Code smells on hot paths

Mateusz Nowak

24.09.2020

Meeting C++ Online

Who am I

Team Manager

Located in Gdańsk, Poland

Who am I

Team Manager

Located in Gdańsk, Poland

Slides writing machine

cppcast listener

/r/cpp reader

Who am I

cppcast listener

All views expressed herein are those of my own and do not represent the opinions of any entity whatsoever with which I have been, am now, or will be affiliated.

Table of contents

What is hot path in code?

Frequently executed sequence of instructions

Performance bottleneck (usually)

Why some constructs may smell on hot paths?

Overhead introduced

Not optimal algotithm used

Doing unnecessary things

Making copies

Why copies are bad?

Copying things takes time

Why copies are bad?

Copying things takes time

May cause side-effects for others

When copies may be good?

When copy can be passed through registry

When doing a copy is cheaper than holding a lock

Missing move constructor and operator= smells

Especially if copy constructor and operator= are defined

Missing move constructor and operator= smells

Especially if copy constructor and operator= are defined

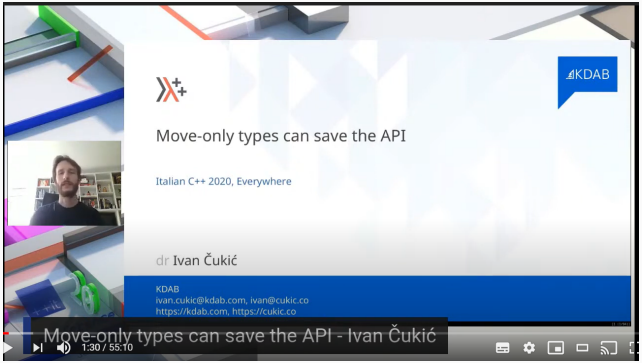
See: Rules for implicit constructors and operator=s

Also see: Rule of 0 and Rule of 5

Not deleted copy constructor and operator= smells

As it gives the user opportunity to make a copy

Not deleted copy constructor and operator= smells



Hot path smell:

Copies

(unless absolutely necessary)

Memory allocations

Allocations smell on hot paths

Allocating and deallocating memory takes time

Allocations smell on hot paths

Allocating and deallocating memory takes time

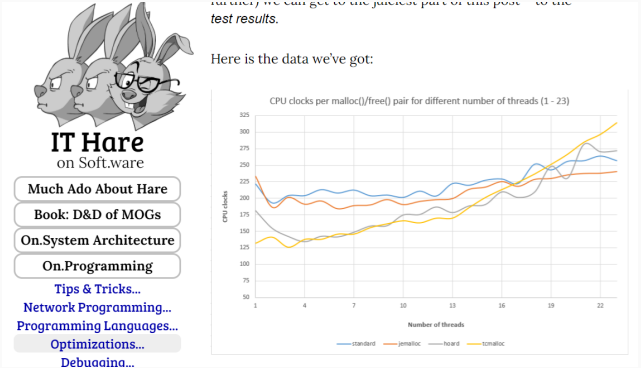
How to reduce its impact then?

Using faster allocator

jemalloc

TCMalloc

Using faster allocator



http://ithare.com/testing-memory-allocators-ptmalloc2-tcmalloc-hoard-jemalloc-while-trying-to-simulate-real-world-loads/

Using memory pools

Using memory pools

Allocate large block of memory once, and use parts of it

Using memory pools

Allocate large block of memory once, and use parts of it

Examples

- std::pmr:: / boost::container::pmr::
- TensorFlow BFCAllocator

vector grow performance - usual vs pmr

Run each test 10 times, each test is adding 4k ints

Intel Core i5-8250U, 8G Ram, Ubuntu on WSL, clang 10

Values are mean performance

vector grow performance - usual vs pmr

Run each test 10 times, each test is adding 4k ints

Intel Core i5-8250U, 8G Ram, Ubuntu on WSL, clang 10

Values are mean performance

	push_back	reserve + push_back
pure vector	5.79k items / s	6.52 items / s
pmr vector + unsynchronized_pool_resource	6.66k items / s	7.80k items / s

Hot path smell:

Unnecessary memory allocations

std:: containers

std::list

Container

Underlying buffer is not contiguous

Good for accessing head and tail, average for sequential access, bad for random accesses

std::vector

Container

Underlying buffer is contiguous

Good for sequential and random accesses

std::deque

Container

Scatter/gather buffer

Good for sequential and random accesses

Good for adding / removing elements at the top / bottom

std::array

Container

Underlying buffer is contiguous

Good for sequential and random accesses

Size is fixed at runtime

std::list vs std::vector vs std::deque vs std::array

In most cases for speed: std::list $<$ std::vector $<$ std::deque $<$ std::array

std::list vs std::vector vs std::deque vs std::array

std::list vs std::vector vs std::deque vs std::array

In most cases for speed: std::list $<$ std::vector $<$ std::deque $<$ std::array

Always measure for your case !

std::list superpowers

std::list superpowers

Inserting and erasing element is O(1)

std::list superpowers

Inserting and erasing element is O(1)

Iterators are never invalidated

std::vector superpowers

std::vector superpowers

Can grow if needed

std::vector superpowers

Can grow if needed

Can reserve space in advance

std::map

Associative container

Contains key value pairs with unique keys

Search, insertion, and removal are on average O(log(n))

std::unordered_map

Associative container

Contains key value pairs with unique keys

Search, insertion, and removal are on average O(1)

std::map vs std::unordered_map

Performance difference can be large

Run each test 10 times, each test is adding or searching 100k items

Intel Core i5-8250U, 8G Ram, Ubuntu on WSL, clang 10

std::map vs std::unordered_map

Performance difference can be large

Run each test 10 times, each test is adding or searching 100k items

Intel Core i5-8250U, 8G Ram, Ubuntu on WSL, clang 10

	std::map	std::unordered_map
insertion mean	91.4k items / s	115.9k items / s
insertion median	86.1k items / s	119.6k items / s
lookup mean	134.6k items / s	146.5k items / s
lookup median	133.3k items / s	145.k items / s

std::map superpowers

std::map superpowers

Elements are sorted by key

std::map superpowers

Elements are sorted by key

Iterating will have determined order

std::map superpowers

Elements are sorted by key

Iterating will have determined order

Can look for first greater or first not less key

std::unordered_map: operator[] vs .at()

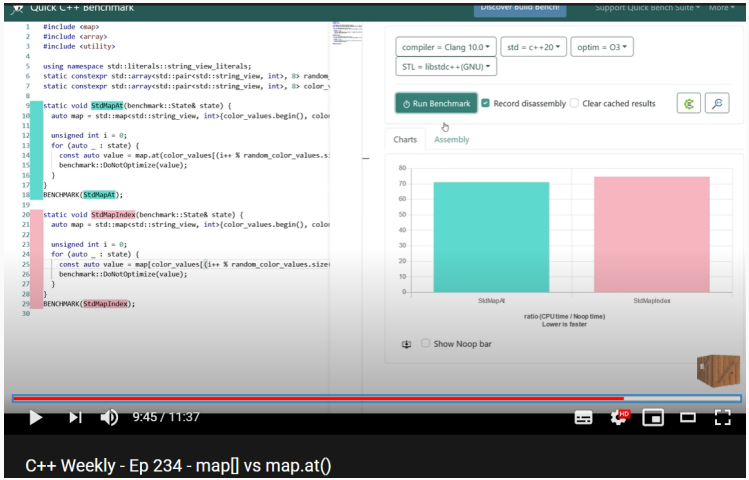
As per C++ Weekly Ep 234

std::unordered_map: operator[] vs .at()

As per C++ Weekly Ep 234

.at() in map is doing less work than operator[]

std::unordered_map: operator[] vs .at()



std:: maps vs alternative maps

2 options:

- 3rd party libraries implementations
- Using std::array

std::unordered_map vs absl::flat_hash_map

Performance difference can be large

Run each test 10 times, each test is adding or searching 100k items

Intel Core i5-8250U, 8G Ram, Ubuntu on WSL, clang 10

std::unordered_map vs absl::flat_hash_map

Performance difference can be large

Run each test 10 times, each test is adding or searching 100k items

Intel Core i5-8250U, 8G Ram, Ubuntu on WSL, clang 10

	std::unordered_map	absl::flat_hash_map
insertion mean	115.9k items / s	146.8k items / s
insertion median	119.6k items / s	145.5k items / s
lookup mean	146.5k items / s	152.1k items / s
lookup median	145.k items / s	154.2k items / s

std::unordered_map vs. constexpr std::array

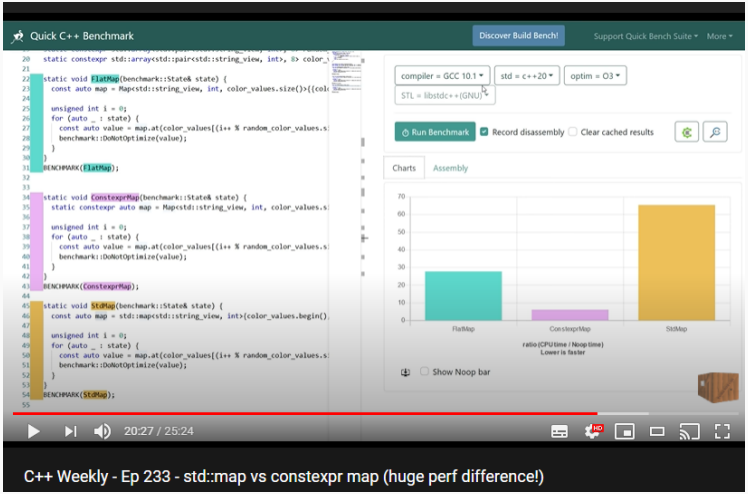
As per C++ Weekly Ep 233

std::unordered_map vs. constexpr std::array

As per C++ Weekly Ep 233

Replace std::map/unordered_map with std::array if string keys are known at compile time

std::unordered_map vs. constexpr std::array



Containers summary

If you need associative container:

Containers summary

If you need associative container:

- Use const array
- Go with 3rd party one
- Only if you absolutely can't, or measure it is not needed go to std:: one

Containers summary

If you need associative container:

- Use const array
- Go with 3rd party one
- Only if you absolutely can't, or measure it is not needed go to std:: one
- First go with unordered (hash)
- Then go to ordered one, only if you need order

Containers summary

If you need associative container:

- Use const array
- Go with 3rd party one
- Only if you absolutely can't, or measure it is not needed go to std:: one
- First go with unordered (hash)
- Then go to ordered one, only if you need order

If you need non-associative container:

Containers summary

If you need associative container:

- Use const array
- Go with 3rd party one
- Only if you absolutely can't, or measure it is not needed go to std:: one
- First go with unordered (hash)
- Then go to ordered one, only if you need order

If you need non-associative container:

- Go with std::array

Containers summary

If you need associative container:

- Use const array
- Go with 3rd party one
- Only if you absolutely can't, or measure it is not needed go to std:: one
- First go with unordered (hash)
- Then go to ordered one, only if you need order

If you need non-associative container:

- Go with std::array
- Then if you need contiguous one or know size in runtime advance go with std::vector

Containers summary

If you need associative container:

- Use const array
- Go with 3rd party one
- Only if you absolutely can't, or measure it is not needed go to std:: one
- First go with unordered (hash)
- Then go to ordered one, only if you need order

If you need non-associative container:

- Go with std::array
- Then if you need contiguous one or know size in runtime advance go with std::vector
- Then use std::deque

Containers summary

If you need associative container:

- Use const array
- Go with 3rd party one
- Only if you absolutely can't, or measure it is not needed go to std:: one
- First go with unordered (hash)
- Then go to ordered one, only if you need order

If you need non-associative container:

- Go with std::array
- Then if you need contiguous one or know size in runtime advance go with std::vector
- Then use std::deque
- Unless you remove and add elements in the middle a lot or need iterator stability, then use std::list

Hot path smell:

Misused containers

Slow strings

The obvious:

Operations on strings are slow

Using string as enum

If building on top of library that:

- returns string from known pool of strings
- accept string from know pool of string

it is tempting to also pass same strings in own code.

Example - external API

```
struct message {
  char type[7];
  uint16_t size;
  uint8_t content[];
};

typedef int (*message_handler)(const message&);

int message_request_handler(const message&);
int message_response_handler(const message&);
int message_ack_handler(const message&);
```

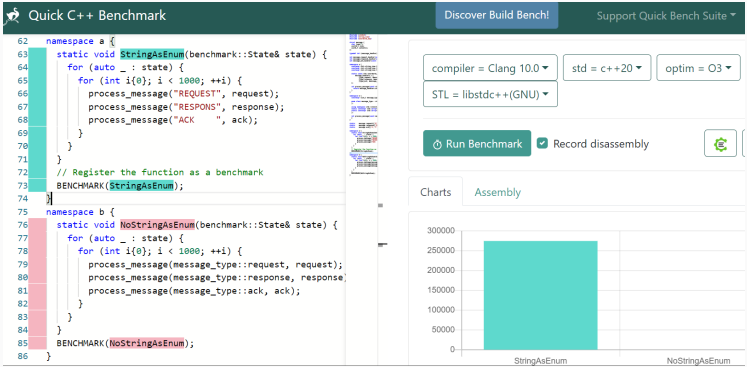
Example - using string as enum

```
constexpr std::string_view type_request = "REQUEST";
constexpr std::string_view type_response = "RESPONS";
constexpr std::string_view type_ack = "ACK     ";

static const std::unordered_map<std::string_view, message_handler>
    message_handlers = {
        {type_request, &message_request_handler},
        {type_response, &message_response_handler},
        {type_ack, &message_ack_handler},
};

int process_message(const std::string_view& type, const message& msg) {
  return message_handlers.at(type)(msg);
}
```

Example - not using string as enum

```
constexpr size_t message_types{3};

enum class message_type : size_t {
  request,
  response,
  ack,
};

using namespace std::literals::string_view_literals;
static constexpr std::array<std::string_view, message_types>
    message_type_string = {"REQUEST"sv, "RESPONS"sv, "ACK____"sv};

static constexpr std::array<message_handler, message_types> message_handlers = {
    &message_request_handler,
    &message_response_handler,
    &message_ack_handler,
};

int process_message(const message_type type, const message& msg) {
  return message_handlers.at((size_t)type)(msg);
}
```

Using string as enum is bad



Hot path smell:

Misusing strings

std::shared_ptr

Smart pointers

Explicitly showing ownership and lifetime

Automatically deleted when all owners go out of scope

std::unique_ptr - single owner, lives as long as owner scope

std::shared_ptr - multiple owners, lives as long as all owners scope

std::weak_ptr - non owning, reference to std::shared_ptr with validity check

Cost of passing std::shared_ptr

shared_ptr is not a pointer

Cost of passing std::shared_ptr

shared_ptr is not a pointer

It has copy & move operations defined

Cost of passing std::shared_ptr

shared_ptr is not a pointer

It has copy & move operations defined

Passing it by value results in copy

Cost of passing std::shared_ptr

shared_ptr is not a pointer

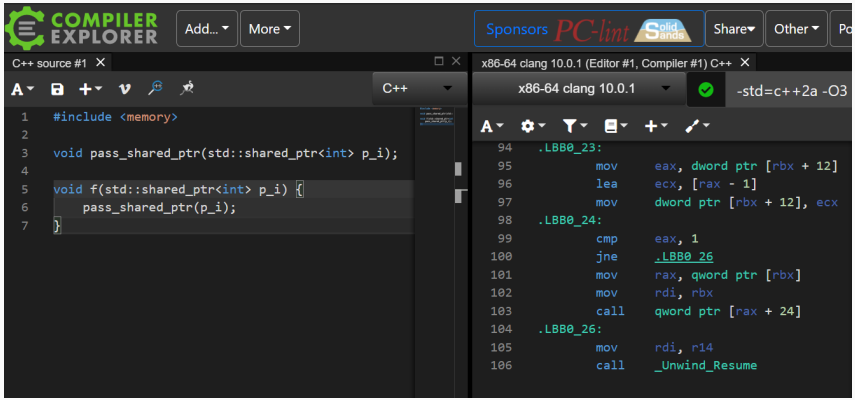
It has copy & move operations defined

Passing it by value results in copy

Copy operation needs to:

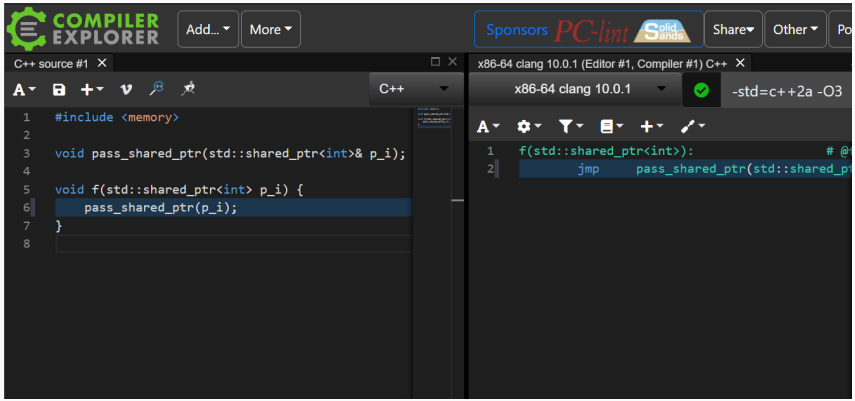
- Take a lock
- Check if is not nullptr
- Increase reference count

Cost of passing std::shared_ptr by value



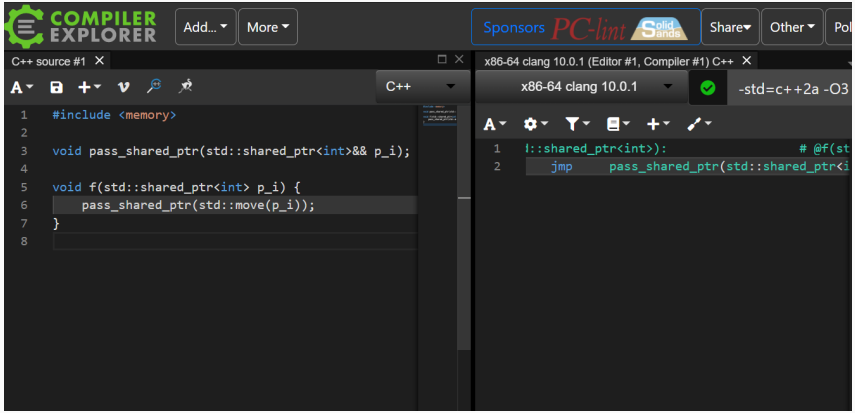
https://godbolt.org/z/KGM6vE

Cost of passing std::shared_ptr by value



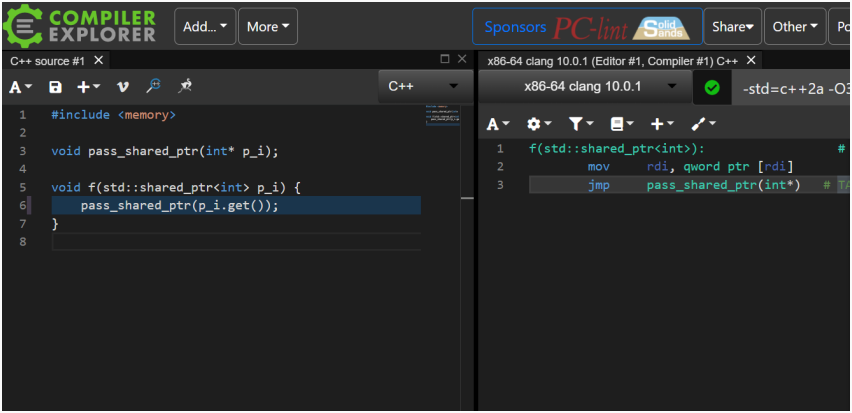
https://godbolt.org/z/nToKEv

Cost of passing std::shared_ptr by value



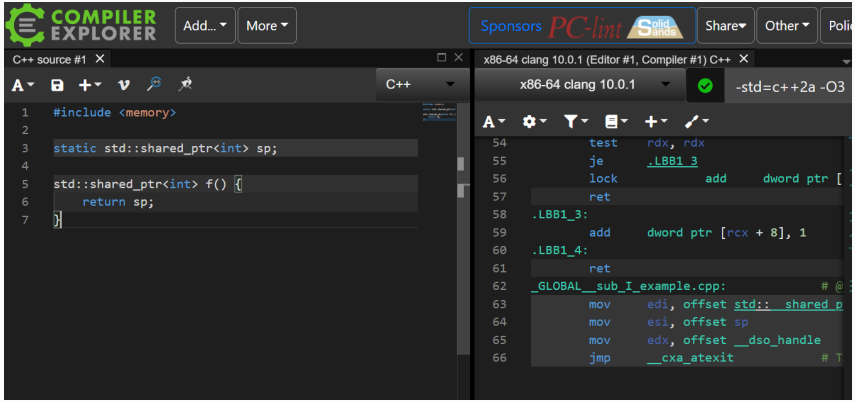
https://godbolt.org/z/vxbYjz

Cost of passing std::shared_ptr by value



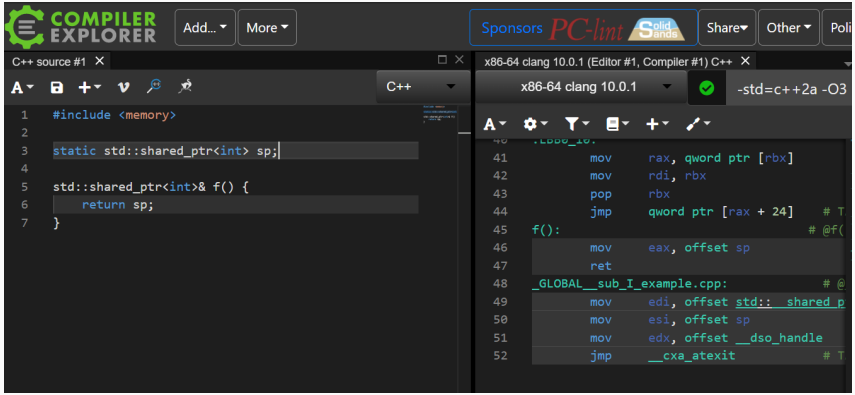
https://godbolt.org/z/6GdqWh

Cost of returning std::shared_ptr by value



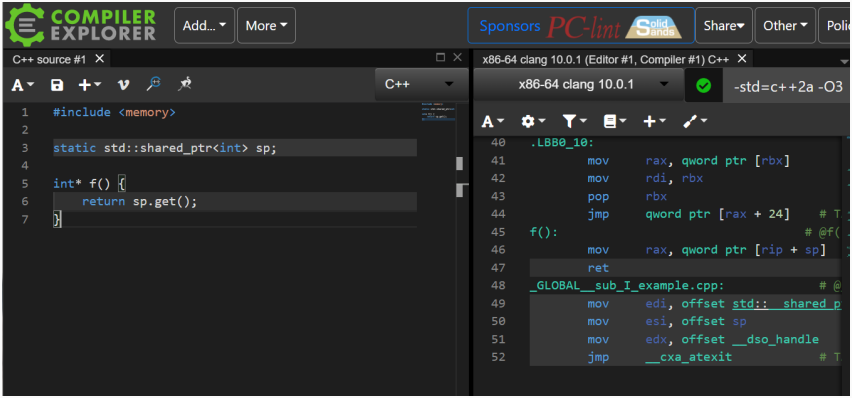
https://godbolt.org/z/Thv5bv

Cost of returning std::shared_ptr by value



https://godbolt.org/z/hab5Ef

Cost of returning std::shared_ptr by value



https://godbolt.org/z/jGc6oG

Rules of passing std::shared_ptr

The ownership is transferred

- Return by value
- Pass by value
- Pass by rvalue reference

Rules of passing std::shared_ptr

The ownership is not transferred

- Return reference to shared_ptr
- Return reference to object
- Return raw pointer to object
- Pass reference to shared_ptr
- Pass reference to object
- Pass raw pointer to object

Rules of passing std::shared_ptr

Please remember that:

- Returning std::shared_ptr by value is safer for the user
- Think if you can use std::unique_ptr instead
- Compiler optimize things, so profile first

Hot path smell:

Overused std::shared_ptr

Unnecessary evaluation

Unnecessary evaluation

Evaluating expression which result will be unused.

Example runtime logging function

```
enum class log_level {
  none = 0,
  error,
  warning,
};

template <typename... Args>
void log(log_level level, std::string format, Args... args);
```

Unnecessary evaluation - very bad example

```
void f() { log(log_level::warning, fmt::format("{} {}", 1, 2)); }
```

Unnecessary evaluation - bad example

```
void f() { log(log_level::warning, "{} {}", 1, 2); }
```

Lazy evaluation

Using language constructs that allow to defer evaluation of expression to the future

Lazy evaluation macro example

```
bool log_level_enabled(log_level level);

#define LOG(level, ...)                          \
  {                                              \
    if ((log_level_enabled(level))) {            \
      [[unlikely]] log(level, __VA_ARGS__);      \
    }                                            \
  }

void f() { LOG(log_level::warning, "{} {}", 1, 2); }
```

Lazy evaluation example

```
struct log_func {
  void operator()(log_level level, std::string format, auto... args);
};

template <typename L>
void log(log_level level, L creator) {
  if (log_level_enabled(level)) {
    [[unlikely]] std::apply(log_func{}, std::tuple_cat(std::tuple{level}, creator()));
  }
}

int main() {
  log(log_level::warning, []() { return std::tuple{std::string{"{} {}"}, 1, 2}; });
}
```

Hot path smell:

Unnecessary evaluation

Performance impact of missing or unnecessary keywords

Some compiler optimizations

Constant folding

Inlining

Compile time evaluation

Some compiler optimizations

Constant folding

Inlining

Compile time evaluation

Copy elision

RVO / NRVO

Helping compiler with optimizations

const everything possible

Helping compiler with optimizations

const everything possible

or better constexpr

Helping compiler with optimizations

const everything possible

or better constexpr

or better consteval

Helping compiler with optimizations

const everything possible

or better constexpr

or better consteval

get rid of unnecessary private member functions

Helping compiler with optimizations

const everything possible

or better constexpr

or better consteval

get rid of unnecessary private member functions

make translation unit local functions static, or put in anonymous namespace

Helping compiler with optimizations

const everything possible

or better constexpr

or better consteval

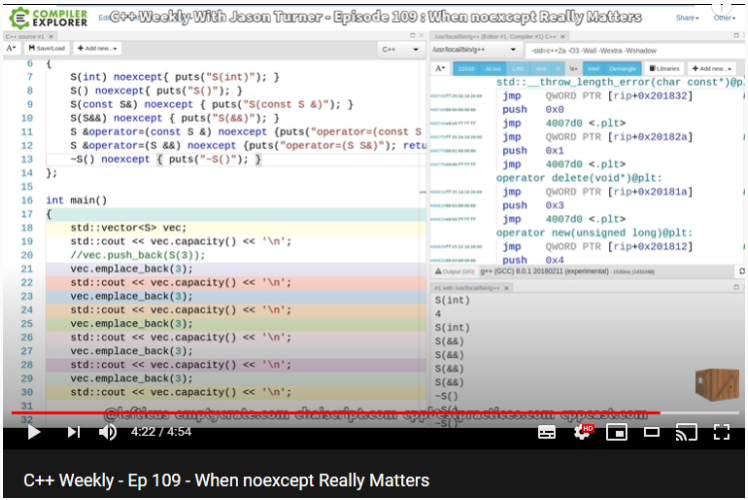
get rid of unnecessary private member functions

make translation unit local functions static, or put in anonymous namespace

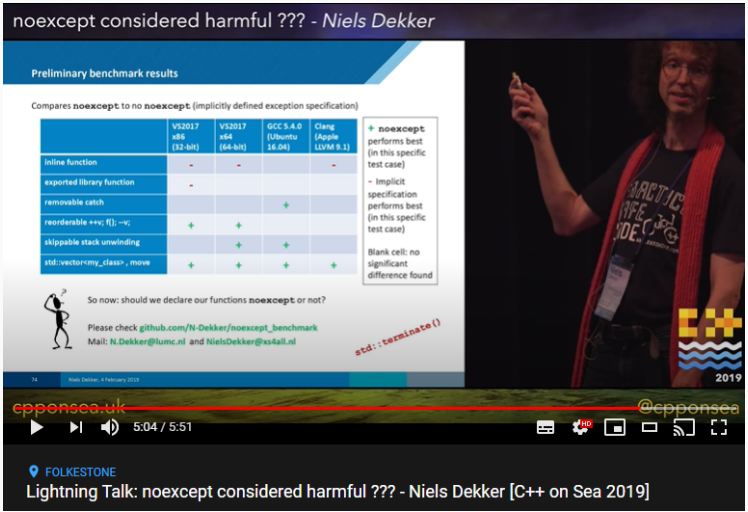
don't use std::move unless absolutely necessary

Don’t miss noexcept

Don't miss noexcept



Don’t miss noexcept



Hot path smell:

Forgetting or blindly typing keywords

Recap on smells:

Copies

Unnecessary memory allocations

Misused containers

Misusing strings

Overused std::shared_ptr

Unnecessary evaluation

Forgetting or blindly typing keywords

Thanks for your attention!

Mateusz Nowak

https://www.linkedin.com/in/mateusz-nowak-qq/

https://github.com/noqqaqq

@noqqaqq

Feedback is appreciated ☺