STRIVING FOR ULTIMATE LOW LATENCY

INTRODUCTION TO DEVELOPMENT OF LOW LATENCY SYSTEMS

Mateusz Pusz

September 18, 2017

LATENCY VS THROUGHPUT

LATENCY VS THROUGHPUT

Latency is the time required to perform some action or to produce some result. Measured in units of time like hours, minutes, seconds, nanoseconds or clock periods.

LATENCY VS THROUGHPUT

Latency is the time required to perform some action or to produce some result. Measured in units of time like hours, minutes, seconds, nanoseconds or clock periods.

Throughput is the number of such actions executed or results produced per unit of time. Measured in units of whatever is being produced per unit of time.

WHAT DO WE MEAN BY LOW LATENCY?

WHAT DO WE MEAN BY LOW LATENCY?

Low Latency allows human-unnoticeable delays between an input being processed and the corresponding output providing real time characteristics.

WHAT DO WE MEAN BY LOW LATENCY?

Low Latency allows human-unnoticeable delays between an input being processed and the corresponding output providing real time characteristics.

Especially important for internet connections utilizing services such as trading, online gaming and VoIP.

WHY DO WE STRIVE FOR LOW LATENCY?

WHY DO WE STRIVE FOR LOW LATENCY?

- In VoIP substantial delays between input from conversation participants may impair their communication

WHY DO WE STRIVE FOR LOW LATENCY?

- In VoIP substantial delays between input from conversation participants may impair their communication
- In online gaming a player with a high latency internet connection may show slow responses in spite of superior tactics or the appropriate reaction time

WHY DO WE STRIVE FOR LOW LATENCY?

- In VoIP substantial delays between input from conversation participants may impair their communication
- In online gaming a player with a high latency internet connection may show slow responses in spite of superior tactics or the appropriate reaction time
- Within capital markets the proliferation of algorithmic trading requires firms to react to market events faster than the competition to increase profitability of trades

HIGH-FREQUENCY TRADING (HFT)

A program trading platform that uses powerful computers to transact a large number of orders at very fast speeds

-- Investopedia

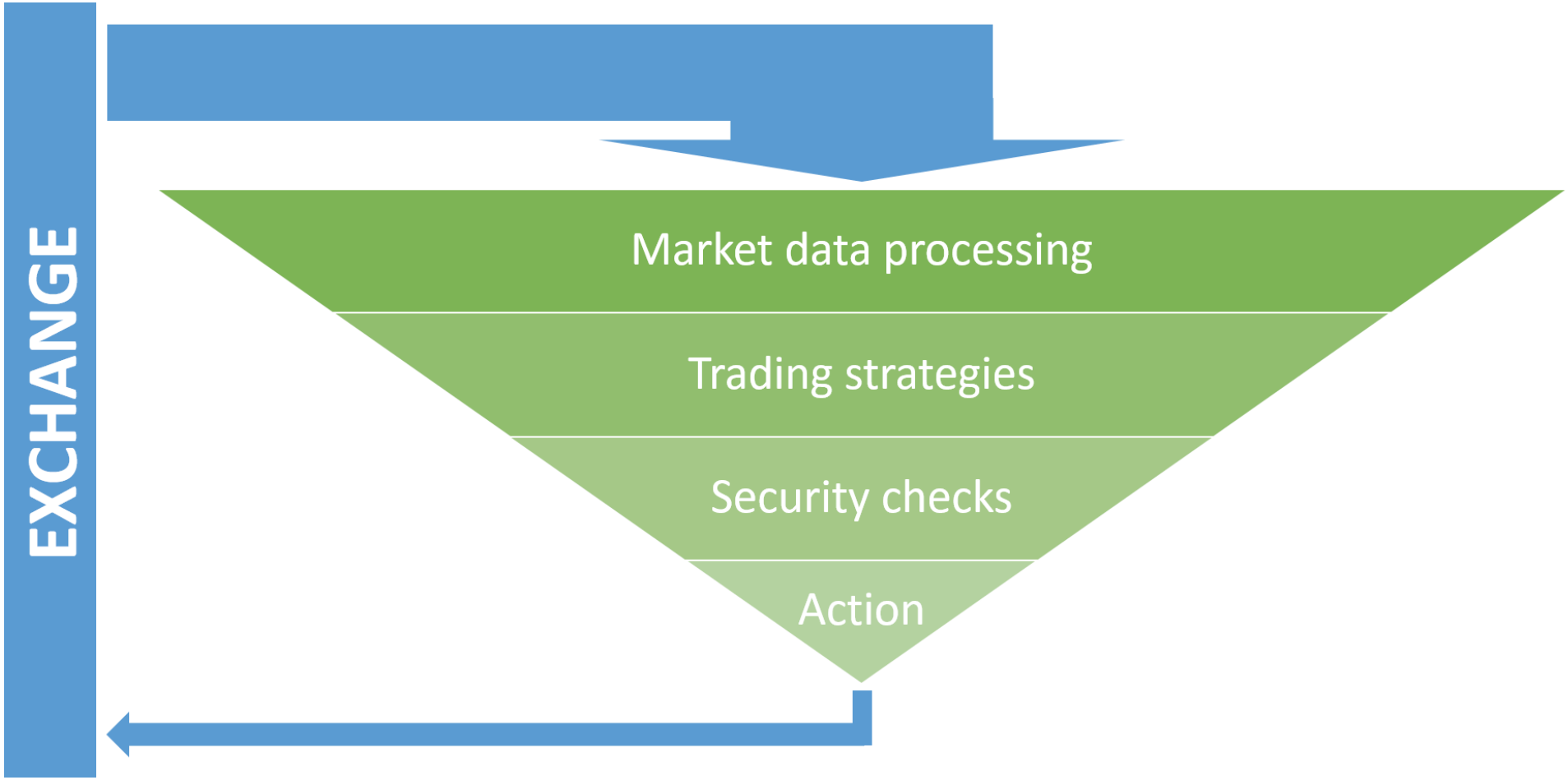
HIGH-FREQUENCY TRADING (HFT)

A program trading platform that uses powerful computers to transact a large number of orders at very fast speeds

-- Investopedia

- Using *complex algorithms* to analyze multiple markets and execute orders based on market conditions
- Buying and selling of securities *many times over a period of time* (often hundreds of times an hour)
- Done to *profit from time-sensitive opportunities* that arise during trading hours
- Implies *high turnover of capital* (i.e. one's entire capital or more in a single day)
- Typically, the traders with **the fastest execution speeds** are more profitable

MARKET DATA PROCESSING



HOW FAST DO WE DO?

ALL SOFTWARE APPROACH

1-10us

ALL HARDWARE APPROACH

100-1000ns

HOW FAST DO WE DO?

ALL SOFTWARE APPROACH

1-10us

ALL HARDWARE APPROACH

100-1000ns

HOW FAST DO WE DO?

ALL SOFTWARE APPROACH

1-10us

ALL HARDWARE APPROACH

100-1000ns

- Average human eye blink takes 350 000us (1/3s)
- Millions of orders can be traded that time

WHAT IF SOMETHING GOES WRONG?

WHAT IF SOMETHING GOES WRONG?

KNIGHT CAPITAL

- In 2012 was the largest trader in U.S. equities
- Market share
  - 17.3% on NYSE
  - 16.9% on NASDAQ
- Had approximately *$365 million* in cash and equivalents
- Average **daily** trading volume
  - 3.3 billion trades
  - trading over *21 billion* dollars

WHAT IF SOMETHING GOES WRONG?

KNIGHT CAPITAL

- In 2012 was the largest trader in U.S. equities
- Market share
  - 17.3% on NYSE
  - 16.9% on NASDAQ
- Had approximately *$365 million* in cash and equivalents
- Average **daily** trading volume
  - 3.3 billion trades
  - trading over *21 billion* dollars
- **pre-tax loss of $440 million in 45 minutes**

-- LinkedIn

How a software bug made Knight Capital lose $500M in a day & almost go bankrupt

C++ OFTEN NOT THE MOST IMPORTANT PART OF THE SYSTEM

- Low Latency network
- Modern hardware
- BIOS profiling
- Kernel profiling
- OS profiling

SPIN, PIN, AND DROP-IN

SPIN

- Don't sleep
- Don't context switch
- Prefer single-threaded scheduling
- Disable locking and thread support
- Disable power management
- Disable C-states
- Disable interrupt coalescing

SPIN, PIN, AND DROP-IN

SPIN

- Don't sleep
- Don't context switch
- Prefer single-threaded scheduling
- Disable locking and thread support
- Disable power management
- Disable C-states
- Disable interrupt coalescing

PIN

- Assign CPU affinity
- Assign interrupt affinity
- Assign memory to NUMA nodes
- Consider the physical location of NICs
- Isolate cores from general OS use
- Use a system with a single physical CPU

SPIN, PIN, AND DROP-IN

SPIN

- Don't sleep
- Don't context switch
- Prefer single-threaded scheduling
- Disable locking and thread support
- Disable power management
- Disable C-states
- Disable interrupt coalescing

PIN

- Assign CPU affinity
- Assign interrupt affinity
- Assign memory to NUMA nodes
- Consider the physical location of NICs
- Isolate cores from general OS use
- Use a system with a single physical CPU

DROP-IN

- Choose NIC vendors based on performance and availability of drop-in kernel bypass libraries
- Use the kernel bypass library

LET'S SCOPE ON THE SOFTWARE

CHARACTERISTICS OF LOW LATENCY SOFTWARE

- Typically only a **small part of code is really important** (fast path)

CHARACTERISTICS OF LOW LATENCY SOFTWARE

- Typically only a **small part of code is really important** (fast path)
- That code is not executed often
- When it is executed it has to
  - **start and finish as soon as possible**
  - have **predictable and reproducible** performance (low jitter)

CHARACTERISTICS OF LOW LATENCY SOFTWARE

- Typically only a **small part of code is really important** (fast path)
- That code is not executed often
- When it is executed it has to
  - **start and finish as soon as possible**
  - have **predictable and reproducible** performance (low jitter)
- Multithreading increases latency
  - it is about low latency and not throughput
  - concurrency (even on different cores) trashes CPU caches above L1, share memory bus, shares IO, shares network

CHARACTERISTICS OF LOW LATENCY SOFTWARE

- Typically only a **small part of code is really important** (fast path)
- That code is not executed often
- When it is executed it has to
  - **start and finish as soon as possible**
  - have **predictable and reproducible** performance (low jitter)
- Multithreading increases latency
  - it is about low latency and not throughput
  - concurrency (even on different cores) trashes CPU caches above L1, share memory bus, shares IO, shares network
- **Mistakes are really costly**
  - good error checking and recovery is mandatory
  - one second is 4 billion CPU instructions (a lot can happen that time)

HOW TO DEVELOP SOFTWARE THAT HAVE PREDICTABLE PERFORMANCE?

HOW TO DEVELOP SOFTWARE THAT HAVE PREDICTABLE PERFORMANCE?

It turns out that the more important question here is...

HOW NOT TO DEVELOP SOFTWARE THAT HAVE PREDICTABLE PERFORMANCE?

HOW NOT TO DEVELOP SOFTWARE THAT HAVE PREDICTABLE PERFORMANCE?

- In Low Latency system we care a lot about WCET (Worst Case Execution Time)
- In order to limit WCET we should limit the usage of specific C++ language features
- This is not only the task for developers but also for code architects

THINGS TO AVOID ON THE FAST PATH

1. C++ tools that trade performance for usability (e.g. `std::shared_ptr<T>`, `std::function<>`)
2. Throwing exceptions on likely code path
3. Dynamic polymorphism
4. Multiple inheritance
5. RTTI
6. Dynamic memory allocations

std::shared_ptr<T>

```
template<class T>
class shared_ptr;
```

- Smart pointer that retains **shared ownership** of an object through a pointer
- Several **`shared_ptr`** objects **may own the same object**
- The shared object is destroyed and its memory deallocated when the last remaining **`shared_ptr`** owning that object is either destroyed or assigned another pointer via **`operator=`** or **`reset()`**
- Support user provided **deleter**

std::shared_ptr<T>

```
template<class T>
class shared_ptr;
```

- Smart pointer that retains **shared ownership** of an object through a pointer
- Several **`shared_ptr`** objects **may own the same object**
- The shared object is destroyed and its memory deallocated when the last remaining **`shared_ptr`** owning that object is either destroyed or assigned another pointer via **`operator=`** or **`reset()`**
- Support user provided **deleter**

Too often overused by C++ programmers

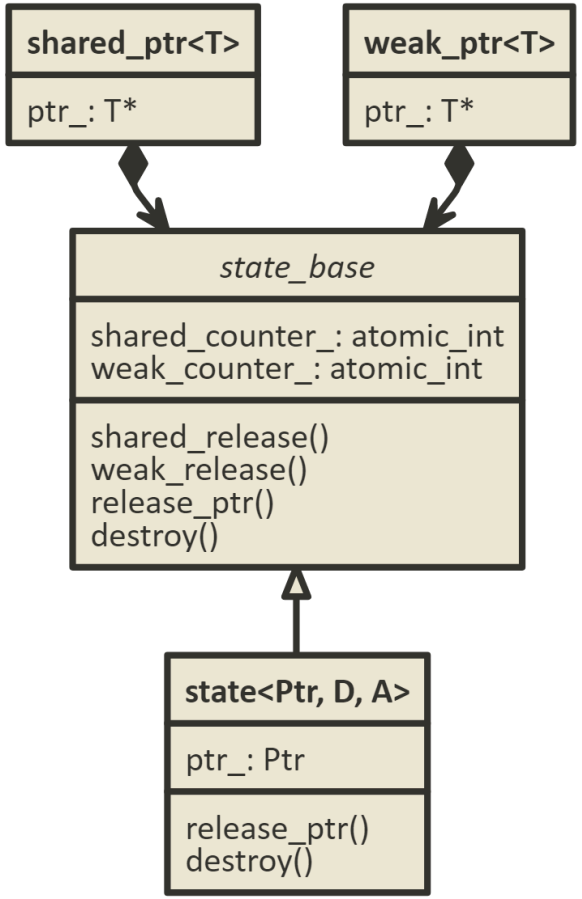
QUESTION: WHAT IS THE DIFFERENCE HERE?

```
void foo()
{
  std::unique_ptr<int> ptr{new int{1}};
  // some code using 'ptr'
}
```

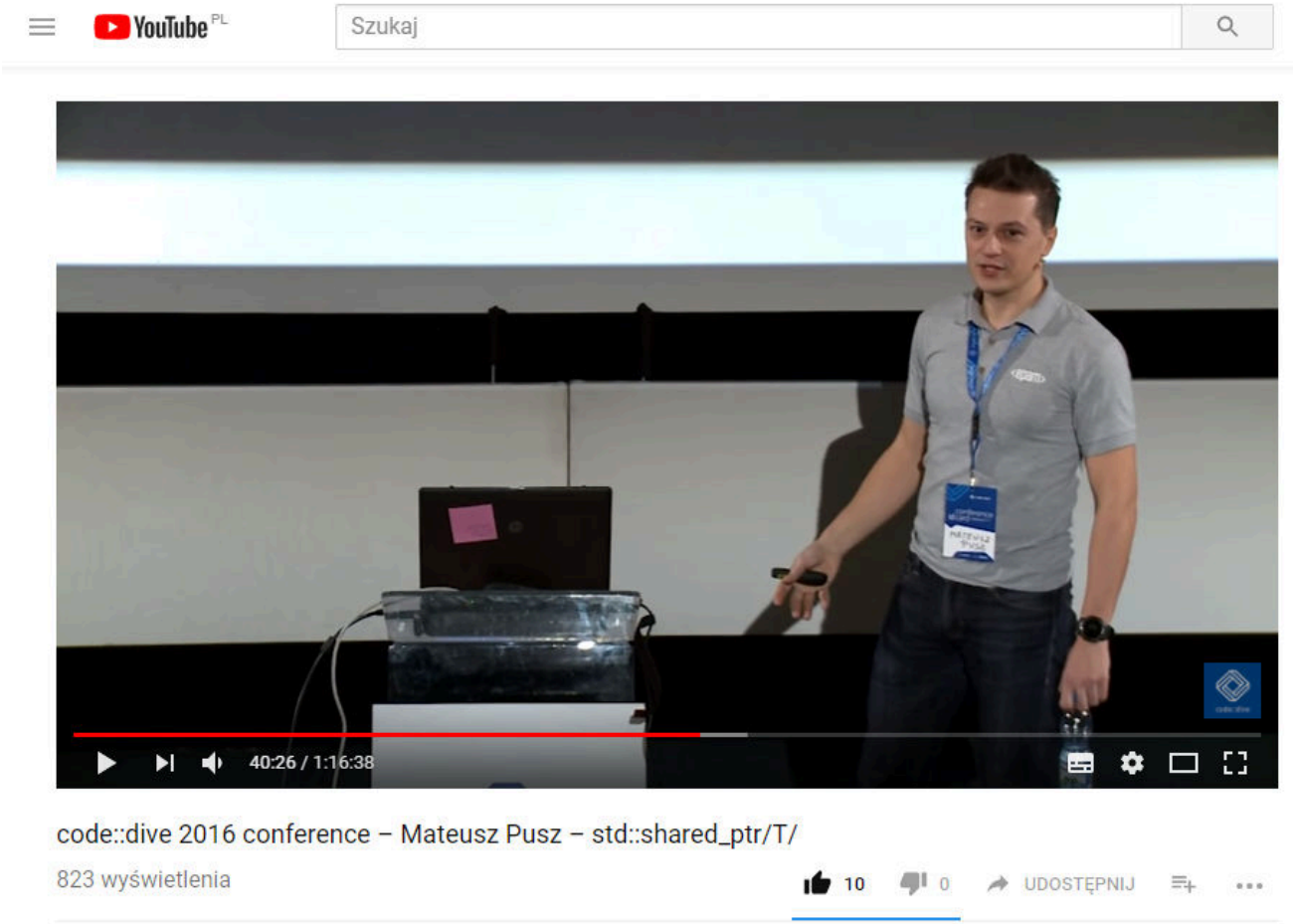
```
void foo()
{
  std::shared_ptr<int> ptr{new int{1}};
  // some code using 'ptr'
}
```

KEY `std::shared_ptr<T>` ISSUES

- Shared state
  - performance + memory footprint
- Mandatory synchronization
  - performance
- Type Erasure
  - performance
- `std::weak_ptr<T>` support
  - memory footprint
- Aliasing constructor
  - memory footprint



MORE INFO ON CODE::DIVE 2016



C++ EXCEPTIONS

C++ EXCEPTIONS

- Code generated by nearly all C++ compilers *does not introduce significant runtime overhead* for C++ exceptions

C++ EXCEPTIONS

- Code generated by nearly all C++ compilers *does not introduce significant runtime overhead* for C++ exceptions
- ... **if they are not thrown**

C++ EXCEPTIONS

- Code generated by nearly all C++ compilers *does not introduce significant runtime overhead* for C++ exceptions
- ... **if they are not thrown**
- Throwing an exception can take *significant and not deterministic time*

C++ EXCEPTIONS

- Code generated by nearly all C++ compilers *does not introduce significant runtime overhead* for C++ exceptions
- ... **if they are not thrown**
- Throwing an exception can take *significant and not deterministic time*
- *Advantages* of C++ exceptions usage
  - (if not thrown) actually can improve application performance
  - cannot be ignored!
  - simplify interfaces
  - make source code of likely path easier to reason about

C++ EXCEPTIONS

- Code generated by nearly all C++ compilers *does not introduce significant runtime overhead* for C++ exceptions
- ... **if they are not thrown**
- Throwing an exception can take *significant and not deterministic time*
- *Advantages* of C++ exceptions usage
  - (if not thrown) actually can improve application performance
  - cannot be ignored!
  - simplify interfaces
  - make source code of likely path easier to reason about

Not using C++ exceptions is not an excuse to write not exception-safe code!

POLYMORPHISM

DYNAMIC

```
class base {
  virtual void setup() = 0;
  virtual void run() = 0;
  virtual void cleanup() = 0;
public:
  virtual ~base() = default;
  void process()
  {
    setup();
    run();
    cleanup();
  }
};

class derived : public base {
  void setup() override   { /* ... */ }
  void run() override     { /* ... */ }
  void cleanup() override { /* ... */ }
};
```

POLYMORPHISM

DYNAMIC

```
class base {
  virtual void setup() = 0;
  virtual void run() = 0;
  virtual void cleanup() = 0;
public:
  virtual ~base() = default;
  void process()
  {
    setup();
    run();
    cleanup();
  }
};

class derived : public base {
  void setup() override   { /* ... */ }
  void run() override     { /* ... */ }
  void cleanup() override { /* ... */ }
};
```

- Additional pointer stored in an object
- Extra indirection (pointer dereference)
- Often not possible to devirtualize
- Not inlined
- Instruction cache miss

POLYMORPHISM

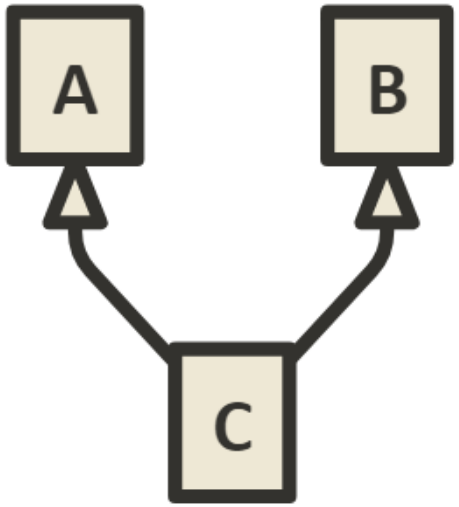
DYNAMIC

```
class base {
  virtual void setup() = 0;
  virtual void run() = 0;
  virtual void cleanup() = 0;
public:
  virtual ~base() = default;
  void process()
  {
    setup();
    run();
    cleanup();
  }
};

class derived : public base {
  void setup() override   { /* ... */ }
  void run() override     { /* ... */ }
  void cleanup() override { /* ... */ }
};
```

STATIC

```
template<class Derived>
class base {
public:
  void process()
  {
    static_cast<Derived*>(this)->setup();
    static_cast<Derived*>(this)->run();
    static_cast<Derived*>(this)->cleanup();
  }
};

class derived : public base<derived> {
  friend class base<derived>;
  void setup()   { /* ... */ }
  void run()     { /* ... */ }
  void cleanup() { /* ... */ }
};
```

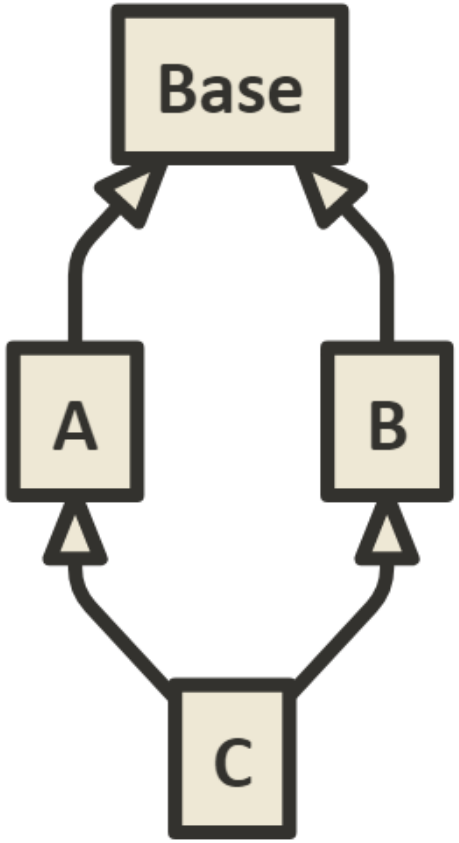
MULTIPLE INHERITANCE



MULTIPLE INHERITANCE

- **`this`** pointer adjustments needed to call member function (for not empty base classes)

MULTIPLE INHERITANCE



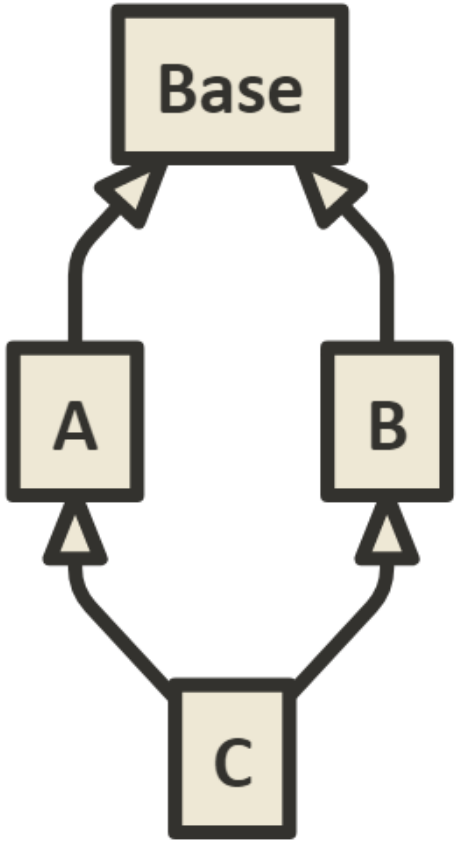
MULTIPLE INHERITANCE

- `this` pointer adjustments needed to call member function (for not empty base classes)

DIAMOND OF DREAD

- Virtual inheritance as an answer
- virtual in C++ means "determined at runtime"
- Extra indirection to access data members

MULTIPLE INHERITANCE



MULTIPLE INHERITANCE

- `this` pointer adjustments needed to call member function (for not empty base classes)

DIAMOND OF DREAD

- Virtual inheritance as an answer
- virtual in C++ means "determined at runtime"
- Extra indirection to access data members

Always consider composition before inheritance!

RUNTIME TYPE IDENTIFICATION (RTTI)

```
class base {
public:
  virtual ~base() = default;
  virtual void foo() = 0;
};
```

```
class derived : public base {
public:
  void foo() override;
  void boo();
};
```

RUNTIME TYPE IDENTIFICATION (RTTI)

```
class base {
public:
  virtual ~base() = default;
  virtual void foo() = 0;
};
```

```
class derived : public base {
public:
  void foo() override;
  void boo();
};
```

```
void foo(base& b)
{
  derived* d = dynamic_cast<derived*>(&b);
  if(d) {
    d->boo();
  }
}
```

RUNTIME TYPE IDENTIFICATION (RTTI)

```
class base {
public:
  virtual ~base() = default;
  virtual void foo() = 0;
};
```

```
class derived : public base {
public:
  void foo() override;
  void boo();
};
```

```
void foo(base& b)
{
  derived* d = dynamic_cast<derived*>(&b);
  if(d) {
    d->boo();
  }
}
```

Often the sign of a *smelly* design

RUNTIME TYPE IDENTIFICATION (RTTI)

```
class base {
public:
  virtual ~base() = default;
  virtual void foo() = 0;
};
```

```
class derived : public base {
public:
  void foo() override;
  void boo();
};
```

```
void foo(base& b)
{
  derived* d = dynamic_cast<derived*>(&b);
  if(d) {
    d->boo();
  }
}
```

- Traversing an inheritance tree
- Comparisons

RUNTIME TYPE IDENTIFICATION (RTTI)

```
class base {
public:
  virtual ~base() = default;
  virtual void foo() = 0;
};
```

```
class derived : public base {
public:
  void foo() override;
  void boo();
};
```

```
void foo(base& b)
{
  derived* d = dynamic_cast<derived*>(&b);
  if(d) {
    d->boo();
  }
}
```

```
void foo(base& b)
{
  if(typeid(b) == typeid(derived)) {
    derived* d = static_cast<derived*>(&b);
    d->boo();
  }
}
```

- Traversing an inheritance tree
- Comparisons

- Only one comparison of **`std::type_info`**
- Often only one runtime pointer compare

DYNAMIC MEMORY ALLOCATIONS

- *General purpose* operation
- *Nondeterministic* execution performance
- Causes memory *fragmentation*
- *Memory leaks* possible if not properly handled
- May *fail* (error handling is needed)

CUSTOM ALLOCATORS TO THE RESCUE

- Address *specific needs* (functionality and hardware constrains)
- Typically *low number of* dynamic memory *allocations*
- *Data structures* needed to manage big chunks of memory

CUSTOM ALLOCATORS TO THE RESCUE

- Address *specific needs* (functionality and hardware constrains)
- Typically *low number of* dynamic memory *allocations*
- *Data structures* needed to manage big chunks of memory

```
template<typename T> struct pool_allocator {
  T* allocate(std::size_t n);
  void deallocate(T* p, std::size_t n);
};
```

```
using pool_string = std::basic_string<char, std::char_traits<char>, pool_allocator>;
```

CUSTOM ALLOCATORS TO THE RESCUE

- Address *specific needs* (functionality and hardware constrains)
- Typically *low number of* dynamic memory *allocations*
- *Data structures* needed to manage big chunks of memory

```
template<typename T> struct pool_allocator {
  T* allocate(std::size_t n);
  void deallocate(T* p, std::size_t n);
};
```

```
using pool_string = std::basic_string<char, std::char_traits<char>, pool_allocator>;
```

Preallocation makes the allocator *jitter more stable*, helps in keeping *related data together* and avoiding long term *fragmentation*.

SMALL OBJECT OPTIMIZATION (SOO / SSO / SBO)

Prevent dynamic memory allocation for the (common) case of dealing with small objects

SMALL OBJECT OPTIMIZATION (SOO / SSO / SBO)

Prevent dynamic memory allocation for the (common) case of dealing with small objects

```
class sso_string {
  char* data_ = u_.sso_;
  size_t size_ = 0;
  union {
    char sso_[16] = "";
    size_t capacity_;
  } u_;
public:
  size_t capacity() const { return data_ == u_.sso_ ? sizeof(u_.sso_) - 1 : u_.capacity_; }
  // ...
};
```

NO DYNAMIC ALLOCATION

```
template<std::size_t MaxSize>
class inplace_string {
  std::array<value_type, MaxSize + 1> chars_;
public:
  // string-like interface
};
```

NO DYNAMIC ALLOCATION

```
template<std::size_t MaxSize>
class inplace_string {
  std::array<value_type, MaxSize + 1> chars_;
public:
  // string-like interface
};
```

```
struct db_contact {
  inplace_string<7> symbol;
  inplace_string<15> name;
  inplace_string<15> surname;
  inplace_string<23> company;
};
```

NO DYNAMIC ALLOCATION

```
template<std::size_t MaxSize>
class inplace_string {
  std::array<value_type, MaxSize + 1> chars_;
public:
  // string-like interface
};
```

```
struct db_contact {
  inplace_string<7> symbol;
  inplace_string<15> name;
  inplace_string<15> surname;
  inplace_string<23> company;
};
```

No dynamic memory allocations or pointer indirections guaranteed with the cost of possibly bigger memory usage

HOW TO DEVELOP SYSTEM WITH LOW-LATENCY CONSTRAINTS

HOW TO DEVELOP SYSTEM WITH LOW-LATENCY CONSTRAINTS

- Keep the **number of threads** close (less or equal) to the number available *physical CPU cores*

HOW TO DEVELOP SYSTEM WITH LOW-LATENCY CONSTRAINTS

- Keep the **number of threads** close (less or equal) to the number available *physical CPU cores*
- Separate **IO threads** from business logic thread (unless business logic is extremely lightweight)

HOW TO DEVELOP SYSTEM WITH LOW-LATENCY CONSTRAINTS

- Keep the **number of threads** close (less or equal) to the number available *physical CPU cores*
- Separate **IO threads** from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads

HOW TO DEVELOP SYSTEM WITH LOW-LATENCY CONSTRAINTS

- Keep the **number of threads** close (less or equal) to the number available *physical CPU cores*
- Separate **IO threads** from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures*, *data locality* principle

HOW TO DEVELOP SYSTEM WITH LOW-LATENCY CONSTRAINTS

- Keep the **number of threads** close (less or equal) to the number available *physical CPU cores*
- Separate **IO threads** from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures*, *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible

HOW TO DEVELOP SYSTEM WITH LOW-LATENCY CONSTRAINTS

- Keep the **number of threads** close (less or equal) to the number available *physical CPU cores*
- Separate **IO threads** from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures*, *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible
- *The simpler the code, the faster it is likely to be*

HOW TO DEVELOP SYSTEM WITH LOW-LATENCY CONSTRAINTS

- Keep the **number of threads** close (less or equal) to the number available *physical CPU cores*
- Separate **IO threads** from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures*, *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible
- *The simpler the code, the faster it is likely to be*
- Do not try to be smarter than the compiler

HOW TO DEVELOP SYSTEM WITH LOW-LATENCY CONSTRAINTS

- Keep the **number of threads** close (less or equal) to the number available *physical CPU cores*
- Separate **IO threads** from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures*, *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible
- *The simpler the code, the faster it is likely to be*
- Do not try to be smarter than the compiler
- Know the *language, tools, and libraries*

HOW TO DEVELOP SYSTEM WITH LOW-LATENCY CONSTRAINTS

- Keep the **number of threads** close (less or equal) to the number available *physical CPU cores*
- Separate **IO threads** from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures*, *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible
- *The simpler the code, the faster it is likely to be*
- Do not try to be smarter than the compiler
- Know the *language, tools, and libraries*
- Know your *hardware*!

HOW TO DEVELOP SYSTEM WITH LOW-LATENCY CONSTRAINTS

- Keep the **number of threads** close (less or equal) to the number available *physical CPU cores*
- Separate **IO threads** from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures*, *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible
- *The simpler the code, the faster it is likely to be*
- Do not try to be smarter than the compiler
- Know the *language, tools, and libraries*
- Know your *hardware*!
- *Bypass the kernel* (100% user space code)

HOW TO DEVELOP SYSTEM WITH LOW-LATENCY CONSTRAINTS

- Keep the **number of threads** close (less or equal) to the number available *physical CPU cores*
- Separate **IO threads** from business logic thread (unless business logic is extremely lightweight)
- Use fixed size *lock free queues* / *busy spins* to pass the data between threads
- Use optimal *algorithms*/*data structures*, *data locality* principle
- **Precompute**, use compile time instead of runtime whenever possible
- *The simpler the code, the faster it is likely to be*
- Do not try to be smarter than the compiler
- Know the *language, tools, and libraries*
- Know your *hardware*!
- *Bypass the kernel* (100% user space code)
- **Measure** performance… **ALWAYS**

THE MOST IMPORTANT RECOMMENDATION

THE MOST IMPORTANT RECOMMENDATION

Always measure your performance!

HOW TO MEASURE THE PERFORMANCE OF YOUR PROGRAMS

- Always measure **Release** version

```
cmake -DCMAKE_BUILD_TYPE=Release
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo
```

HOW TO MEASURE THE PERFORMANCE OF YOUR PROGRAMS

- Always measure **Release** version

```
cmake -DCMAKE_BUILD_TYPE=Release
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo
```

- Prefer *hardware based black box* performance meassurements

HOW TO MEASURE THE PERFORMANCE OF YOUR PROGRAMS

- Always measure **Release** version

```
cmake -DCMAKE_BUILD_TYPE=Release
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo
```

- Prefer *hardware based black box* performance meassurements
- In case that is not possible or you want to debug specific performance issue use *profiler*
- To gather meaningful stack traces *preserve frame pointer*

```
set(CMAKE_CXX_FLAGS_RELWITHDEBINFO
    "${CMAKE_CXX_FLAGS_RELWITHDEBINFO} -fno-omit-frame-pointer")
```

- Familiarize yourself with linux perf tools (xperf on Windows) and flame graphs
- Use tools like Intel VTune

HOW TO MEASURE THE PERFORMANCE OF YOUR PROGRAMS

- Always measure **Release** version

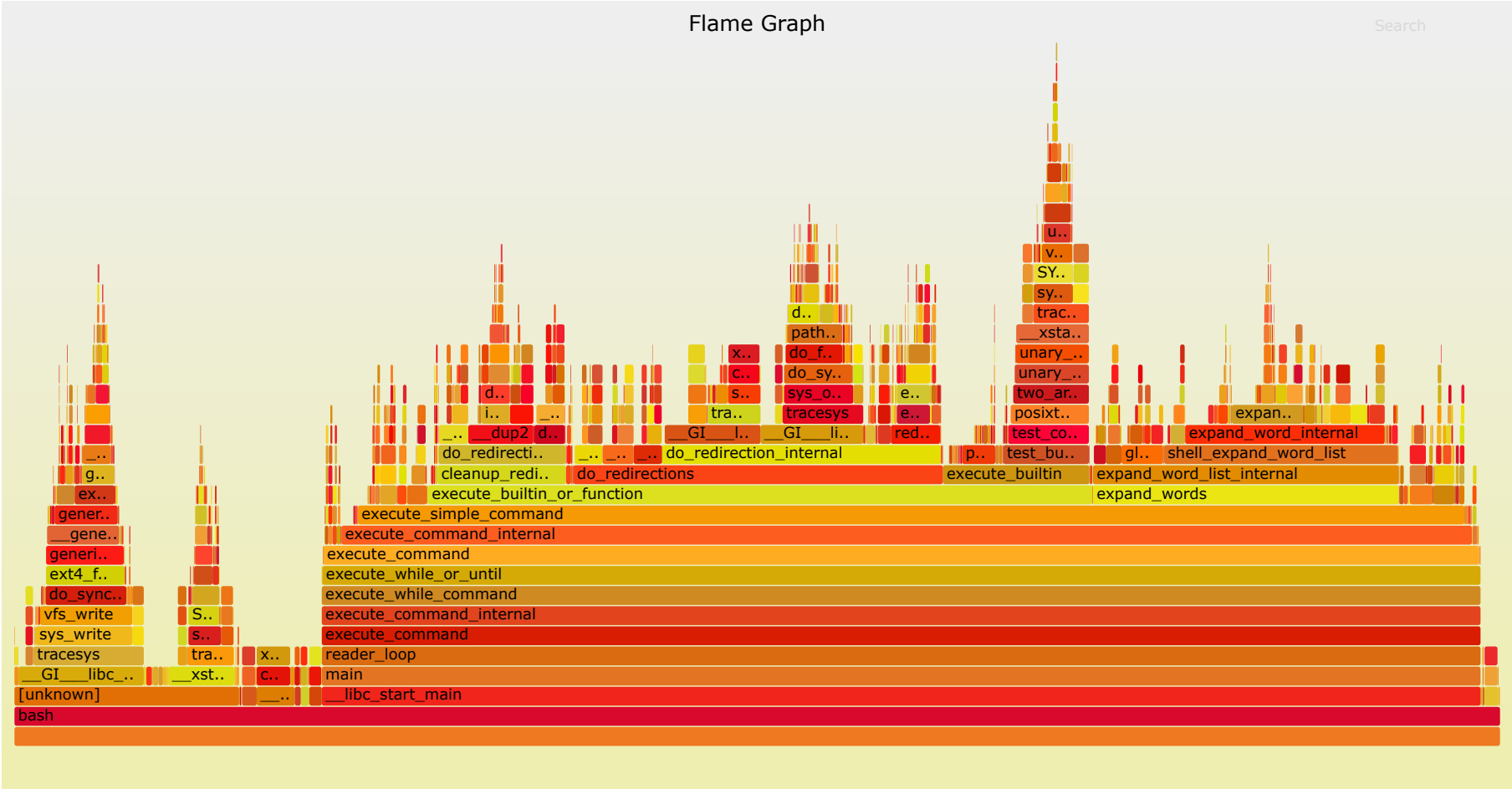
```
cmake -DCMAKE_BUILD_TYPE=Release
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo
```

- Prefer *hardware based black box* performance meassurements
- In case that is not possible or you want to debug specific performance issue use *profiler*
- To gather meaningful stack traces *preserve frame pointer*

```
set(CMAKE_CXX_FLAGS_RELWITHDEBINFO
    "${CMAKE_CXX_FLAGS_RELWITHDEBINFO} -fno-omit-frame-pointer")
```

- Familiarize yourself with linux perf tools (xperf on Windows) and flame graphs
- Use tools like Intel VTune
- *Verify output assembly code*

FLAMEGRAPH


Flame Graph
Search
u..
v..
SY..
sy..
trac..
__xsta..
unary_..
unary_..
two_ar..
posixt..
test_co..
test_bu..
expan..
expand_word_internal
gl..
shell_expand_word_list
expand_word_list_internal
expand_words
execute_builtin
p..
d..
path..
do_f..
do_sy..
sys_o..
tracesys
e..
e..
red..
x..
c..
s..
tra..
__GI____l..
__GI____li..
d..
i..
_..
__dup2
d..
do_redirecti..
do_redirection_internal
do_redirections
cleanup_redi..
execute_builtin_or_function
execute_simple_command
execute_command_internal
execute_command
execute_while_or_until
execute_while_command
execute_command_internal
execute_command
reader_loop
main
__libc_start_main
g..
ex..
gener..
__gene..
generi..
ext4_f..
do_sync..
vfs_write
sys_write
tracesys
__GI____libc_..
[unknown]
bash
S..
s..
tra..
__xst..
x..
c..
__..

CAUTION
Programming
is addictive
(and too much fun)