

# 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

1. Making copies
2. Memory allocations
3. `std::` containers
4. Slow strings
5. `std::shared_ptr`
6. Unnecessary evaluation
7. Performance impact of missing or unnecessary keywords

# 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 algorithm 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

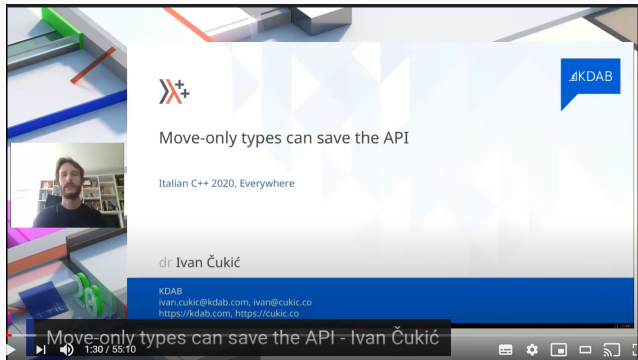
See: Rules for implicit constructors and operator=s

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

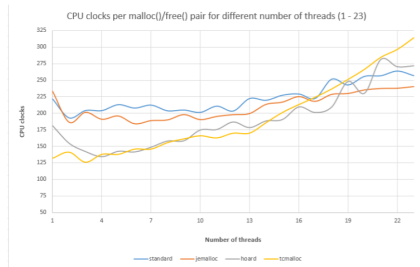
TCMalloc

# Using faster allocator



allocator, we can get to the juiciest part of this post – to the test results.

Here is the data we've got:



<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**

---

Container

Underlying buffer is not contiguous

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

Container

Underlying buffer is contiguous

Good for sequential and random accesses

Container

Scatter/gather buffer

Good for sequential and random accesses

Good for adding / removing elements at the top / bottom



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`

Blog blog("Baptiste Wicht");

## C++ Containers Benchmark: vector/list/deque and plf::colony

Baptiste Wicht — 2017-05-21 12:46 — 6 Comments — [Source](#)

## `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

Inserting and erasing element is  $O(1)$

Inserting and erasing element is  $O(1)$

Iterators are never invalidated





Can grow if needed

Can grow if needed

Can reserve space in advance

Associative container

Contains key value pairs with unique keys

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

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

Elements are sorted by key



Elements are sorted by key

Iterating will have determined order

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()`

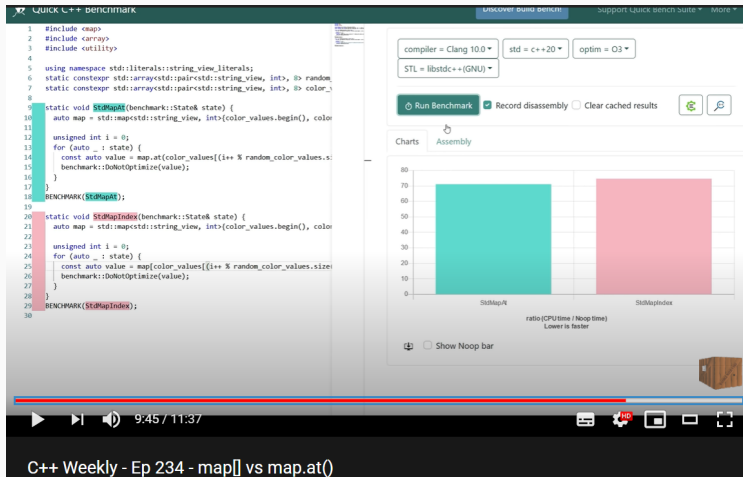
As per C++ Weekly Ep 234

## `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

	<code>std::unordered_map</code>	<code>absl::flat_hash_map</code>
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`

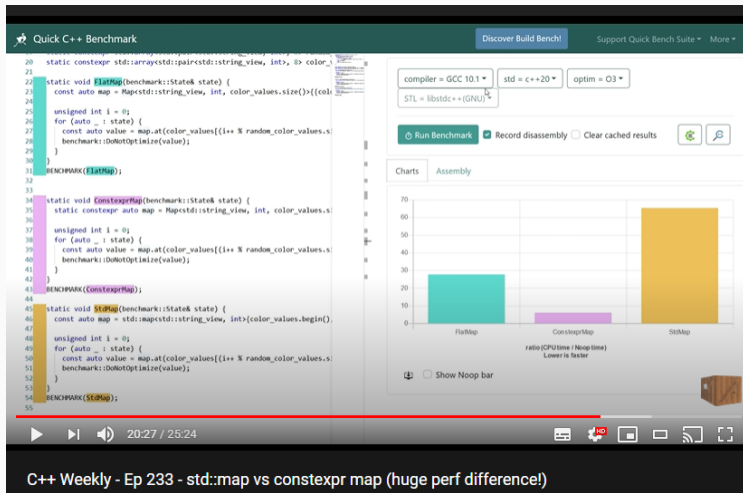
As per C++ Weekly Ep 233

## `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



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

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 known 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};

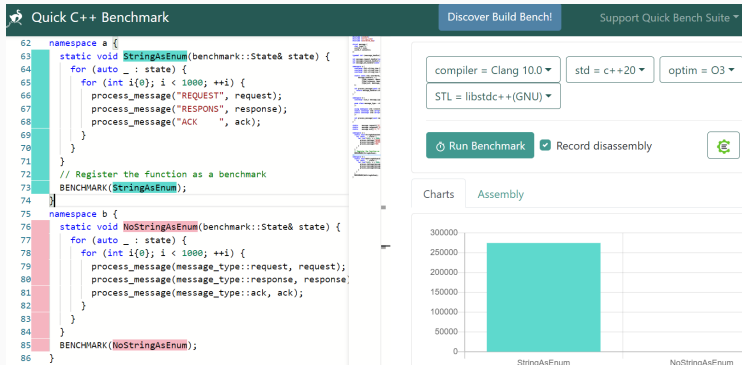
enum class message_type : size_t {
    request,
    response,
    ack,
};

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**

---

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

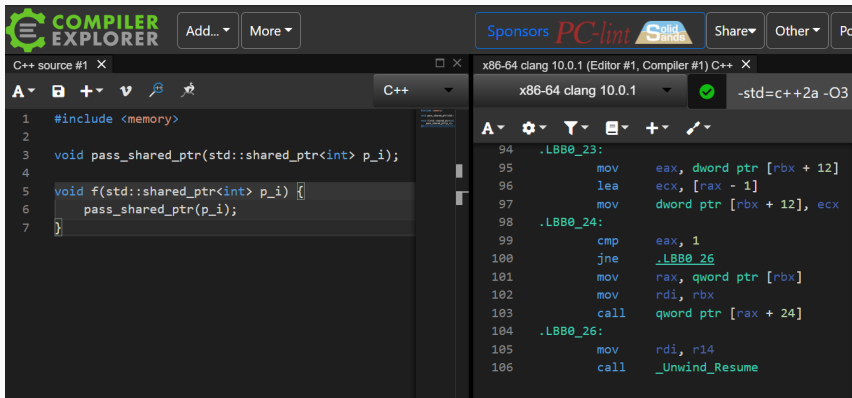
Copy operation needs to:

- Take a lock

- Check if is not `nullptr`

- Increase reference count

# Cost of passing std::shared\_ptr by value



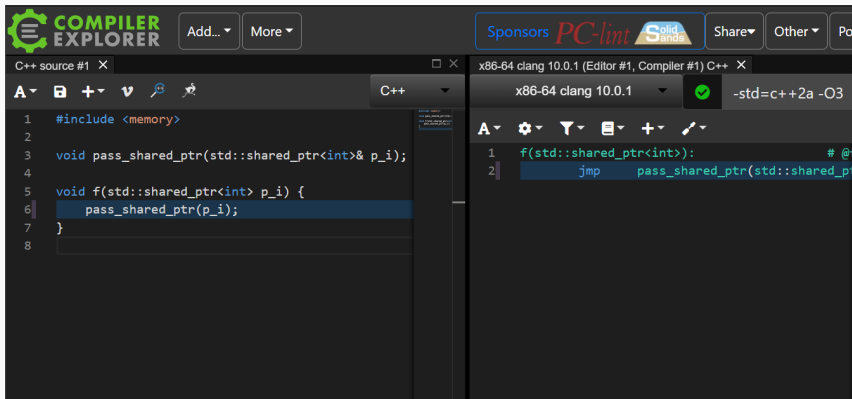
The screenshot shows the Compiler Explorer interface. The left pane displays C++ source code for a function `f` that takes a `std::shared_ptr<int>` by value and calls `pass_shared_ptr`. The right pane shows the generated assembly for x86-64 using clang 10.0.1 with optimization level -O3. The assembly includes instructions for moving pointers, calculating addresses, and calling the function.

```
C++ source #1 X
1  #include <memory>
2
3  void pass_shared_ptr(std::shared_ptr<int> p_i);
4
5  void f(std::shared_ptr<int> p_i) {
6      pass_shared_ptr(p_i);
7  }
```

```
x86-64 clang 10.0.1 (Editor #1, Compiler #1) C++ X
x86-64 clang 10.0.1 -std=c++2a -O3
94  .LBB0_23:
95      mov     eax, dword ptr [rbx + 12]
96      lea     ecx, [rax - 1]
97      mov     dword ptr [rbx + 12], ecx
98  .LBB0_24:
99      cmp     eax, 1
100     jne     .LBB0_26
101     mov     rax, qword ptr [rbx]
102     mov     rdi, rbx
103     call    qword ptr [rax + 24]
104  .LBB0_26:
105     mov     rdi, r14
106     call    _Unwind_Resume
```

<https://godbolt.org/z/KGM6vE>

# Cost of passing std::shared\_ptr by value



The screenshot shows the Compiler Explorer interface. The left pane displays the C++ source code:

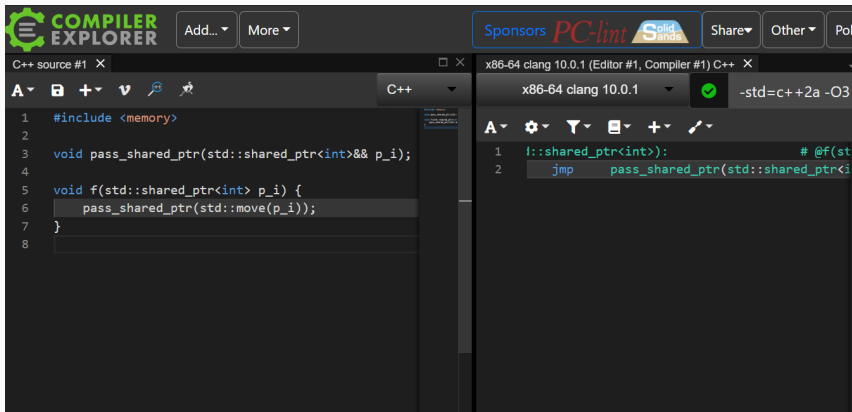
```
1 #include <memory>
2
3 void pass_shared_ptr(std::shared_ptr<int>& p_i);
4
5 void f(std::shared_ptr<int> p_i) {
6     pass_shared_ptr(p_i);
7 }
8
```

The right pane shows the assembly output for x86-64 clang 10.0.1 with flags -std=c++2a -O3. The assembly for the function f is shown:

```
1 f(std::shared_ptr<int>):
2     jmp     pass_shared_ptr(std::shared_ptr<int>)
```

<https://godbolt.org/z/nToKEv>

# Cost of passing std::shared\_ptr by value



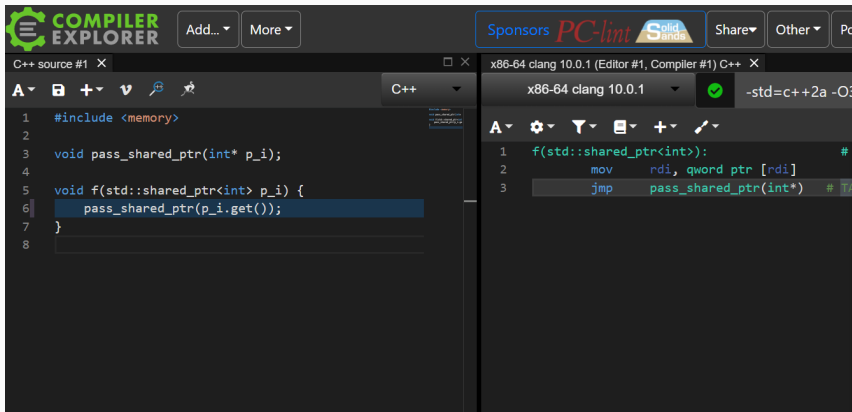
The screenshot shows the Compiler Explorer interface. The left pane displays C++ source code for a function `pass_shared_ptr` and a test function `f`. The right pane shows the generated assembly for x86-64 using clang 10.0.1. The assembly for `pass_shared_ptr` is a single `jmp` instruction, indicating a very low cost for passing the pointer by value.

```
1 #include <memory>
2
3 void pass_shared_ptr(std::shared_ptr<int>&& p_i);
4
5 void f(std::shared_ptr<int> p_i) {
6     pass_shared_ptr(std::move(p_i));
7 }
8
```

```
1 l::shared_ptr<int>:                                # @f(st
2     jmp     pass_shared_ptr(std::shared_ptr<i
```

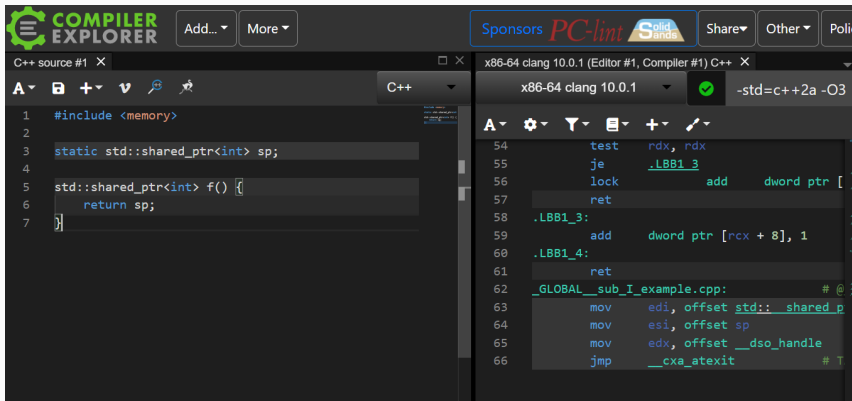
<https://godbolt.org/z/vxbYjz>

# Cost of passing std::shared\_ptr by value



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

# Cost of returning `std::shared_ptr` by value



The screenshot shows the Compiler Explorer interface. The left pane displays the C++ source code for a function `f()` that returns a `std::shared_ptr<int>` by value. The right pane shows the generated assembly code for x86-64 using clang 10.0.1 with the flags `-std=c++2a -O3`. The assembly code includes instructions for testing registers, jumping to a label, locking and adding to a pointer, and returning. It also shows global symbols for the example and the C++ standard library.

```
#include <memory>

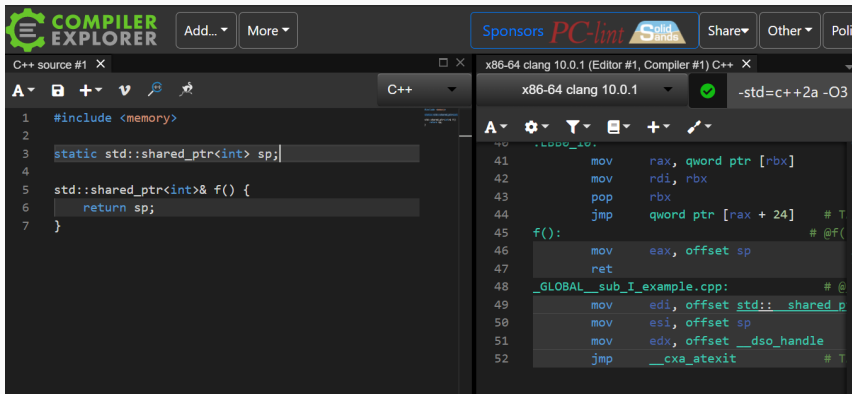
static std::shared_ptr<int> sp;

std::shared_ptr<int> f() {
    return sp;
}
```

```
54      test    rdx, rdx
55      je      .LBB1_3
56      lock    add    dword ptr [
57      ret
58  .LBB1_3:
59      add     dword ptr [rcx + 8], 1
60  .LBB1_4:
61      ret
62  _GLOBAL__sub_I_example.cpp:          # @
63      mov     edi, offset std::shared_p
64      mov     esi, offset sp
65      mov     edx, offset __dso_handle
66      jmp     __cxa_atexit             # T
```

<https://godbolt.org/z/Thv5bv>

# Cost of returning `std::shared_ptr` by value



The screenshot shows the Compiler Explorer interface. The left pane displays the C++ source code, and the right pane shows the generated assembly for x86-64 clang 10.0.1.

**C++ source code:**

```
1 #include <memory>
2
3 static std::shared_ptr<int> sp;
4
5 std::shared_ptr<int>& f() {
6     return sp;
7 }
```

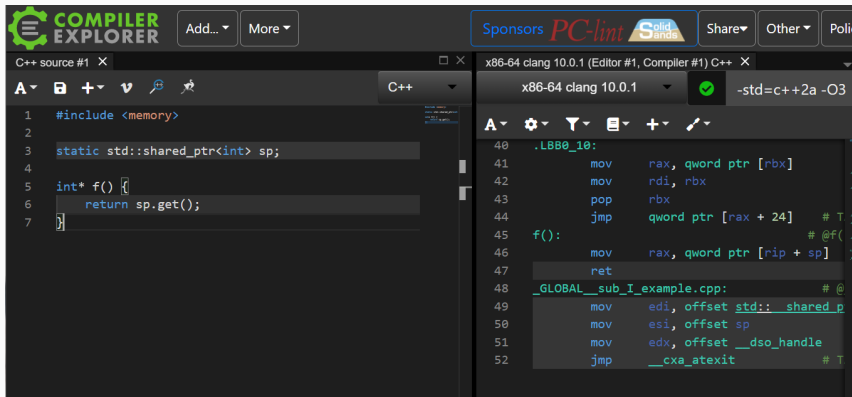
**Assembly output (x86-64 clang 10.0.1):**

```
41     mov     rax, qword ptr [rbx]
42     mov     rdi, rbx
43     pop     rbx
44     jmp     qword ptr [rax + 24]    # T
45 f():                                # @(
46     mov     eax, offset sp
47     ret
48 _GLOBAL__sub_I_example.cpp:        # @
49     mov     edi, offset std::shared_p
50     mov     esi, offset sp
51     mov     edx, offset __dso_handle
52     jmp     __cxa_atexit           # T
```

<https://godbolt.org/z/hab5Ef>



# Cost of returning `std::shared_ptr` by value



The screenshot shows the Compiler Explorer interface. The left pane displays the C++ source code, and the right pane shows the generated assembly for x86-64 clang 10.0.1 with flags `-std=c++2a -O3`.

```
1  #include <memory>
2
3  static std::shared_ptr<int> sp;
4
5  int* f() {
6      return sp.get();
7  }
```

The assembly output on the right shows the function `f()` returning a pointer to the `sp` variable, which is a static global. This illustrates that returning a `std::shared_ptr` by value is cheap because it only returns a pointer to a shared object.

```
40  .LBB0_10:
41      mov     rax, qword ptr [rbx]
42      mov     rdi, rbx
43      pop     rbx
44      jmp     qword ptr [rax + 24]    # T
45  f():
46      mov     rax, qword ptr [rip + sp]
47      ret
48  _GLOBAL__sub_I_example.cpp:
49      mov     edi, offset std::_shared_p
50      mov     esi, offset sp
51      mov     edx, offset __dso_handle
52      jmp     __cxa_atexit            # T
```

<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::weak_ptr` instead

- Compiler optimize things, so profile first

Hot path smell:

**O**verused `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

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 `constexpr`

# Helping compiler with optimizations

const everything possible

or better constexpr

or better constexpr

get rid of unnecessary private member functions

# Helping compiler with optimizations

const everything possible

or better constexpr

or better constexpr

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 constexpr

get rid of unnecessary private member functions

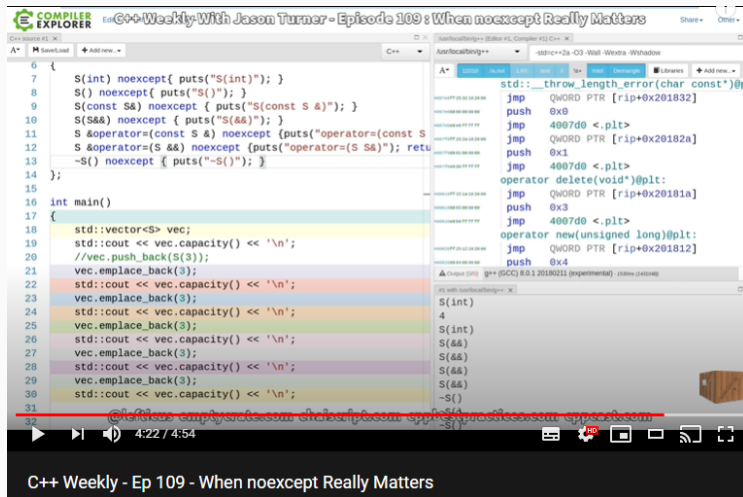
make translation unit local functions static, or put in anonymous namespace

don't use `std::move` unless absolutely necessary

# Don't miss noexcept



# Don't miss noexcept



The screenshot shows a video player interface. The main content area displays a C++ code editor window titled "C++ Weekly With Jason Turner - Episode 109: When noexcept Really Matters". The code defines a struct `S` with several `noexcept` functions: `S(int)`, `S()`, `S(const S&)`, `S(S&&)`, `S&operator=(const S&)`, `S&operator=(S&&)`, and `~S()`. The `main` function creates a `std::vector<S>` and performs a series of capacity checks and `emplace_back` operations, printing the capacity after each. The video player controls at the bottom show the video is at 4:22 / 4:54. The title bar of the video player reads "C++ Weekly - Ep 109 - When noexcept Really Matters".

```
6 {
7     S(int) noexcept { puts("S(int)"); }
8     S() noexcept { puts("S()"); }
9     S(const S&) noexcept { puts("S(const S &)"); }
10    S(S&&) noexcept { puts("S(&&)"); }
11    S&operator=(const S&) noexcept { puts("operator=(const S &)"); }
12    S&operator=(S&&) noexcept { puts("operator=(S &&)"); return *this; }
13    ~S() noexcept { puts("~S()"); }
14 };
15
16 int main()
17 {
18     std::vector<S> vec;
19     std::cout << vec.capacity() << '\n';
20     //vec.push_back(S(3));
21     vec.emplace_back(3);
22     std::cout << vec.capacity() << '\n';
23     vec.emplace_back(3);
24     std::cout << vec.capacity() << '\n';
25     vec.emplace_back(3);
26     std::cout << vec.capacity() << '\n';
27     vec.emplace_back(3);
28     std::cout << vec.capacity() << '\n';
29     vec.emplace_back(3);
30     std::cout << vec.capacity() << '\n';
31 }
32
```

Assembly output (x86\_64):

```
std::_throw_length_error(char const*)@plt:
    jmp     QWORD PTR [rip+0x201832]
    push    0x0
    jmp     4007d0 <.plt>
    jmp     QWORD PTR [rip+0x20182a]
    push    0x1
    jmp     4007d0 <.plt>
operator delete(void*)@plt:
    jmp     QWORD PTR [rip+0x20181a]
    push    0x3
    jmp     4007d0 <.plt>
operator new(unsigned long)*@plt:
    jmp     QWORD PTR [rip+0x201812]
    push    0x4
```

# Don't miss noexcept

## noexcept considered harmful ??? - Niels Dekker

### Preliminary benchmark results

Compares `noexcept` to no `noexcept` (implicitly defined exception specification)

	VS2017 x86 (32-bit)	VS2017 x64 (64-bit)	GCC 5.4.0 (Ubuntu 16.04)	Clang (Apple LLVM 9.1)
inline function	-	-		-
exported library function	-			
removable catch			+	
reorderable <code>++v; f(); --v;</code>	+	+		
skippable stack unwinding		+	+	
<code>std::vector&lt;my_class&gt;, move</code>	+	+	+	+

+ `noexcept`  
performs best  
(in this specific  
test case)

- Implicit  
specification  
performs best  
(in this specific  
test case)

Blank cell: no  
significant  
difference found



So now: should we declare our functions `noexcept` or not?

Please check [github.com/N-Dekker/noexcept\\_benchmark](https://github.com/N-Dekker/noexcept_benchmark)  
Mail: [N.Dekker@lumc.nl](mailto:N.Dekker@lumc.nl) and [NielsDekker@xs4all.nl](mailto:NielsDekker@xs4all.nl)

*std::terminate()*



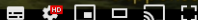
2019

[epponsea.uk](http://epponsea.uk)

@epponsea



5:04 / 5:51



FOLKESTONE

Lightning Talk: `noexcept` considered harmful ??? - Niels Dekker [C++ on Sea 2019]

Hot path smell:

**F**orgetting 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 😊