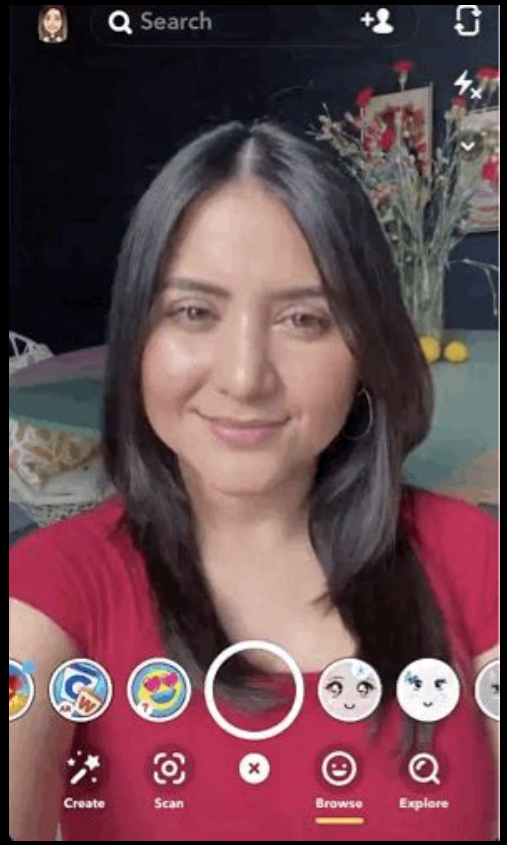
Hidden Overhead of a Function API

OLEKSANDR BACHERIKOV

Cppcon
The C++ Conference

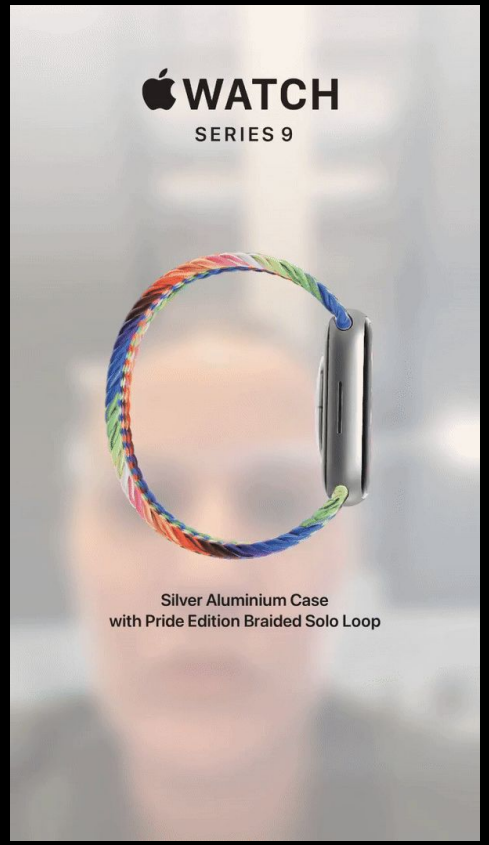
2024
September 15 - 20

What we do at Snap with C++



Neural style transfer
Face tracking

Full body tracking
Cloth simulation



Ray tracing
Wrist tracking

Thank you, Serhii Huralnik and Eduardo Madrid!!

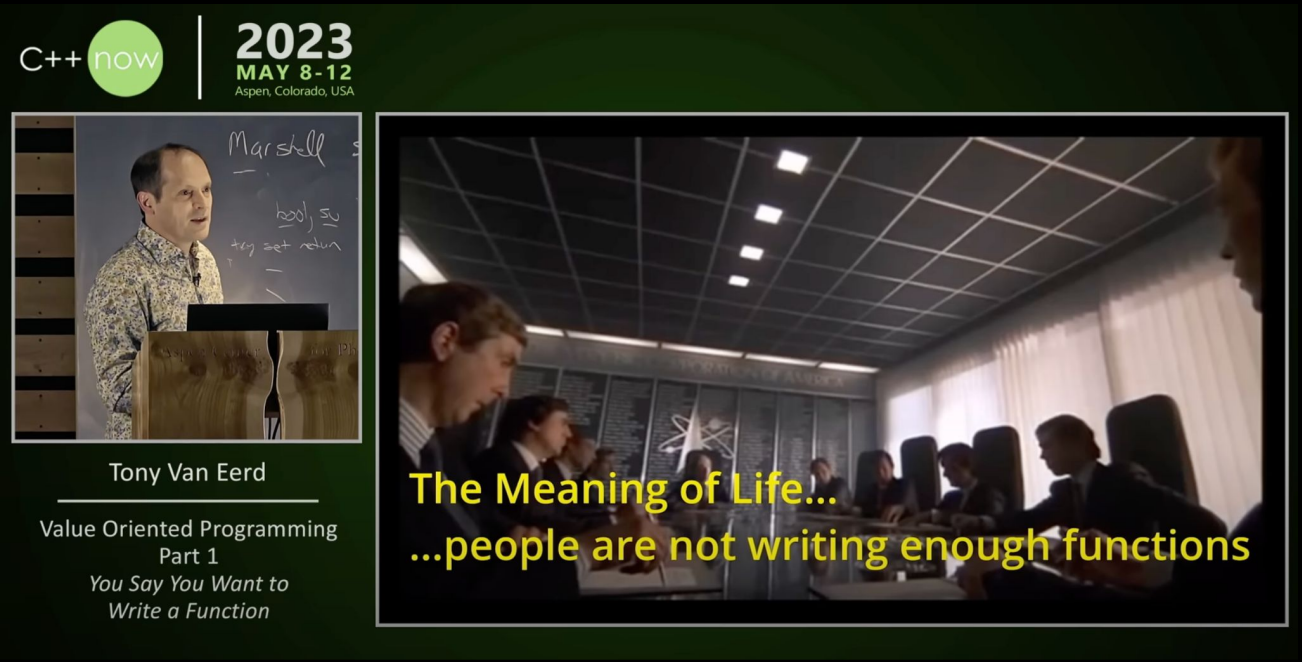
Section 0. Introduction

Section 1. Return value

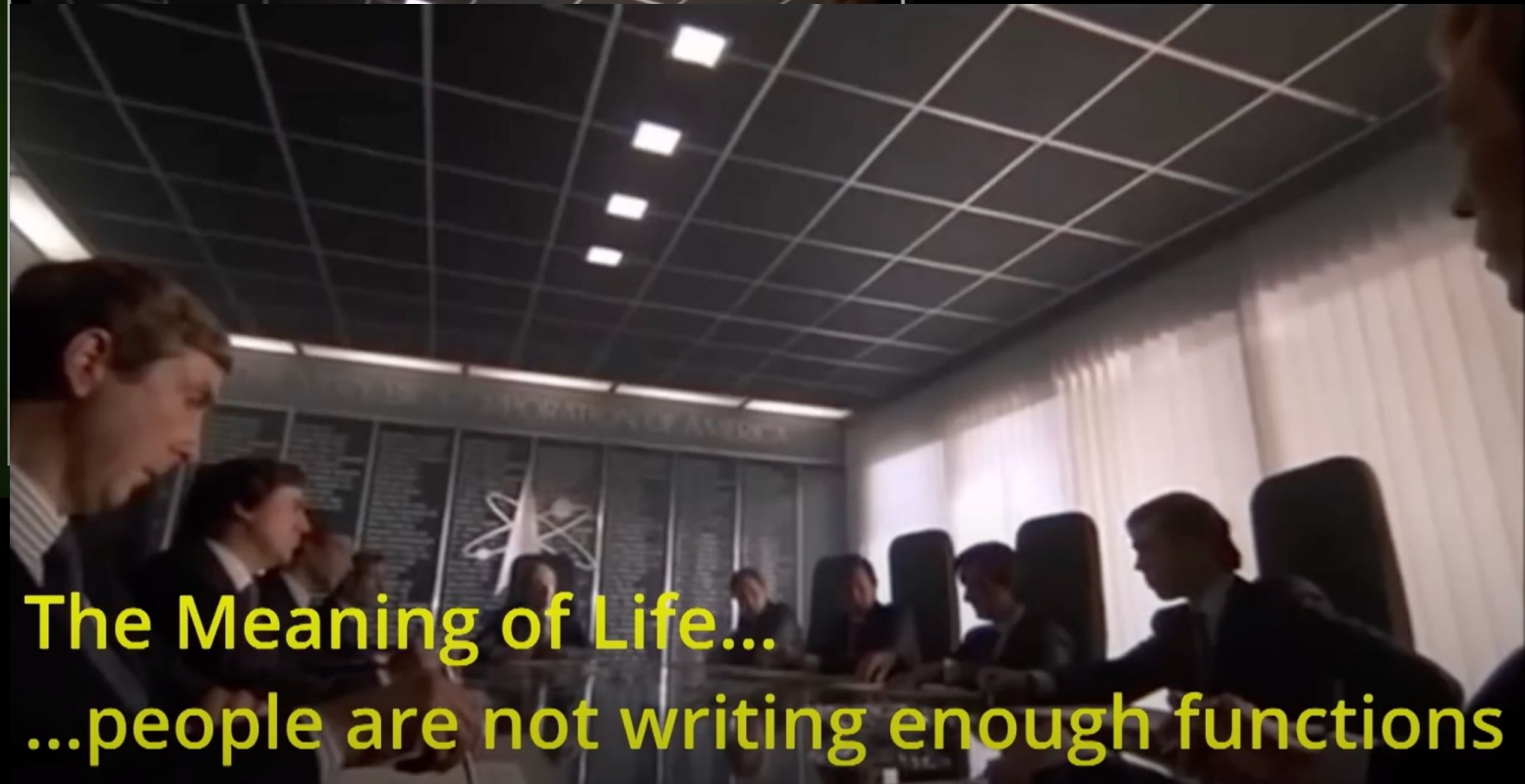
Section 2. Parameter passing

Section 3. Multiple parameters

Tony Van Eerd: “people are not writing enough functions”

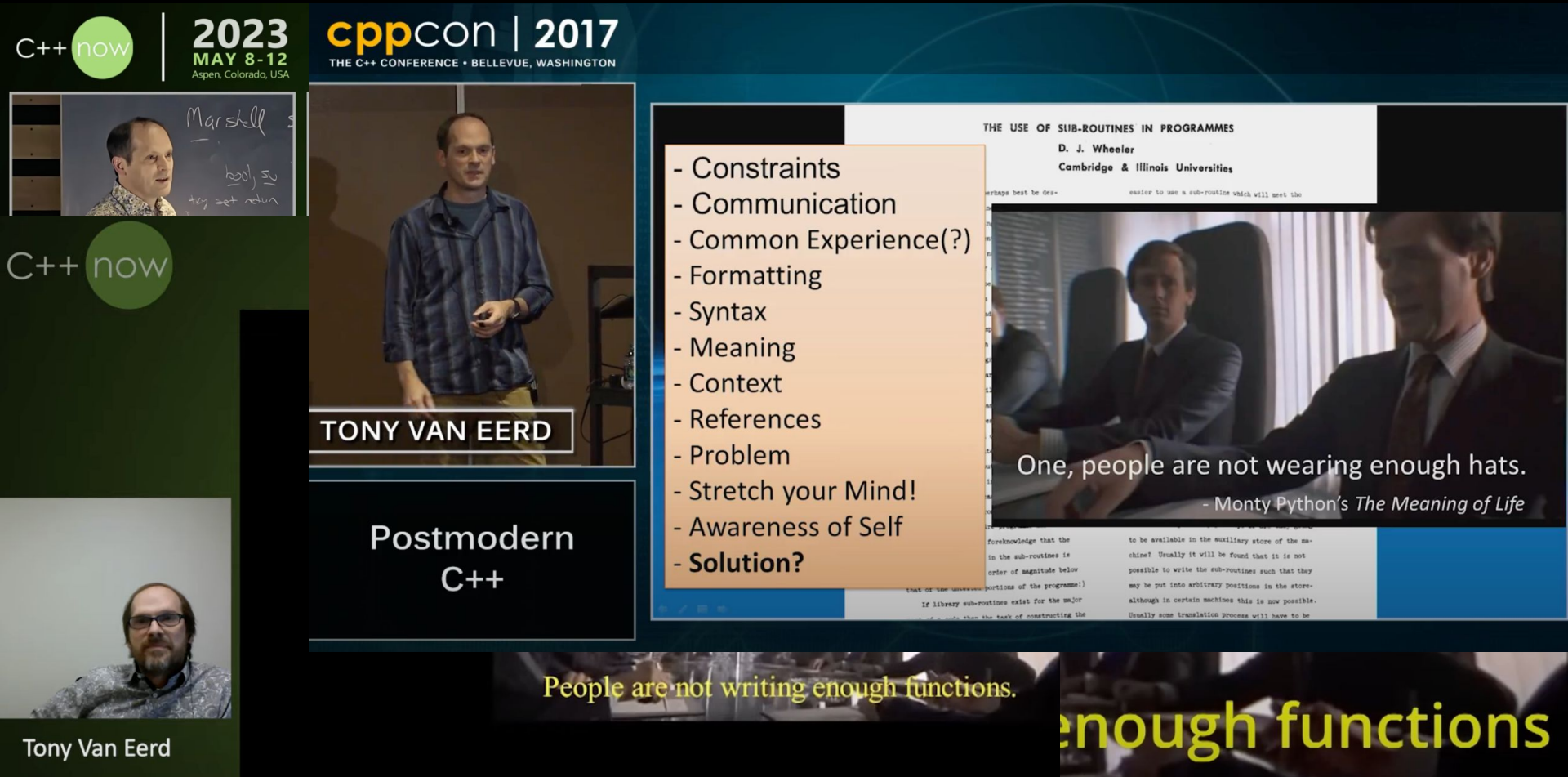


Tony Van Eerd: “people are not writing enough functions”



Tony Van Eerd: “people are not writing enough functions”

Tony Van Eerd: “people are not writing enough functions”



When people finally start writing more functions, we’d prefer to get only the well designed ones!

When talking about performance, we typically think about the function logic. We'll see that a well designed function API can have an even larger impact.

How will we compare performance?

- Benchmarks at this low level are not too reliable,

  and also don’t represent performance in large projects well.

How will we compare performance?

- Benchmarks at this low level are not too reliable,

  and also don’t represent performance in large projects well.
- Dynamic instruction count is more reliable on modern CPUs.

How will we compare performance?

- Benchmarks at this low level are not too reliable,

  and also don’t represent performance in large projects well.
- Dynamic instruction count is more reliable on modern CPUs.
- We’ll use simple examples, so that we can just compare

  the number of instructions generated by a compiler.

Accelerate large-scale applications with BOLT (link)

"… machine code … can range from 10s to 100s of megabytes in size, which is often too large to fit in any modern CPU instruction cache. As a result, the hardware spends a considerable amount of processing time — nearly 30 percent, in many cases — getting an instruction stream from memory to the CPU."

Disclaimer:

Our discussion is relevant

only for non-inlined functions

ISO C++ wiki: Do inline functions improve performance?

Yes and no. Sometimes. Maybe.

There are no simple answers. `inline` functions might make the code faster, they might make it slower. They might make the executable larger, they might make it smaller. They might cause thrashing, they might prevent thrashing. And they might be, and often are, totally irrelevant to speed.

ISO C++ wiki: Do inline functions improve performance?

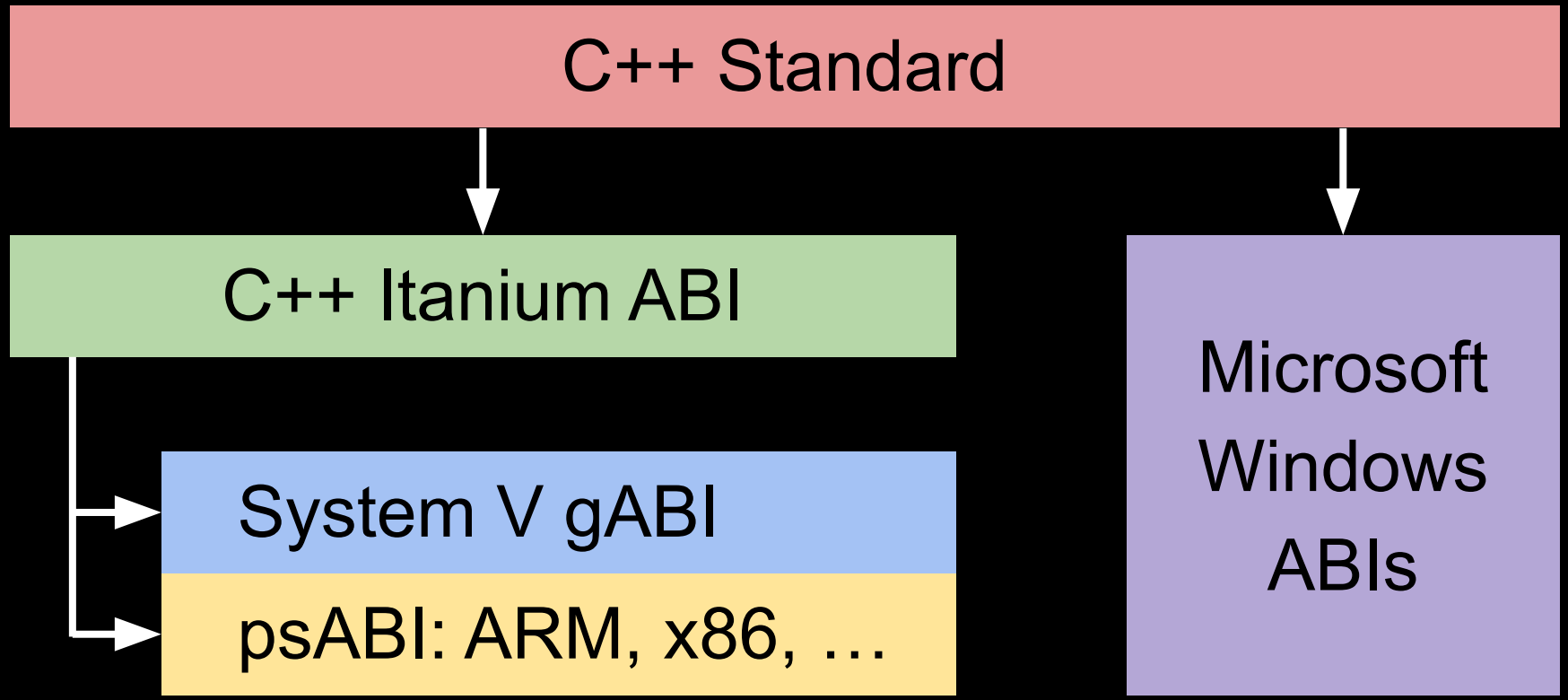
Yes and no. Sometimes. Maybe.

There are no simple answers. `inline` functions might make the code faster, they might make it slower. They might make the executable larger, they might make it smaller. They might cause thrashing, they might prevent thrashing. And they might be, and often are, totally irrelevant to speed.

Credit to Khalil Estell:

Firefox function distribution	
157946	functions above (127B)
167404	functions below (127B)

Understanding how machine code is generated from C++



armv8-a System V ABI - iPhone - M1 Mac and newer - Android smartphone	x86-64 (AMD64) System V ABI - Linux server - old Mac	x86-64 (AMD64) Microsoft ABI - Windows device
armv7-a System V ABI - ancient iPhone - low-end Android smartphone	x86 (IA-32) System V ABI - ancient Linux server	x86 (IA-32) Microsoft ABI - ancient Windows device

armv8-a System V ABI Procedure Call Standard for the Arm® 64-bit Architecture armv8-a clang 18.1.0 `-O2 -std=c++20`	x86-64 (AMD64) System V ABI AMD64 Architecture Processor Supplement x86-64 gcc 14.2 `-O2 -std=c++20`	x86-64 (AMD64) Microsoft ABI x64 calling convention x64 msvc v19.40 VS17.10 `-O2 /std:c++20`
armv7-a System V ABI Procedure Call Standard for the Arm® Architecture armv7-a clang 11.0.1 `-O2 -std=c++20`	x86 (IA-32) System V ABI Intel386 Architecture Processor Supplement x86-64 gcc 14.2 `-O2 -std=c++20 -m32`	x86 (IA-32) Microsoft ABI calling conventions x86 msvc v19.40 VS17.10 `-O2 /std:c++20`

Things are complicated

We'll be looking for simple guidelines to navigate this complexity.

C++ Core Guidelines seem like a good candidate.

Section 0. Introduction

Section 1. Return value

Section 2. Parameter passing

Section 3. Multiple parameters

C++ Core Guidelines

F.20: For “out” output values, prefer return values to output parameters

Reason A return value is self-documenting, whereas a `&` could be either in-out or out-only and is liable to be misused.

Returning std::unique_ptr

```
#include <memory>

std::unique_ptr<int> value_ptr() {
    return nullptr;
}

void output_ptr(std::unique_ptr<int>& dst) {
    dst = nullptr;
}
```

- return by value

- output parameter

https://godbolt.org/z/ea9M3G94s

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	

`str xzr, [x8]`
`ret`
	`mov QWORD PTR [rdi], 0`
	`mov rax, rdi`
	`ret`
		`sub rsp, 24`
		`xor eax, eax`
		`mov QWORD PTR [rcx], rax`
		`mov rax, rcx`
		`add rsp, 24`
		`ret 0`
			VALUE

`mov x8, x0`
`ldr x0, [x0]`
`str xzr, [x8]`
`cbz x0, .LBB1_2`
`b operator delete(void*)`
`.LBB1_2:`
`ret`
	`mov rax, QWORD PTR [rdi]`
	`mov QWORD PTR [rdi], 0`
	`test rax, rax`
	`je .L3`
	`mov esi, 4`
	`mov rdi, rax`
	`jmp operator delete(void*)`
	`.L3:`
	`ret`
		`mov rax, QWORD PTR [rcx]`
		`mov QWORD PTR [rcx], 0`
		`test rax, rax`
		`je SHORT $LN34@output_ptr`
		`mov edx, 4`
		`mov rcx, rax`
		`jmp operator delete(void*)`
		`$LN34@output_ptr:`
		`ret 0`
			OUTPUT

Returning std::unique_ptr

```
#include <memory>

std::unique_ptr<int> value_ptr() {
    return nullptr;
}

void output_ptr(std::unique_ptr<int>& dst) {
    dst = nullptr;
}
```

- return by value

- output parameter

This might be non-empty

Returning `std::unique_ptr` : call site

```
#include <memory>

std::unique_ptr<int> value_ptr();
void output_ptr(std::unique_ptr<int>& dst);

int value_ptr_call() {
    auto ptr = value_ptr();
    return *ptr;
}

int output_ptr_call() {
    std::unique_ptr<int> ptr;
    output_ptr(ptr);
    return *ptr;
}
```

- definitions removed to avoid inlining

- return by value

- output parameter

https://godbolt.org/z/G9aPehqM1

armv8-a clang 18.1.0

```
stp   x29, x30, [sp, #-32]!
str   x19, [sp, #16]
mov   x29, sp
add   x8, x29, #24
bl    value_ptr()
ldr   x0, [x29, #24]
ldr   w19, [x0]
bl    operator delete(void*)
mov   w0, w19
ldr   x19, [sp, #16]
ldp   x29, x30, [sp], #32
ret
```

x86-64 gcc 14.2

```
push  rbx
sub   rsp, 16
lea   rdi, [rsp+8]
call  value_ptr()
mov   rdi, QWORD PTR [rsp+8]
mov   esi, 4
mov   ebx, DWORD PTR [rdi]
call  operator delete(void*)
add   rsp, 16
mov   eax, ebx
pop   rbx
ret
```

x64 msvc v19.40 VS17.10

```
push  rbx
sub   rsp, 32
lea   rcx, QWORD PTR ptr$[rsp]
call  value_ptr()
mov   rcx, QWORD PTR ptr$[rsp]
mov   edx, 4
mov   ebx, DWORD PTR [rcx]
call  operator delete(void*)
mov   eax, ebx
add   rsp, 32
pop   rbx
ret   0
```

VALUE

```
stp   x29, x30, [sp, #-32]!
str   x19, [sp, #16]
mov   x29, sp
str   xzr, [x29, #24]
add   x0, x29, #24
bl    output_ptr(std::unique_ptr&)
ldr   x0, [x29, #24]
ldr   w19, [x0]
bl    operator delete(void*)
mov   w0, w19
ldr   x19, [sp, #16]
ldp   x29, x30, [sp], #32
ret
ldr   x8, [x29, #24]
mov   x19, x0
cbz   x8, .LBB1_4
mov   x0, x8
bl    operator delete(void*)
.LBB1_4:
mov   x0, x19
bl    _Unwind_Resume

DW.ref.__gxx_personality_v0:
.xword __gxx_personality_v0
```

```
push  rbx
sub   rsp, 16
mov   QWORD PTR [rsp+8], 0
lea   rdi, [rsp+8]
call  output_ptr(std::unique_ptr&)
mov   rdi, QWORD PTR [rsp+8]
mov   esi, 4
mov   ebx, DWORD PTR [rdi]
call  operator delete(void*)
add   rsp, 16
mov   eax, ebx
pop   rbx
ret
mov   rbx, rax
jmp   .L5
```

```
$stateUnwindMap$int output_ptr_call()
DB    02H
DB    0aH
DD    imagerel
      std::unique_ptr::~unique_ptr()
DB    060H

push  rbx
sub   rsp, 32
mov   QWORD PTR ptr$[rsp], 0
lea   rcx, QWORD PTR ptr$[rsp]
call  output_ptr(std::unique_ptr&)
mov   rcx, QWORD PTR ptr$[rsp]
mov   ebx, DWORD PTR [rcx]
mov   edx, 4
call  operator delete(void*)
mov   eax, ebx
add   rsp, 32
pop   rbx
ret   0
```

OUTPUT

armv8-a clang 18.1.0

```
stp   x29, x30, [sp, #-32]!
str   x19, [sp, #16]
mov   x29, sp
add   x8, x29, #24
bl    value_ptr()
ldr   x0, [x29, #24]
ldr   w19, [x0]
bl    operator delete(void*)
mov   w0, w19
ldr   x19, [sp, #16]
ldp   x29, x30, [sp], #32
ret
```

```
stp   x29, x30, [sp, #-32]!
str   x19, [sp, #16]
mov   x29, sp
str   xzr, [x29, #24]
add   x0, x29, #24
bl    output_ptr(std::unique_ptr&)
ldr   x0, [x29, #24]
ldr   w19, [x0]
bl    operator delete(void*)
mov   w0, w19
ldr   x19, [sp, #16]
ldp   x29, x30, [sp], #32
ret
ldr   x8, [x29, #24]
mov   x19, x0
cbz   x8, .LBB1_4
mov   x0, x8
bl    operator delete(void*)
.LBB1_4:
mov   x0, x19
bl    _Unwind_Resume

DW.ref.__gxx_personality_v0:
.xword __gxx_personality_v0
```

x86-64 gcc 14.2

```
push  rbx
sub   rsp, 16
lea   rdi, [rsp+8]
call  value_ptr()
mov   rdi, QWORD PTR [rsp+8]
mov   esi, 4
mov   ebx, DWORD PTR [rdi]
call  operator delete(void*)
add   rsp, 16
mov   eax, ebx
pop   rbx
ret
```

```
push  rbx
sub   rsp, 16
mov   QWORD PTR [rsp+8], 0
lea   rdi, [rsp+8]
call  output_ptr(std::unique_ptr&)
mov   rdi, QWORD PTR [rsp+8]
mov   esi, 4
mov   ebx, DWORD PTR [rdi]
call  operator delete(void*)
add   rsp, 16
mov   eax, ebx
pop   rbx
ret
mov   rbx, rax
jmp   .L5
```

x64 msvc v19.40 VS17.10

```
push  rbx
sub   rsp, 32
lea   rcx, QWORD PTR ptr$[rsp]
call  value_ptr()
mov   rcx, QWORD PTR ptr$[rsp]
mov   edx, 4
mov   ebx, DWORD PTR [rcx]
call  operator delete(void*)
mov   eax, ebx
add   rsp, 32
pop   rbx
ret   0
```

```
$stateUnwindMap$int output_ptr_call()
DB    02H
DB    0aH
DD    imagerel
      std::unique_ptr::~unique_ptr()
DB    060H
```

```
push  rbx
sub   rsp, 32
mov   QWORD PTR ptr$[rsp], 0
lea   rcx, QWORD PTR ptr$[rsp]
call  output_ptr(std::unique_ptr&)
mov   rcx, QWORD PTR ptr$[rsp]
mov   ebx, DWORD PTR [rcx]
mov   edx, 4
call  operator delete(void*)
mov   eax, ebx
add   rsp, 32
pop   rbx
ret   0
```

Stack unwinding on exception

VALUE

OUTPUT

Returning `std::unique_ptr` : call site

```
#include <memory>

std::unique_ptr<int> value_ptr();
void output_ptr(std::unique_ptr<int>& dst);

int value_ptr_call() {
    auto ptr = value_ptr();
    return *ptr;
}

int output_ptr_call() {
    std::unique_ptr<int> ptr;
    output_ptr(ptr);
    return *ptr;
}
```

- return by value

Has to be destroyed if `output_ptr` throws

- output parameter

value_ptr_call	output_ptr_call
`push rbx`	`push rbx`
`sub rsp, 16`	`sub rsp, 16`
	`mov QWORD PTR [rsp+8], 0`
`lea rdi, [rsp+8]`	`lea rdi, [rsp+8]`
`call value_ptr()`	`call output_ptr(std:unique_ptr&)`
`mov rdi, QWORD PTR [rsp+8]`	`mov rdi, QWORD PTR [rsp+8]`
`mov esi, 4`	`mov esi, 4`
`mov ebx, DWORD PTR [rdi]`	`mov ebx, DWORD PTR [rdi]`
`call operator delete(void*)`	`call operator delete(void*)`
`add rsp, 16`	`add rsp, 16`
`mov eax, ebx`	`mov eax, ebx`
`pop rbx`	`pop rbx`
`ret`	`ret`

puts 0 into memory

Returning std::unique_ptr : call site

```
#include <memory>

std::unique_ptr<int> value_ptr();
void output_ptr(std::unique_ptr<int>& dst);

int value_ptr_call() {
    auto ptr = value_ptr();
    return *ptr;
}

int output_ptr_call() {
    std::unique_ptr<int> ptr;
    output_ptr(ptr);
    return *ptr;
}
```

- return by value

Default constructed here

- output parameter

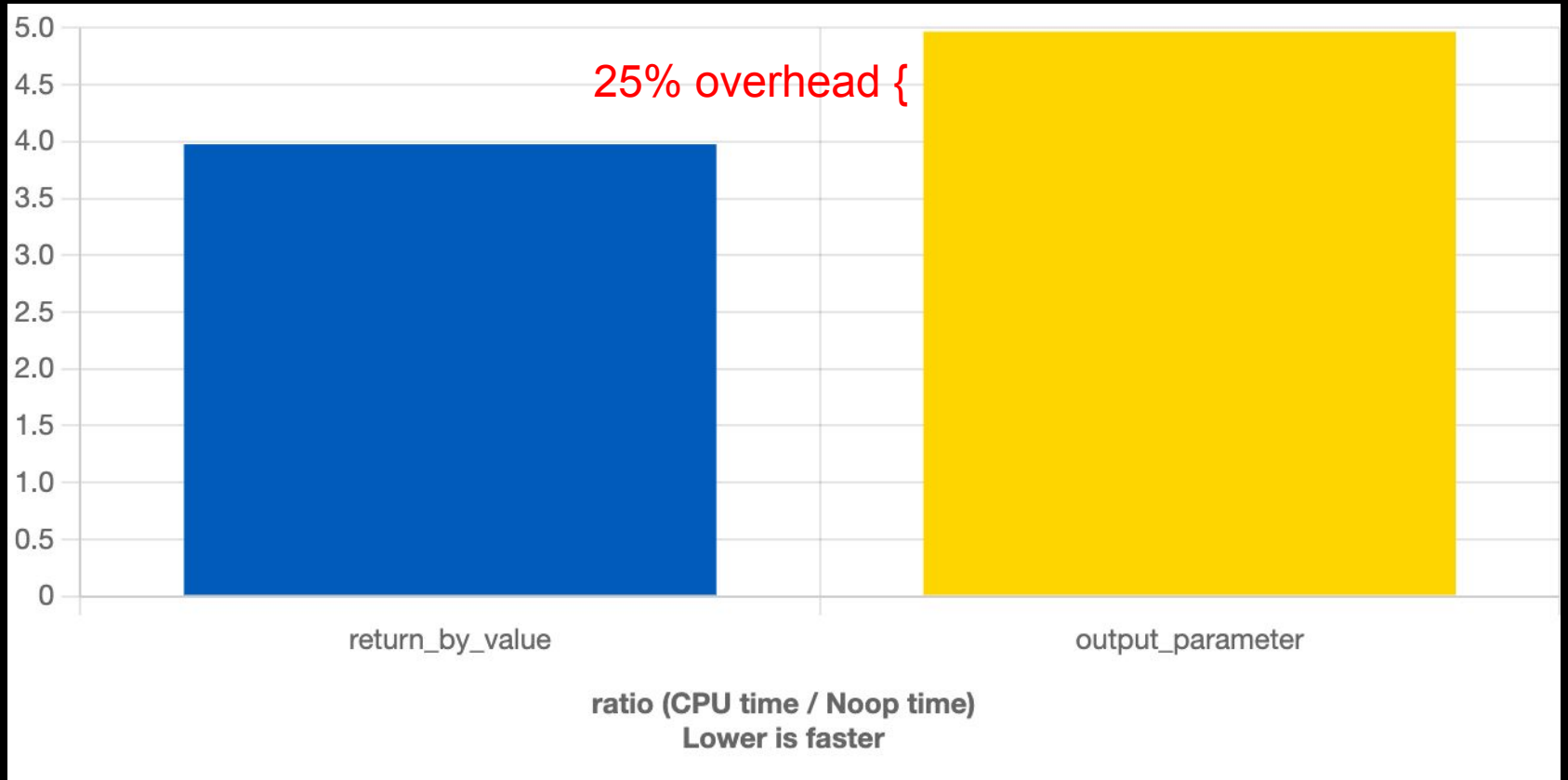
C++ Core Guidelines

ES.20: Always initialize an object

Reason Avoid used-before-set errors and their associated undefined behavior. Avoid problems with comprehension of complex initialization. Simplify refactoring.

Quick benchmark

https://quick-bench.com/q/mOAHh7zZeagJlCJ63GtWVMPngsw



void foo(T& out) - How to fix output parameters

26 Oct 2016 by Jonathan

Share this post:

```
deferred_construction<std::string> output;

read_strings(in, out(output));
```

https://www.foonathan.net/2016/10/output-parameter/

Does it solve our problems?

Pros:

- Default constructor before the function call is avoided

Does it solve our problems?

Pros:

- Default constructor before the function call is avoided

Cons (unless we fully trust the user and don’t have exceptions):

- Stack unwind on exception is still necessary
- Extra bool flag is required to know if the object
  - was actually initialized
  - not initialized more than once

Does it solve our problems?

Pros:

- Default constructor before the function call is avoided

Cons (unless we fully trust the user and don’t have exceptions):

- Stack unwind on exception is still necessary
- Extra bool flag is required to know if the object
  - was actually initialized
  - not initialized more than once

Any C++ compiler checks that every execution path in a function ends with a return statement. We just need to return by value.

Hopefully, you're convinced that output parameter is a bad idea.

Now, let's see how return by value works, specifically for C++ abstractions.

C++ Core Guidelines

F.26: Use a `unique_ptr<T>` to transfer ownership where a pointer is needed

Reason Using `unique_ptr` is the cheapest way to pass a pointer safely.

Returning a pointer

```
#include <memory>

int* raw_ptr() {
    return nullptr;
}

std::unique_ptr<int> smart_ptr() {
    return nullptr;
}
```

- returning raw pointer

- returning smart pointer

https://godbolt.org/z/ExaoKfT1z

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	
`mov x0, xzr` `ret`	`xor eax, eax` `ret`	`xor eax, eax` `ret 0`	RAW

`str xzr, [x8]`
`ret`
	`mov QWORD PTR [rdi], 0`
	`mov rax, rdi`
	`ret`
		`sub rsp, 24`
		`xor eax, eax`
		`mov QWORD PTR [rcx], rax`
		`mov rax, rcx`
		`add rsp, 24`
		`ret 0`
			SMART

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	
`mov x0, xzr` `ret`	`xor eax, eax` `ret`	`xor eax, eax` `ret 0`	RAW

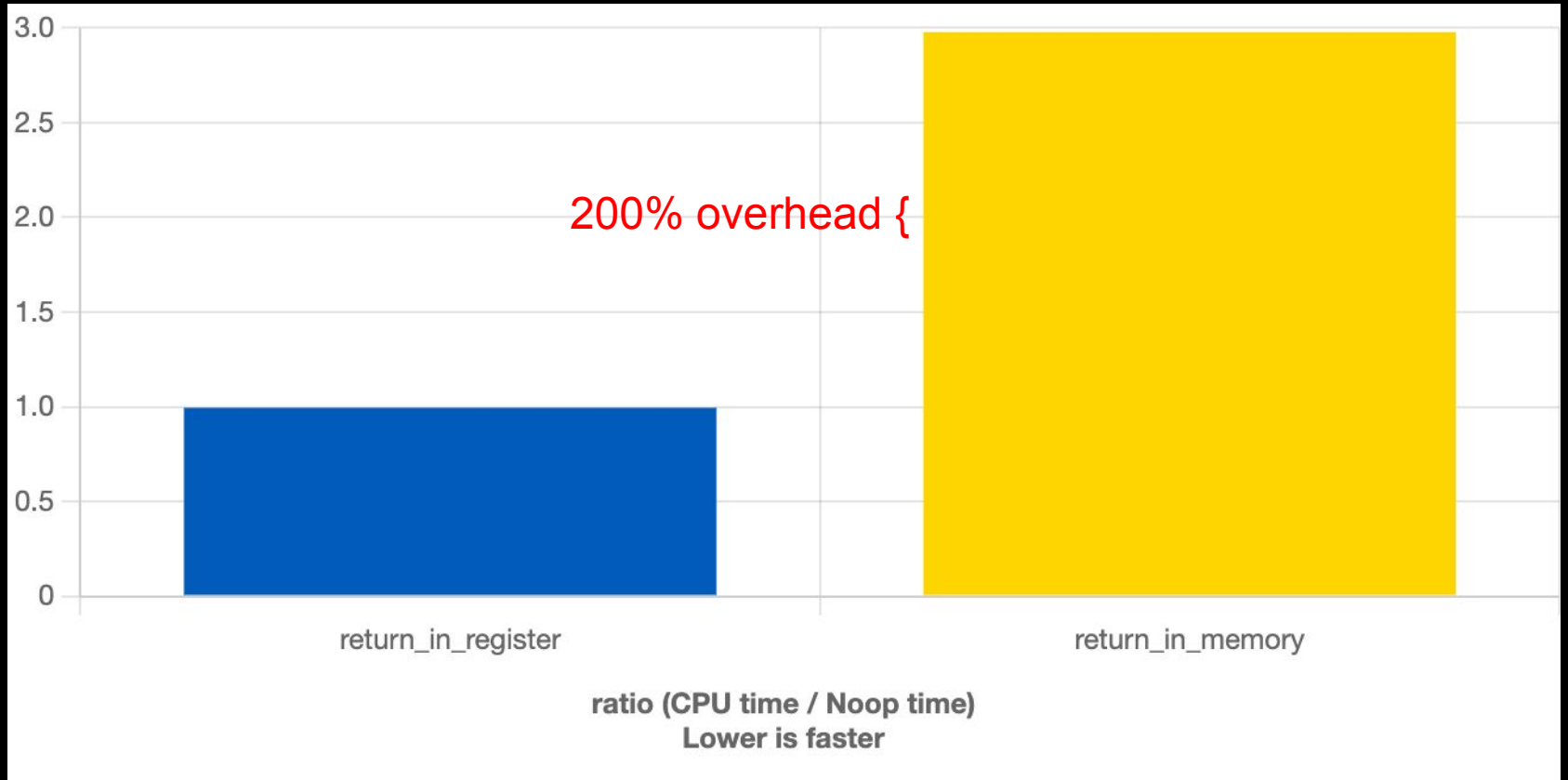
x0, **xzr**, **eax**, **rsp**… are machine registers.
They are the fastest storage available on a machine.

Register in square brackets means it stores an address,
and we're accessing memory at that address: **[x8]**, **QWORD PTR [rdi]**

`str xzr, [x8]`
`ret`
	`mov QWORD PTR [rdi], 0`
	`mov rax, rdi`
	`ret`
		`sub rsp, 24`
		`xor eax, eax`
		`mov QWORD PTR [rcx], rax`
		`mov rax, rcx`
		`add rsp, 24`
		`ret 0`
			SMART

Quick benchmark

https://quick-bench.com/q/pJ3z9L_Q1M16qob8-sq8lM3-T60



C++ Core Guidelines

F.26: Use a `unique_ptr<T>` to transfer ownership where a pointer is needed

Reason Using `unique_ptr` is the cheapest way to pass a pointer safely.

Not free

Wrapper over `int`

```
struct INT {
    int value;

    INT(int value = 0) : value{value} {}
    INT(INT&& src) : value{src.value} {}
    INT& operator=(INT&& src) {
        value = src.value;
        return *this;
    }
    INT(INT const& src) : value{src.value} {}
    INT& operator=(INT const& src) {
        value = src.value;
        return *this;
    }
    ~INT() {}
};
```

Libraries that wrap integers

- Smart pointers similarly wrap raw pointers
- `std::chrono`
- All other units libraries
- Safe integers
- Bindings for other languages

Wrapper over `int`

```
struct INT {
    int value;

    INT(int value = 0) : value{value} {}
    INT(INT&& src) : value{src.value} {}
    INT& operator=(INT&& src) {
        value = src.value;
        return *this;
    }
    INT(INT const& src) : value{src.value} {}
    INT& operator=(INT const& src) {
        value = src.value;
        return *this;
    }
    ~INT() {}
};
```

Wrapper over `int`

```
struct INT {
    int value;

    INT(int value = 0) : value{value} {}
    INT(INT&& src) : value{src.value} {}
    INT& operator=(INT&& src) {
        value = src.value;
        return *this;
    }
    INT(INT const& src) : value{src.value} {}
    INT& operator=(INT const& src) {
        value = src.value;
        return *this;
    }
    ~INT() {}
};
```

```
int int_seconds() {
    return 60;
}

INT INT_seconds() {
    return 60;
}
```

https://godbolt.org/z/Tddd4E6hs

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	
`mov w0, #60` `ret`	`mov eax, 60` `ret`	`mov eax, 60` `ret 0`	int

`mov w9, #60`
`str w9, [x8]`
`ret`
	`mov DWORD PTR [rdi], 60`
	`mov rax, rdi`
	`ret`
		`mov QWORD PTR [rcx], 60`
		`mov rax, rcx`
		`ret 0`
			INT

armv7-a clang 11.0.1	**x86-64 gcc 14.2 (-m32)**	**x86 msvc v19.40 VS17.10**	
`mov r0, #60` `bx lr`	`mov eax, 60` `ret`	`mov eax, 60` `ret 0`	int

`mov r1, #60`
`str r1, [r0]`
`bx lr`
	`mov eax, DWORD PTR [esp+4]`
	`mov DWORD PTR [eax], 60`
	`ret 4`
		`mov eax, DWORD PTR ___$ReturnUdt$[esp-4]`
		`mov DWORD PTR [eax], 60`
		`ret 0`
			INT

The problem

Itanium C++ ABI

3.1.3.1 Non-trivial Return Values

If the return type is a class type that is non-trivial for the purposes of calls, the caller passes an address as an implicit parameter. The callee then constructs the return value into this address.

The problem

Itanium C++ ABI

3.1.3.1 Non-trivial Return Values

If the return type is a class type that is non-trivial for the purposes of calls, the caller passes an address as an implicit parameter. The callee then constructs the return value into this address.

C++ reference: Trivial class

A trivial class is a class that

- is trivially copyable, and
- has one or more eligible default constructors such that each is trivial.

Wrapper over `int` : no custom copy and move

```
struct INT {
    int value;

    INT(int value = 0) : value{value} {}
    // INT(INT&&) = default;
    // INT& operator=(INT&&) = default;
    // INT(INT const&) = default;
    // INT& operator=(INT const&) = default;
    ~INT() {}
};
```

```
int int_seconds() {
    return 60;
}

INT INT_seconds() {
    return 60;
}
```

https://godbolt.org/z/f6s1P96Tx

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	
`mov w0, #60` `ret`	`mov eax, 60` `ret`	`mov eax, 60` `ret 0`	int

`mov w9, #60`
`str w9, [x8]`
`ret`
	`mov DWORD PTR [rdi], 60`
	`mov rax, rdi`
	`ret`
		`mov QWORD PTR [rcx], 60`
		`mov rax, rcx`
		`ret 0`
			INT

armv7-a clang 11.0.1	**x86-64 gcc 14.2 (-m32)**	**x86 msvc v19.40 VS17.10**	
`mov r0, #60` `bx lr`	`mov eax, 60` `ret`	`mov eax, 60` `ret 0`	int

`mov r1, #60`
`str r1, [r0]`
`bx lr`
	`mov eax, DWORD PTR [esp+4]`
	`mov DWORD PTR [eax], 60`
	`ret 4`
		`mov eax, DWORD PTR ___$ReturnUdt$[esp-4]`
		`mov DWORD PTR [eax], 60`
		`ret 0`
			INT

The problem : digging deeper

Itanium C++ ABI

non-trivial for the purposes of calls

A type is considered non-trivial for the purposes of calls if:

- it has a non-trivial copy constructor, move constructor, or destructor, or
- all of its copy and move constructors are deleted.

different from

C++ reference: Trivially copyable class

Also requires trivial copy and move assignment operators.

Wrapper over `int` : no custom destructor

```
struct INT {
    int value;

    INT(int value = 0) : value{value} {}
};
```

```
int int_seconds() {
    return 60;
}

INT INT_seconds() {
    return 60;
}
```

https://godbolt.org/z/qeq9rEE8T

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	
`mov w0, #60` `ret`	`mov eax, 60` `ret`	`mov eax, 60` `ret 0`	int

`mov w0, #60`
`ret`
	`mov eax, 60`
	`ret`
		`mov QWORD PTR [rcx], 60`
		`mov rax, rcx`
		`ret 0`
			INT

armv7-a clang 11.0.1	**x86-64 gcc 14.2 (-m32)**	**x86 msvc v19.40 VS17.10**	
`mov r0, #60` `bx lr`	`mov eax, 60` `ret`	`mov eax, 60` `ret 0`	int

`mov r0, #60`
`bx lr`
	`mov eax, DWORD PTR [esp+4]`
	`mov DWORD PTR [eax], 60`
	`ret 4`
		`mov eax, DWORD PTR ___$ReturnUdt$[esp-4]`
		`mov DWORD PTR [eax], 60`
		`ret 0`
			INT

Wrapper over `int` : no custom constructor

```
struct INT {
    int value = 0;
};
```

```
int int_seconds() {
    return 60;
}

INT INT_seconds() {
    return INT{60};
}
```

https://godbolt.org/z/c6GbTdjc7

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	
`mov w0, #60` `ret`	`mov eax, 60` `ret`	`mov eax, 60` `ret 0`	int
`mov w0, #60` `ret`	`mov eax, 60` `ret`	`mov eax, 60` `ret 0`	INT
armv7-a clang 11.0.1	**x86-64 gcc 14.2 (-m32)**	**x86 msvc v19.40 VS17.10**	
`mov r0, #60` `bx lr`	`mov eax, 60` `ret`	`mov eax, 60` `ret 0`	int

`mov r0, #60`
`bx lr`
	`mov eax, DWORD PTR [esp+4]`
	`mov DWORD PTR [eax], 60`
	`ret 4`
		`mov eax, 60`
		`ret 0`
			INT

armv8-a System V	x86-64 System V	x86-64 Microsoft
If the argument type is a Composite Type that is larger than 16 bytes, then the argument is copied to memory. … the result is returned in the same registers as would be used for such an argument. Otherwise, … The address… shall be passed … in **`x8`**.	If a C++ object is non-trivial for the purpose of calls, as specified in the C++ ABI, it is passed by invisible reference… in **`%rdi`**… If the class is INTEGER, the next available register of the sequence **`%rax`**, **`%rdx`** is used.	To return a user-defined type by value in **`RAX`**, it must have a length of 1, 2, 4, 8, 16, 32, or 64 bits. It must also have no user-defined constructor, destructor, or copy assignment operator… This definition is essentially the same as a C++03 POD type.
armv7-a System V	**x86 System V**	**x86 Microsoft**
A Composite Type not larger than 4 bytes is returned in **`r0`**… A Composite Type larger than 4 bytes, or whose size cannot be determined statically by both caller and callee, is stored in **memory** at an address passed as an extra argument.	Some fundamental types and all aggregate types are returned in memory.	Return values are … returned in the **`EAX`** register, except for 8-byte structures, which are returned in the **`EDX:EAX`** register pair. Larger structures are returned in the **`EAX`** register as pointers to hidden return structures… Structures that are not PODs will not be returned in registers.

General purpose registers allocation for function parameters and return values

Architecture	ABI	Composite types returned in registers
armv8-a	System V	≤ 16 bytes
armv7-a	System V	≤ 4 bytes
x86-64	System V	≤ 16 bytes
x86	System V	fundamental only
x86-64	Microsoft	1,2,4,8 bytes, C++03 POD
x86	Microsoft	1,2,4,8 bytes, C++03 POD

Composite types are required to be “trivial” to get into registers!

C++ Core Guidelines

C.20: If you can avoid defining default operations, do

Reason It's the simplest and gives the cleanest semantics.

Note This is known as "the rule of zero".

C++ Core Guidelines

C.20: If you can avoid defining default operations, do

Reason It's the simplest and gives the cleanest semantics.

Note This is known as "the rule of zero".

Approved

Surely, this problem

is handled properly

in the popular libraries,

right?

std::chrono

```
#include <chrono>

int64_t int_seconds() {
    return 60;
}

std::chrono::seconds chrono_seconds() {
    return std::chrono::seconds{60};
}

static_assert(std::is_same_v<int64_t, std::chrono::seconds::rep>);
```

https://godbolt.org/z/E5e1nGY94

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	
`mov w0, #60` `ret`	`mov eax, 60` `ret`	`mov eax, 60` `ret 0`	int

`mov w0, #60`
`ret`
	`mov eax, 60`
	`ret`
		`mov QWORD PTR [rcx], 60`
		`mov rax, rcx`
		`ret 0`
			CHR

armv7-a clang 11.0.1	**x86-64 gcc 14.2 (-m32)**	**x86 msvc v19.40 VS17.10**	

`mov r0, #60`
`mov r1, #0`
`bx lr`
	`mov eax, 60`
	`xor edx, edx`
	`ret`
		`mov eax, 60`
		`xor edx, edx`
		`ret 0`
			int

`mov r1, #0`
`mov r2, #60`
`str r2, [r0]`
`str r1, [r0, #4]`
`bx lr`
	`mov eax, DWORD PTR [esp+4]`
	`mov DWORD PTR [eax], 60`
	`mov DWORD PTR [eax+4], 0`
	`ret 4`
		`mov eax, DWORD PTR ___$ReturnUdt$[esp-4]`
		`mov DWORD PTR [eax], 60`
		`mov DWORD PTR [eax+4], 0`
		`ret 0`
			CHR

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	
`mov w0, #60` `ret`	`mov eax, 60` `ret`	`mov eax, 60` `ret 0`	int

`mov w0, #60`
`ret`
	`mov eax, 60`
	`ret`
		`mov QWORD PTR [rcx], 60`
		`mov rax, rcx`
		`ret 0`
		not a POD
			CHR

armv7-a clang 11.0.1	**x86-64 gcc 14.2 (-m32)**	**x86 msvc v19.40 VS17.10**	

`mov r0, #60`
`mov r1, #0`
`bx lr`
	`mov eax, 60`
	`xor edx, edx`
	`ret`
		`mov eax, 60`
		`xor edx, edx`
		`ret 0`
			int

`mov r1, #0`
`mov r2, #60`
`str r2, [r0]`
`str r1, [r0, #4]`
`bx lr`
size > 4
	`mov eax, DWORD PTR [esp+4]`
	`mov DWORD PTR [eax], 60`
	`mov DWORD PTR [eax+4], 0`
	`ret 4`
	not fundamental
		`mov eax, DWORD PTR ___$ReturnUdt$[esp-4]`
		`mov DWORD PTR [eax], 60`
		`mov DWORD PTR [eax+4], 0`
		`ret 0`
		not a POD
			CHR

Can we do something about it?

- `std::chrono` would have to give up encapsulation to be maximally efficient on Windows.
- It cannot use a type smaller than `int64_t` just to optimize code on armv7-a.

`std::pair` and `std::tuple`

- `std::pair` copy and move constructors are defaulted according to the C++ standard.
- Only since C++17 `std::pair` is trivially destructible if its elements are trivially destructible.

  This is an ABI breakage, but a quick search gave only one complaint.
- Copy and move assignment operators are trivial only on MSVC.

  This is not a problem for the function calls.
  But a problem for `std::memcpy` and `std::bit_cast`.
- `std::tuple` is never trivially move constructible.

https://godbolt.org/z/r7McGEb8o

Can we do something about it?

Don’t use `std::pair` and especially `std::tuple`.

Named struct is better for both readability and performance.

Can we do something about `std::unique_ptr`?

```
#include <memory>

namespace detail {

int* smart_ptr_impl() {
    return nullptr;
}

} // namespace detail

[[always_inline]] std::unique_ptr<int> smart_ptr() {
    return std::unique_ptr<int>{detail::smart_ptr_impl()};
}
```

Return Value Optimization (copy elision)

C++ reference: copy elision

Since C++17, a prvalue (“pure” rvalue) is not materialized until needed, and then it is constructed directly into the storage of its final destination.

RVO: how it works

Itanium C++ ABI

3.1.3.1 Non-trivial Return Values

If the return type is a class type that is non-trivial for the purposes of calls, the caller passes an address as an implicit parameter. The callee then constructs the return value into this address.

… the pointer is passed as if it were the first parameter in the function prototype, preceding all other parameters, including the `this` and `VTT` parameters.

RVO: how it works

Itanium C++ ABI

3.1.3.1 Non-trivial Return Values

If the return type is a class type that is non-trivial for the purposes of calls, the caller passes an address as an implicit parameter. The callee then constructs the return value into this address.

… the pointer is passed as if it were the first parameter in the function prototype, preceding all other parameters, including the `this` and `VTT` parameters.

It's an output parameter done right by the compiler, and only when necessary!

Can You RVO?

Using Return Value Optimization for Performance in Bloomberg C++ Codebases

MICHELLE FAE D'SOUZA

Cppcon
The C++ Conference

2024
September 15 - 20

RVO: inserting a function result into a container

```
#include <optional>

struct large {
    large();
    large(large&&);
    large& operator=(large&&);
    large(large const&);
    large& operator=(large const&);
    ~large();
};
large make_large();

std::optional<large> optional_large() {
    return std::optional<large>{make_large()};
}
```

https://godbolt.org/z/bcdsx7aP4

armv8-a clang 18.1.0

```
stp   x29, x30, [sp, #-32]!
str   x19, [sp, #16]
mov   x29, sp
mov   x19, x8
add   x8, x29, #31
bl    make_large()
add   x1, x29, #31
mov   x0, x19
bl    large::large(large&&)
mov   w8, #1
add   x0, x29, #31
strb  w8, [x19, #1]
bl    large::~large()
ldr   x19, [sp, #16]
ldp   x29, x30, [sp], #32
ret
mov   x19, x0
add   x0, x29, #31
bl    large::~large()
mov   x0, x19
bl    _Unwind_Resume

DW.ref.__gxx_personality_v0:
.xword __gxx_personality_v0
```

x86-64 gcc 14.2

```
push  rbp
push  rbx
mov   rbx, rdi
sub   rsp, 24
lea   rbp, [rsp+15]
mov   rdi, rbp
call  make_large()
mov   rsi, rbp
mov   rdi, rbx
call  large::large(large&&)
mov   BYTE PTR [rbx+1], 1
mov   rdi, rbp
call  large::~large()
add   rsp, 24
mov   rax, rbx
pop   rbx
pop   rbp
ret
mov   rbx, rax
jmp   .L2
optional_large() [clone .cold]:
.L2:
mov   rdi, rbp
call  large::~large()
mov   rdi, rbx
call  _Unwind_Resume
```

x64 msvc v19.40 VS17.10

```
$stateUnwindMap$std::optional<
large> optional_large() DB 02H
DB    0aH
DD    imagerel
      large::~large()
DB    080H

mov   QWORD PTR [rsp+8], rcx
push  rbx
sub   rsp, 48
mov   rbx, rcx
lea   rcx, QWORD PTR $T1[rsp]
call  make_large()
npad  1
mov   rdx, rax
mov   rcx, rbx
call  large::large(large&&)
mov   BYTE PTR [rbx+1], 1
lea   rcx, QWORD PTR $T1[rsp]
call  large::~large()
mov   rax, rbx
add   rsp, 48
pop   rbx
ret   0
```

The problem

https://en.cppreference.com/w/cpp/utility/optional/optional

```
template < class U = T >
constexpr optional( U&& value );
```

turns prvalue into rvalue,
which is then forwarded into the storage

The problem

https://en.cppreference.com/w/cpp/utility/optional/optional

```
template < class U = T >
constexpr optional( U&& value );
```

Affects constructors / emplace / insert into all the containers:

- `std::optional`
- `std::variant`
- `std::vector` and all other sequence containers
- `std::map` and all other associative containers

There is a solution!

https://quuxplusone.github.io/blog/2018/05/17/super-elider-round-2/

Arthur O'Dwyer

Recent Date

Stuff mostly about C++

Superconstructing super elider, round 2

```
template<class F>
class with_result_of_t {
```

There is a solution!

https://quuxplusone.github.io/blog/2018/05/17/super-elider-round-2/

Arthur O'Dwyer

Recent Date

Stuff mostly about C++

Superconstructing super elider, round 2

```
template<class F>
class with_result_of_t {
```

https://akrzemi1.wordpress.com/2018/05/16/rvalues-redefined/

Rvalues redefined

Posted on May 16, 2018 by Andrzej Krzemieński

```
1 template <typename F>
2 class rvalue
```

There is a solution!

https://quuxplusone.github

Arthur O'Dwyer

Stuff mostly about C++

Superconstructing su

```
template<class F>
class with_result_of_t {
```

https://akrzemi1.wordpress

Rvalues redefined

Posted on May 16, 2018 by Andrzej Krzemieński

```
template <typename F>
class rvalue
```

C++ now 2024 Aspen, Colorado

CppNow.org

Video Sponsorship Provided By

millennium think-cell

```
template<typename Function>
struct elide {
  explicit constexpr elide(Function f) noexcept(
    std::is_nothrow_move_constructible_v<Function>)
    : f_(std::move(f))
  {}
  template<typename Self>
  constexpr operator decltype(std::declval<Self>().f_())(this Self&& self)
    noexcept(noexcept(std::forward<Self>(self).f_()))
  {
    return std::forward<Self>(self).f_();
  }
private:
  Function f_;
};
```

121

An Adventure in Modern Library Design

Robert Leahy

P3288R3

std::elide

New Proposal, 2024-06-27

There is a solution!

https://quuxplusone.github

Arth

Stuff n

Supercon

```
template<clas
class with_re
```

https://akrze

Rvalues

Posted on May 16, 2018 by Andrzej Krzemieński

```
1 template <typename F>
2 class rvalue
```

C++ now 2024 Aspen, Colorado

Meeting C++ 2023

Prog C++ - Ivan Čukić

think-cell

LAZY LAMBDA

```
template<typename Fn>
class lazy {
public:
    using value_type = std::invoke_result_t<Fn>;

private:
    Fn m_function;
    mutable std::once_flag m_once;
    mutable std::optional<value_type> m_data;

    // ...
```

INTRODUCTION
WRAPS
SWAPS
STATES
ERRORS
VALUES
SAFETY

Meeting C++ 2023 Ivan Čukić kdab.com, cukic.co


```
)(this Self&& self)
```

121

sign

std::elide

New Proposal, 2024-06-27

Lazy evaluation with ac::lazy

https://alcash07.github.io/ACTL/actl/functional/lazy.html

```
template<class Function>
struct lazy {
    operator std::invoke_result_t<Function>() {
        return function();
    }

    Function function;
};
template<class Function>
lazy(Function&&) -> lazy<Function>;

std::optional<large> lazy_optional_large() {
    return std::optional<large>{lazy{make_large}};
}
```

https://godbolt.org/z/PYq6KTPKh

LAZY

armv8-a clang 18.1.0

```
stp   x29, x30, [sp, #-32]!
str   x19, [sp, #16]
mov   x29, sp
mov   x19, x8
bl    make_large()
mov   w8, #1
strb  w8, [x19, #1]
ldr   x19, [sp, #16]
ldp   x29, x30, [sp], #32
ret
```

x86-64 gcc 14.2

```
push  rbx
mov   rbx, rdi
call  make_large()
mov   rax, rbx
mov   BYTE PTR [rbx+1], 1
pop   rbx
ret
```

x64 msvc v19.40 VS17.10

```
push  rbx
sub   rsp, 48
mov   rbx, rcx
call  make_large()
mov   rax, rbx
mov   BYTE PTR [rbx+1], 1
add   rsp, 48
pop   rbx
ret   0
```

LAZY

armv8-a clang 18.1.0

```
stp   x29, x30, [sp, #-32]!
str   x19, [sp, #16]
mov   x29, sp
mov   x19, x8
bl    make_large()
mov   w8, #1
strb  w8, [x19, #1]
ldr   x19, [sp, #16]
ldp   x29, x30, [sp], #32
ret
```

x86-64 gcc 14.2

```
push  rbx
mov   rbx, rdi
call  make_large()
mov   rax, rbx
mov   BYTE PTR [rbx+1], 1
pop   rbx
ret
```

x64 msvc v19.40 VS17.10

```
push  rbx
sub   rsp, 48
mov   rbx, rcx
call  make_large()
mov   rax, rbx
mov   BYTE PTR [rbx+1], 1
add   rsp, 48
pop   rbx
ret   0
```

Negative-overhead abstraction!

C++ Core Guidelines

F.20: For “out” output values, prefer return values to output parameters

Reason A return value is self-documenting, whereas a `&` could be either in-out or out-only and is liable to be misused.

C++ Core Guidelines

F.20: For “out” output values, prefer return values to output parameters

Reason A return value is self-documenting, whereas a `&` could be either in-out or out-only and is liable to be misused.

Approved

Valid use cases for output parameters

```
std::ranges::transform(x, y, z);
std::ranges::sort(x);
```

Return value cannot be allocated on stack,

for example, because it’s a range with run-time size.

If we decouple memory allocation and data processing,

the code is more reusable.

ac::out and ac::inout

https://alcash07.github.io/ACTL/actl/functional/out_inout.html

```
template<class InRange, class OutRange, class Function>
void transform(InRange const& src, ac::out<OutRange&> dst, Function f);

template<class Range>
void sort(ac::inout<Range&> range);

template<class Range>
[[nodiscard]] Range sort(Range const& range);

transform(x, ac::out{y}, z);
sort(ac::inout{x});
auto y = sort(x);
```

Section 0. Introduction

Section 1. Return value

Section 2. Parameter passing

Section 3. Multiple parameters

C++ Core Guidelines

F.16: For “in” parameters, pass cheaply-copied types by value and others by reference to `const`

Reason Both let the caller know that a function will not modify the argument, and both allow initialization by rvalues.

What is “cheap to copy” depends on the machine architecture, but two or three words (doubles, pointers, references) are usually best passed by value.

int parameter

```
bool value_is_zero(int x) {
    return x == 0;
}
```

- passing by value

```
bool ref_is_zero(int const& x) {
    return x == 0;
}
```

- passing by (const) reference

https://godbolt.org/z/ofznMvKWc

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	

`cmp w0, #0`
`cset w0, eq`
`ret`
	`test edi, edi`
	`sete al`
	`ret`
		`test ecx, ecx`
		`sete al`
		`ret 0`
			VALUE

`ldr w8, [x0]`
`cmp w8, #0`
`cset w0, eq`
`ret`
	`mov eax, DWORD PTR [rdi]`
	`test eax, eax`
	`sete al`
	`ret`
		`cmp DWORD PTR [rcx], 0`
		`sete al`
		`ret 0`
			REF

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	

`cmp w0, #0`
`cset w0, eq`
`ret`
	`test edi, edi`
	`sete al`
	`ret`
		`test ecx, ecx`
		`sete al`
		`ret 0`
			VALUE

`ldr w8, [x0]`
`cmp w8, #0`
`cset w0, eq`
`ret`
	`mov eax, DWORD PTR [rdi]`
	`test eax, eax`
	`sete al`
	`ret`
		`cmp DWORD PTR [rcx], 0`
		`sete al`
		`ret 0`
			REF

Reference has to be dereferenced

int parameter : call site

```
bool value_is_zero(int x);
bool ref_is_zero(int const& x);

bool value_is_zero_call() {
    return value_is_zero(1);
}

bool ref_is_zero_call() {
    return ref_is_zero(1);
}
```

- passing by value

- passing by (const) reference

https://godbolt.org/z/fzshqhd85

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	
`mov w0, #1` `b value_is_zero(int)`	`mov edi, 1` `jmp value_is_zero(int)`	`mov ecx, 1` `jmp value_is_zero(int)`	VALUE

`sub sp, sp, #32`
`stp x29, x30, [sp, #16]`
`add x29, sp, #16`
`mov w8, #1`
`sub x0, x29, #4`
`stur w8, [x29, #-4]`
`bl ref_is_zero(int const&)`
`and w0, w0, #0x1`
`ldp x29, x30, [sp, #16]`
`add sp, sp, #32`
`ret`
	`sub rsp, 24`
	`lea rdi, [rsp+12]`
	`mov DWORD PTR [rsp+12], 1`
	`call ref_is_zero(int const&)`
	`add rsp, 24`
	`ret`
		`sub rsp, 40`
		`lea rcx, QWORD PTR $T1[rsp]`
		`mov DWORD PTR $T1[rsp], 1`
		`call ref_is_zero(int const&)`
		`add rsp, 40`
		`ret 0`
			REF

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	
`mov w0, #1` `b value_is_zero(int)`	`mov edi, 1` `jmp value_is_zero(int)`	`mov ecx, 1` `jmp value_is_zero(int)`	VALUE

Here, we just put constant 1 into a register and call the function.

Below, we put constant 1 on the stack and pass its address, and after the function call we restore the stack.

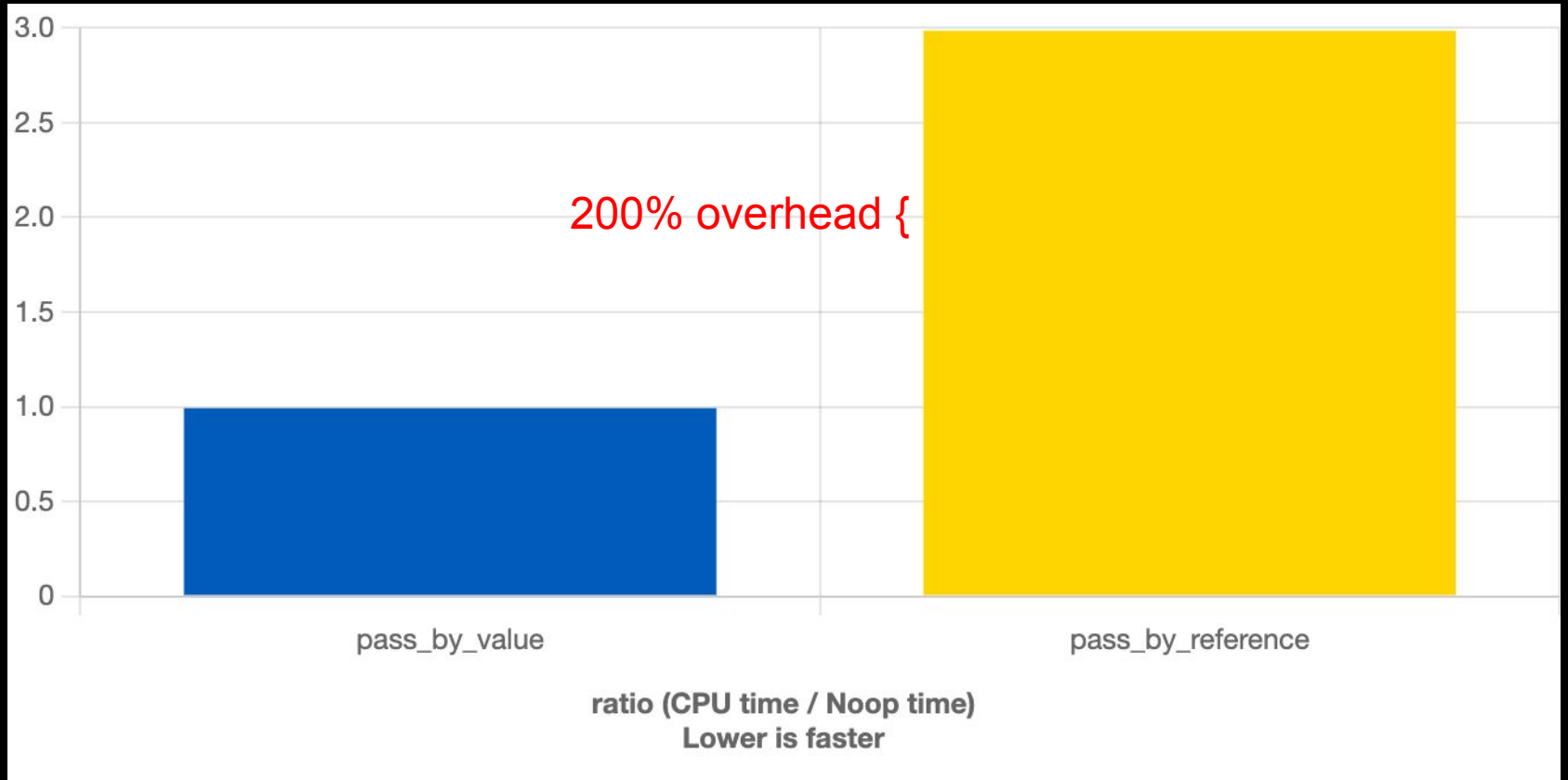
`sub sp, sp, #32`
`stp x29, x30, [sp, #16]`
`add x29, sp, #16`
`mov w8, #1`
`sub x0, x29, #4`
`stur w8, [x29, #-4]`
`bl ref_is_zero(int const&)`
`and w0, w0, #0x1`
`ldp x29, x30, [sp, #16]`
`add sp, sp, #32`
`ret`
	`sub rsp, 24`
	`lea rdi, [rsp+12]`
	`mov DWORD PTR [rsp+12], 1`
	`call ref_is_zero(int const&)`
	`add rsp, 24`
	`ret`
		`sub rsp, 40`
		`lea rcx, QWORD PTR $T1[rsp]`
		`mov DWORD PTR $T1[rsp], 1`
		`call ref_is_zero(int const&)`
		`add rsp, 40`
		`ret 0`
			REF

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	
`mov w0, #1` `b value_is_zero(int)`	`mov edi, 1` `jmp value_is_zero(int)`	`mov ecx, 1` `jmp value_is_zero(int)`	VALUE
No profiler will guide you here			

`sub sp, sp, #32`
`stp x29, x30, [sp, #16]`
`add x29, sp, #16`
`mov w8, #1`
`sub x0, x29, #4`
`stur w8, [x29, #-4]`
`bl ref_is_zero(int const&)`
`and w0, w0, #0x1`
`ldp x29, x30, [sp, #16]`
`add sp, sp, #32`
`ret`
	`sub rsp, 24`
	`lea rdi, [rsp+12]`
	`mov DWORD PTR [rsp+12], 1`
	`call ref_is_zero(int const&)`
	`add rsp, 24`
	`ret`
		`sub rsp, 40`
		`lea rcx, QWORD PTR $T1[rsp]`
		`mov DWORD PTR $T1[rsp], 1`
		`call ref_is_zero(int const&)`
		`add rsp, 40`
		`ret 0`
			REF

Quick benchmark

https://quick-bench.com/q/qVbxyQvoqxN76wfqnWVrFF8kqvQ



int parameter : extra function

```
void some_extra_function();

bool value_extra_function(int x) {
    int const copy = x;
    some_extra_function();
    return copy == x;
}

bool ref_extra_function(int const& x) {
    int const copy = x;
    some_extra_function();
    return copy == x;
}
```

- passing by value

- passing by (const) reference

https://godbolt.org/z/4r946xh8T

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	

`stp x29, x30, [sp, #-16]!`
`mov x29, sp`
`bl some_extra_function()`
`mov w0, #1`
`ldp x29, x30, [sp], #16`
`ret`
	`sub rsp, 8`
	`call some_extra_function()`
	`mov eax, 1`
	`add rsp, 8`
	`ret`
		`sub rsp, 40`
		`call some_extra_function()`
		`mov al, 1`
		`add rsp, 40`
		`ret 0`
			VALUE

`stp x29, x30, [sp, #-32]!`
`stp x20, x19, [sp, #16]`
`mov x29, sp`
`ldr w20, [x0]`
`mov x19, x0`
`bl some_extra_function()`
`ldr w8, [x19]`
`cmp w20, w8`
`cset w0, eq`
`ldp x20, x19, [sp, #16]`
`ldp x29, x30, [sp], #32`
`ret`
	`push rbp`
	`push rbx`
	`mov rbx, rdi`
	`sub rsp, 8`
	`mov ebp, DWORD PTR [rdi]`
	`call some_extra_function()`
	`cmp DWORD PTR [rbx], ebp`
	`sete al`
	`add rsp, 8`
	`pop rbx`
	`pop rbp`
	`ret`
		`mov QWORD PTR [rsp+8], rbx`
		`push rdi`
		`sub rsp, 32`
		`mov ebx, DWORD PTR [rcx]`
		`mov rdi, rcx`
		`call some_extra_function()`
		`cmp ebx, DWORD PTR [rdi]`
		`mov rbx, QWORD PTR [rsp+48]`
		`sete al`
		`add rsp, 32`
		`pop rdi`
		`ret 0`
			REF

int parameter : extra function

```
void some_extra_function();

bool value_extra_function(int x) {
    int const copy = x;
    some_extra_function();
    return copy == x;
}

bool ref_extra_function(int const& x) {
    int const copy = x;
    some_extra_function();
    return copy == x;
}
```

- passing by value

- passing by (const) reference

- can change the referenced value

Perfect forwarding

“In C++, perfect forwarding is the act of passing a function’s parameters to another function while preserving its reference category.” link

The main purpose is to replace copies with moves when possible.

```
template<class T, class... Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(
        new T(std::forward<Args>(args)...));
}
```

Perfect forwarding is not perfect!

"In C++, perfect forwarding is the act of passing a function's parameters to another function while preserving its reference category." link

The main purpose is to replace copies with moves when possible.

```
template<class T, class... Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(
        new T(std::forward<Args>(args)...));
}
```

breaks RVO

Perfect forwarding is not perfect!

"In C++, perfect forwarding is the act of passing a function's parameters to another function while preserving its reference category." link

The main purpose is to replace copies with moves when possible.

```
template<class T, class... Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(
        new T(std::forward<Args>(args)...));
}
```

forwarding reference is still a reference, so it prevents passing in registers

breaks RVO

Hopefully, you’re convinced that built-in types should be passed by value. Now, let’s see which C++ abstractions should also be passed by value.

Chandler Carruth: There Are No Zero-Cost Abstractions

Cppcon The C++ Conference | 2019 cppcon.org

There Are No
Zero-Cost Abstractions

```
#include <memory>

void bar(int* ptr);

// Takes ownership.
void baz(int* ptr);

void foo(int* ptr) {
  if (*ptr > 42) {
    bar(ptr);
    *ptr = 42;
  }
  baz(ptr);
}
```

```
#include <memory>

void bar(int* ptr);

// Takes ownership.
void baz(unique_ptr<int> ptr);

void foo(unique_ptr<int> ptr) {
  if (*ptr > 42) {
    bar(ptr.get());
    *ptr = 42;
  }
  baz(std::move(ptr));
}
```

The problem

Itanium C++ ABI

3.1.2.3 Non-Trivial Parameters

If a parameter type is a class type that is non-trivial for the purposes of calls, the caller must allocate space for a temporary and pass that temporary by reference.

For such types, passing by reference is likely more efficient,
because it avoids making an extra copy on the stack
(unless you need that copy anyway).

C++ Core Guidelines

F.24: Use a `span<T>` or a `span_p<T>` to designate a half-open sequence

Reason Informal/non-explicit ranges are a source of errors.

“use span” + “use a span”: 16 occurrences

C++20 std::span vs raw pointer and size

```
#include <span>

int raw_back(int const* ptr, size_t size) {
    return ptr[size - 1];
}

int span_back(std::span<int const> span) {
    return span[span.size() - 1];
}
```

https://godbolt.org/z/bez7c5PMK

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	

`add x8, x0, x1, lsl #2`
`ldur w0, [x8, #-4]`
`ret`
	`mov eax, DWORD PTR [rdi-4+rsi*4]`
	`ret`
		`mov eax, DWORD PTR [rcx+rdx*4-4]`
		`ret 0`
			RAW

`add x8, x0, x1, lsl #2`
`ldur w0, [x8, #-4]`
`ret`
	`mov eax, DWORD PTR [rdi-4+rsi*4]`
	`ret`
		`mov rdx, QWORD PTR [rcx+8]`
		`mov rax, QWORD PTR [rcx]`
		`mov eax, DWORD PTR [rax+rdx*4-4]`
		`ret 0`
			SPAN

armv7-a clang 11.0.1	**x86-64 gcc 14.2 (-m32)**	**x86 msvc v19.40 VS17.10**	

`add r0, r0, r1, lsl #2`
`ldr r0, [r0, #-4]`
`bx lr`
	`mov eax, DWORD PTR [esp+4]`
	`mov edx, DWORD PTR [esp+8]`
	`mov eax, DWORD PTR [eax-4+edx*4]`
	`ret`
		`mov ecx, DWORD PTR _size$[esp-4]`
		`mov eax, DWORD PTR _ptr$[esp-4]`
		`mov eax, DWORD PTR [eax+ecx*4-4]`
		`ret 0`
			RAW

`add r0, r0, r1, lsl #2`
`ldr r0, [r0, #-4]`
`bx lr`
	`mov eax, DWORD PTR [esp+8]`
	`lea eax, [-4+eax*4]`
	`add eax, DWORD PTR [esp+4]`
	`mov eax, DWORD PTR [eax]`
	`ret`
		`mov ecx, DWORD PTR _span$[esp]`
		`mov eax, DWORD PTR _span$[esp-4]`
		`mov eax, DWORD PTR [eax+ecx*4-4]`
		`ret 0`
			SPAN

C++23 std::mdspan vs raw pointer and sizes

```
#include <cstddef>

int raw_back2(int const* ptr, size_t width, size_t height) {
    return ptr[width * height - 1];
}

struct mdspan2 {
    int const* ptr;
    size_t width;
    size_t height;
};

int mdspan_back2(mdspan2 span) {
    return span.ptr[span.width * span.height - 1];
}
```

https://godbolt.org/z/EcfanMoYf

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	

`mul x8, x2, x1`
`add x8, x0, x8, lsl #2`
`ldur w0, [x8, #-4]`
`ret`
	`imul rsi, rdx`
	`mov eax, DWORD PTR [rdi-4+rsi*4]`
	`ret`
		`imul rdx, r8`
		`mov eax, DWORD PTR [rcx+rdx*4-4]`
		`ret 0`
			RAW

`ldp x8, x9, [x0, #8]`
`mul x8, x9, x8`
`ldr x9, [x0]`
`add x8, x9, x8, lsl #2`
`ldur w0, [x8, #-4]`
`ret`
	`mov rax, QWORD PTR [rsp+16]`
	`imul rax, QWORD PTR [rsp+24]`
	`mov rdx, QWORD PTR [rsp+8]`
	`mov eax, DWORD PTR [rdx-4+rax*4]`
	`ret`
		`mov rdx, QWORD PTR [rcx+16]`
		`imul rdx, QWORD PTR [rcx+8]`
		`mov rax, QWORD PTR [rcx]`
		`mov eax, DWORD PTR [rax+rdx*4-4]`
		`ret 0`
			SPAN

armv7-a clang 11.0.1	**x86-64 gcc 14.2 (-m32)**	**x86 msvc v19.40 VS17.10**	

`mul r3, r2, r1`
`add r0, r0, r3, lsl #2`
`ldr r0, [r0, #-4]`
`bx lr`
	`mov eax, DWORD PTR [esp+12]`
	`imul eax, DWORD PTR [esp+8]`
	`mov edx, DWORD PTR [esp+4]`
	`mov eax, DWORD PTR [edx-4+eax*4]`
	`ret`
		`mov ecx, DWORD PTR _width$[esp-4]`
		`imul ecx, DWORD PTR _height$[esp-4]`
		`mov eax, DWORD PTR _ptr$[esp-4]`
		`mov eax, DWORD PTR [eax+ecx*4-4]`
		`ret 0`
			RAW

`mul r3, r1, r2`
`add r0, r0, r3, lsl #2`
`ldr r0, [r0, #-4]`
`bx lr`
	`mov eax, DWORD PTR [esp+8]`
	`imul eax, DWORD PTR [esp+12]`
	`mov edx, DWORD PTR [esp+4]`
	`mov eax, DWORD PTR [edx-4+eax*4]`
	`ret`
		`mov ecx, DWORD PTR _span$[esp+4]`
		`imul ecx, DWORD PTR _span$[esp]`
		`mov eax, DWORD PTR _span$[esp-4]`
		`mov eax, DWORD PTR [eax+ecx*4-4]`
		`ret 0`
			SPAN

armv8-a System V	x86-64 System V	x86-64 Microsoft
If the argument type is a Composite Type that is larger than 16 bytes, then the argument is copied to memory allocated by the caller and the argument is replaced by a pointer to the copy.	If the class is MEMORY, pass the argument on the stack… If the size of the aggregate exceeds two eightbytes and the first eightbyte isn't SSE or any other eightbyte isn't SSEUP, the whole argument is passed in memory.	Any argument that doesn't fit in 8 bytes, or isn't 1, 2, 4, or 8 bytes, must be passed by reference. A single argument is never spread across multiple registers.

armv7-a System V	x86 System V	x86 Microsoft
When a Composite Type argument is assigned to core registers (either fully or partially), the behavior is as if the argument had been stored to memory at a word-aligned (4-byte) address and then loaded into consecutive registers using a suitable load-multiple instruction.	Most parameters are passed on the stack. - The first three parameters of type `__m64` are passed in `%mm0`, `%mm1`, and `%mm2`…	Parameters are pushed onto the stack from right to left. `__fastcall`: Classes, structs, and unions: Treated as "multibyte" types (regardless of size) and passed on the stack.

General purpose registers allocation for function parameters and return values

Architecture	ABI	Composite types returned in registers	Composite types passed in registers
armv8-a	System V	≤ 16 bytes	≤ 16 bytes
armv7-a	System V	≤ 4 bytes	≤ 16 bytes
x86-64	System V	≤ 16 bytes	≤ 16 bytes
x86	System V	fundamental only	SIMD only
x86-64	Microsoft	1,2,4,8 bytes, C++03 POD	1,2,4,8 bytes
x86	Microsoft	1,2,4,8 bytes, C++03 POD	not even fundamental
x86 __fastcall	Microsoft	1,2,4,8 bytes, C++03 POD	fundamental only

Composite types are required to be “trivial” to get into registers!

C++ Core Guidelines

F.24: Use a `span<T>` or a `span_p<T>` to designate a half-open sequence

Reason Informal/non-explicit ranges are a source of errors.

Not free

Empty parameter : use cases

- Predicates and transform function passed to STL algorithms:

```
std::ranges::find_if(range, predicate);

std::ranges::transform(input_range, output, unary_op);
```

- Tag dispatch (somewhat obsolete after C++20 concepts)

```
template <class InputIter, class Diff = iter_difference_t<InputIter>>
void advance(InputIter& iter, Diff n, input_iterator_tag) {
    for (; n > 0; --n)
        ++iter;
}
template <class RandIter, class Diff = iter_difference_t<RandIter>>
void advance(RandIter& iter, Diff n, random_access_iterator_tag) {
    iter += n;
}
```

- Access token to make some API available only inside the library (like the default “package private” access modifier in Java)

Empty parameter : tag dispatch

```
int raw_rand();

struct mt19937 {};
int tagged_rand(mt19937);

int raw_rand_call() {
    return raw_rand();
}

int tagged_rand_call() {
    return tagged_rand(mt19937{});
}
```

https://godbolt.org/z/vxG4eY1r4

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	120
`b raw_rand()`	`jmp raw_rand()`	`jmp raw_rand()`	RAW
`b tagged_rand(mt19937)`	`jmp tagged_rand(mt19937)`	`xor ecx, ecx` `jmp tagged_rand(mt19937)`	TAG
armv7-a clang 11.0.1	**x86-64 gcc 14.2 (-m32)**	**x86 msvc v19.40 VS17.10**	
`b raw_rand()`	`jmp raw_rand()`	`jmp raw_rand()`	RAW

`b tagged_rand(mt19937)`	`sub esp, 24`
	`push 0`
	`call tagged_rand(mt19937)`
	`add esp, 28`
	`ret`
		`push ecx`
		`mov BYTE PTR $T1[esp+4], 0`
		`push DWORD PTR $T1[esp+4]`
		`call tagged_rand(mt19937)`
		`add esp, 8`
		`ret 0`
			TAG

Empty parameter

Itanium C++ ABI

2.2 POD Data Types

If the base ABI does not specify rules for empty classes, then an empty class has size and alignment 1.

3.1.2.6 Empty Parameters

Arguments of empty class types that are not non-trivial for the purposes of calls are passed no differently from ordinary classes.

C++ Core Guidelines

F.16: For “in” parameters, pass cheaply-copied types by value and others by reference to `const`

Reason Both let the caller know that a function will not modify the argument, and both allow initialization by rvalues.

What is “cheap to copy” depends on the machine architecture, but two or three words (doubles, pointers, references) are usually best passed by value.

C++ Core Guidelines

F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to `const`

Reason Both let the caller know that a function will not modify the argument, and both allow initialization by rvalues.

What is "cheap to copy" depends on the machine architecture, but two or three words (doubles, pointers, references) are usually best passed by value.

Approved*

Class member functions

Itanium C++ ABI

3.1.2.1 this Parameters

Non-static member functions, including constructors and destructors, take an implicit `this` parameter of pointer type. It is passed as if it were the first parameter in the function prototype…

This isn’t efficient if the class is small enough to be passed by value.

Effect on empty function objects

```
template<class T>
struct plus {
    constexpr T operator()(T const& lhs, T const& rhs) const {
        return lhs + rhs;
    }
};
```

Simple function objects like `std::plus` above would most likely be inlined, but more complex empty function objects would introduce overhead if not inlined.

Effect on empty function objects

```
template<class T>
struct plus {
    constexpr T operator()(T const& lhs, T const& rhs) const {
        return lhs + rhs;
    }
};
```

Simple function objects like `std::plus` above would most likely be inlined, but more complex empty function objects would introduce overhead if not inlined.

Luckily, C++23 introduces `static operator()` and `[]`.

Section 0. Introduction

Section 1. Return value

Section 2. Parameter passing

Section 3. Multiple parameters

Chain of function calls

```
int sum(int x1, int x2);

int sum_12_3(int x1, int x2, int x3) {
    return sum(sum(x1, x2), x3);
}
int sum_13_2(int x1, int x2, int x3) {
    return sum(sum(x1, x3), x2);
}
int sum_23_1(int x1, int x2, int x3) {
    return sum(sum(x2, x3), x1);
}
```

Chain of function calls

```
int sum(int x1, int x2);

int sum_12_3(int x1, int x2, int x3) {
    return sum(sum(x1, x2), x3);
}
int sum_13_2(int x1, int x2, int x3) {
    return sum(sum(x1, x3), x2);
}
int sum_23_1(int x1, int x2, int x3) {
    return sum(sum(x2, x3), x1);
}
int sum_21_3(int x1, int x2, int x3) {
    return sum(sum(x2, x1), x3);
}
```

https://godbolt.org/z/MsjeT8TTK

	armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10
`sum_12_3`	9 instructions	7 instructions	9 instructions
`sum_13_2`	10 instructions	8 instructions	10 instructions
`sum_23_1`	11 instructions	9 instructions	12 instructions
`sum_21_3`	12 instructions	10 instructions	12 instructions

sum_12_3	sum_13_2
`push rbx`	`push rbx`
`mov ebx, edx`	`mov ebx, esi`
	`mov esi, edx`
`call sum(int, int)`	`call sum(int, int)`
`mov esi, ebx`	`mov esi, ebx`
`pop rbx`	`pop rbx`
`mov edi, eax`	`mov edi, eax`
`jmp sum(int, int)`	`jmp sum(int, int)`

Order of parameters is fixed in every ABI

```
int sum(int x1, int x2);


int sum_12_3(int x1, int x2, int x3) {
    return sum(sum(x1, x2), x3);
}
int sum_13_2(int x1, int x2, int x3) {
    return sum(sum(x1, x3), x2);
}
int sum_23_1(int x1, int x2, int x3) {
    return sum(sum(x2, x3), x1);
}
int sum_12_3(int x1, int x2, int x3) {
    return sum(sum(x2, x1), x3);
}
```

swap is required (3 moves)

	armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10
`sum_12_3`	9 instructions	7 instructions	9 instructions
`sum_13_2`	10 instructions	8 instructions	10 instructions
`sum_23_1`	11 instructions	9 instructions	12 instructions
`sum_21_3`	12 instructions	10 instructions	12 instructions

Eduardo Madrid: about the overhead of `std::function`

Cppcon The C++ Conference September 13-18

Eduardo Madrid

Not Leaving Performance On The Jump Table

godbolt.org

Watch C++ Weekly to learn new C++ features

C++ source #1

```
#include <functional>

using Signature = void(
    void *, // <-- notice this eccentricity
    int, int, int, int
);

using OrderConsumer = std::function<Signature>;

void invokeConsumer(
    void *data,
    int instrumentId, int ticks, int side, int quantity,
    OrderConsumer &oc
) {
    oc(data, instrumentId, ticks, side, quantity);
}
```

x86-64 clang 10.0.1 (Editor #1, Compiler #1) C++

x86-64 clang 10.0.1 -std=c++17 -O2

```
invokeConsumer(void*, int, int, int, int, std::function
        sub     rsp, 24
        mov     qword ptr [rsp + 16], rdi
        mov     dword ptr [rsp + 12], esi
        mov     dword ptr [rsp + 8], edx
        mov     dword ptr [rsp + 4], ecx
        mov     dword ptr [rsp], r8d
        cmp     qword ptr [r9 + 16], 0
        je      .LBB0_2
        mov     rax, r9
        lea     rsi, [rsp + 16]
        lea     rdx, [rsp + 12]
        lea     rcx, [rsp + 8]
        lea     r8, [rsp + 4]
        mov     r9, rsp
        mov     rdi, rax
        call    qword ptr [rax + 24]
        add     rsp, 24
        ret
.LBB0_2:
        call    std::__throw_bad_function_call()
```

Eduardo Madrid: about the overhead of `std::function`

Cppcon The C++ Conference September 13-18

Eduardo Madrid

Not Leaving Performance On The Jump Table

godbolt.org — COMPILER EXPLORER

C++ source #1

```
#include <functional>

using Signature = void(
    void *, // <-- notice this eccentricity
    int, int, int, int
);

using OrderConsumer = std::function<Signature>;

void invokeConsumer(
    void *data,
    int instrumentId, int ticks, int side, int quantity,
    OrderConsumer &oc
) {
    oc(data, instrumentId, ticks, side, quantity);
}
```

x86-64 clang 10.0.1 (Editor #1, Compiler #1) C++ — x86-64 clang 10.0.1 — -std=c++17 -O2

```
invokeConsumer(void*, int, int, int, int, std::function
        sub     rsp, 24
        mov     qword ptr [rsp + 16], rdi
        mov     dword ptr [rsp + 12], esi
        mov     dword ptr [rsp + 8], edx
        mov     dword ptr [rsp + 4], ecx
        mov     dword ptr [rsp], r8d
        cmp     qword ptr [r9 + 16], 0
        je      .LBB0_2
        mov     rax, r9
        lea     rsi, [rsp + 16]
        lea     rdx, [rsp + 12]
        lea     rcx, [rsp + 8]
        lea     r8, [rsp + 4]
        mov     r9, rsp
        mov     rdi, rax
        call    qword ptr [rax + 24]
        add     rsp, 24
        ret
.LBB0_2:
        call    std::__throw_bad_function_call()
```

Knowledge needed

- `this` parameter passing, because `std::function` is a function object
- consistent parameters order
- enhanced “perfect forwarding”, which preserves passing in registers

C++ Core Guidelines

I.13: Do not pass an array as a single pointer

Example Consider:

```
void copy_n(const T* p, T* q, int n); // copy from [p:p+n) to [q:q+n)
```

What if there are fewer than `n` elements in the array pointed to by `q`? Then, we overwrite some probably unrelated memory. What if there are fewer than `n` elements in the array pointed to by `p`? Then, we read some probably unrelated memory. Either is undefined behavior and a potentially very nasty bug.

Alternative Consider using explicit spans:

```
void copy(span<const T> r, span<T> r2); // copy r to r2
```

Copy of a byte span

```
#define NDEBUG
#include <cassert>
#include <cstddef>
#include <cstring>

void raw_copy(std::byte* dst, std::byte const* src, size_t size) {
    std::memcpy(dst, src, size);
}

void checked_copy( // imagine 2 std::spans here
    std::byte* dst, std::byte const* src, size_t dst_size, size_t src_size
) {
    assert(src_size == dst_size);
    std::memcpy(dst, src, dst_size);
}
```

https://godbolt.org/z/3Tqs849eh

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	139
`b memcpy`	`jmp memcpy`	`jmp memcpy`	RAW
`b memcpy`	`jmp memcpy`	`jmp memcpy`	CHECKED

Copy of a byte span : call site

```
#include <array>
#include <cstddef>

void raw_copy(std::byte* dst, std::byte const* src, size_t size);
void checked_copy(
    std::byte* dst, std::byte const* src, size_t dst_size, size_t src_size
);
std::array<std::byte, 8> arr;

void raw_copy_call() {
    raw_copy(arr.data(), arr.data(), 8);
}
void checked_copy_call() {
    checked_copy(arr.data(), arr.data(), 8, 8);
}
```

https://godbolt.org/z/7M45xz9ha

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	

`adrp x0, arr`
`add x0, x0, :lo12:arr`
`mov w2, #8`
`mov x1, x0`
`b raw_copy`
	`mov esi, OFFSET FLAT:arr`
	`mov edx, 8`
	`mov rdi, rsi`
	`jmp raw_copy`
		`mov r8d, 8`
		`lea rdx, OFFSET FLAT:arr`
		`lea rcx, OFFSET FLAT:arr`
		`jmp raw_copy`
			RAW

`adrp x0, arr`
`add x0, x0, :lo12:arr`
`mov w2, #8`
`mov x1, x0`
`mov w3, #8`
`b checked_copy`
	`mov esi, OFFSET FLAT:arr`
	`mov ecx, 8`
	`mov edx, 8`
	`mov rdi, rsi`
	`jmp checked_copy`
		`mov r9d, 8`
		`lea rdx, OFFSET FLAT:arr`
		`mov r8d, r9d`
		`lea rcx, OFFSET FLAT:arr`
		`jmp checked_copy`
			CHECKED

armv8-a clang 18.1.0	x86-64 gcc 14.2	x64 msvc v19.40 VS17.10	

`adrp x0, arr`
`add x0, x0, :lo12:arr`
`mov w2, #8`
`mov x1, x0`
`b raw_copy`
	`mov esi, OFFSET FLAT:arr`
	`mov edx, 8`
	`mov rdi, rsi`
	`jmp raw_copy`
		`mov r8d, 8`
		`lea rdx, OFFSET FLAT:arr`
		`lea rcx, OFFSET FLAT:arr`
		`jmp raw_copy`
			RAW

`adrp x0, arr`
`add x0, x0, :lo12:arr`
`mov w2, #8`
`mov x1, x0`
`mov w3, #8`
`b checked_copy`
	`mov esi, OFFSET FLAT:arr`
	`mov ecx, 8`
	`mov edx, 8`
	`mov rdi, rsi`
	`jmp checked_copy`
		`mov r9d, 8`
		`lea rdx, OFFSET FLAT:arr`
		`mov r8d, r9d`
		`lea rcx, OFFSET FLAT:arr`
		`jmp checked_copy`
			CHECKED

Size is passed twice!

C++ Core Guidelines

I.13: Do not pass an array as a single pointer

Example Consider:

```
void copy_n(const T* p, T* q, int n); // copy from [p:p+n) to [q:q+n)
```

Alternative Consider using explicit spans:

```
void copy(span<const T> r, span<T> r2); // copy r to r2
```

Not free

C++ Core Guidelines

I.23: Keep the number of function arguments low

Reason Having many arguments opens opportunities for confusion. Passing lots of arguments is often costly compared to alternatives.

Discussion The two most common reasons why functions have too many parameters are:

1. Missing an abstraction. …
2. Violating “one function, one responsibility.” …

Triple product (wiki)

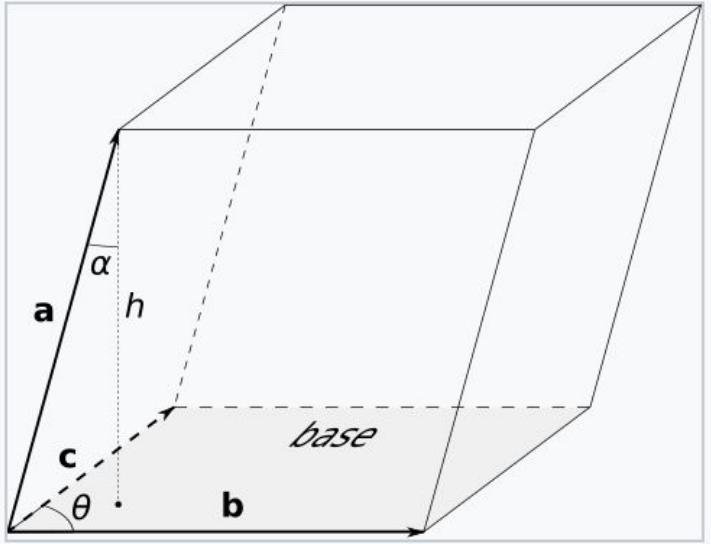
Geometrically, the scalar triple product is the (signed) volume of the parallelepiped defined by the three vectors.

The scalar triple product is unchanged under a circular shift of its three operands **(a, b, c)**:

a · (b x c) = b · (c x a) = c · (a x b)

Swapping the positions of the operators without re-ordering the operands leaves the triple product unchanged:

a · (b x c) = (a x b) · c



Three vectors defining a parallelepiped

Triple product : all `int`

```
struct vector3 {
    int x, y, z;
};

int dot_product(int ax, int ay, int az, int bx, int by, int bz);
vector3 cross_product(int ax, int ay, int az, int bx, int by, int bz);

int triple_product(
    int ax, int ay, int az,
    int bx, int by, int bz,
    int cx, int cy, int cz
) {
    vector3 d = cross_product(ax, ay, az, bx, by, bz);
    return dot_product(d.x, d.y, d.z, cx, cy, cz);
}
```

Triple product : vector3

```
struct vector3 {
    int x, y, z;
};

int vector_dot_product(vector3 const& a, vector3 const& b);
vector3 vector_cross_product(vector3 const& a, vector3 const& b);

int vector_triple_product(
    vector3 const& a,
    vector3 const& b,
    vector3 const& c
) {
    return vector_dot_product(vector_cross_product(a, b), c);
}
```

https://godbolt.org/z/PfdcEjo1h

VECTOR

armv8-a clang 18.1.0

```
sub   sp, sp, #48
stp   x29, x30, [sp, #16]
str   x19, [sp, #32]
add   x29, sp, #16
mov   x19, x2
bl    vector_cross_product
str   x0, [sp]
mov   x0, sp
str   w1, [sp, #8]
mov   x1, x19
bl    vector_dot_product
ldp   x29, x30, [sp, #16]
ldr   x19, [sp, #32]
add   sp, sp, #48
ret
```

x86-64 gcc 14.2

```
push  rbx
mov   rbx, rdx
sub   rsp, 16
call  vector_cross_product
lea   rdi, [rsp+4]
mov   rsi, rbx
mov   QWORD PTR [rsp+4], rax
mov   DWORD PTR [rsp+12], edx
call  vector_dot_product
add   rsp, 16
pop   rbx
ret
```

x64 msvc v19.40 VS17.10

```
push  rbx
sub   rsp, 80
mov   rax, QWORD PTR
      __security_cookie
xor   rax, rsp
mov   QWORD PTR
      $ArrayPad$[rsp], rax
mov   rbx, r8
mov   r8, rdx
mov   rdx, rcx
lea   rcx, QWORD PTR $T1[rsp]
call  vector_cross_product
mov   rdx, rbx
lea   rcx, QWORD PTR $T2[rsp]
movsd xmm0, QWORD PTR [rax]
movsd QWORD PTR $T2[rsp], xmm0
mov   eax, DWORD PTR [rax+8]
mov   DWORD PTR $T2[rsp+8], eax
call  vector_dot_product
mov   rcx, QWORD PTR
      __$ArrayPad$[rsp]
xor   rcx, rsp
call  __security_check_cookie
add   rsp, 80
pop   rbx
ret   0
```

Keep an eye out for buffer security checks

by Nicholas Frechette

VECTOR

armv8-a clang 18.1.0

```
sub   sp, sp, #48
stp   x29, x30, [sp, #16]
str   x19, [sp, #32]
add   x29, sp, #16
mov   x19, x2
bl    vector_cross_product
str   x0, [sp]
mov   x0, sp
str   w1, [sp, #8]
mov   x1, x19
bl    vector_dot_product
ldp   x29, x30, [sp, #16]
ldr   x19, [sp, #32]
add   sp, sp, #48
ret
```

x86-64 gcc 14.2

```
push  rbx
mov   rbx, rdx
sub   rsp, 16
call  vector_cross_product
lea   rdi, [rsp+4]
mov   rsi, rbx
mov   QWORD PTR [rsp+4], rax
mov   DWORD PTR [rsp+12], edx
call  vector_dot_product
add   rsp, 16
pop   rbx
ret
```

x64 msvc v19.40 VS17.10

```
push  rbx
sub   rsp, 64
mov   rbx, r8
mov   r8, rdx
mov   rdx, rcx
lea   rcx, QWORD PTR $T2[rsp]
call  vector_cross_product
mov   rdx, rbx
lea   rcx, QWORD PTR $T1[rsp]
movsd xmm0, QWORD PTR [rax]
movsd QWORD PTR $T1[rsp], xmm0
mov   eax, DWORD PTR [rax+8]
mov   DWORD PTR $T1[rsp+8], eax
call  vector_dot_product
add   rsp, 64
pop   rbx
ret   0
```

```
__declspec(safebuffers)
```

RAW

armv8-a clang 18.1.0

```
stp   x29, x30, [sp, #-48]!
str   x21, [sp, #16]
stp   x20, x19, [sp, #32]
mov   x29, sp
ldr   w21, [x29, #48]
mov   w19, w7
mov   w20, w6
bl    cross_product
lsr   x8, x0, #32
mov   w2, w1
mov   w3, w20
mov   w4, w19
mov   w1, w8
mov   w5, w21
ldp   x20, x19, [sp, #32]
ldr   x21, [sp, #16]
ldp   x29, x30, [sp], #48
b     dot_product
```

x86-64 gcc 14.2

```
push  r12
push  rbp
push  rbx
sub   rsp, 16
mov   ebx, DWORD PTR [rsp+48]
mov   ebp, DWORD PTR [rsp+56]
mov   r12d, DWORD PTR [rsp+64]
call  cross_product
add   rsp, 16
mov   r8d, ebp
mov   rcx, rax
mov   r9d, r12d
mov   edi, eax
shr   rcx, 32
mov   esi, ecx
mov   ecx, ebx
pop   rbx
pop   rbp
pop   r12
jmp   dot_product
```

x64 msvc v19.40 VS17.10

```
sub   rsp, 104
mov   eax, DWORD PTR z2$[rsp]
mov   DWORD PTR [rsp+48], eax
mov   eax, DWORD PTR y2$[rsp]
mov   DWORD PTR [rsp+40], eax
mov   DWORD PTR [rsp+32], r9d
mov   r9d, r8d
mov   r8d, edx
mov   edx, ecx
lea   rcx, QWORD PTR $T1[rsp]
call  cross_product
mov   r9d, DWORD PTR x3$[rsp]
movsd xmm0, QWORD PTR [rax]
mov   r8d, DWORD PTR [rax+8]
mov   eax, DWORD PTR z3$[rsp]
mov   DWORD PTR z2$[rsp], eax
mov   eax, DWORD PTR y3$[rsp]
movsd QWORD PTR v4$[rsp], xmm0
mov   rcx, QWORD PTR v4$[rsp]
mov   rdx, rcx
mov   DWORD PTR y2$[rsp], eax
shr   rdx, 32
add   rsp, 104
jmp   dot_product
```

RAW

armv8-a clang 18.1.0

```
stp   x29, x30, [sp, #-48]!
str   x21, [sp, #16]
stp   x20, x19, [sp, #32]
mov   x29, sp
ldr   w21, [x29, #48]
mov   w19, w7
mov   w20, w6
bl    cross_product
lsr   x8, x0, #32
mov   w2, w1
mov   w3, w20
mov   w4, w19
mov   w1, w8
mov   w5, w21
ldp   x20, x19, [sp, #32]
ldr   x21, [sp, #16]
ldp   x29, x30, [sp], #48
b     dot_product
```

x86-64 gcc 14.2

```
push  r12
push  rbp
push  rbx
sub   rsp, 16
mov   ebx, DWORD PTR [rsp+48]
mov   ebp, DWORD PTR [rsp+56]
mov   r12d, DWORD PTR [rsp+64]
call  cross_product
add   rsp, 16
mov   r8d, ebp
mov   rcx, rax
mov   r9d, r12d
mov   edi, eax
shr   rcx, 32
mov   esi, ecx
mov   ecx, ebx
pop   rbx
pop   rbp
pop   r12
jmp   dot_product
```

A lot of moving!

x64 msvc v19.40 VS17.10

```
sub   rsp, 104
mov   eax, DWORD PTR z2$[rsp]
mov   DWORD PTR [rsp+48], eax
mov   eax, DWORD PTR y2$[rsp]
mov   DWORD PTR [rsp+40], eax
mov   DWORD PTR [rsp+32], r9d
mov   r9d, r8d
mov   r8d, edx
mov   edx, ecx
lea   rcx, QWORD PTR $T1[rsp]
call  cross_product
mov   r9d, DWORD PTR x3$[rsp]
movsd xmm0, QWORD PTR [rax]
mov   r8d, DWORD PTR [rax+8]
mov   eax, DWORD PTR z3$[rsp]
mov   DWORD PTR z2$[rsp], eax
mov   eax, DWORD PTR y3$[rsp]
movsd QWORD PTR v4$[rsp], xmm0
mov   rcx, QWORD PTR v4$[rsp]
mov   rdx, rcx
mov   DWORD PTR y2$[rsp], eax
shr   rdx, 32
add   rsp, 104
jmp   dot_product
```

RAW

armv8-a clang 18.1.0

```
stp   x29, x30, [sp, #-48]!
str   x21, [sp, #16]
stp   x20, x19, [sp, #32]
mov   x29, sp
ldr   w21, [x29, #48]
mov   w19, w7
mov   w20, w6
bl    cross_product
lsr   x8, x0, #32
mov   w2, w1
mov   w3, w20
mov   w4, w19
mov   w1, w8
mov   w5, w21
ldp   x20, x19, [sp, #32]
ldr   x21, [sp, #16]
ldp   x29, x30, [sp], #48
b     dot_product
```

x86-64 gcc 14.2

```
push  r12
push  rbp
push  rbx
sub   rsp, 16
mov   ebx, DWORD PTR [rsp+48]
mov   ebp, DWORD PTR [rsp+56]
mov   r12d, DWORD PTR [rsp+64]
call  cross_product
add   rsp, 16
mov   r8d, ebp
mov   rcx, rax
mov   r9d, r12d
mov   edi, eax
shr   rcx, 32
mov   esi, ecx
mov   ecx, ebx
pop   rbx
pop   rbp
pop   r12
jmp   dot_product
```

x64 msvc v19.40 VS17.10

```
sub   rsp, 104
mov   eax, DWORD PTR z2$[rsp]
mov   DWORD PTR [rsp+48], eax
mov   eax, DWORD PTR y2$[rsp]
mov   DWORD PTR [rsp+40], eax
mov   DWORD PTR [rsp+32], r9d
mov   r9d, r8d
mov   r8d, edx
mov   edx, ecx
lea   rcx, QWORD PTR $T1[rsp]
call  cross_product
mov   r9d, DWORD PTR x3$[rsp]
movsd xmm0, QWORD PTR [rax]
mov   r8d, DWORD PTR [rax+8]
mov   eax, DWORD PTR z3$[rsp]
mov   DWORD PTR z2$[rsp], eax
mov   eax, DWORD PTR y3$[rsp]
movsd QWORD PTR v4$[rsp], xmm0
mov   rcx, QWORD PTR v4$[rsp]
mov   rdx, rcx
mov   DWORD PTR y2$[rsp], eax
shr   rdx, 32
add   rsp, 104
jmp   dot_product
```

Stack pointer is heavily used

armv8-a System V	x86-64 System V	x86-64 Microsoft
The first eight registers, **`r0`**-**`r7`**, are used to pass argument values into a subroutine and to return result values from a function.	If the class is INTEGER, the next available register of the sequence **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%rcx`**, **`%r8`** and **`%r9`** is used.	By default, the x64 calling convention passes the first four arguments to a function in registers.

armv7-a System V	x86 System V	x86 Microsoft
The first four registers **`r0`**-**`r3`** (**`a1`**-**`a4`**) are used to pass argument values into a subroutine and to return a result value from a function.	Most parameters are passed on the stack. - The first three parameters of type **`__m64`** are passed in **`%mm0`**, **`%mm1`**, and **`%mm2`**…	Parameters are pushed onto the stack from right to left. **`__fastcall`**: The first two DWORD or smaller arguments that are found in the argument list from left to right are passed in **`ECX`** and **`EDX`** registers.

General purpose registers allocation for function parameters and return values

Architecture	ABI	Composite types returned in registers	Composite types passed in registers	Number of registers for parameters + return
armv8-a	System V	≤ 16 bytes	≤ 16 bytes	8 total
armv7-a	System V	≤ 4 bytes	≤ 16 bytes	4 total
x86-64	System V	≤ 16 bytes	≤ 16 bytes	6 + 2
x86	System V	fundamental only	SIMD only	0 + 2
x86-64	Microsoft	1,2,4,8 bytes, C++03 POD	1,2,4,8 bytes	4 + 1
x86	Microsoft	1,2,4,8 bytes, C++03 POD	not even fundamental	0 + 2
x86 __fastcall	Microsoft	1,2,4,8 bytes, C++03 POD	fundamental only	2 + 2

Composite types are required to be “trivial” to get into registers!

C++ Core Guidelines

I.23: Keep the number of function arguments low

Reason Having many arguments opens opportunities for confusion. Passing lots of arguments is often costly compared to alternatives.

Approved*

Conclusions

- Compilers do unexpected things to your code, because they have to follow all the specifications
- Compiler Explorer is you friend
  https://godbolt.org/
- C++ Core Guidelines are pretty reasonable from performance point of view

Most important guidelines to avoid function call overhead

- Return by value
- Pass “trivial” types by value, others by reference
- Follow the Rule of 0 (or at least support trivial copy)
- Make APIs consistent
- Understand abstractions cost on your target platform

Thank you for attention!