The Speed Game: Automated Trading in C++

Carl Cook, Ph.D.

optiver

Contents

- How fast is fast
- The role of C++ in automated trading systems
- Low latency C++ coding techniques
- Downsides of C++
- SG14

About me

- Ph.D. in software engineering
  - Static code analysis/semantic modelling of code
- M.Sc. in computer science
  - Programmable/active networks
- 10+ years of commercial software development
  - All within the finance industry - focussed on order execution
- Member of SG14 (with proposals to WG21)
  - Representing the trading community
- Currently work at Optiver (an electronic market maker)
  - I work on order execution, risk systems, performance, ...

Disclaimer

This isn’t a talk about general optimization

For that, please see talks by:

- Andrei Alexandrescu
- Mike Acton

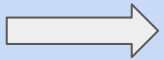
This is a talk about squeezing latency out of your code

I’ve also removed any overlap from other presentations

Motivating example: Spending money for a weekend in London

EUR-GBP	0.9/1.1

EUR-USD	1.1/0.9



USD-GBP	0.82/1.22

1.0 * 1.1 * 0.82 = 0.9
(no arbitrage opportunity)

1.0 * **1.2** * 0.82 = 0.98

Arbitrage opportunity: convert to USD**1.2**, convert that to GBP**0.98**, convert that back to EUR**1.10**

Generally, markets are efficient, where participants will quickly bring prices back into line

The fastest wins

and it doesn’t matter by how much

How fast is fast?

From:

Receiving the exchange's market data,
to spotting the opportunity,
to performing risk checks,
to sending the buy/sell message back to the exchange

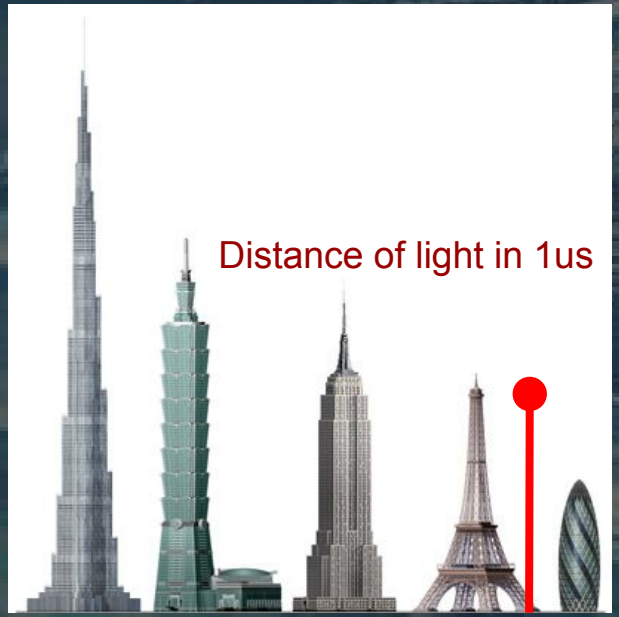
An all software approach: **1-10us**
An all hardware approach: **100-1000ns**
(Hybrid approaches are also possible)

How fast is fast?

From:

Receiving the exchange's market data,
to spotting the opportunity,
to performing risk checks,
to sending the buy/sell message back to the exchange

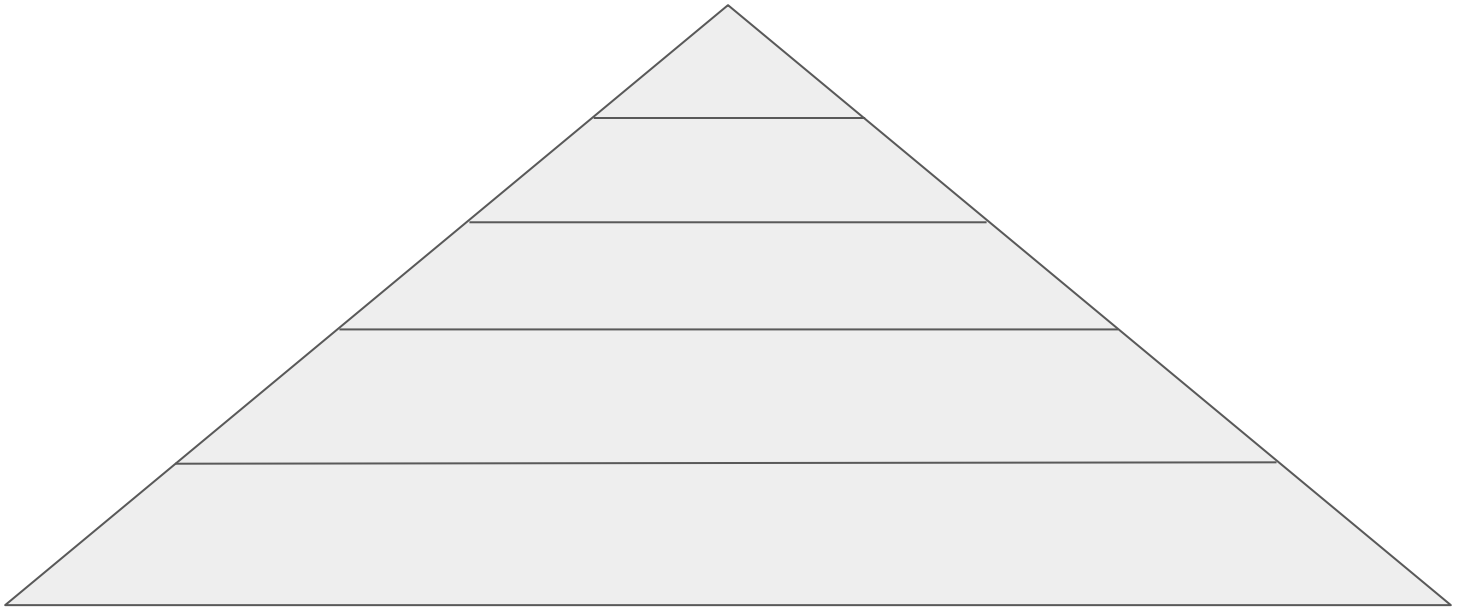
An all software approach: **1-10us**
An all hardware approach: **100-1000ns**
(Hybrid approaches are also possible)



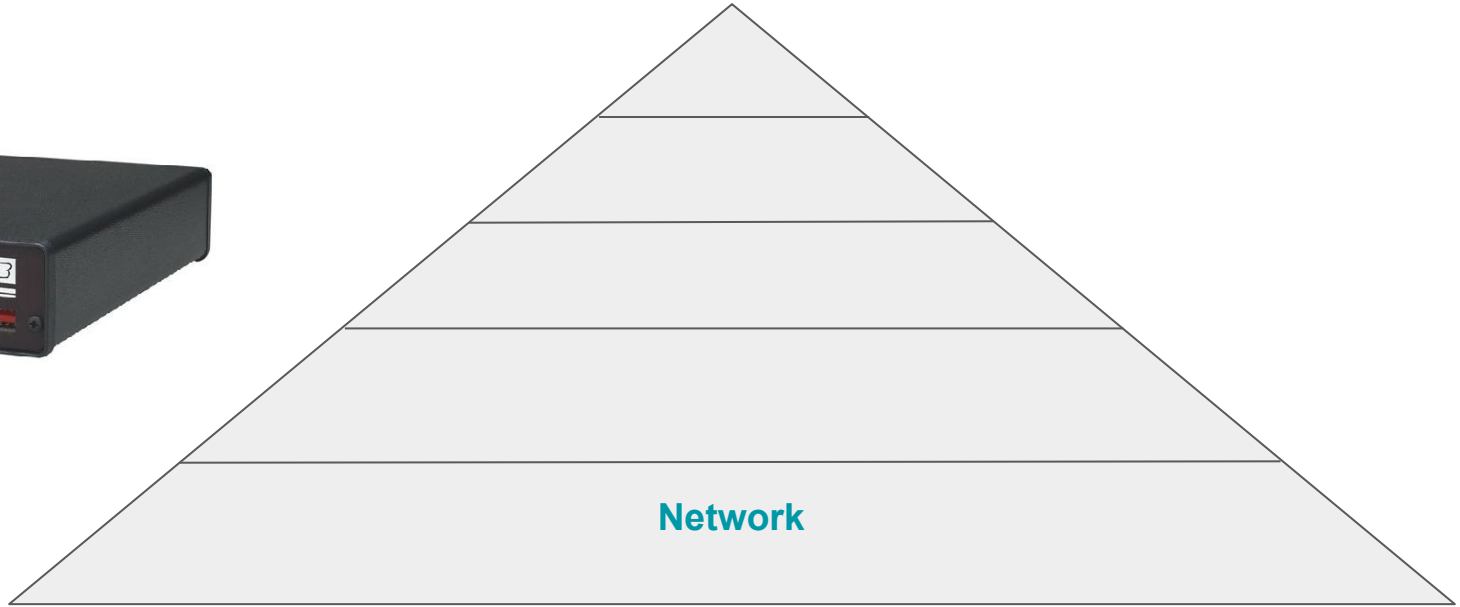
Characteristics of automated trading systems

- Only a few lines of code in the entire system are important (buy/sell)
- These lines are not exercised often (relative to say graphics rendering)
- Lots of market data events (millions of events per second)
- Jitter is a killer (delays in reaction time can be very costly)
- Very little threading/instruction vectorisation/etc
  - Remember - latency not throughput
- No mistakes (and very good recovery from mistakes)
  - One second is 4 billion CPU instructions
  - Companies have gone bankrupt through system error before, and markets/investors have also been impacted
  - A highly regulated and highly visible industry

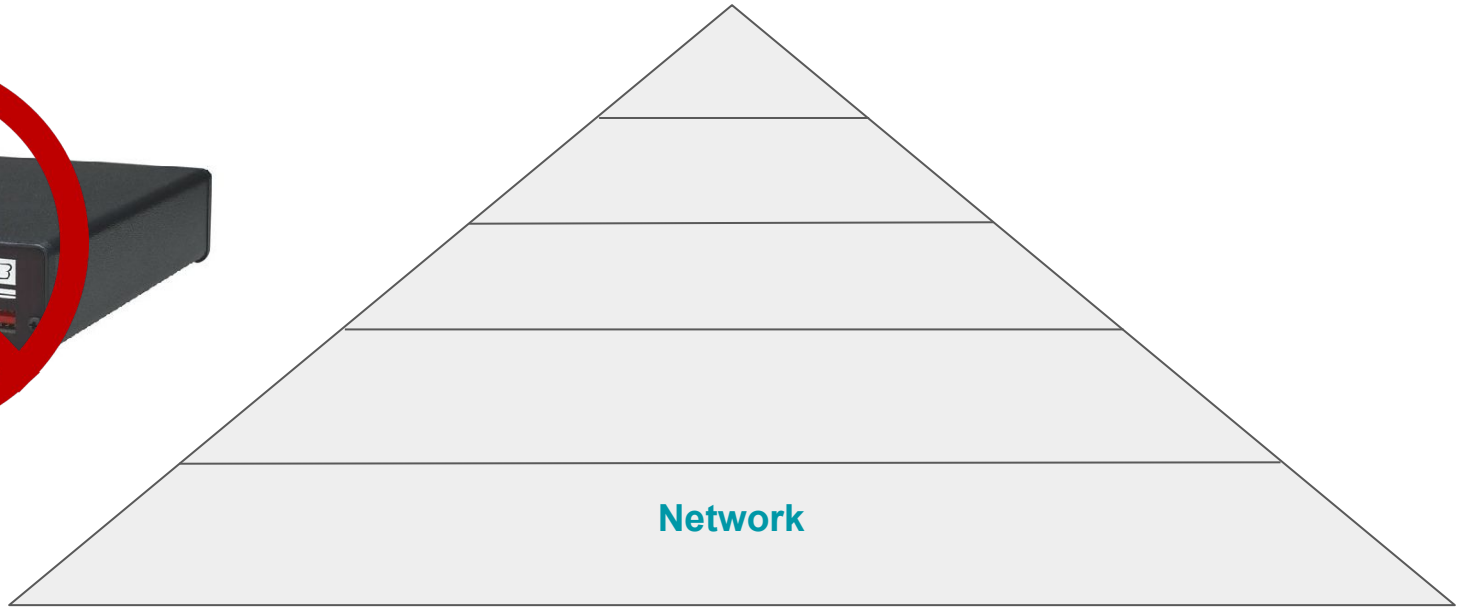
The role of C++ in trading systems



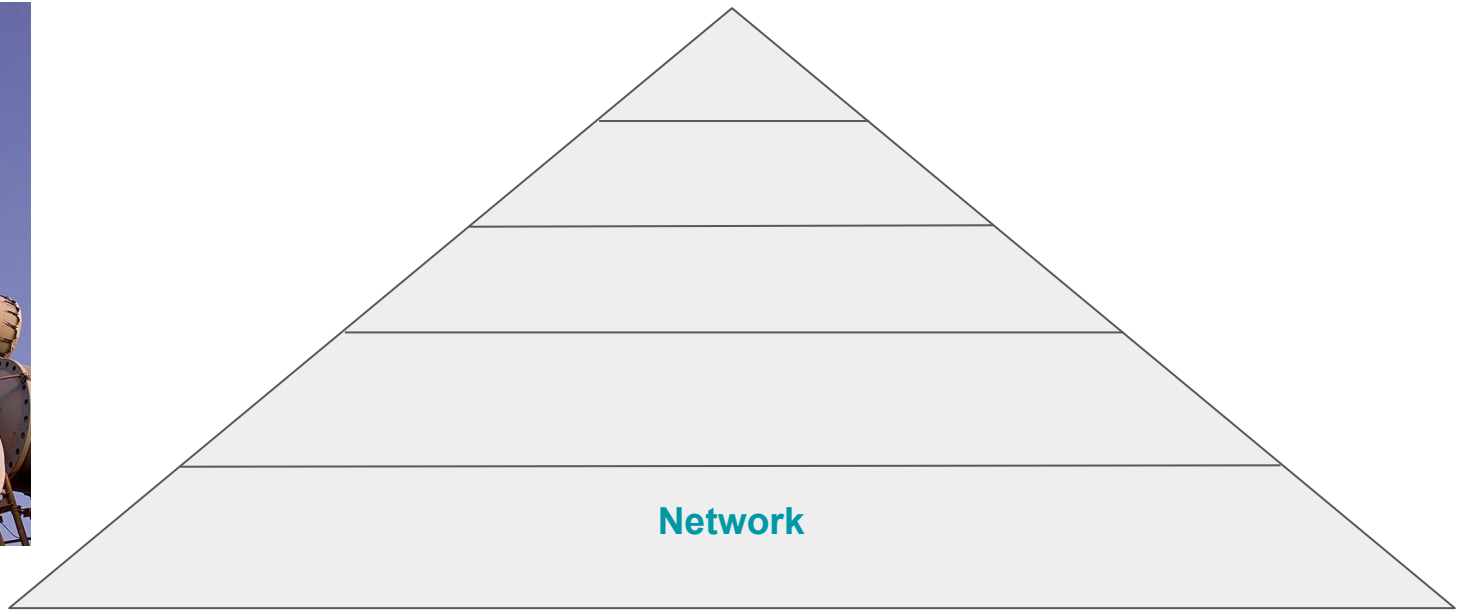
The role of C++ in trading systems



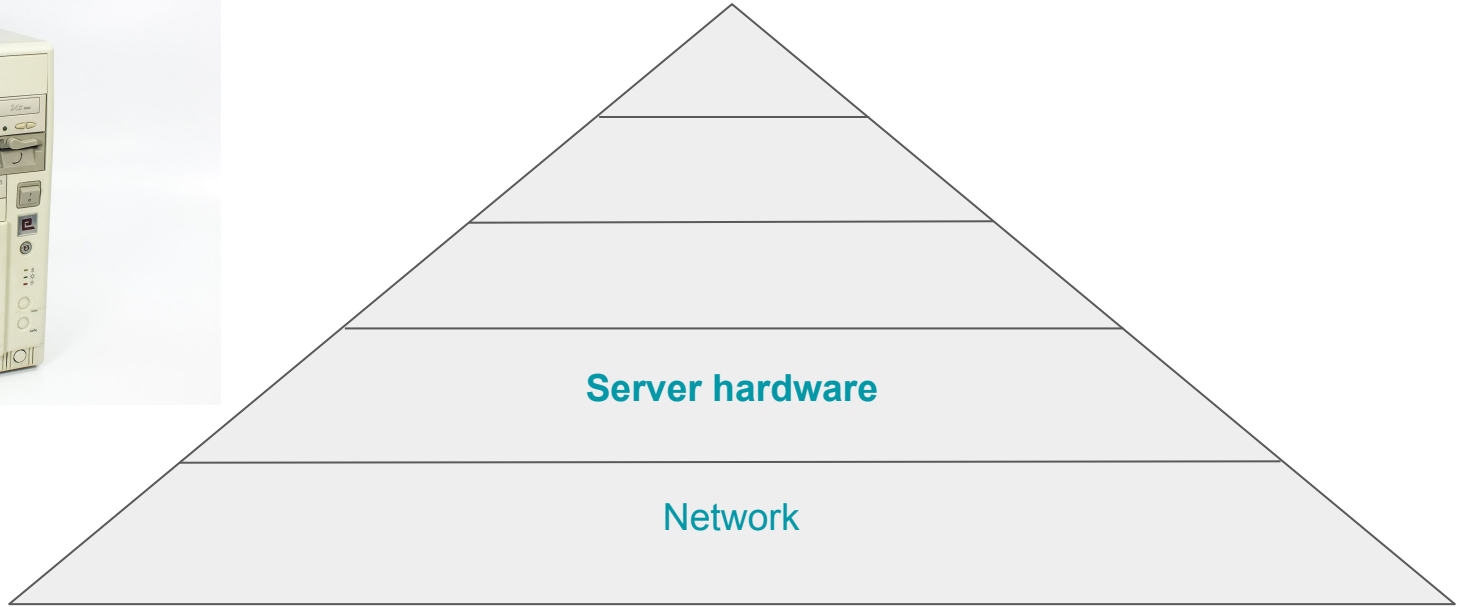
The role of C++ in trading systems



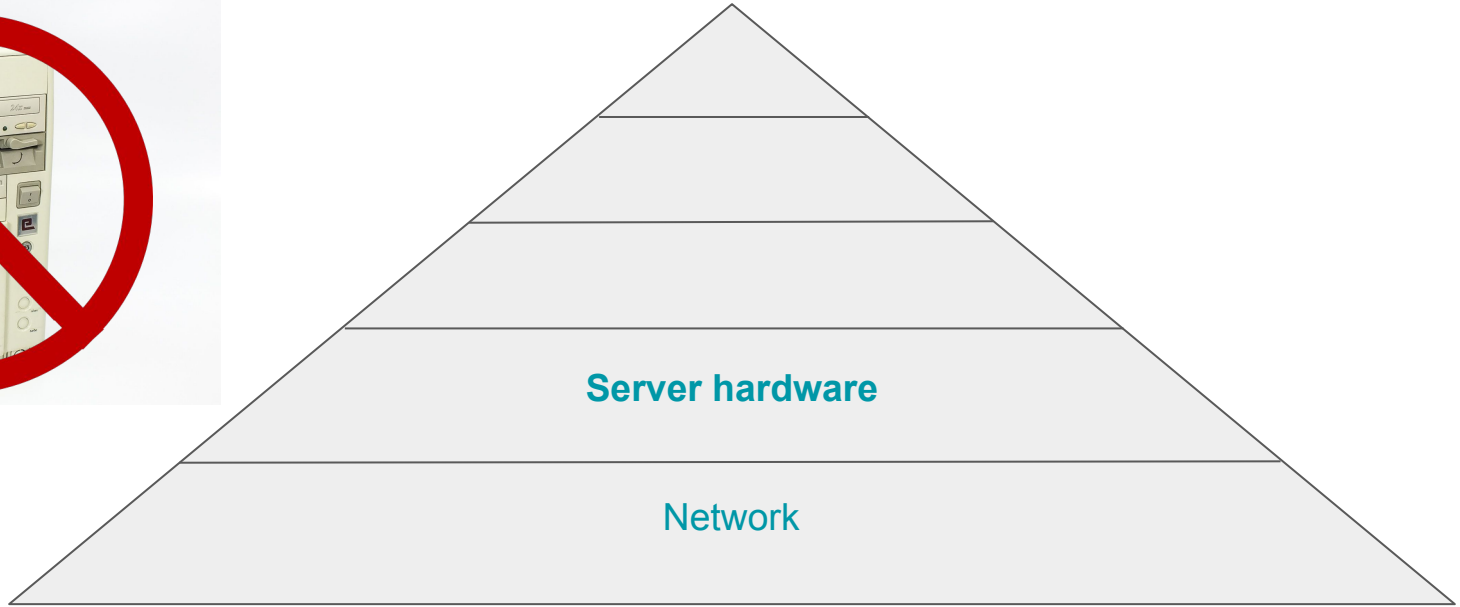
The role of C++ in trading systems



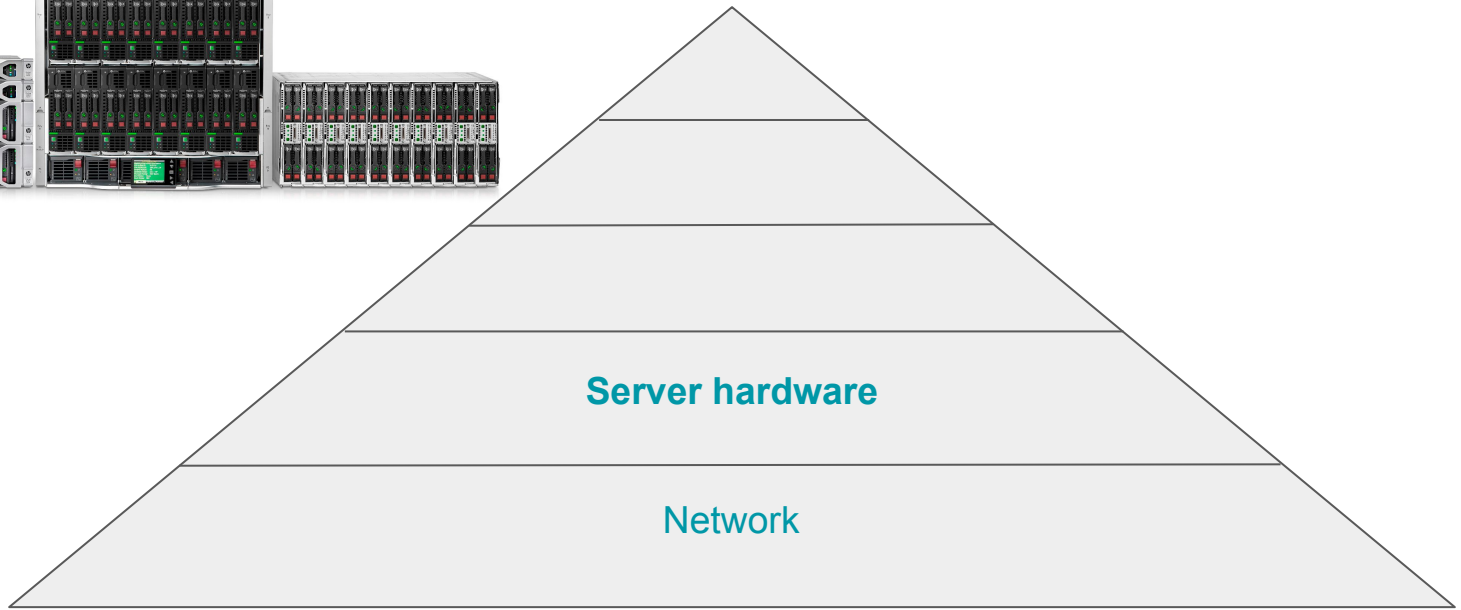
The role of C++ in trading systems



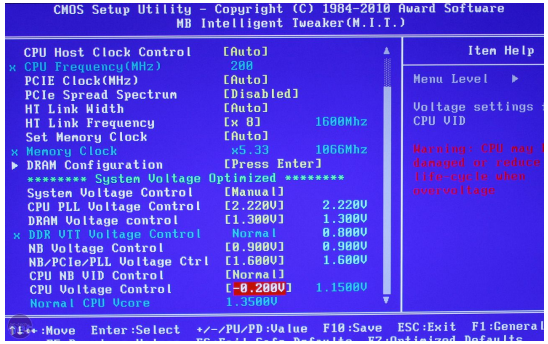
The role of C++ in trading systems

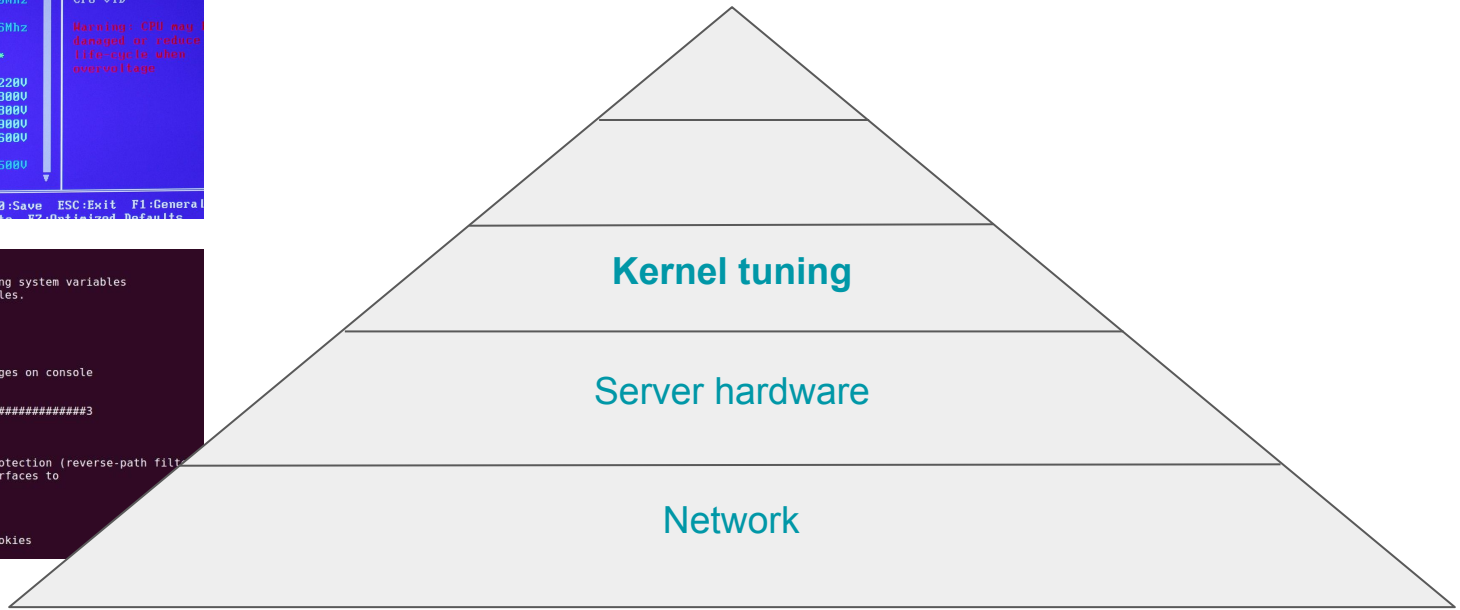


The role of C++ in trading systems

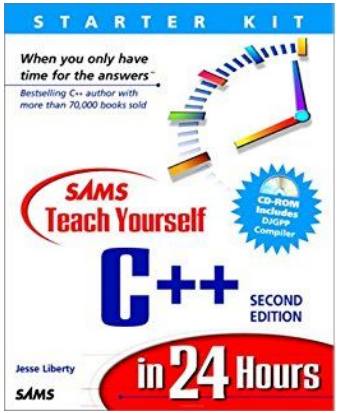


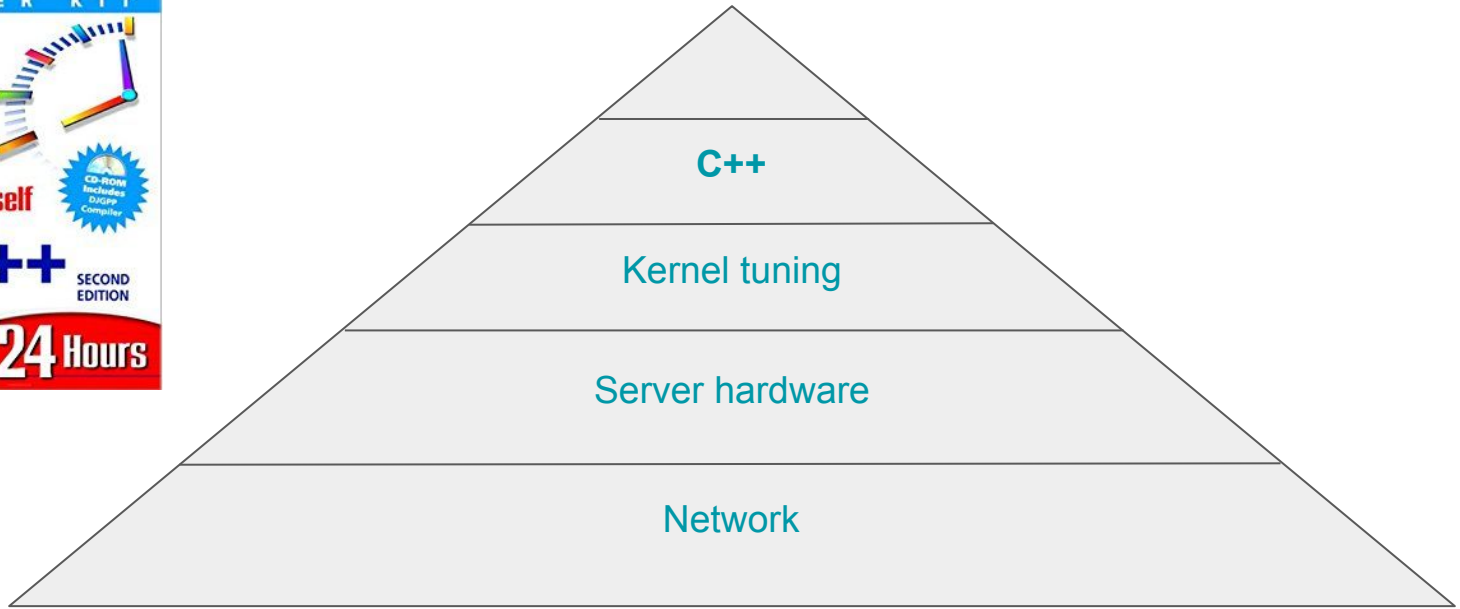
The role of C++ in trading systems



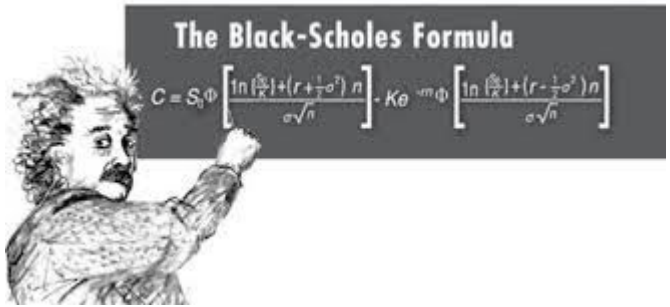


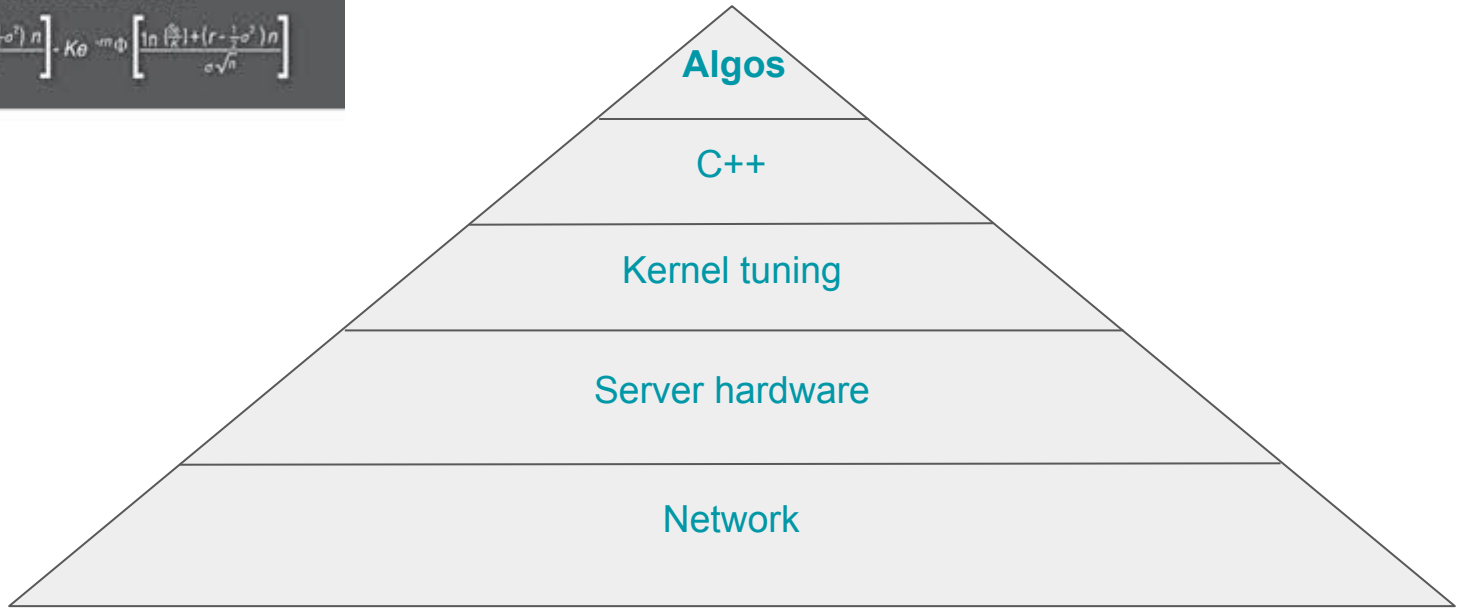
The role of C++ in trading systems



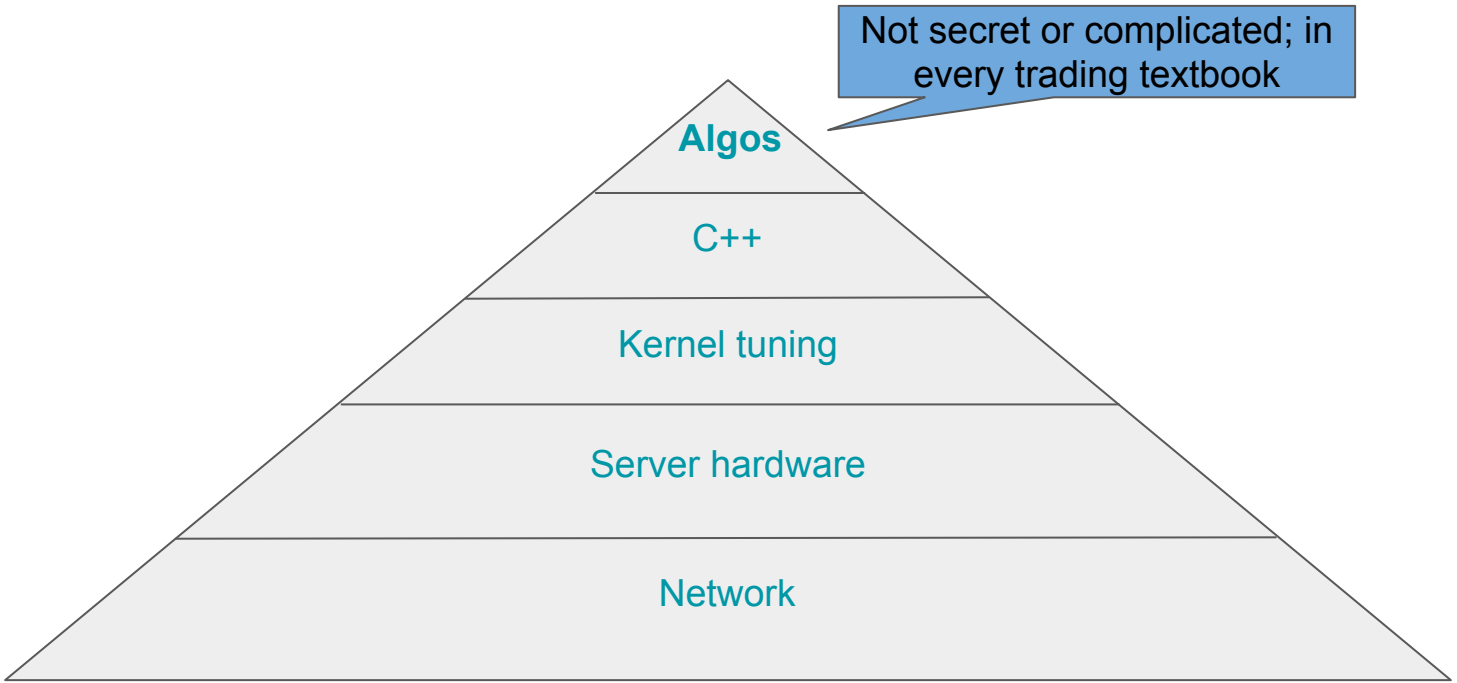


The role of C++ in trading systems

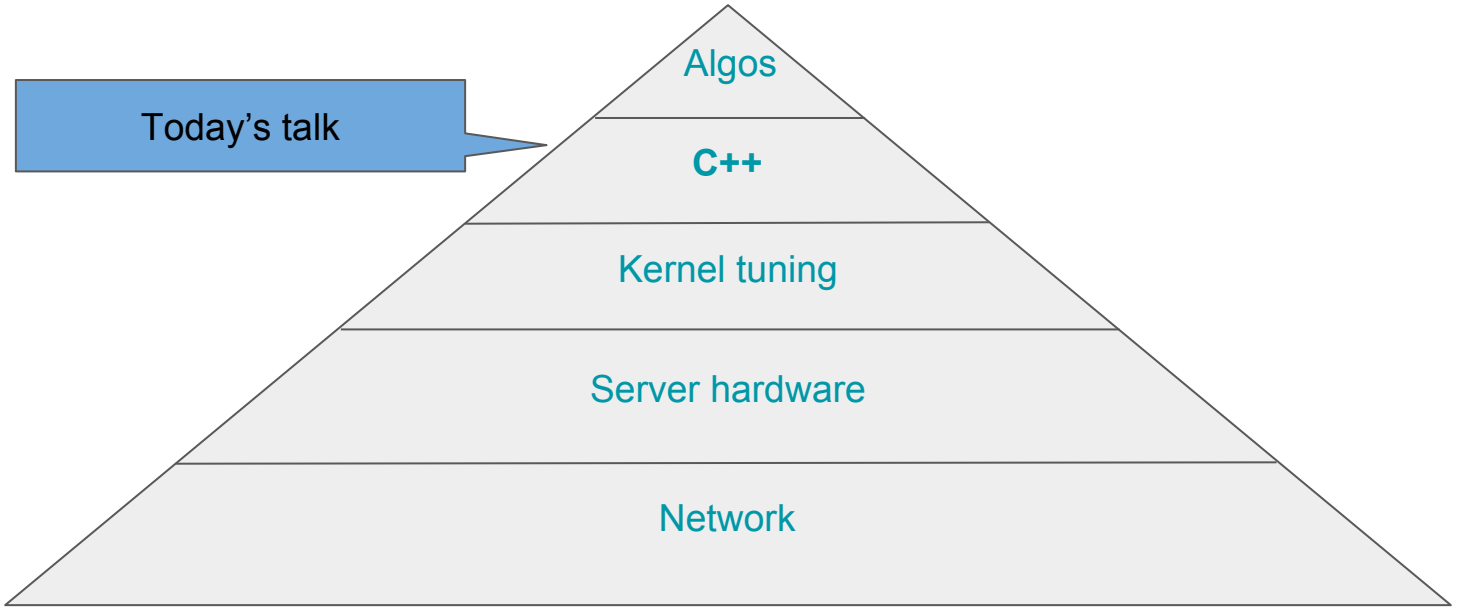




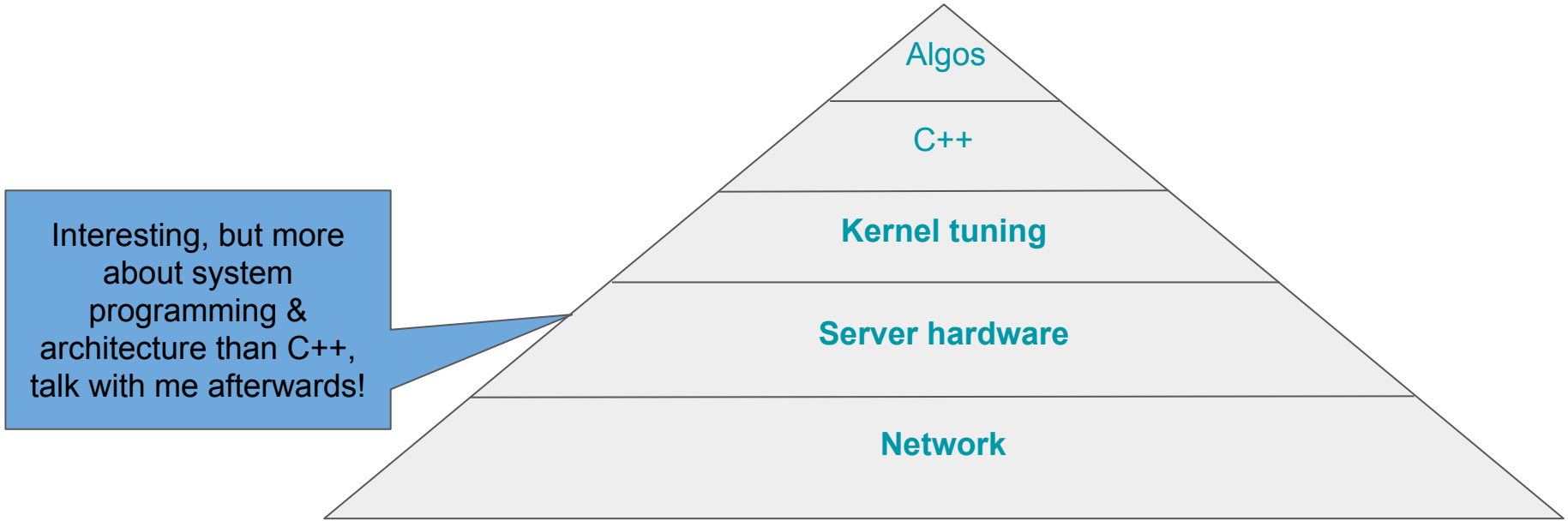
The role of C++ in trading systems



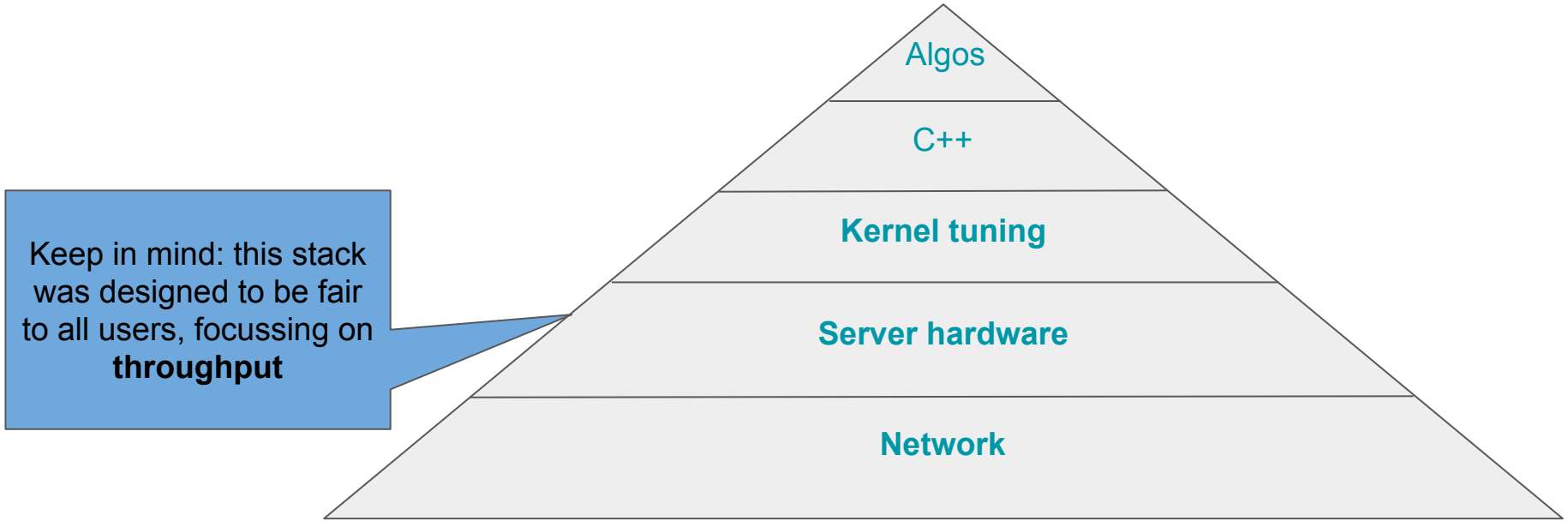
The role of C++ in trading systems



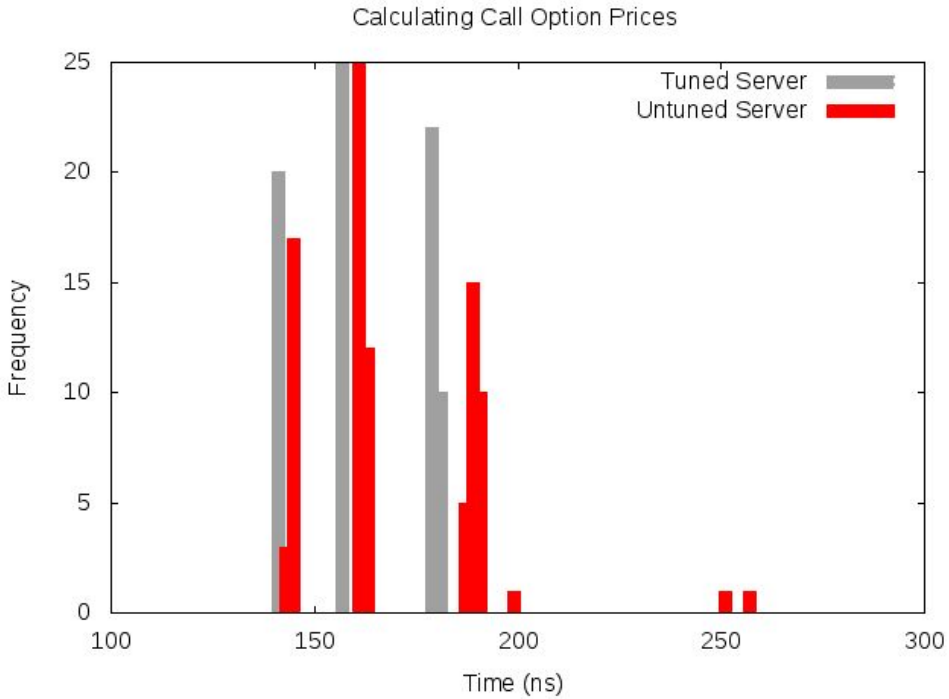
The role of C++ in trading systems



The role of C++ in trading systems



The effect of kernel tuning



Computation: Black-Scholes options pricing (CPU intensive, minimal data access)

Low latency C++

Smartness

- Problem transformation to faster forms
- Then, move decisions from runtime to compile time
- Sometimes, heuristics are acceptable (price interpolation, some flexibility on limits, etc)

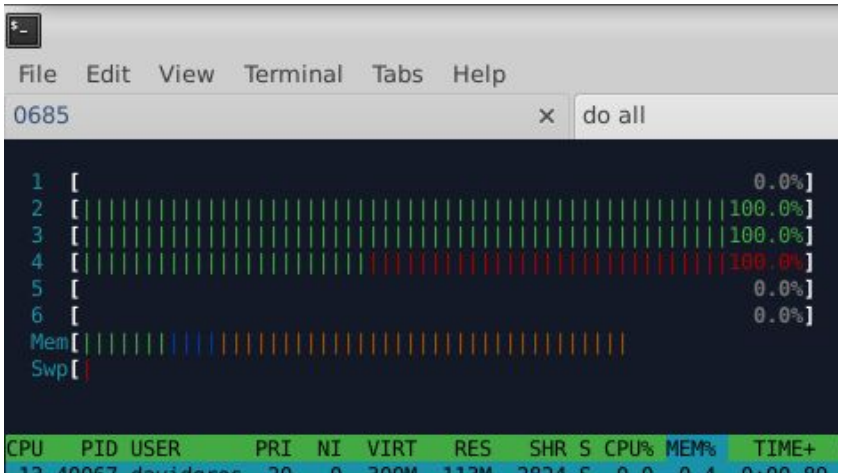
Implementation

- The simpler the code, the faster it is likely to be
- Don’t under-estimate optimising compilers, or try to be the optimising compiler
- Know the language, and become very familiar with the STL (algorithm selection)
- Know your hardware!
- Bypass the kernel - aim for 100% userspace code (C/C++), including any network IO
- Measure, measure, measure

Specifics

- Read the resultant assembly
- Cache warming helps (instruction and data cache misses are a large cost)

Remember: bypass the operating system!



C++ low latency coding techniques

- General considerations
- Compile-time dispatch
- Constexpr
- Variadic templates
- Loop unrolling
- Expression short-circuiting
- Signed vs unsigned comparisons
- Mixing float and doubles
- Branch prediction/reduction
- Exceptions
- Slowpath removal
- Avoiding allocations
- Fast containers
- Lambda functions

C++ low latency coding techniques

- **General considerations**
- Compile-time dispatch
- Constexpr
- Variadic templates
- Loop unrolling
- Expression short-circuiting
- Signed vs unsigned comparisons
- Mixing float and doubles
- Branch prediction/reduction
- Exceptions
- Slowpath removal
- Avoiding allocations
- Fast containers
- Lambda functions

Without repeating examples from other MeetingCpp talks this year:

- Move semantics
- Static assert
- Data member layout, padding and alignment
- False sharing
- Cache locality

C++ low latency coding techniques

- General considerations
- **Compile-time dispatch**
- Constexpr
- Variadic templates
- Loop unrolling
- Expression short-circuiting
- Signed vs unsigned comparisons
- Mixing float and doubles
- Branch prediction/reduction
- Exceptions
- Slowpath removal
- Avoiding allocations
- Fast containers
- Lambda functions

An old C vs C++ example, but very significant on modern hardware

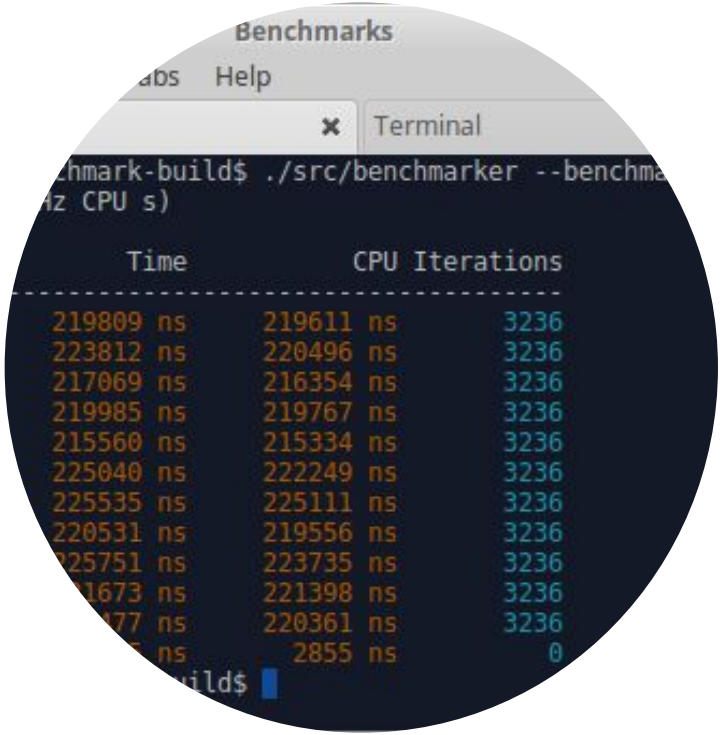
```
constexpr N = 10000;
int array[N];

std::sort(array, array + N, [](int a, int b) { return b < a; });
```

71us, std deviation 1.5us

```
int comparer(const void* a, const void* b) { return *(int*)a - *(int*)b; }
qsort(arr, N, sizeof(int), comparer);
```

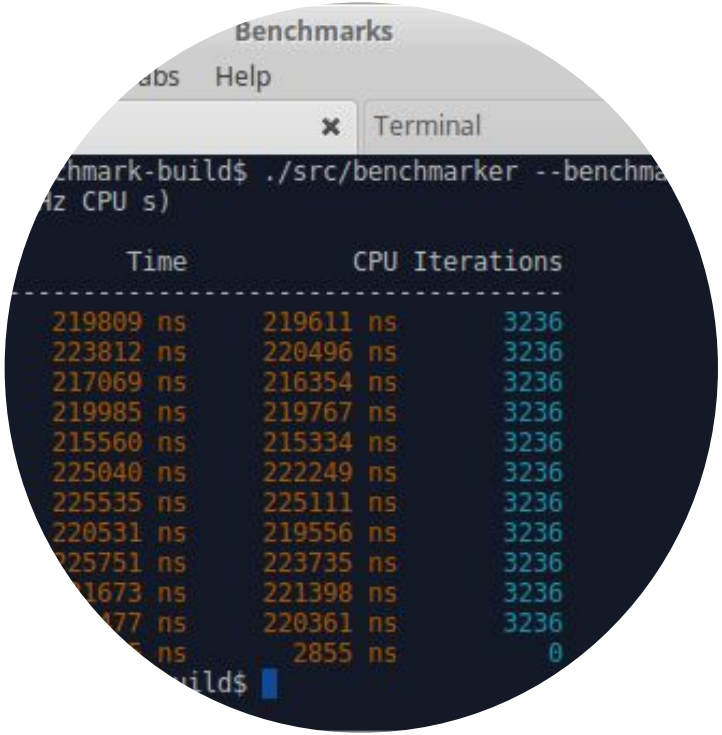
223us, std deviation 7us

By the way, google-benchmark is great



By the way, google-benchmark is great

More about measurement later on...



C++ low latency coding techniques

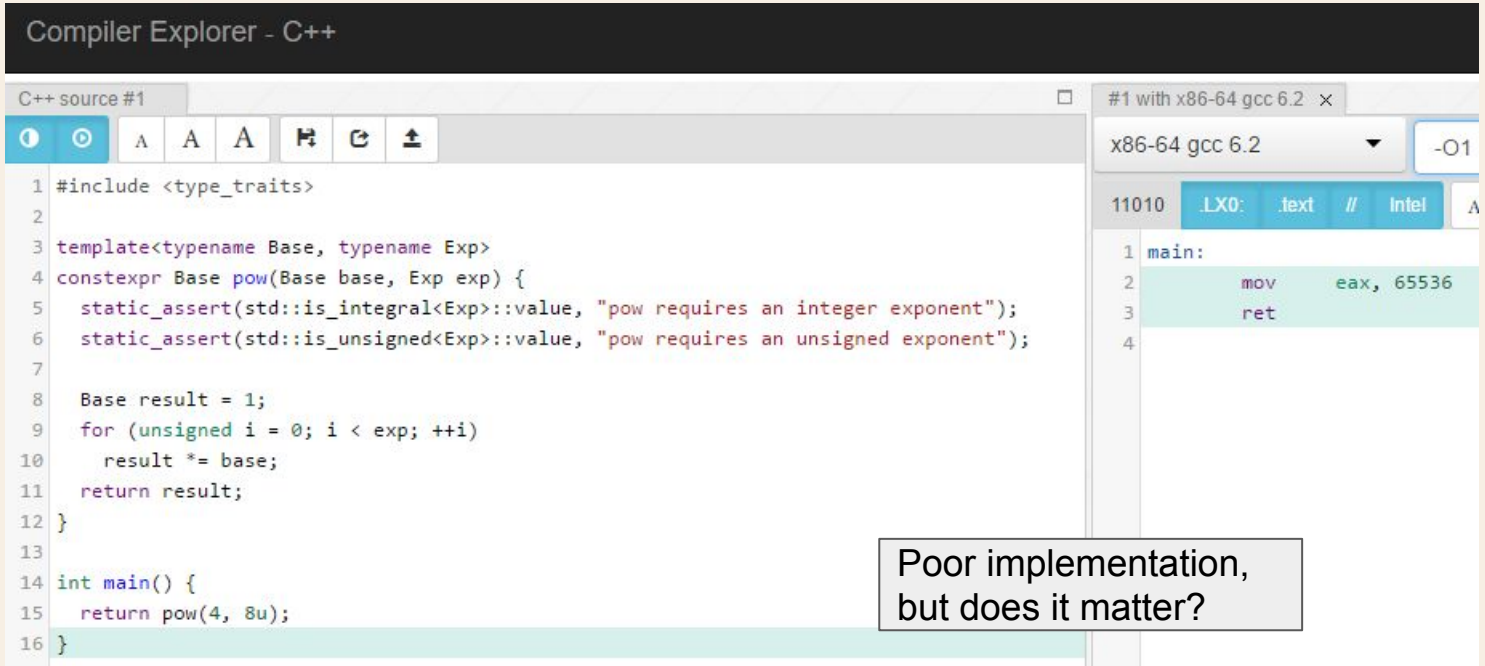
- General considerations
- Compile-time dispatch
- **Constexpr**
- Variadic templates
- Loop unrolling
- Expression short-circuiting
- Signed vs unsigned comparisons
- Mixing float and doubles
- Branch prediction/reduction
- Exceptions
- Slowpath removal
- Avoiding allocations
- Fast containers
- Lambda functions

Another case of trying to do as much at compile time as possible

- Yields faster compile times than templates

Another case of trying to do as much at compile time as possible

- Yields faster compile times than templates



C++ low latency coding techniques

- General considerations
- Compile-time dispatch
- Constexpr
- **Variadic templates**
- Loop unrolling
- Expression short-circuiting
- Signed vs unsigned comparisons
- Mixing float and doubles
- Branch prediction/reduction
- Exceptions
- Slowpath removal
- Avoiding allocations
- Fast containers
- Lambda functions

Motivating example: fast, typesafe and simple logging:

```
int main(int argc, char* args[]) {
  LOG("Hello myInt=% myChar=% myDouble=%", 1, ‘a’, 42.3);
}

$ ./logger
$ myInt=1 myChar=a myDouble=42.3
```

Disassembly (of hotpath):

```
movl $1, writeBuffer(%rip)
movb $97, writeBuffer+4(%rip)
movq %rax, writeBuffer+5(%rip)
```

The general idea:

- Do argument checking at compile time
- Also at compile time, define types that know how to:
  - Serialise the argument data
  - Deserialise/format the arguments given the serialised data
- In the hotpath, just serialise the data

```
LOG("Hello myInt=% myChar=% myDouble=%", 1, 'a', 42.3);
```

1	0	0	0	97	102	102	102	102	102	38	69	64

1 'a' 42.3

```
// API
#define LOG(formatString, ...) \
  static_assert(CountPlaceholders(formatString) == sizeof_args(__VA_ARGS__), \
                "Number of arguments mismatch"); \
  WriteLog(formatString, ##__VA_ARGS__);

// Useful helper functions
constexpr size_t CountPlaceholders(const char* formatString) {
  size_t result {0};
  while (formatString[0] != '\0') {
    result += (formatString[0] == '%') ? 1u : 0u;
    formatString++;
  }
  return result;
}

template <typename... Types>
constexpr unsigned sizeof_args(Types&&...) {
  return sizeof...(Types);
}
```

```
template <typename... Args>
static void WriteLog(const char* formatString, const Args&... args) {
  // calculate space needed in buffer, then get the buffer, then store the format
  // string, and pointer to static object that knows how to format these arguments
  char* argsBuffer = /* minor details */

  // copy args in the buffer
  CopyArgs(argsBuffer, args...);
}

// write a single arg to the buffer and continue with the tail
template<typename Arg, typename ... Args>
static char* CopyArgs(char* argsData, const Arg& arg, const Args&... args) {
  argsData = CopyArg(argsData, arg);
  return CopyArgs(argsData, args...);
}

// base case (terminator)
inline char* CopyArgs(char* argsData) { return argsData; }
```

```
// specialisation of CopyArg for trivially copyable types
template <typename T>
static char* CopyArg(char* argsData, T arg) {
  static_assert(std::is_trivially_copyable<T>::value, "trivially-copyable types only");
  *reinterpret_cast<T*>(argsData) = arg;
  return argsData + sizeof(arg);
}
```

Key point: all of this code gets inlined into simple `mov` statements, with non-trivial types only slightly more expensive

Have a look at the following repositories for working examples:

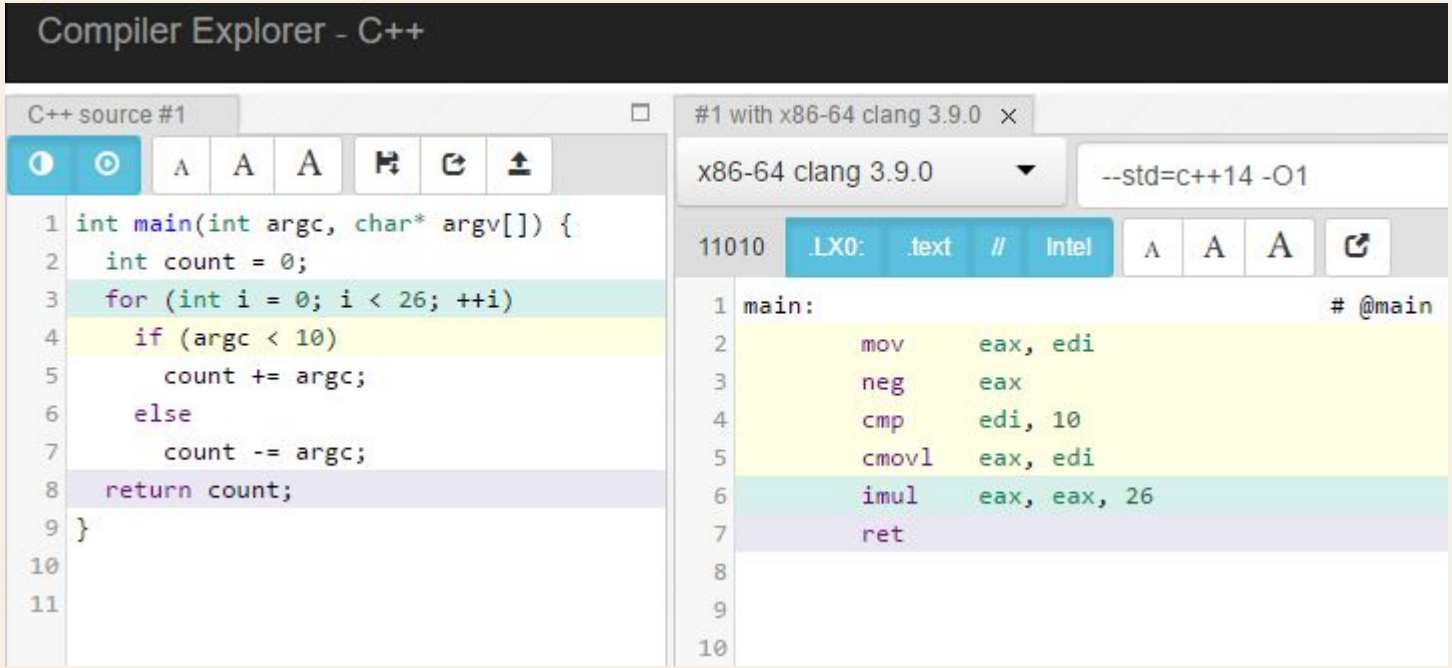
- https://github.com/maciekgajewski/Fast-Log
- https://github.com/carlcook/variadicLogging

C++ low latency coding techniques

- General considerations
- Compile-time dispatch
- Constexpr
- Variadic templates
- **Loop unrolling**
- Expression short-circuiting
- Signed vs unsigned comparisons
- Mixing float and doubles
- Branch prediction/reduction
- Exceptions
- Slowpath removal
- Avoiding allocations
- Fast containers
- Lambda functions

Generally, don’t bother, the compiler will figure it out

- In fact, some compilers ask you to not manually unroll



C++ low latency coding techniques

- General considerations
- Compile-time dispatch
- Constexpr
- Variadic templates
- Loop unrolling
- **Expression short-circuiting**
- Signed vs unsigned comparisons
- Mixing float and doubles
- Branch prediction/reduction
- Exceptions
- Slowpath removal
- Avoiding allocations
- Fast containers
- Lambda functions

Try to short circuit wherever possible

- Rewrite:

```
if (expensiveCheck() && inexpensiveCheck()) {}
```

- As:

```
if (inexpensiveCheck() && expensiveCheck()) {}
```

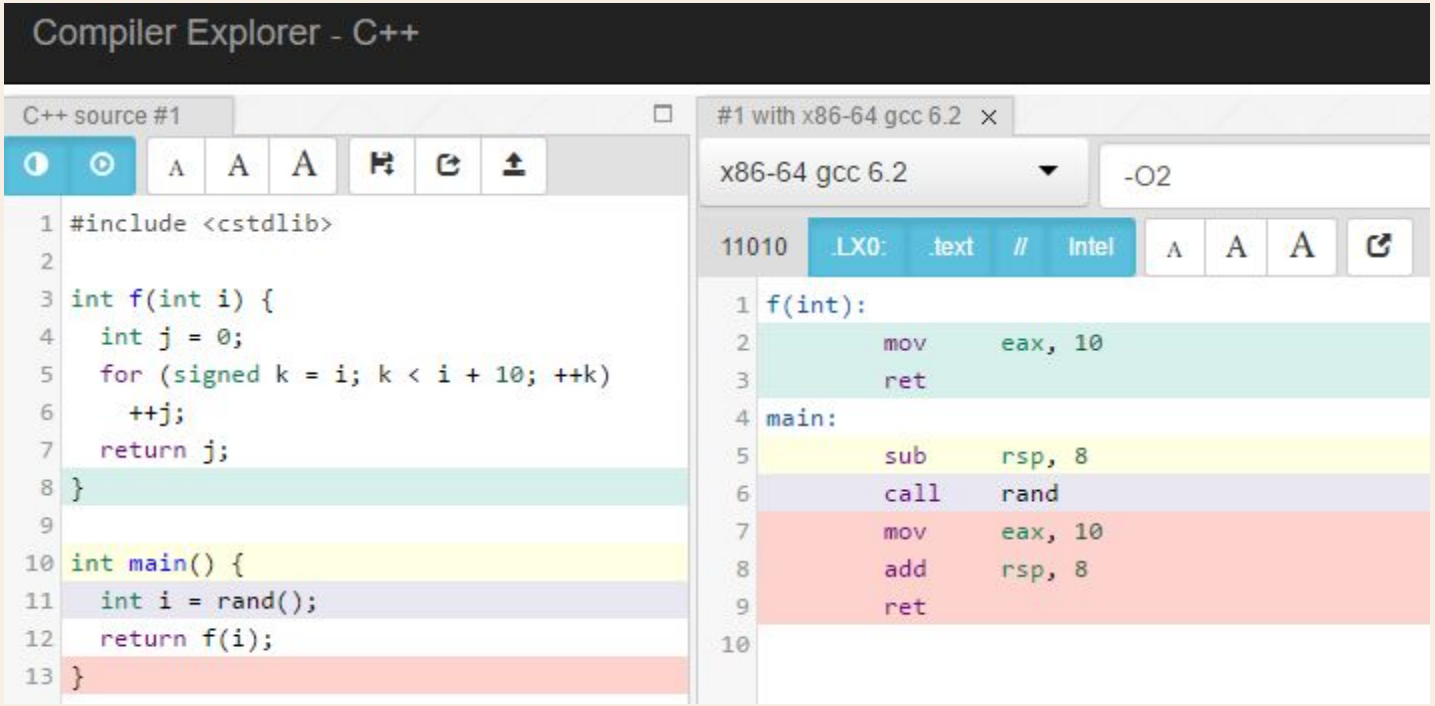
I know I just said not to try to beat the optimizer… but compilers can’t reorder statements where they are not sure of what the side effects are of doing so

C++ low latency coding techniques

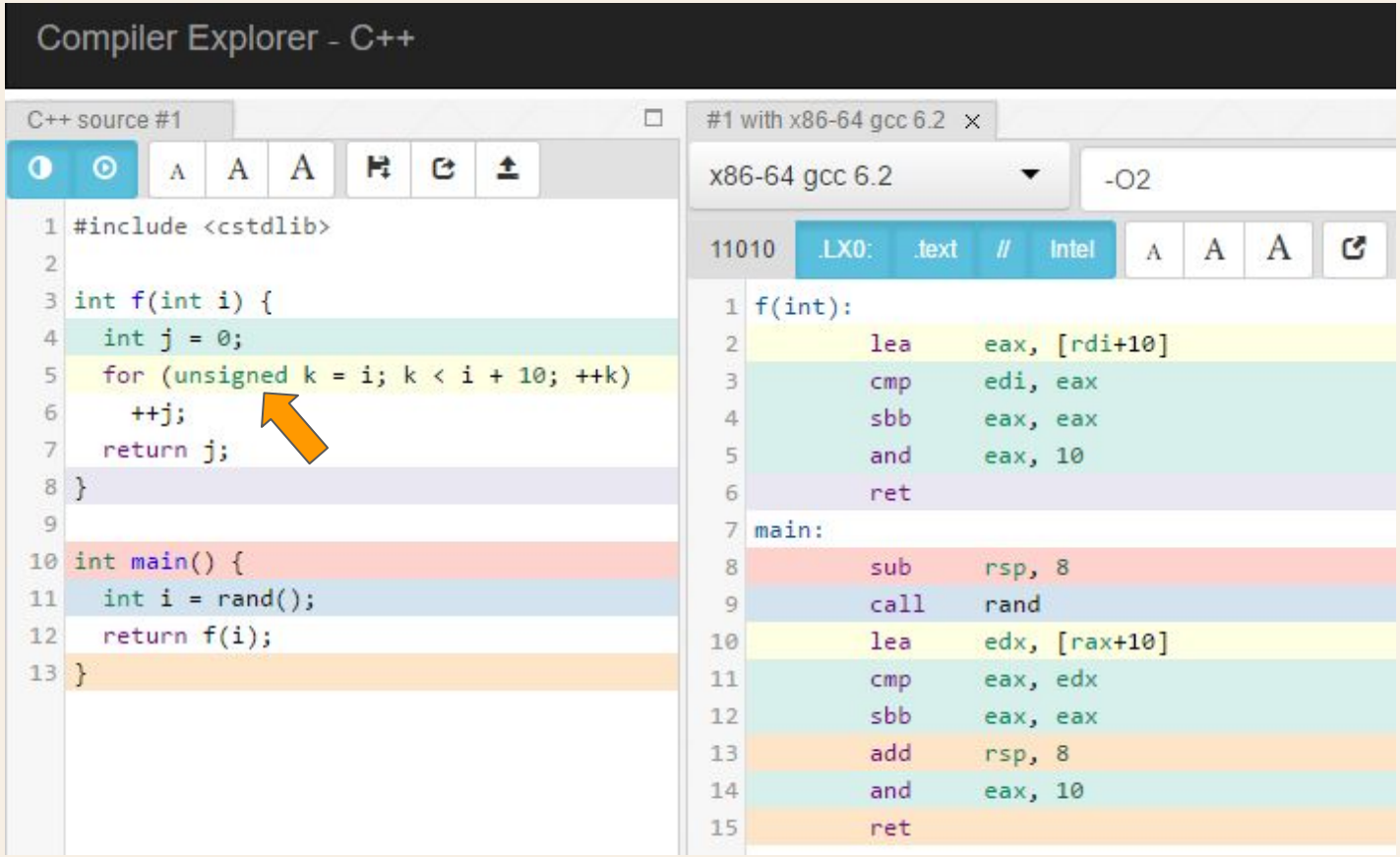
- General considerations
- Compile-time dispatch
- Constexpr
- Variadic templates
- Loop unrolling
- Expression short-circuiting
- **Signed vs unsigned comparisons**
- Mixing float and doubles
- Branch prediction/reduction
- Exceptions
- Slowpath removal
- Avoiding allocations
- Fast containers
- Lambda functions

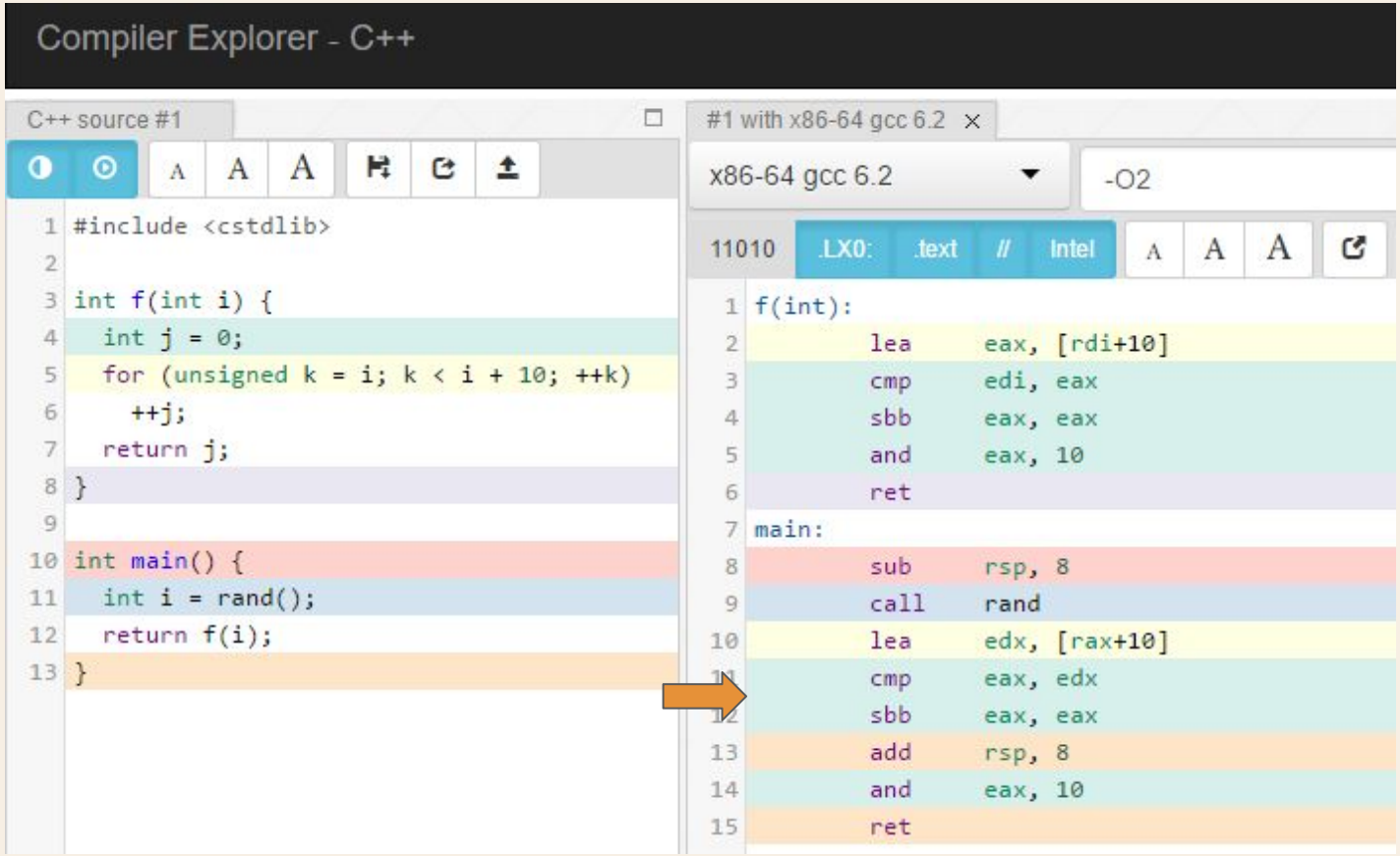
Key points:

1. There is no such thing as overflow for signed integers in C++
    - Writing a program that overflows is Undefined Behaviour
2. Unsigned integers in C++ must wrap (which can have a cost)
    - I.e. it is not Undefined Behaviour



Credit for example: http://www.airs.com/blog/archives/120


Compiler Explorer - C++
C++ source #1
#1 with x86-64 gcc 6.2
x86-64 gcc 6.2
-O2
11010
.LX0:
.text
//
Intel
1 #include <cstdlib>
2
3 int f(int i) {
4 int j = 0;
5 for (unsigned k = i; k < i + 10; ++k)
6 ++j;
7 return j;
8 }
9
10 int main() {
11 int i = rand();
12 return f(i);
13 }
1 f(int):
2 lea eax, [rdi+10]
3 cmp edi, eax
4 sbb eax, eax
5 and eax, 10
6 ret
7 main:
8 sub rsp, 8
9 call rand
10 lea edx, [rax+10]
11 cmp eax, edx
12 sbb eax, eax
13 add rsp, 8
14 and eax, 10
15 ret


Compiler Explorer - C++
C++ source #1
#1 with x86-64 gcc 6.2
x86-64 gcc 6.2
-O2
11010
.LX0:
.text
//
Intel
1 #include <cstdlib>
2
3 int f(int i) {
4 int j = 0;
5 for (unsigned k = i; k < i + 10; ++k)
6 ++j;
7 return j;
8 }
9
10 int main() {
11 int i = rand();
12 return f(i);
13 }
1 f(int):
2 lea eax, [rdi+10]
3 cmp edi, eax
4 sbb eax, eax
5 and eax, 10
6 ret
7 main:
8 sub rsp, 8
9 call rand
10 lea edx, [rax+10]
11 cmp eax, edx
12 sbb eax, eax
13 add rsp, 8
14 and eax, 10
15 ret

C++ low latency coding techniques

- General considerations
- Compile-time dispatch
- Constexpr
- Variadic templates
- Loop unrolling
- Expression short-circuiting
- Signed vs unsigned comparisons
- **Mixing float and doubles**
- Branch prediction/reduction
- Exceptions
- Slowpath removal
- Avoiding allocations
- Fast containers
- Lambda functions

Mixing floats and double can have a minor runtime cost:

- http://www.agner.org/optimize/optimizing_cpp.pdf

```
int main() {
  float a, b = rand();
  a = b * 1.23;
  return a;
}
```

Mixing floats and double can have a minor runtime cost

- http://www.agner.org/optimize/optimizing_cpp.pdf

```
int main() {
  float a, b = rand();
  a = b * 1.23;
  return a;
}
```

Default type of a floating point literal in C++ is `double`, not `float`

- Watch out for conversions
- Side note: If you are brave, consider `-ffast-math`

C++ low latency coding techniques

- General considerations
- Compile-time dispatch
- Constexpr
- Variadic templates
- Loop unrolling
- Expression short-circuiting
- Signed vs unsigned comparisons
- Mixing float and doubles
- **Branch prediction/reduction**
- Exceptions
- Slowpath removal
- Avoiding allocations
- Fast containers
- Lambda functions

- Compile-time branch prediction hints are a topic of much discussion (particularly within SG14)
  - I.e. gcc's `__builtin_expect`
- Remember, the hardware branch predictor is extremely good
  - This will (should) dominate any compiler reordering of branches
- Some exceptions to the rule:
  - A function that isn't called often, but when it is, it needs to be fast
  - Data/code layout *might* be better with compiler hints
- Branch prediction hints are definitely a quick win
  - But returns may diminish as you start to manually tune your code

Avoid this:

```
if (checkForErrorA())
  handleErrorA();
else if (checkForErrorB())
  handleErrorB();
else if (checkForErrorC())
  handleErrorC();
else
  executeHotpath();
```

Aim for this:

```
uint32_t errorFlags;
...
if (errorFlags)
  HandleError(errorFlags)
else
{
  ... hotpath
}
```

- I.e. handle/prepare for errors when they happen, not in the hotpath
- Also consider tearing down entire control path upon error
  - This means the control path does less error checking itself

C++ low latency coding techniques

- General considerations
- Compile-time dispatch
- Constexpr
- Variadic templates
- Loop unrolling
- Expression short-circuiting
- Signed vs unsigned comparisons
- Mixing float and doubles
- Branch prediction/reduction
- **Exceptions**
- Slowpath removal
- Avoiding allocations
- Fast containers
- Lambda functions

Trying is cheap, catching is expensive*

*assuming a table-driven approach, as supported by llvm/gcc

- Unthrown exceptions don't appear to have a measurable cost
  - **Disclaimer**: for server applications (not necessarily other domains)
- For the non-exceptional flow, it's just evaluating a conditional jump
  - The branch predictor is going guess correctly
- I've timed this with multiple autotrading applications compiled with exception handling disabled
  - No speedup (in fact, quite a fast form of error handling)
- Admittedly, it's hard to measure this, as there is an *Observer Effect*
    - By disabling exceptions you are altering the codepath

C++ low latency coding techniques

- General considerations
- Compile-time dispatch
- Constexpr
- Variadic templates
- Loop unrolling
- Expression short-circuiting
- Signed vs unsigned comparisons
- Mixing float and doubles
- Branch prediction/reduction
- Exceptions
- **Slowpath removal**
- Avoiding allocations
- Fast containers
- Lambda functions

Code is data - keep it minimal, and keep the slowpath code away from the hotpath

```
if (okay) {
  ...
  hotpath
}
else {
  sendToClient(“Order failed”);
  removeFromMap(orderId);
  logMessage(“Order failed”);
  ...
}
```

```
if (okay) {
  ...
  hotpath
}
else {
  HandleError();
}
```

- Keep functions that are used together close to each other
- Keep data that are used together close to each other
- Don’t declare slowpath functions as `inline`. In fact, consider using:

```
void __attribute__ ((noinline)) myUnimportantFunction()
{
  ...
}
```

C++ low latency coding techniques

- General considerations
- Compile-time dispatch
- Constexpr
- Variadic templates
- Loop unrolling
- Expression short-circuiting
- Signed vs unsigned comparisons
- Mixing float and doubles
- Branch prediction/reduction
- Exceptions
- Slowpath removal
- **Avoiding allocations**
- Fast containers
- Lambda functions

`new` and `delete` have an associated cost

- That cost is variable, but can be several microseconds
  - It can involve a system call (and page faults and TLB misses)
- Multithreaded applications also incur overhead in terms of allocations

Allocate ahead of time, hold onto memory forever (placement new)

- This also helps keeps related data together
- This also helps avoid long term fragmentation

Tip: don’t use swap. Memory is cheap. Buy more!

C++ low latency coding techniques

- General considerations
- Compile-time dispatch
- Constexpr
- Variadic templates
- Loop unrolling
- Expression short-circuiting
- Signed vs unsigned comparisons
- Mixing float and doubles
- Branch prediction/reduction
- Exceptions
- Slowpath removal
- Avoiding allocations
- **Fast containers**
- Lambda functions

`std::vector` is the data structure of choice

- Use it liberally (including as the underlying container for hash tables)
- Reserve ahead of time, and don't bother deleting items at runtime

Aim to denormalise data:

- Avoid doing lookups if the space/time tradeoff is acceptable
- Consider passing pointers around if you know the collection is stable

Lookups on associative containers can really thrash the cache

- Consider storing expensive hash values with the key
- Consider storing only pointers to objects in the container

http://www.reedbeta.com/blog/data-oriented-hash-table

C++ low latency coding techniques

- General considerations
- Compile-time dispatch
- Constexpr
- Variadic templates
- Loop unrolling
- Expression short-circuiting
- Signed vs unsigned comparisons
- Mixing float and doubles
- Branch prediction/reduction
- Exceptions
- Slowpath removal
- Avoiding allocations
- Fast containers
- **Lambda functions**

If only at runtime you know what the target is, you have no choice but to use `std::function`

If you know at compile time which target is to be run, then prefer lambdas

```
template <typename T>
void SendMessage(T&& target) {
  // populate and send message
  target(mBuffer);
  send(mBuffer);
}
```

```
SendMessage([&](auto& message) {
   message.field1 = x;
   ...
   message.field3 = z;
 });
```

Surprises and side notes

std::string

Older versions of gcc's implementation had copy on write semantics

- The reference counting mechanism was threadsafe
- You will see atomic reads/writes upon string copying/destruction

gcc 4.9.1

```
Benchmark                Time           CPU Iterations
----------------------------------------------------
StringOperations         141 ns         141 ns    4902860
```

gcc 5.4.0

```
Benchmark                Time           CPU Iterations
----------------------------------------------------
StringOperations         84 ns          84 ns     8239460
```

Build and link

- A few more statements in your code and something might no longer be inlined (and vice versa)
  - Frequently check the disassembly
- Profile guided optimisation (PGO):
  - Can give a speedup, but can also overfit the model
- Static linking tends to help
- Debug symbols have little impact (and are very useful)
  - Debug info just sits in non-cached memory until a core dump occurs
- Consider using Link Time Optimisation (LTO)
- Consider machine-specific compiler flags (`-march`, `-mtune`, etc)
- Do experiment with inlining

Userspace

Userspace networking, such as OpenOnload, allows you to read/write packets without any system calls

- This is great for sending order messages to exchanges
- But, if too many (potentially irrelevant) packets require reading, this will trample over your cache (as the default implementation is to handle all data that you are subscribed for)

I ended up putting userspace onto a different CPU and then pushed key data into a shared memory queue, which was read on demand

Concurrency

- One process is generally better than two
  - But not if the market data becomes the dominant factor
- Concurrency doesn’t necessarily gain you much
  - Sure you get multiple CPUs, but:
    - Shared caches (above L1)
    - Shared memory bus
    - Shared IO
    - Shared network
- If you use multiple processes, minimize communication between them
- Scale horizontally - split workload across multiple independent servers
  - Don’t think in terms of code, think in terms of problem, then hardware, then code

So you have a performance issue and plan on multi-threading?

you Now two problems. have

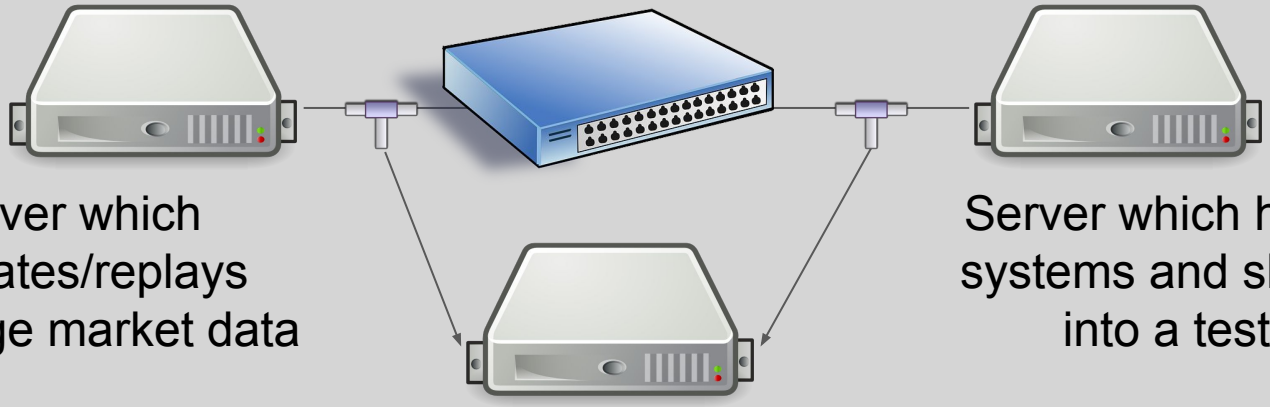
Accurate measurement

It’s easier to measure and then correct, than to be correct in the first place

I’m nearly always *wrong* when I try to make educated guesses about how different factors affect performance

High-resolution packet in/packet out timestamping is the source of truth

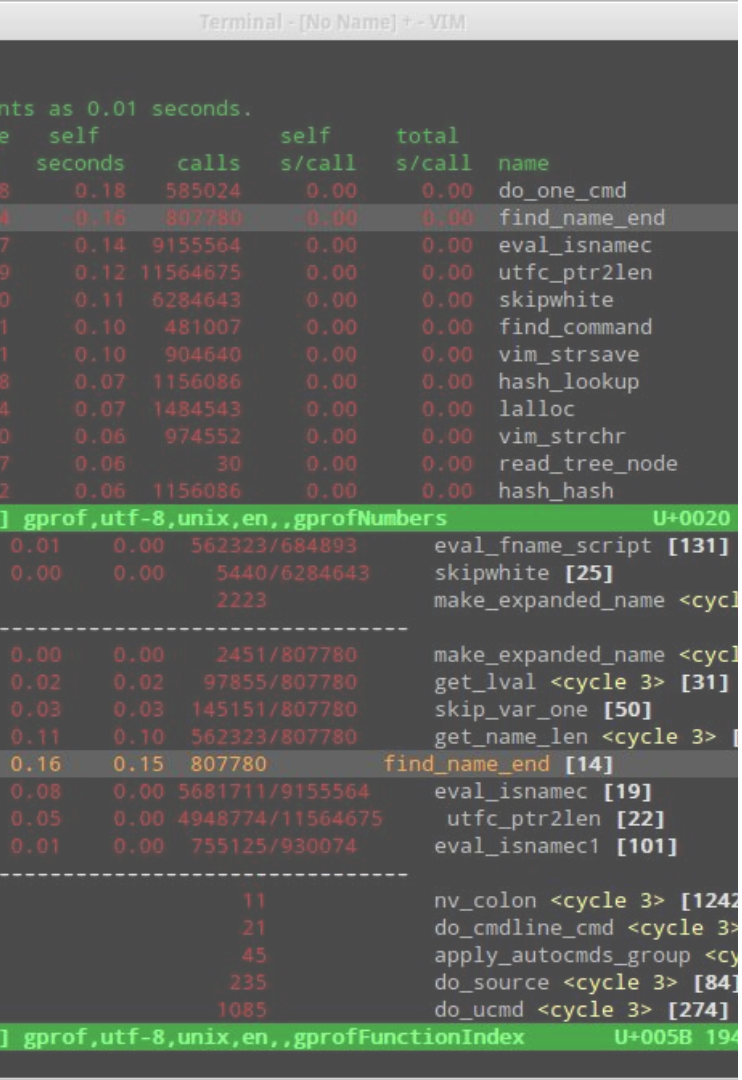
Switch with high precision oscillator and hardware timestamping (appended to each packet)



Server which simulates/replays exchange market data

Server which hosts trading systems and shoots orders into a test market

Server which captures and parses each network packet it sees, and calculates packet in/packet out time (accurate to a few nanoseconds)



Profiling is useful, but more as a debugger for poor performance, to complement any packet in/packet out timing you are doing

Profiling results are often too sensitive to the specific benchmark

Downsides of C++

You *do* pay for what you don't use

- Zero sized vectors may have a cost
- std::function allocates
- x86 has a stronger memory model than what C++11's memory model provides
  - You see fewer concurrency bugs, but at a performance cost
- Standard containers and allocators are somewhat non-deterministic in runtime cost

SG14

I don’t want to have to:

- Perform cache warming by hand
- Perform process control, thread affinity, memory affinity
- Perform performance tricks by hand, and use compiler-specific options
- Write my own containers, std::functions, strings, etc to avoid unnecessary allocations
- Write my own IPC (because the standard library doesn’t provide it)

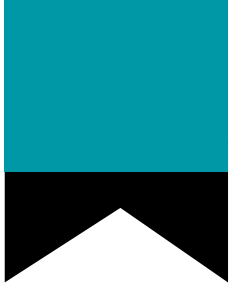
This is what SG14 aims to take care of

Side note: please see D0419 (LEWG) for my proposal of a non-allocating std::function (to be presented at WG21 in Kona next year)

stdext::inplace_function

```
stdext::inplace_function<void(int)> function = [] (int i)
{
  std::cout << i << std::endl;
};
function(42);

stdext::inplace_function<void(int), 32> function = [] (int i)
{
  ClassOf64Bytes c64{i};
  std::cout << c64 << std::endl;
};
// error: static assertion failed: Function too big to fit into buffer
```

https://github.com/WG21-SG14/SG14/blob/master/SG14/inplace_function.h



Summary

- Automated trading systems are about low latency, not throughput
  - Everything by default is (unfortunately) geared towards throughput
- Speed comes from:
  - Problem transformation/simplification/use of heuristics
  - Knowing C++ well, and exploiting it
  - A foundation of fast networks and well tuned servers
  - Measurement, measurement, measurement
- C++ does have some downsides/unexpected costs
- SG14 is fighting the good fight for us

Questions?

carl.cook@gmail.com

SG14

https://groups.google.com/a/isocpp.org/forum/#!forum/sg14

https://github.com/WG21-SG14/SG14