multiple Producer Multiple Consumer Lockless Queue

https://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++.htm

4

OS & IPC (Inter-Process Communication)

Shared memory is the most common topic in IPC for the interviews. It is used for in-memory data persistence by processes(to not lose some crucial data if process has to restart) and also as underlying memory for shared queues between processes(for faster data sharing between processes).

Q1

What happens to the shared memory segment when the last process attached to it dies unexpectedly?

A1

The shared memory segment will persist. It must be explicitly deleted with shmctl.

Q2

Why is Shared Memory, the fastest form of IPC?

A2

Because data is not copied from one process address space to another process address space, the downside being synchronization is now up to the processes using the shared memory.

Q3

Can we increase shared memory segment size during runtime as needed (in Linux)?

A3

No, shared memory segment once created is fixed in size.

Q4

What is the advantage of using mmap to share memory between two processes? What happens if one process writes to that memory?

A4

The primary advantage being as long as the processes are doing read-only operations on the mmap memory, they all use the same underlying physical memory region, thus lowering memory usage. If one process writes, then a separate copy is made for that process, similar to a copy on write strategy.

Q5

Why is virtual memory needed?

A5

A couple of reasons below.

With virtual memory, every process has access to large addressable space because the OS can translate the virtual memory address to physical memory address and give more pages of memory to the process by saving some other pages to disk.

With virtual memory, one process cannot access the address space of another process because virtual memory system as an intermediate address translation module ensures process can only access physical memory pages of itself.

Virtual memory enables us to run multiple programs simultaneously. Compiled binaries have functions at fixed addresses in memory, if virtual memory didn't exist, two programs could not be loaded and run at the same time, because they may need different functions at the same physical address.

Q6

What is the difference between the Application Binary Interface (ABI) and Application Programming Interface (API)?

A6

API defines the objects and methods that a module makes available to its clients at the source code level. Things like how to instantiate an object, the arguments and return type of a method, etc.

ABI defines how the bits and bytes are passed at a lower level between the module and its clients. Things like, is the data pushed to registers or stack, etc., who will pop the stack after a function call returns, etc.

5

Network Programming

You are expected to know how TCP, UDP protocols, sockets, and IP Multicast work under the hood. Understand how non-blocking sockets work and help in designing event-driven systems.

Q1

Explain TCP handshake

A1

As per TCP protocol, a TCP connection between two endpoints is established with a 3-way handshake. To establish a TCP connection between A(originator) and B, A sends SYN packet to B, B responds with SYN-ACK packet to A and finally, A responds with ACK packet to B.

Q2

How can the TCP server find the status of client connections that are dead without notifying the server and close them?

A2

One way to do is using the TCP “keepalive” flag. With this flag, a timer is set up by TCP and when the timer expires, the server-side will send “keepalive” probe packets to the clients and if no reply is received in the configured duration, the connection can be considered dead and associated resources can be released.

Q3

How does TCP deal with fast sender and slow receiver?

A3

TCP has a mechanism called Flow Control that is initiated from the receiving side. It ensures the sender cannot flood the receiver. When receiver gets data packet from sender, it will reply an ACK packet to the sender which also contains a field that tells how much capacity is available in the receiver’s buffer, which serves as a hint and feedback mechanism for the sender to not flood the receiver.

Q4

What is MTU?

A4

MTU is the Maximum Transmission Unit, the maximum IP packet size that can travel through a given link. Packets bigger than MTU of a link are fragmented typically by the router and reassembled at the destination host.

Q5

Compare TCP and UDP

A5

TCP is a connection-oriented protocol. TCP protocol ensures packets arrive in the correct order without loss at the receiving endpoint.

UDP is a connectionless protocol. In UDP, packets can arrive out of order, and also packets may be lost. Due to no guarantees on data delivery, UDP is faster than TCP.

Q6

How do clients join the IP Multicast stream?

A6

Clients interested in joining IP Multicast stream send IGMP message. Routers remember the clients interested in the multicast stream based on this message. When the data packet of a multicast IP stream reaches the router, the router will send the data to the clients that expressed interest with the IGMP message.

Q7

What is the advantage of using non-blocking sockets?

A7

non-blocking sockets do not put the thread calling the send/recv operations on socket to sleep if no data is available to be sent/read. With non-blocking sockets, a thread can do some other useful work instead of going to sleep. This is better approach than using select system call with timeout and blocking sockets because sometimes select system call can return for socket being ready but the read/write can block for blocking socket as sometimes the packet may be discarded for various reasons.

Q8

Compare select and poll system calls

A8

With select system call, even if we are monitoring only a few file descriptors, we need to loop the number of file descriptors up to the value of maximum file descriptor. Also, the bitmasks used to mark the file descriptors for interested activity will get overwritten by the kernel after each select call, so we need to reset the bitmasks to original values before calling select in a loop.

With poll system call, we only need a number of structures equal to the number of file descriptors monitored and these structures can be reused.

Additional Reading

- **pselect** https://linux.die.net/man/2/pselect

- **epoll** on Linux and level triggered vs edge triggered modes
  http://man7.org/linux/man-pages/man7/epoll.7.html

6

Low Latency Programming & Techniques

You should know the common techniques used to achieve low latency. Some common techniques are understanding how CPU caches work, kernel bypass, non-blocking programming, compiler optimizations, underlying hardware (CPU pipelines, FPGA etc.), efficient programming techniques using algorithms & data structures, etc.

Things to know about CPU Caches

- **Cacheline -** Data to/from main memory from/to caches is always copied in size of cacheline, example: CPU sends request for an integer variable(4 to 8 bytes), but the whole cacheline is fetched (64 bytes)
- **Types of Caches -** Instruction Cache(for program instructions), Data Cache(for data used by program), TLB(caching for virtual memory address to physical memory address translation)
- **Cache Hierarchy -high speed to low speed -** CPU registers -> L1 Cache -> L2 Cache -> L3 Cache -> Main Memory (for i7 processor)
- **Cache Coherency** - Cache Coherence deals with the uniformity of data between different caches and main memory
- **Temporal Locality** - data being used recently in CPU will likely be referenced again in near future

- **Spatial Locality** - if data at a certain location in memory is used recently in CPU, it is likely data in memory locations nearby that location will be referenced in near future
- **Cache Warming** - when data is referenced first in CPU, it is brought from main memory to cache. Cache warming is the technique to bring the anticipated data from main memory into cache in advance to reduce cache misses
- **CPU pinning** - set processor affinity for process or related processes so they all can share the data faster via same caches

Q1

```
// snippet1 - row major order
std::vector<double> v( n * n );
double sum{ 0.0 };

for (std::size_t i = 0; i < n; i++) {
    for (std::size_t j = 0; j < n; j++) {
        sum += v[i * n + j];
    }
}

// snippet2 - column major order
std::vector<double> v( n * n );
double sum{ 0.0 };

for (std::size_t j = 0; j < n; j++) {
    for (std::size_t i = 0; i < n; i++) {
        sum += v[i * n + j];
    }
}
```

Which of the above code is faster snippet1 or snippet2?

A1

snippet1. Code performs better with row major access because data is laid out in main memory linearly in row major order. Since caches fetch data in cache lines and also do

linear prefetching in the direction of access, there are lower cache misses when traversed in row major order.

Q2

```
struct Data {
  char c;
  char d;
};

Data data;

// function that thread1 runs
void thread1_func() {
   int sum=0;
   for (int i=0; i < 1000000000; ++i) {
       sum += data.c;
   }
}

// function that thread2 runs
void thread2_func() {
    for (int i=0; i < 1000000000; ++i) {
        data.d = i%256;
    }
}

Assume thread1 and thread2 started around same time and running concurrently. Both threads are
working on different members of data object.
How can we improve performance of thread1_func()?
```

A2

Because most likely data members c and d(size one byte each) fall on same cache line(typically 64 bytes on most processors), so updates to member d from thread2 will invalidate the cache line making thread1 refetch member c again and again from main memory, although thread1 is not updating data object. It can be fixed by using alignas function to place data members c and d on separate cache lines.

Q3

```
long long sum = 0;

// same thread function run by 3 threads
void sumUp(){
    std::lock_guard<std::mutex> lockGuard(myMutex);
    for (int i=0; i<10000; ++i) {
        sum += i;
    }
}

If 3 threads are started to concurrently run the above function, how can we improve the
performance?
```

A3

In the above code, the thread that gets the lock will do the computation until then other threads will block.

Consider code below,

```
long long sum = 0;

// same thread function run by 3 threads
void sumUp(){
    long long tmp=0;
    for (int i=0; i<10000; ++i) {
        tmp += i;
    }

    std::lock_guard<std::mutex> lockGuard(myMutex);
    sum += tmp;
}
```

Here, we did the computation locally before adding to global sum there by reducing the critical section, which will ensure the 3 threads will finish faster than the code in the question.

About kernel bypass

Kernel bypass is a technique to manage in the user space the network stack and hardware to avoid going through the OS. This will improve performance because there is zero-copy as all data is in the user space.

About avoiding OS locks like mutex etc.

For threads synchronization at the hardware level instead of using OS locks like mutex etc., we can use atomic operations tagged with appropriate memory ordering for synchronization.

About non-blocking programming

If application gets blocked, it goes to sleep and then needs to be woken up when the resource is available. With non-blocking approach, the application can do other work or just spin for short duration until it gets some work to do, this avoids all the latency from context switching.

For low latency, use non blocking I/O (sockets for example) and design your app in an event driven way using asynchronous I/O.

About compiler optimizations

Give compiler hints if you know a particular branch is more likely using _builtin_expect. This will generate code that will take advantage of CPU pipelines.

```
if (x) {
    foo();
    ...
```

```
}
else {
    bar();
    ....
}
```

Let’s say in above code x is most likely to be false, we can write code like below.

```
if (__builtin_expect(x, 0)) {
    foo();
    ...
}
else {
    bar();
     ...
}
```

Above code will then generate assembly something like:

```
cmp   $x, 0
  jne   _foo
_bar:
  call  bar
  ...
  jmp   after_if
_foo:
  call  foo
  ...
after_if:
```

As opposed to the C++ code, above we can see bar case precedes foo case. Since foo case is unlikely, and instructions of bar are pushed to the pipeline, thrashing the pipeline is unlikely. This is a good exploitation of a modern CPU which has long pipeline.

About efficient programming techniques

- Avoid dynamic memory allocations

- Representing working data compactly ensures data fits better in cache
- Using bitwise operations is faster. example: division by 2, n >> 1 is faster than n/2 operation
- Integer operations are faster than floating point operations. fixed precision floating point numbers for example stock prices with upto 4 decimals, can be represented as integers by scaling factor of 4 and operating on them as integers
- For smaller data set lookups, linear search will perform better than binary search due to cache prefetching.
- See if your compiler can do vectorization where possible

About CPU branch prediction

Q4

Which of the following code is faster ? Why?

```
// case 1
int main(){
    const unsigned arraySize = 32768;
    int data[arraySize];
    for (unsigned c = 0; c < arraySize; ++c)
         data[c] = std::rand() % 256;

    // Test
    clock_t start = clock();
    long long sum = 0;
    for (unsigned i = 0; i < 100000; ++i){
        // Primary loop
        for (unsigned c = 0; c < arraySize; ++c){
            if (data[c] >= 128)
                sum += data[c];
        }
    }

    double elapsedTime =
       static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
}
```

```
// case 2
int main(){
    const unsigned arraySize = 32768;
    int data[arraySize];
    for (unsigned c = 0; c < arraySize; ++c)
         data[c] = std::rand() % 256;

    // ONLY DIFFERENCE is this line between the two programs
    std::sort(data, data + arraySize);

    // Test
    clock_t start = clock();
    long long sum = 0;
    for (unsigned i = 0; i < 100000; ++i){
        // Primary loop
        for (unsigned c = 0; c < arraySize; ++c){
            if (data[c] >= 128)
                sum += data[c];
        }
    }

    double elapsedTime =
       static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
}
```

A4

Modern CPUs have longer pipelines and execute instructions ahead in advance based on branch predictor. Branch predictor typically predicts based on previous history of the branch. If the branch predictor is wrong, CPU needs to rollback the computation and refetch the correct instructions which will cause delay at CPU level.

case 2 is much faster because it aligns well with the CPU branch predictor. To understand, look at this code snippet below from above programs.

```
if (data[c] >= 128)
    sum += data[c];
```

When data is unsorted, the number of branch mispredictions are higher and random while it is a constant if data is sorted.

Miscellaneous Low Latency Techniques

Colocation: Financial trading firms typically run their servers in the same data centers of the exchanges to reduce data transmission latency due to physical distance. This is called colocation.

FPGA: Some ultra low latency firms use specialized programmable hardware like FPGAs to reduce latency. FPGA will bypass CPU for that particular programmed case and will be significantly faster and power efficient. One such common use case is for handling market data feed which is high volume of data coming in a short time.

Additional Things to know about:

- Kernel/OS tuning
- TCP/IP tuning
- Choosing the correct Network Interface Cards for your network intensive application

7

Systems Design

Some form of systems design interview has become a must in any tech interview for a software developer. Typical topics asked in trading systems engineer interviews are related to the systems they are working on. Answers are expected at a high level or low level depending on the background. Topics to focus on - distributed systems, partitioning data, sharding, load balancing, efficient data structure design, algorithm design for large data sets, etc. For experienced developers from the same industry, questions on the design of trading systems related components like order book, matching engine, feed handler, memory pool, etc. may be asked.

Q1

You are building a stock exchange matching engine that supports trading of 5000 different stocks. There could be few millions of orders per stock on each trading day. How do you go about designing this system for scale?

A1

Every stock has a unique identifier - stock symbol. Also every stock's order book is completely independent of any other stock's order book. Based on the historical and predicted volumes for each stock, and the CPU/Memory capacity of each matching engine machine, we can run multiple instances of the matching engine software by horizontally partitioning the stocks onto different machines to distribute the load across several machines. Example: stocks with symbols starting with A-B trade on machine 1, C-F machine 2, G-P machine 3, etc. This design will horizontally scale and easy to upgrade if we need to reconfigure the partitioning.

Q2

Design data structures for fast limit order book so that New Order, Trade, Cancel Order, Get Best Bid/Offer are as fast as possible.

A2

```
// Order book will be represented by two binary search trees of
// Limit objects, one for buy-side and another for sell-side

// lowestSell and highestBuy need to be updated with updates to the
// binary tree to keep them at O(1) lookup

// order book
struct OrderBook {
  Limit *buyTree; // buy-side tree
  Limit *sellTree; // sell-side tree
  Limit *lowestSell; // points to best offer
  Limit *highestBuy; // points to best bid
};

// representing a single limit price either on buy-side or sell-side
// each limit object will have doubly linked list of orders
// at that price

struct Limit {
  int limitPrice;
  int size; // number of shares at this limit price and side
```

```
  Limit *parent; // bst parent node for this node
  Limit *leftChild; // bst left child
  Limit *rightChild; // bst right child
  Order *headOrder; // head of linked list at each limit price orders
  Order *tailOrder; // tail
};

//There will be one order book per symbol, so we don't need symbol information in the data
structures
//represents a single order

struct Order {
  int orderId;
  bool buyOrSell;
  int numShares;
  int limitPrice; // scaled price
  Order *nextOrder;
  Order *prevOrder;
  Limit *parentLimit; // limit object in binary tree it belongs to
}
```

Apart from the above data structures we create two maps such that each Order is an entry in a map with key as orderId, and each Limit is an entry in a map with key as limitPrice.

We can now achieve time complexities as below for different operations.

```
Let M be the number of limit objects in a buy or sell tree
New Order – O(log M) for the first order at a limit price, O(1) for all others
Explanation: if no limit price on order exists, we need to add a node to tree, otherwise, we
just go to that limit object node via map and add to the doubly linked list

Cancel Order – O(1)
// lookup from the map with order id

Trade – O(1)
// lookup from the map with buy order id and sell order id

GetVolumeAtLimitPrice – O(1) // lookup from map with key limit price

GetBestBid/Offer – O(1) // using lowestSell and highestBuy fields in OrderBook structure
defined above
```

Q3

Very large files size (size at multiple TB for example) need to be transferred from data center A to data center B, each located to far off geographically from each other. What are your thoughts on design of file transfer mechanism for such large files?

A3

If the file is transferred as a whole and if there are network issues after a file transferred partially, then we need to re-transmit the large file again from the beginning. We can overcome this by breaking the original file into small parts at the source and then reassemble at the destination without losing the data integrity. To do this, we can take the MD5 sum of the original file, split the large file into parts and take their MD5 sums and transmit the file parts and the file containing MD5 sums to the destination machine. At destination host, we verify MD5 sums of each of the parts. We can then assemble the large file from parts and verify the MD5 sum of the original file. We can also take advantage of transmitting the file's parts parallelly if there is enough bandwidth.

Q4

You have to design a system where the users can send automated messages using an API to your system. How do you scale the system to millions of users?

A4

Give every authorized user an API key that identifies the user uniquely. When the user sends messages, the API key will be a field in the message. Now, we can partition the

users onto multiple app servers based on a hash function that takes the user API key as the input. With this strategy, we can easily scale horizontally. Based on the message from the user, we can look at the API key and compute the hash and go to the app server that will process the message of this user.

Q5

There is a stream of integers coming into your app in the range [x, y] (x and y are integers, 1 <= y-x <= 1000). You may get many millions of integers in the stream to your app in that range. How can we efficiently calculate the median of the stream seen so far at any moment? Assume you have only 1MB memory for your app.

A5

Create a global_counter variable that keeps count of the number of integers seen so far.

Create a counter for each number in the range [x, y] and increment the corresponding counter as each input integer seen. Example, if x+1 integer is seen in stream 3 times so far, counter[x+1]. will have a value of 3.

If we have all the input integers seen so far in a sorted manner, to calculate the median, we need to find the element at index (global_counter+1)/2 if the global_counter is odd, and the average of elements at indices global_counter/2 and global_counter/2+1 if global_counter is even.

We do not need to sort the input seen so far, instead we can take advantage that counters for each of [x, y] if traversed from x to y, we are traversing in a sorted manner. Starting from counter of x, we can traverse towards counter of y in the order to find the intended elements.

Example, x=2, y=100; stream is 2, 3, 4, 3,3 then counter[2]=1, counter[3]=3, counter[4]=1, global_count=5, so median is element 3. to locate element 3, start with first bucket, bucket[2] has 1 element, next bucket[3] has 3 elements, so element 3 falls in bucket[3], hence median is 3.

Q6

```
Let MAX_POSITIVE_INT be the maximum positive integer on the system. Assume integer is 4 bytes
(32 bits)

int nums[MAX_POSITIVE_INT];
```

You are given array nums above which contains all positive integers (in range 1 to MAX_POSITIVE_INT) some of which may be duplicates. Design an algorithm to query if a given positive number exists in nums. There will be multiple queries with different inputs, inputs being positive integers. You can modify the array nums if needed and make your solution run time and memory efficient.

A6

pre-processing - iterate through all elements of nums and for each element multiply the number at nums[element value - 1] with -1 to make negative. example: if nums[0] is 253 we will make element at index 252 multiplied by -1. i.e element at index 0, will mark presence of 1, element at index 5 will mark presence of 6 etc.

```
for (int i=0; i < MAX_POSITIVE_INT; ++i) {
    int tmp = abs(nums[i]);
  // take care of duplicates by not multiplying more than once
    if (nums[tmp-1] > 0) nums[tmp-1] *= -1;
}
```

To answer the Query - check if the element at input number -1 index. If it is negative that element is somewhere in the nums array otherwise it is missing.

8

Design Patterns

Commonly the question is about what design patterns you have used in the past. It is good to prepare in advance which design patterns were used in your past projects and the rationale for using them.

Some design patterns to know:

- Singleton
- Factory
- Messaging Pattern
- Producer Consumer
- Actor Model
- Reactor Pattern
- Event Driven Architecture
- Thread Pools
- Observer
- Strategy

If you are not confident about design patterns, I recommend you get the book “Head First Design Patterns” or go through “Design Patterns” on https://refactoring.guru/

9

Coding Interview

Tech Coding Interviews have become harder these days in general in the software industry. Everyone looking for a programming job at least in the top companies are expected to face algorithm/problem solving based coding questions. It is common for software developers to practice coding questions on websites like leetcode.com, hackerrank.com, geeks4geeks, etc. before attending any job interview these days.

For the trading systems developer interviews, the coding questions are typically on -

- linked data structures like linked list, binary tree, trie
- general algorithm/problem-solving questions using hash tables, stacks, queues, and priority queues
- questions on arrays/strings related to searching, sorting, etc.
- Bit Manipulation comes up some times because they are a common technique in writing low latency code
- Hard Dynamic Programming problems are not common, but I recommend to be familiar with some common DP problems of easy/medium difficulty
- Hard Graph problems are not common, I recommend solving some graph problems using BFS(breadth-first search)/DFS(depth-first search) of medium difficulty

The above topics should be sufficient for most of the companies. If you are going for interviews at top tier companies like Citadel, Hudson River Trading, Two Sigma, etc. it is better to be comfortable with Leetcode.com Medium level problems or with the problems in Elements of Programming Interviews / Cracking the coding interview books.

Q1

Your application accepts orders from customers and sends them into a trading engine. We want to throttle customer messages if they exceed a threshold in the given trailing time interval. We allow only N messages in the trailing T microseconds. If a message gets rejected because it got throttled, it doesn't count toward the limit. Please note that the streaming nature of this input and structure your program accordingly.

```
Input format:
The first line contains 2 space separated integers denoting N and T respectively. The remaining
lines contain the (microsecond) timestamps of arrival times of messages. The timestamps on any
given row would be greater than or equal to the timestamp from the previous row.

Output format:
For each of the timestamp rows, print the timestamp of the message and a pass or fail separated
by space.

Sample Input:

5 20
1547222542000010
1547222542000010
1547222542000012
1547222542000014
1547222542000014
1547222542000014
1547222542000015
1547222542000020
1547222542000037
1547222542000039

Sample Output:
```

```
1547222542000010 pass
1547222542000010 pass
1547222542000012 pass
1547222542000014 pass
1547222542000014 pass
1547222542000014 fail
1547222542000015 fail
1547222542000020 fail
1547222542000037 pass
1547222542000039 pass
```

A1

Here we need to allow only N messages in last T micro seconds. So, we need some data structure that can let us add new messages on one end, and let us delete the old messages from the other end i.e. those arrived more than T micro seconds ago. In C++, we have std::deque that is much more efficient than doubly linked list for this purpose.

Each time we get a message, we will remove the expired messages if any and then try to insert the new message into the deque only if the deque size is less than N.

The RateLimiter class holds the logic for above data structure and algorithm while the main program below that used the RateLimiter class to demonstrate the use of RateLimiter class on the input mentioned in this problem.

Rate Limiter class

```
#include <deque>

class RateLimiter {
public:
    RateLimiter(int N, int T) : _numMsgs(N), _timeWindow(T) {
    }

    bool isRateLimiterOK(uint64_t ts) {
        if (_dq.empty()) {
            _dq.push_front(ts);
```

```
            return true;
        }

        while (!_dq.empty() && (ts - _dq.back() >= _timeWindow)) {
            _dq.pop_back();
        }

        if (_dq.size() < _numMsgs) {
            _dq.push_front(ts);
            return true;
        }

        return false;
    }

    int getCountInWindow() {
        return _dq.size();
    }

private:
    std::deque<uint64_t> _dq;
    int _numMsgs;
    int _timeWindow;
};
```

Main program that processes input and applies the Rate Limiter class

```
#include <iostream>
#include <string>
#include <fstream>
#include <limits>
#include <cstdlib>
#include <cerrno>
using namespace std;

#include "RateLimiter.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        cout << "Usage: ./throttler <input-file-path>" << endl;
        exit(1);
    }

    ifstream inp(argv[1]);
    if (!inp.is_open()) {
        cout << "Unable to open input file " << argv[1] << endl;
        exit(1);
    }
```

```
    int N=0, T=0;
    if (inp >> N >> T)
    {
        inp.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    }
    else {
        cout << "First line format wrong in input file" << endl;
        exit(1);
    }

    RateLimiter rl(N, T);
    string line;
    uint64_t ts=0;
    while (getline(inp, line)) {
        //ignore empty lines
        if (line.size() == 0) continue;
        ts = strtoull(line.c_str(), NULL, 10);
        if (ts == 0 || errno == ERANGE) {
           errno = 0;
           cout << "Warning: timestamp out of range or could not be              converted to
uint64_t for " << line << "\n";
           continue;
        }

        if (rl.isRateLimiterOK(ts)) {
            cout << ts << " pass\n";
        }
        else {
            cout << ts << " fail\n";
        }
    }

    cout.flush();
    return 0;
}
```

Q2

Implement insert/search functions of Trie (pre-fix tree)

A2

```
class Trie {
    struct Node {
        Node* child[26];
        bool isWord;
        Node()  {
            for (int i=0; i < 26; i++) {
                child[i] = nullptr;
            }
            isWord = false;
        }
    };
    Node * root;
public:
    Trie() {
        root = new Node();
    }

    void insert(string word) {
        Node * curr = root;
        for (auto ch: word) {
            if (curr->child[ch-'a']) {
                curr = curr->child[ch-'a'];
                continue;
            }
            curr->child[ch-'a'] = new Node();
            curr = curr->child[ch-'a'];
        }
        curr->isWord = true;
    }

    bool search(string word) {
        Node * curr = root;
        for (auto ch: word) {
            if (!curr->child[ch-'a']) {
                return false;
            }
            curr = curr->child[ch-'a'];
        }
        return curr->isWord ? true : false;
    }
};
```

Q3

Say you have an array, **A**, for which the **i**th element is the price of a given stock on day **i**.

If you were only permitted to complete at most one transaction (i.e, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

You cannot sell a stock before you buy one.

Example:

```
Input: [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
```

A3

As we process the array, we will track the minimum price seen so far and the max profit that is possible so far.

```
int maxProfit(vector<int>& prices) {
        int minSoFar = numeric_limits<int>::max();
        int maxProfit = 0;

        for (auto & p: prices) {
            maxProfit = max(maxProfit, p-minSoFar);
            minSoFar = min(minSoFar, p);
        }

        return maxProfit;
    }
```

Q4

Say you have an array for which the i^{th} element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most *two* transactions.

You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example:

```
Input: [3,3,5,0,0,3,1,4]
Output: 6
Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.
Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.
```

A4

The idea here is that given the prices from days 1 to N for N days, in Step 1, we compute on each day i, 1<= i<= N, similar to problem 3 above, we compute the max profit assuming only one trade. Then in step 2, we compute backwards for each day i, from N to 1, such that max profit at day i will be max profit possible assuming only prices from days i to N exist. Then we record a global max profit for each day i from N to 1, which combines max profit at day i from step 2 with max profit at day i-1 from step 1.

```
int maxProfit(vector<int>& prices) {
        int min_so_far = numeric_limits<int>::max();
        vector<int> profit_first_trade(prices.size(), 0);
        int max_profit=0;

        //step 1 from above explanation
        for (int i=0; i < prices.size(); ++i) {
            min_so_far = min(min_so_far, prices[i]);
            max_profit = max(max_profit, prices[i]-min_so_far);
            profit_first_trade[i] = max_profit;
        }

        int max_so_far = numeric_limits<int>::min();
        // steps 2 and 3 combined from above explanation
        for (int i=prices.size()-1; i>0; --i) {
```

```
            max_so_far = max(prices[i], max_so_far);
            max_profit =
            max(
             max_profit, max_so_far-prices[i]+profit_first_trade[i-1]
            );
        }

        return max_profit;
    }
```

Q5

Reverse bits of a 32-bit unsigned integer

A5

Here we start with two indices, one at first bit and another at last bit and if the bits there are different, we swap and then move inwards to swap the next pair of bits if needed.

```
uint32_t reverseBits(uint32_t n) {
       for (int i=31,j=0; i > j; --i,++j) {
           bool iset = (n & (1<<i)) != 0;
           bool jset = (n & (1<<j)) != 0;
           if ((iset && !jset) || (!iset && jset)) {
               n = (n ^ (1<<i));
               n = (n ^ (1<<j));
           }
       }

       return n;
   }
```

Q6

Implement pow(*x*, *n*), which calculates *x* raised to the power *n* (x^n). x is double, n is integer.

A6

pow(x,n) = pow(x,a)*pow(x,b)*pow(x,c)*… if n = a+b+c+…

We can look at the bits in power i.e. n here to find a,b,c, etc. as above i.e a 1 at first bit means 1, a 1 at second bit means 2, a 1 at third bit means 4, and so on. We also take care of negative power by first ignoring it until we calculate pow(x, abs(n)) and for negative powers we divide 1.0 with pow(x, abs(n)) to get the final result.

```
double myPow(double x, int n) {
        long long pow = n;
        bool isNeg = n < 0;
        pow = isNeg ? -pow : pow;
        double res = 1.0;
        double xpow = x;
        while (pow != 0) {
            if (pow & 1) {
                res *= xpow;
            }
            xpow *= xpow;
            pow >>= 1;
        }

        return isNeg ? 1.0/res : res;
    }
```

Q7

Given a positive integer, return its corresponding column title as it appears in an Excel sheet. For example:

```
    1 -> A
    2 -> B
    3 -> C
    ...
    26 -> Z
    27 -> AA
```

```
    28 -> AB
    ...
```

A7

Excel column titles are string of characters which are upper case english A to Z, so this is equivalent to converting the number to base 26 where 0 is A, 1 is B, …, 25 is Z

```
string convertToTitle(int n) {
        string col;
        while (n > 26) {
            col += 'A' + (n/26-1);
            n /= 26;
        }

        if (n > 0)
            col += 'A' + n-1;

        return col;
}
```

Q8

Given a linked list, remove the *n*-th node from the end of list and return its head.

Example:

```
Given linked list: 1->2->3->4->5, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5.
```

A8

We need two pointers here to locate the nth last node from end in one pass of the list. We can have two pointers - leading pointer that will first move n units ahead in the list before the trailing pointer starts. Now when both these pointers moving ahead in the list, by the time the leading pointer reaches end of list, the trailing pointer should point to nth last node from end.

```
ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummyHead = new ListNode(0);
        dummyHead->next = head;
        ListNode *prev, *lead, *trail;
        prev = lead = trail = dummyHead;

        while (n--) {
            lead = lead->next;
        }

        while (lead) {
            prev = trail;
            trail = trail->next;
            lead = lead->next;
        }

        prev->next = trail->next;
        return dummyHead->next;
    }
```

Q9

Given a string containing just the characters ‘(', ‘)', ‘{', ‘}', ‘[' and ‘]', determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

Note that an empty string is also considered valid.

Example 1:

```
Input: "()"
Output: true
```

Example 2:

```
Input: "()[]{}"
Output: true
```

Example 3:

```
Input: "(]"
Output: false
```

A9

As we traverse the string, we can push the character to a stack if it is one of the open parentheses and if the character is one of closed parentheses we can compare immediately to the top of the stack to check validity. If a matching open parentheses to the closed parentheses is found on top of stack, we can pop the top of stack and continue. If the matching fails, we return false immediately. After the whole string is processed, the stack should be empty for validity as per rules mentioned in the problem.

```
    bool isOpen(char ch) {
        return ch == '(' || ch == '[' || ch == '{';
    }

    bool isClose(char ch) {
        return ch == ')' || ch == ']' || ch == '}';
    }

    char getClose(char ch) {
        if (ch == '(') return ')';
        else if (ch == '[') return ']';
        else return '}';
    }
```

```
    bool isValid(string s) {
        stack<char> stk;
        for (auto ch: s) {
            if (isOpen(ch)) {
                stk.push(ch);
            }
            else {
                if (stk.empty()) return false;
                if (getClose(stk.top()) != ch) return false;
                stk.pop();
            }
        }

        return stk.empty();
    }
```

Q10

Given an array of strings, group anagrams together.

Example:

```
Input: ["eat", "tea", "tan", "ate", "nat", "bat"],
Output:
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

A10

The idea here is if two strings are anagrams then their sorted versions must be same. To implement this idea, we use a map, where key is sorted string input and value is a list of strings whose sorted version matches the key.

```
vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>> anagMap;
```

```
        for (auto& s: strs) {
            string tmp=s;
            sort(tmp.begin(), tmp.end());
            anagMap[tmp].push_back(s);
        }

        vector<vector<string>> res;
        for (auto& x: anagMap) {
            res.push_back(x.second);
        }

        return res;
    }
```

Q11

Implement **next permutation**, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (i.e, sorted in ascending order).

The replacement must be **in-place** and use only constant extra memory.

Here are some examples. Inputs(given in vector<int> nums) are in the left-hand column and its corresponding outputs are in the right-hand column.

1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1

A11

Iterate digits from LSB(at the last index in the given vector) to MSB(first index in given vector) and look for first “i” where nums[i] > nums[i-1] i.e. to check if there is a possibility that the digits from LSB to MSB are not in ascending order at some point i. For example in input 1,2,3 at i=2, nums[i]>nums[i-1]. If no such i exists, then input is the highest in the lexicographical order, so we just sort and return it as per problem statement.

If such i is found, then we need to find the first digit “j” in order from LSB to MSB that is greater than nums[i-1], then we can swap nums[j] and nums[i-1] and finally sort the portion of vector from position i to end of vector. This is because we want the next number in lexicographical order, not just any greater number than the input. Example: in input 1,2,4,3,1, 4>2 so i=2, then 3>2 so j=3, after swapping we get 1,3,4,2,1. Finally after sorting from i to end of vector, we get 1,3,1,2,4.

The above algorithm and code below is intuitive when run on an example input.

```
void nextPermutation(vector<int>& nums) {
    //step 1
    int i=0;
    for (i=nums.size()-1; i>0; --i) {
        if (nums[i] > nums[i-1]) break;
    }

    // case where input is max in lexicographic order
    if (i == 0) {
        sort(nums.begin(), nums.end());
        return;
    }

    //step 2
    int j=0;
    for (j=nums.size()-1; j>0; --j) {
        if (nums[j] > nums[i-1]) break;
    }
    swap(nums[i-1], nums[j]);

    //step 3
    sort(nums.begin()+i, nums.end());
 }
```

Q12

We have a list of points on the plane. Find the K closest points to the origin (0, 0). Here, the distance between two points on a plane is the Euclidean distance.

Example 1:

```
Input: points = [[1,3],[-2,2]], K = 1 Output: [[-2,2]] Explanation:
The distance between (1, 3) and the origin is sqrt(10).
The distance between (-2, 2) and the origin is sqrt(8).
Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.
We only want the closest K = 1 points from the origin, so the answer is just [[-2,2]].
```

Example 2:

```
Input: points = [[3,3],[5,-1],[-2,4]], K = 2 Output: [[3,3],[-2,4]]
```

A12

We use max priority queue to hold K closest points. Each time the size of priority queue is already K and there is still an input to be processed, we compare the distance of the current point to the top of the priority queue and choose the one that is the closest and add into the priority queue.

```
struct Point {
    int index;
    double distance;
    Point(int i, double d) : index(i), distance(d) {}
};

class PointCompare {
public:
    bool operator()(Point & p1, Point & p2) {
        return p1.distance - p2.distance < 0.0 ? true : false;
    }
};

vector<vector<int>> kClosest(vector<vector<int>>& points, int K) {
```

```
    priority_queue<Point, vector<Point>, PointCompare> pq;
    for (int i=0; i < points.size(); i++) {
        double dist =
            sqrt(points[i][0]*points[i][0]+points[i[1]*points[i][1]);

        if (pq.size() == K) {
            if (dist < pq.top().distance) {
                pq.pop();
                pq.push(Point(i, dist));
            }
        }
        else {
            pq.push(Point(i, dist));
        }
    }

    vector<vector<int>> result;
    for (int i=0; i < K; i++) {
        result.push_back(points[pq.top().index]);
        pq.pop();
    }
    return result;
}
```

Q13

Implement int sqrt(int x).

Compute and return the square root of *x*, where *x* is guaranteed to be a non-negative integer.

Since the return type is an integer, the decimal digits are truncated and only the integer part of the result is returned.

Example 1:

```
Input: 4
Output: 2
```

Example 2:

```
Input: 8
Output: 2
Explanation: The square root of 8 is 2.82842..., and since
             the decimal part is truncated, 2 is returned.
```

A13

Here we do binary search between 0 and x/2 to find the integer square root. For numbers >=2, the square root after truncating decimal part won't exceed x/2.

```
int mySqrt(int x) {
    if (x < 2) return x;
    int lo=0, hi=x/2;
    long long tmp;
    while (lo <= hi) {
        int mid = lo+(hi-lo)/2;
        tmp = (long long)mid * mid;
        if (tmp == x) return mid;
        else if (tmp > x) hi=mid-1;
        else lo=mid+1;
    }

    return hi;
}
```

Q14

You are climbing a stair case. It takes *n* steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Note: Given *n* will be a positive integer.

Example 1:

```
Input: 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

A14

Number of ways to climb to stair at step n is sum of the number of ways to reach step n from step n-1 and step n-2, as we can reach from steps n-1 and steps n-2 to step n.

```
int climbStairs(int n) {
    if (n==1) return 1;
    if (n==2) return 2;

    int prev2=1, prev1=2;
    int count = 0;
    while (n-2 != 0) {
        count = prev1+prev2;
        prev2 = prev1;
        prev1 = count;
        n--;
    }

    return count;
}
```

Q15

Shuffle an array of numbers (no duplicates in input). Any permutation of the input array must be equally likely after the shuffle.

Example:

```
// Init an array with set 1, 2, and 3.
int[] nums = {1,2,3};
Solution solution = new Solution(nums);

// Shuffle the array [1,2,3] and return its result. Any permutation of [1,2,3] must equally
likely to be returned.
solution.shuffle();

// Resets the array back to its original configuration [1,2,3].
solution.reset();

// Returns the random shuffling of array [1,2,3].
solution.shuffle();
```

A15

We choose each element of the result of shuffle with equal probability in the shuffle method below.

```
class Solution {
    vector<int> orig;
public:
    Solution(vector<int> nums) : orig(nums) {
        srand(time(nullptr));
    }

    //Resets the array to its original configuration and return it
    vector<int> reset() {
        return orig;
    }

    // Returns a random shuffling of the array
    vector<int> shuffle() {
```

```
        vector<int> result(orig);
        for (int i=result.size()-1; i > 0; i--) {
            int randIdx = rand()%(i+1);
            swap(result[i], result[randIdx]);
        }
        return result;
    }
};
```

Q16

There are **N** students in a class. Some of them are friends, while some are not. Their friendship is transitive in nature. For example, if A is a **direct** friend of B, and B is a **direct** friend of C, then A is an **indirect** friend of C. And we defined a friend circle is a group of students who are direct or indirect friends.

Given a **N*N** matrix **M** representing the friend relationship between students in the class. If M[i][j] = 1, then the i_{th} and j_{th} students are **direct** friends with each other, otherwise not. And you have to output the total number of friend circles among all the students.

Example:

```
Input:
[[1,1,0],
 [1,1,0],
 [0,0,1]]
Output: 2
Explanation:The 0th and 1st students are direct friends, so they are in a friend circle. The
2nd student himself is in a friend circle. So return 2.
```

A16

If A is friend of B, B is friend of C, then friend circles will have A,B,C as they all can be reached via common friend.

We run a depth first search(dfs) on all unvisited yet students to visit all the students reachable from them and keep a count of friend circles each time we start with an unvisited student who was not visited in previous dfs visits.

```
//helper function
void dfs(vector<vector<int>>& M, vector<bool>& visited, int i) {
    visited[i] = true;
    for (int j=0; j < M[i].size(); j++) {
        if (M[i][j] == 1 && !visited[j]) dfs(M, visited, j);
    }
}

// function that finds number of friend circles
int findCircleNum(vector<vector<int>>& M) {
    int n = M.size();
    int m = n > 0 ? M[0].size() : 0;
    if (n == 0 || m == 0 || n != m) return 0;

    vector<bool> visited(n, false);
    int count=0;
    for (int i=0; i<n; ++i) {
        if (!visited[i]) {
            dfs(M, visited, i);
            ++count;
        }
    }
    return count;
}
```

Q17

Given a collection of intervals, merge all overlapping intervals.

Example:

```
Input: [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
```

A17

We first sort the intervals with start value and if they are equal by close value. Then we check for overlap in order, if there is overlap we combine to one otherwise add to the result.

```
// functor that compares two intervals
struct IntervalComp {
  bool operator()(const vector<int>& a, const vector<int>& b) {
    return a[0]==b[0] ? a[1] < b[1] : a[0] < b[0];
  }
};

// actual merge function
vector<vector<int>> merge(vector<vector<int>>& intervals) {
    if (intervals.size() <=1 ) return intervals;

    vector<vector<int>> res;
    sort(intervals.begin(), intervals.end(), IntervalComp());

    vector<int> curr = intervals[0];
    for (int i=1; i<intervals.size(); ++i) {
        if (curr[0] <= intervals[i][1] &&
            intervals[i][0] <= curr[1]) {
            curr[0] = min(curr[0], intervals[i][0]);
            curr[1] = max(curr[1], intervals[i][1]);
        }
        else {
            res.emplace_back(curr);
            curr = intervals[i];
        }
    }
    res.emplace_back(curr);

    return res;
}
```

Q18

Given an array nums of *n* integers, are there elements *a*, *b*, *c* in nums such that *a* + *b* + *c* = 0? Find all unique triplets in the array which gives the sum of zero.

Note:

The solution set must not contain duplicate triplets.

Example:

```
Given array nums = [-1, 0, 1, 2, -1, -4],

solution set is:
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

A18

We first sort the input numbers. Then for each i, 0<=i<nums.size(), for each nums[i], look for pair of two numbers in range of indices i+1 to nums.size()-1 in nums array, that when added to nums[i] will yield 0.

```
vector<vector<int>> threeSum(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    vector<vector<int>> res;

   for (int i=0; i<nums.size(); ++i) {
        if (i>0 && nums[i] == nums[i-1]) continue;
        int lo=i+1, hi=nums.size()-1;
        int t=-nums[i];

        while (lo < hi) {
            if (nums[lo]+nums[hi] == t) {
                vector<int> tmp({nums[i], nums[lo], nums[hi]});
```

```
                res.emplace_back(tmp);
                while (lo < hi && nums[lo] == nums[lo+1]) ++lo;
                while (lo < hi && nums[hi] == nums[hi-1]) --hi;
                ++lo; --hi;
            }
            else if (t > nums[lo]+nums[hi]) ++lo;
            else --hi;
        }
    }

    return res;
}
```

Q19

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

A19

We recursively check for BST validity at each node, top to bottom, and then check its left and right subtrees recursively.

```
struct TreeNode {
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

bool isValidBSTHelper(TreeNode* root, TreeNode* min, TreeNode* max) {
    if (!root) return true;
    if ((min && root->val <= min->val) ||
```

```
        (max && root->val >= max->val))
        return false;

    return isValidBSTHelper(root->left, min, root) &&
           isValidBSTHelper(root->right, root, max);
}

bool isValidBST(TreeNode* root) {
    return isValidBSTHelper(root, nullptr, nullptr);
}
```

Q20

Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level).

For example:
Given binary tree [3,9,20,null,null,15,7],

```
    3
   / \
  9  20
    /  \
   15   7
```

return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

A20

We need in FIFO order, so we can use queue. To differentiate between two levels, we use two queues, one which has the nodes from current level and another with nodes from one level below i.e. children of nodes at current level. Then once we process the current level, the queue with children becomes current level and will add their children to the queue that holds nodes at one level below them.

```
struct TreeNode {
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        if (!root) return result;
        queue<TreeNode*> q1, q2;
        q1.push(root);
        queue<TreeNode*> * getq, * putq;
        vector<int> level;
        TreeNode * curr;
        while (!q1.empty() || !q2.empty()) {
            putq = q1.empty() ? &q1 : &q2;
            getq = q1.empty() ? &q2 : &q1;
            while (!getq->empty()) {
                curr = getq->front();
                if (curr->left) putq->push(curr->left);
                if (curr->right) putq->push(curr->right);
                level.push_back(curr->val);
                getq->pop();
            }
            result.push_back(level);
            level.clear();
        }
        return result;
    }
```

Q21

Given a string, sort it in decreasing order based on the frequency of characters.

Example:

```
Input:
"tree"

Output:
"eert"

Explanation:
'e' appears twice while 'r' and 't' both appear once.
So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.
```

A21

First we record the frequency of each character into a hash table. Then we take the key, value pairs from the hash table into a vector to sort them by frequency. Next step is building the output string from the sorted vector.

```
struct Node {
  char c;
  int freq;
  Node(char ch, int ifreq) : c(ch), freq(ifreq) {}
};

struct compare_func {
  inline bool operator() (const Node& n1, const Node& n2) {
    return n1.freq > n2.freq;
  }
};

string frequencySort(string s) {
  unordered_map<char, int> cmap;
  for (auto & ch: s) {
    cmap[ch]++;
  }

  vector<Node> chfreqs;
  for (auto & elem: cmap) {
    chfreqs.push_back(Node(elem.first, elem.second));
  }

  sort(chfreqs.begin(), chfreqs.end(), compare_func());
  stringstream ss;
```

```
  for (auto & node: chfreqs) {
    for (int i=0; i<node.freq; i++) {
      ss << node.c;
    }
  }

  return ss.str();
}
```

Q22

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
put(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

The cache is initialized with a **positive** capacity.

Do both operations get and put in **O(1)** time complexity

Example:

```
LRUCache cache = new LRUCache( 2 /* capacity */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
cache.put(4, 4);    // evicts key 1
cache.get(1);       // returns -1 (not found)
```

```
cache.get(3);       // returns 3
cache.get(4);       // returns 4
```

A22

We maintain two data structures. One is a list of Items where each Item holds the key and value. Another is a hash table that holds key of the Item to its position in the list in the form of list iterator that points to the list node of that key. With these, it is easy to find the cache size; locate an item by key and fetch its value or delete it; and adding a new (key, value) pair in O(1) time.

```
class LRUCache {
    struct Item {
        int key;
        int val;
        Item(int k, int v) : key(k), val(v) {}
    };
    list<Item> cache;
    unordered_map<int, list<Item>::iterator> keysMap;
    int capacity;
public:
    LRUCache(int capacity) : capacity(capacity) {
    }

    int get(int key) {
        if (keysMap.count(key)) { //key exists
            int val = (*keysMap[key]).val;
            put(key, val);
            return val;
        }

        return -1;
    }

    void put(int key, int value) {
        if (keysMap.count(key)) {
            cache.erase(keysMap[key]);
            keysMap.erase(key);
        }

        if (cache.size() == capacity) {
            int key = (*cache.begin()).key;
```

```
            cache.pop_front();
            keysMap.erase(key);
        }

        cache.push_back(Item(key, value));
        auto iter = cache.end();
        advance(iter, -1);
        keysMap[key] = iter;
    }
};
```

10

Math Puzzles

Math Puzzles are not commonly asked in trading systems developer interviews, they are more common for quants and traders. However some firms do ask, and you should not be surprised.

Most of the math puzzles are based on probability, statistics, discrete math, number theory, and logic.

Q1

There are three ants on a triangle, one at each corner. At a given moment in time, they all set off for a corner at random. What is the probability that they don't collide?

A1

Collision doesn't happen only in two cases -

- if all ants move in the clockwise direction
- if all ants move in the counter-clockwise direction

Since each ant has two choices i.e. pick either of the two edges going through the corner on which ant is initially sitting, there are a total of 2x2x2 possibilities. Out of 8 possibilities, only 2 doesn't cause a collision. So, the probability that the ants don't collide is 2/8.

Q2

You have 1000 bottles of wine and exactly one bottle is poisoned. You need to find the poisoned wine before your party starts in an hour. You have rats available to test on to find out which bottle is deadly. The poison takes effect after an hour of consumption, so you only have one chance to run your rat poison experiment, meaning you can't feed some rats wine and wait an hour before feeding them more wine. Assume each rat can drink as much wine as you feed it. What is the minimum number of rats required to find the poisoned wine?

A2

If we divide 1000 bottles into two groups of 500 each, to isolate which group has the poisoned bottle, we need two rats. Label the bottles with numbers from 1 to 1000 in binary, feed one rat with drops from bottles with first bit 0, and another rat with drops from bottles with first bit 1.

Similarly, to isolate one bottle from 1000, we need 10 rats as 2^10 = 1024, 2^9 = 512 i.e. we need minimum of 10 bits to represent all 1000 bottles in binary. Applying the same logic as above, label rats 1 to 10, feed rat 1 with drops from bottles whose first bit is 1, feed rat 2 with drops from bottles whose second bit is 1, so on. Example: If rats numbered 5, 7, and 9 die, then bottle number 42 (Binary 0000101010) is poisoned.

Q3

If there is a party with N guests and every guest shakes hands with each other once, how many handshakes happened?

A3

If there are N guests, then each guest will shake hands with N-1 other guests. If N guests make N-1 handshakes, it appears there are N*(N-1) handshakes. In this method, we are counting each handshake twice i.e Guest1's handshake to Guest2 and Guest2's handshake to Guest1, so we must divide by 2.

Total number of handshakes will be N*(N-1)/2

Q4

N tennis players are competing in a tournament. How many matches need to be played to decide the winner?

A4

N-1. To decide the winner, remaining players need to lose. N-1 players will lose in N-1 matches.

Q5

Two trains enter a tunnel 200 miles long, traveling at 100 mph at the same time from opposite directions. As soon as they enter the tunnel a supersonic bee flying at 1000 mph starts from one train and heads toward the other one. As soon as it reaches the other one it turns around and heads back toward the first, going back and forth between

the trains until the trains collide in a fiery explosion in the middle of the tunnel. How far did the bee travel?

A5

time taken by both trains to meet is 200miles/(100mph+100mph) = 1 hour

In that time, the bee will travel 1000mph x 1 hour = 1000 miles

11

Tools

The work culture in this industry is somewhat fast-paced. So, it helps if the engineer is proficient with the tools required for the job. It is common to ask a simple question on a commonly used feature in the tools you used in the past to see if you are hands on or a theory person.

Some tools below commonly used by trading system developers working with C++ in a Linux environment.

git - used for source code management, especially useful when distributed teams working on the same repository

gdb - debugger. very important to master the debugger in your environment as fixing the production issues quickly is crucial to success.

valgrind - used to check for memory leaks in programs

perf - profiler tool that comes with Linux used for performance analysis

12

Domain Knowledge

If you have no experience in the financial industry, you won't get these questions. If you do have experience in the financial industry, there might be some simple relevant questions based on the areas you worked on. It is recommended to have some general knowledge of technologies and keywords relevant to the financial industry.

Q1

Compare FIX and binary protocols (OUCH etc.)

A1

FIX (Financial Information Exchange) protocol is a text-based protocol to exchange financial and trading information between different financial firms. FIX uses a sequence of key-value pairs with delimiter to specify the information.

Binary protocols transmit using a fixed record layout in binary. Binary protocols are not verbose and take fewer data compared to FIX as there is no need to transmit key names and also data is in binary format.

Q2

Some trading industry-specific keywords to know

A2

Market Microstructure - Market microstructure deals with issues of market structure and design, price formation, price discovery, transaction and timing cost, information & disclosure, and investor behavior

RegNMS - a set of rules that defines how trading works in the U.S. for all listed stocks. Reg NMS also mandates market-wide cross-connectivity, allowing for a competitive and distributed market

Volcker Rule - aims to protect bank customers by preventing banks from making certain types of speculative investments

VWAP - Volume weighted average price is weighted average calculated based on all of the trades for a financial instrument in any time interval. If the trades are 100 quantity at 10 price and 200 quantity at 5 price, vwap will be calculated as (100*10+200*5)/(100+200). It is is an important tool used to benchmark an order fill.

ETF - ETF is a financial instrument that tracks a basket of financial instruments with various criteria and custom allocations chosen by the ETF manager. ex: SPY ETF tracks stocks in the S&P 500 index

Alpha - how well (or badly) a stock has performed in comparison to the market

Beta - how volatile a stock's price has been in comparison to the market

13

Behavioral Interview

This interview is to know about your experience, gauge how you will work in a team, and see if you are a cultural fit for the company. This is usually not a separate interview by itself but instead is a part of the technical interview where the interviewer will spend some time on this. However, some firms have one dedicated interview round just for this. I have listed some important things to prepare for below.

- **Past projects** - prepare to talk at least about 3 challenging technical projects you worked on in the past. Be ready to explain the reasoning behind the design decisions, technical obstacles you encountered and how you overcame them, etc.
- **Challenging bugs** - prepare 3 interesting and challenging bugs you have fixed
- **Team conflicts** - how you handled any conflicts with your coworkers and management
- **People skills** - how you work with external(if you need to interact with external customers) and internal(product, business, sales) clients for your projects
- **Project planning** - how you estimate the deadlines for the project, how you deal with when you encounter unplanned obstacles
- **Career goals** - long term career goals and how this job is a stepping stone in that direction

Thank You

Thanks for purchasing my book. I hope the book was helpful to you and I wish you good luck with your interviews.

It was a pleasure writing this book although it was a bit hectic managing my day job and spending time with my wife and daughter.

I'm always looking to grow and improve as an author. It is reassuring to hear what works, as well as receive constructive feedback on what should improve. Second, starting out as an unknown author is exceedingly difficult, and reviews go a long way toward making the journey out of anonymity possible. Please take a few minutes to write a review.

If this book was helpful to you in some way, could you please do me a favor by leaving a high star rating and review for the book on the website you bought this book from.

FAQ on tech/quantitative finance career for software engineers

Are software developers respected and well-treated at hedge funds, HFT, and similar financial companies?

Financial firms that are driven fundamentally by tech such as HFT, quant/algorithmic trading firms, hedge funds, etc. treat software developers really well.

How should I prepare for interviews of software engineer roles in top hedge funds, HFT companies, etc. (Citadel, HRT, etc.)?

You need to be good at problem-solving and algorithm/data structure design similar to top tech companies' interviews.

Apart from that, you also need to know a good deal about C++, lock-free programming, low latency engineering, IPC, OS/Network tuning, etc.

Should I(software engineer) work at a startup, or at a hedge fund/trading firm? How do the pay and work compare?

Go work at a startup if your goal is to start your own company in the future and making top compensation is not your priority in short term.

Go work at a hedge fund or in a trading team of a top financial firm if you want to make big paychecks right away.

You need to have an entrepreneurial mindset to thrive in both places.

How much do software developers make at big hedge funds compared to those who work at Google, Facebook, etc.?

In the current market, the pay at FAANGs and top hedge funds like Citadel, Two Sigma, HRT, etc. are comparably similar.

Can a software developer at a finance company move into quant and trader roles?

Yes, but with some effort and luck. Two things need to be done.

Improve your math background – stats, probability, working with time series, relevant calculus/linear algebra, discrete math, etc. Get an MFE (Masters in Financial Engineering) part-time/online from a reputed school if you lack the math background.

Network with the quants and traders at your firm.

Resources

C++

Effective C++ - Scott Meyers

More Effective C++ - Scott Meyers

Effective Modern C++ - Scott Meyers

C++ Templates - David Vandevoorde

Quantitative Finance : An Object-Oriented approach in C++ - Erik Schlogl

Multithreading

C++ Concurrency in Action - Anthony Williams

Linux

Linux Kernel Development - Robert Love

Linux Programming Interface - Michael Kerrisk

Networks

Understanding Linux Network Internals - Christian Benvenuti

TCP/IP Illustrated - Richard Stevens

Coding Interview

Elements of Programming Interviews - Adnan Aziz et al.

Cracking the Coding Interview - Gayle McDowell

Design Patterns

Head First Design Patterns - Eric Freeman et al.

Quant Interviews

Heard on the Street – Timothy Falcon Crack

Options, Futures, and other Derivatives – John C. Hull