Cpp North 2025

# Closing Keynote

## Misusing reinterpret_cast ?! You Probably Are :)

Alex Dathskovsky

# Misusing reinterpret_cast

## You Probably Are ;)

# About Me:


**alex.datskovsky@speedata.io**

**www.linkedin.com/in/alexdathskovsky**

**www.cppnext.com**

**https://www.youtube.com/@cppnext-alexd**

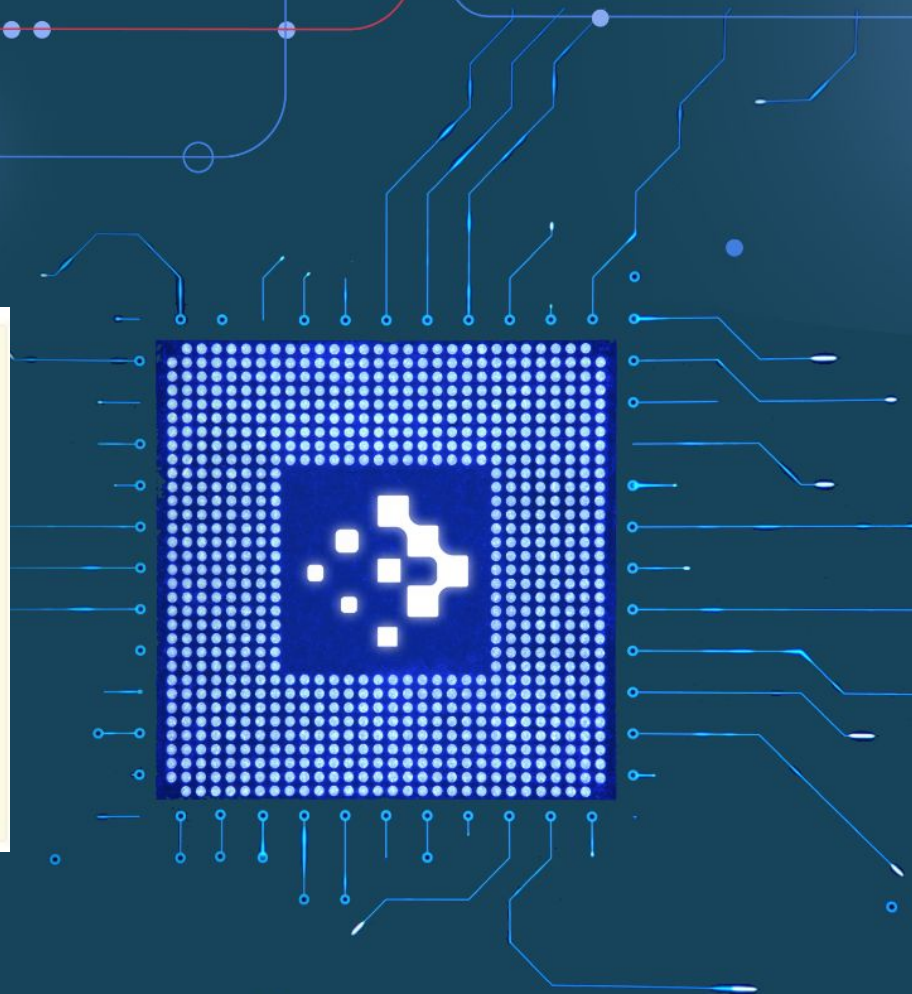
# About Me:

SPEEDATA

your keystrokes.

TWEET THIS!

I have only 331,775,982 keystrokes left before I die.



# Story Time

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Story Time: It All Started When

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Story Time: It Progressed to


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

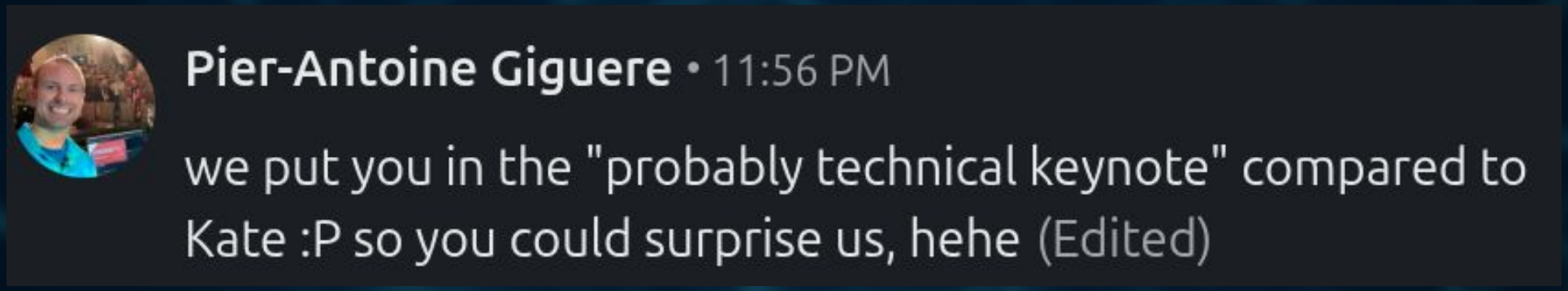
# Story Time: Bad Photos

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

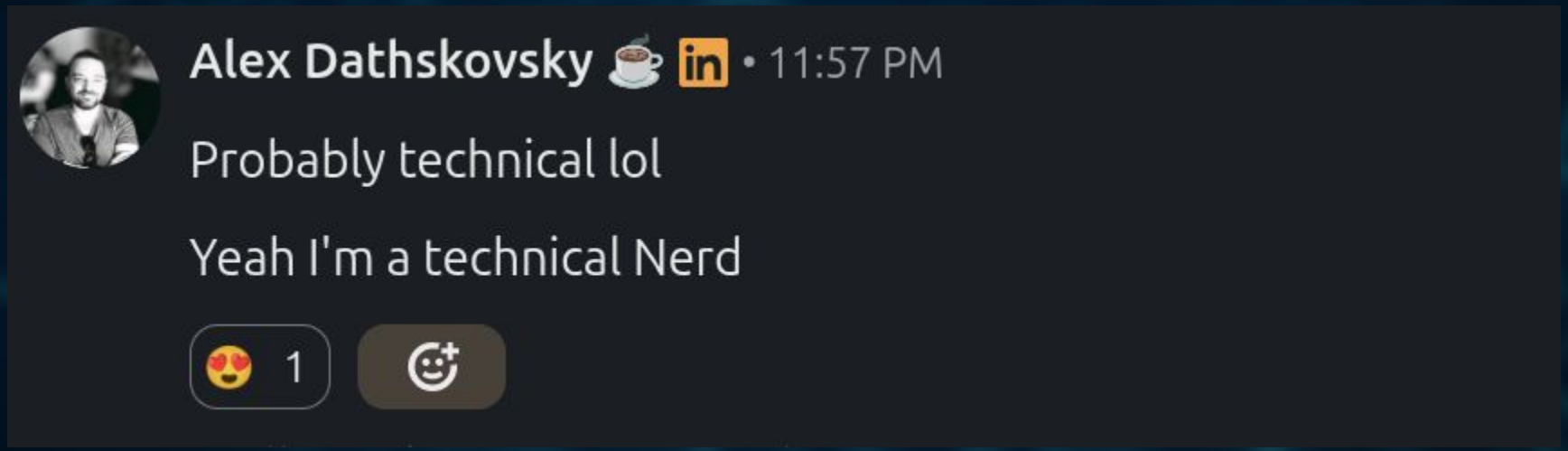
Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Story Time: I Just Asked



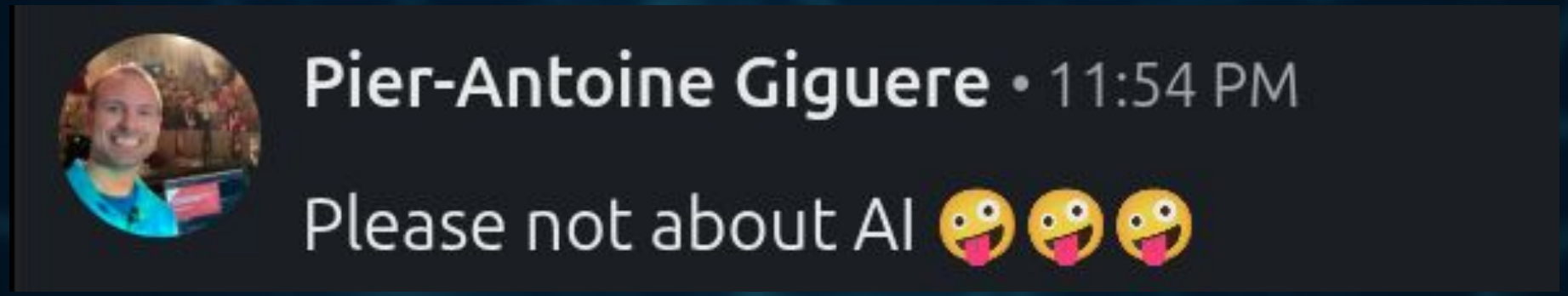

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Story Time: I Just Asked




Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Story Time: Now I Have To Speak About AI

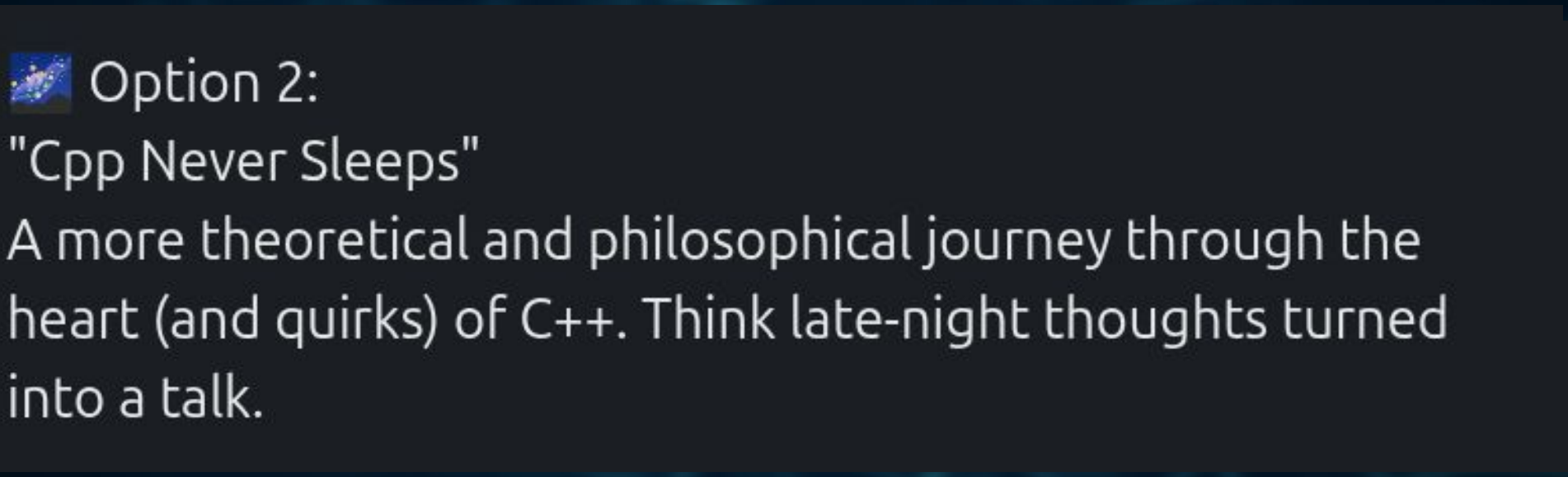



Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Story Time: Survey

🎯 Option 1:
"Misusing reinterpret_cast, probably yes"
A deep technical dive with all the juicy bits — classic me. Think of it like my CppIndia keynote, but cranked up.

# Story Time: Survey



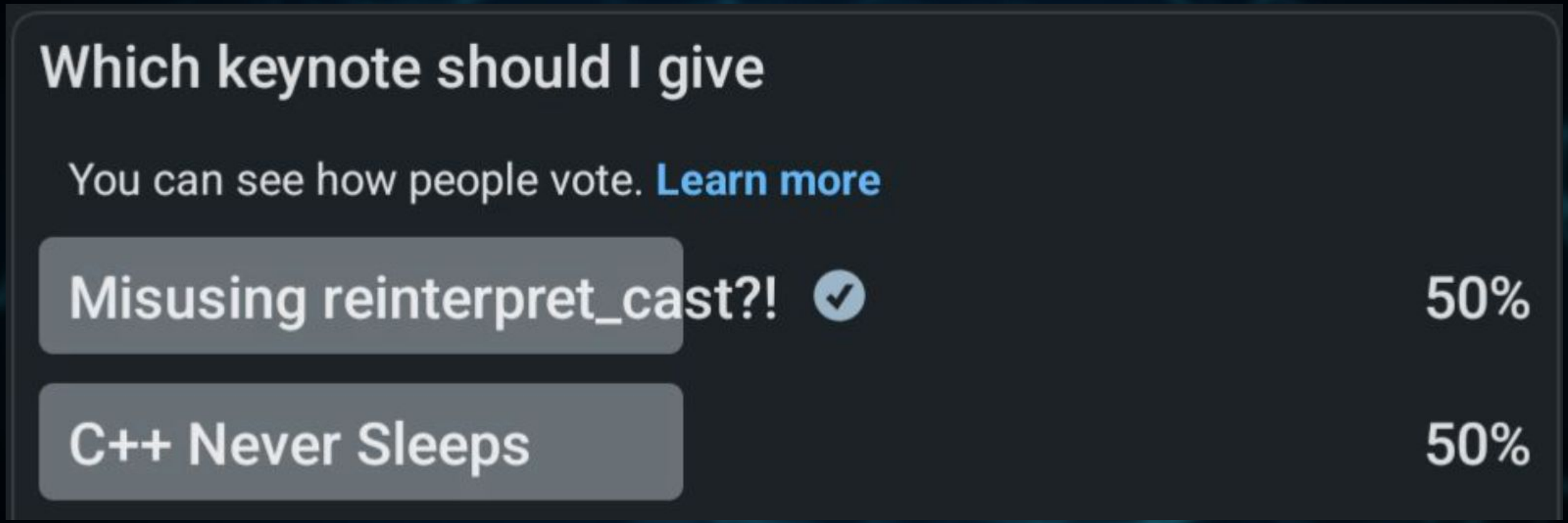

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Story Time: Survey




Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Story Time: A Warning From a Friend

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Let's Start ~~Misusing~~ Using Correctly

# Definitions

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Different Casts in C++

| Cast Type | Purpose | Safety Level | Runtime Check | Typical Use |
|---|---|---|---|---|
| static_cast | Compile-time conversion | ✅ Safe | ❌ No | Numeric, up/downcast |
| reinterpret_cast | Bit-level type punning | ❌ Dangerous | ❌ No | Low-level hacks |
| dynamic_cast | Checked polymorphic downcast | ✅ Safe | ✅ Yes | RTTI-based hierarchies |
| const_cast | Remove const/volatile | ⚠️ Dangerous | ❌ No | Interop, legacy API |

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# What is reinterpret_cast

- Translates between types
- Unlike static_cast usually compiles to nothing, unless casting to/from integral type
- Tells the compiler to treat a type as a different type

# What is reinterpret_cast

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# What is reinterpret_cast

## Explanation

Unlike `static_cast`, but like `const_cast`, the `reinterpret_cast` expression does not compile to any CPU instructions (except when converting between integers and pointers, or between pointers on obscure architectures where pointer representation depends on its type). It is primarily a compile-time directive which instructs the compiler to treat *expression* as if it had the type *target-type*.

Only the following conversions can be done with `reinterpret_cast`, except when such conversions would cast away constness (or volatility).

1) An expression of integral, enumeration, pointer, or pointer-to-member type can be converted to its own type. The resulting value is the same as the value of *expression*.

2) A pointer can be converted to any integral type large enough to hold all values of its type (e.g. to `std::uintptr_t`).

3) A value of any integral or enumeration type can be converted to a pointer type. A pointer converted to an integer of sufficient size and back to the same pointer type is guaranteed to have its original value, otherwise the resulting pointer cannot be dereferenced safely (the round-trip conversion in the opposite direction is not guaranteed; the same pointer may have multiple integer representations) The null pointer constant `NULL` or integer zero is not guaranteed to yield the null pointer value of the target type; `static_cast` or implicit conversion should be used for this purpose.

4) Any value of type `std::nullptr_t`, including `nullptr` can be converted to any integral type as if it were `(void*)0`, but no value, not even `nullptr` can be converted to `std::nullptr_t`: `static_cast` should be used for that purpose. (since C++11)

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# What is reinterpret_cast

5) Any object pointer type T1* can be converted to another object pointer type *cv* T2*. This is exactly equivalent to `static_cast<cv T2*>(static_cast<cv void*>(expression))` (which implies that if T2's alignment requirement is not stricter than T1's, the value of the pointer does not change and conversion of the resulting pointer back to its original type yields the original value). In any case, the resulting pointer may only be dereferenced safely if the dereferenced value is type-accessible.

6) An lvalue(until C++11) glvalue(since C++11) expression of type T1 can be converted to reference to another type T2. The result is that of `*reinterpret_cast<T2*>(p)`, where `p` is a pointer of type "pointer to T1" to the object or function designated by *expression*. No temporary is materialized or(since C++17) created, no copy is made, no constructors or conversion functions are called. The resulting reference can only be accessed safely if it is type-accessible.

7) Any pointer to function can be converted to a pointer to a different function type. The result is unspecified, but converting such pointer back to pointer to the original function type yields the pointer to the original function. The resulting pointer can only be called safely if it function type is call-compatible with the original function type.

8) On some implementations (in particular, on any POSIX compatible system as required by `dlsym`), a function pointer can be converted to `void*` or any other object pointer, or vice versa. If the implementation supports conversion in both directions, conversion to the original type yields the original value, otherwise the resulting pointer cannot be dereferenced or called safely.

9) The null pointer value of any pointer type can be converted to any other pointer type, resulting in the null pointer value of that type. Note that the null pointer constant `nullptr` or any other value of type `std::nullptr_t` cannot be converted to a pointer with `reinterpret_cast`: implicit conversion or `static_cast` should be used for this purpose.

10) A pointer to member function can be converted to pointer to a different member function of a different type. Conversion back to the original type yields the original value, otherwise the resulting pointer cannot be used safely.

11) A pointer to member object of some class T1 can be converted to a pointer to another member object of another class T2. If T2's alignment is not stricter than T1's, conversion back to the original type T1 yields the original value, otherwise the resulting pointer cannot be used safely.

As with all cast expressions, the result is:

- an lvalue if *target-type* is an lvalue reference type or an rvalue reference to function type(since C++11);
- an xvalue if *target-type* is an rvalue reference to object type; (since C++11)
- a prvalue otherwise.

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# What is reinterpret_cast

## Type aliasing

### Type accessibility

If a type `T_ref` is similar to any of the following types, an object of dynamic type `T_obj` is *type-accessible* through a lvalue(until C++11) glvalue(since C++11) of type `T_ref`:

- `char`, `unsigned char` or `std::byte`(since C++17): this permits examination of the object representation of any object as an array of bytes.
- `T_obj`
- the signed or unsigned type corresponding to `T_obj`

If a program attempts to read or modify the stored value of an object through a lvalue(until C++11) glvalue(since C++11) through which it is not type-accessible, the behavior is undefined.

This rule enables type-based alias analysis, in which a compiler assumes that the value read through a glvalue of one type is not modified by a write to a glvalue of a different type (subject to the exceptions noted above).

Note that many C++ compilers relax this rule, as a non-standard language extension, to allow wrong-type access through the inactive member of a union (such access is not undefined in C).

### Call compatibility

If any of the following conditions is satisfied, a type `T_call` is *call-compatible* with a function type `T_func`:

- `T_call` is the same type as `T_func`.
- `T_func*` can be converted to `T_call*` via a function pointer conversion. (since C++17)

If a function is called through an expression whose function type is not call-compatible with the type of the called function's definition, the behavior is undefined.

## Notes

Assuming that alignment requirements are met, a `reinterpret_cast` does not change the value of a pointer outside of a few limited cases dealing with *pointer-interconvertible* objects:

# What is reinterpret_cast

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# What is reinterpret_cast

- In simple words
  - Types must be layout-compatible and aliasable under C++'s strict aliasing rules
  - Converted types can be looked at as the “same” type


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Example

```
3    struct A{int a;};
4    struct B{int a; void foo() {};};
5    struct C: public A {};
6    struct D: public A, B {};
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Example

```
10      B b;
11      auto* a = reinterpret_cast<A*>(&b);
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Example

```
10      B b;
11      auto* a = reinterpret_cast<A*>(&b);
12      a->a;
```


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Example

```
11      C c;
12      auto* a = reinterpret_cast<A*>(&c);
13      a->a;
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Upcasting Known Classes


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# reinterpret_cast vs dynamic_cast

```
auto* a1 =
reinterpret_cast<A*>(c);
mov qword ptr [rbp - 16], rax
```

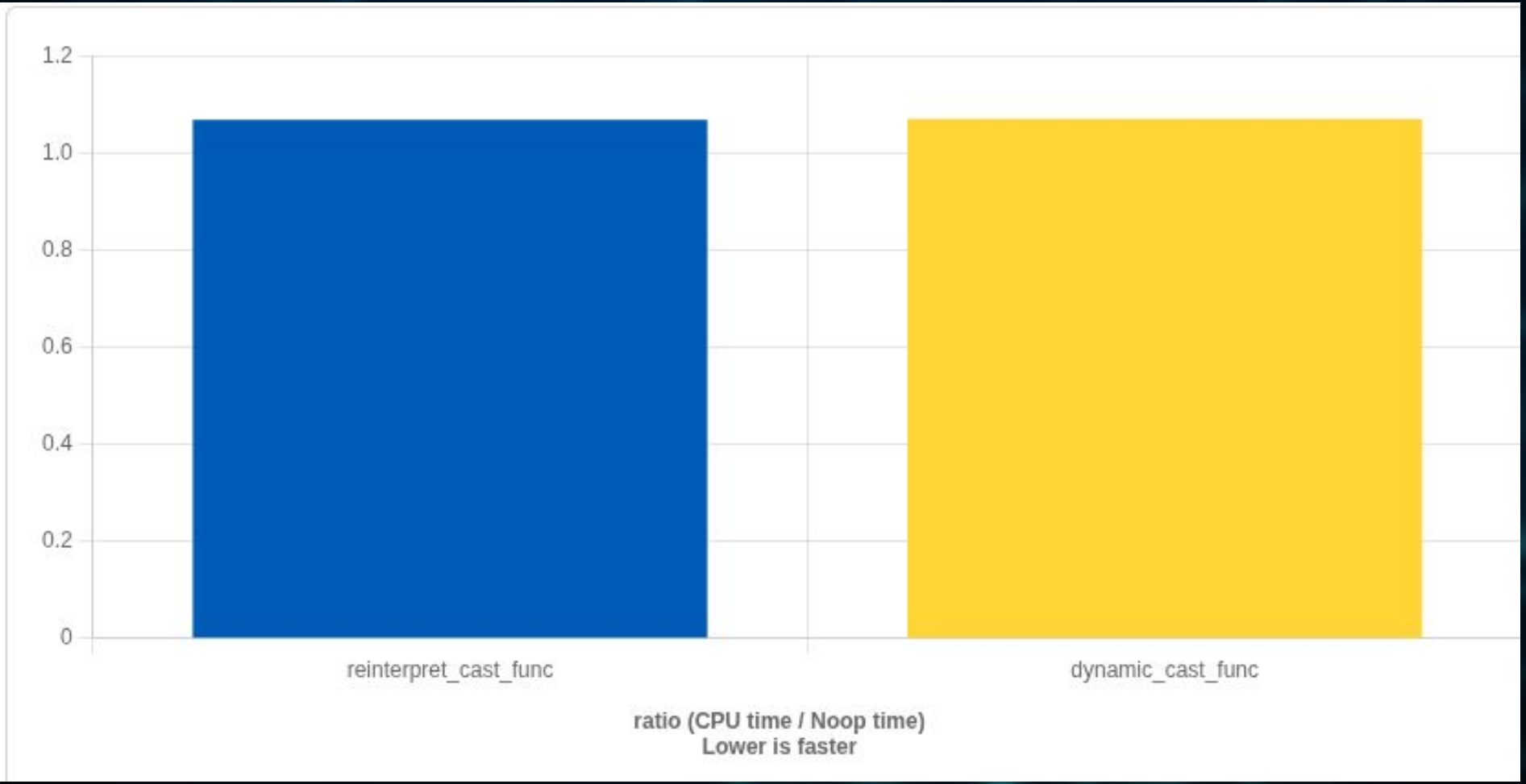
Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# reinterpret_cast vs dynamic_cast

```
auto* a2 =
dynamic_cast<A*>(c);
mov qword ptr [rbp - 24], rax
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# reinterpret_cast vs dynamic_cast




Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# reinterpret_cast vs dynamic_cast

```
auto* a1 =
reinterpret_cast<A*>(c);
```

```
lw a1, -12(s0)
sw a1, -16(s0)
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# reinterpret_cast vs dynamic_cast

```
auto* a2 =
dynamic_cast<A*>(c);
```

```
lw a1, -12(s0)
sw a1, -20(s0)
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Prefer Using dynamic_cast

```
11      auto* b = new B{0};
12
13      auto* a1 =
14      reinterpret_cast<A*>(b);
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Prefer Using dynamic_cast

```
16      auto* a2 =
17      dynamic_cast<A*>(b);
```

```
error: 'B' is not polymorphic x86-64 clang 20.1.0 #1
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Prefer Using dynamic_cast

```
11      void* b = new C{0};
12
13      auto* a1 =
14      reinterpret_cast<A*>(b);
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Prefer Using dynamic_cast

```
16      auto* a2 =
17      dynamic_cast<A*>(b);
```

```
error: 'void' is not a class type x86-64 clang 20.1.0 #1

error: cannot 'dynamic_cast' 'b' (of type 'void*') to
type 'struct A*' (source is not a pointer to class) x86-
64 gcc 15.1 #Executor 1
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Prefer Using C++ 😇

```
15      auto* a2 =
16      dynamic_cast<A*>(std::any_cast<C*>(b));
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Example - Reminder

```
3    struct A{int a;};
4    struct B{int a; void foo() {};};
5    struct C: public A {};
6    struct D: public A, B {};
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Example 2

```
13      auto* d = new D;
14
15      auto* b = reinterpret_cast<B*>(d);
```

# Example 2

```
warning: 'reinterpret_cast' from class 'D *' to its base
at non-zero offset 'B *' behaves differently from
'static_cast' [-Wreinterpret-base-class] x86-64 clang
20.1.0 #1
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Strict Aliasing Broken


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Strict Aliasing

```
11  int foo(int* i, float* f){
12      *i = 1;
13      *f = 0.1f;
14      return *i;
15  };
```

The compiler is allowed to assume 1 is returned

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Strict Aliasing

```
25      int i{0};
26      return foo(&i, reinterpret_cast<float*>(&i));
```

```
Program returned: 1
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Let’s Inspect the Assembly

```
1   init_vars(int*, float*):
2           mov     dword ptr [rdi], 1
3           mov     dword ptr [rsi], 1036831949
4           mov     eax, 1
5           ret
6
7   main:
8           mov     eax, 1
9           ret
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Speaking of ints to floats


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Most Common Mistake

```
28      int x{10};
29
30      auto f = *reinterpret_cast<float*>(&x);
31
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Byte Array

```
32      auto* c = reinterpret_cast<char*>(&x);
33      auto f  = *reinterpret_cast<float*>(c);
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

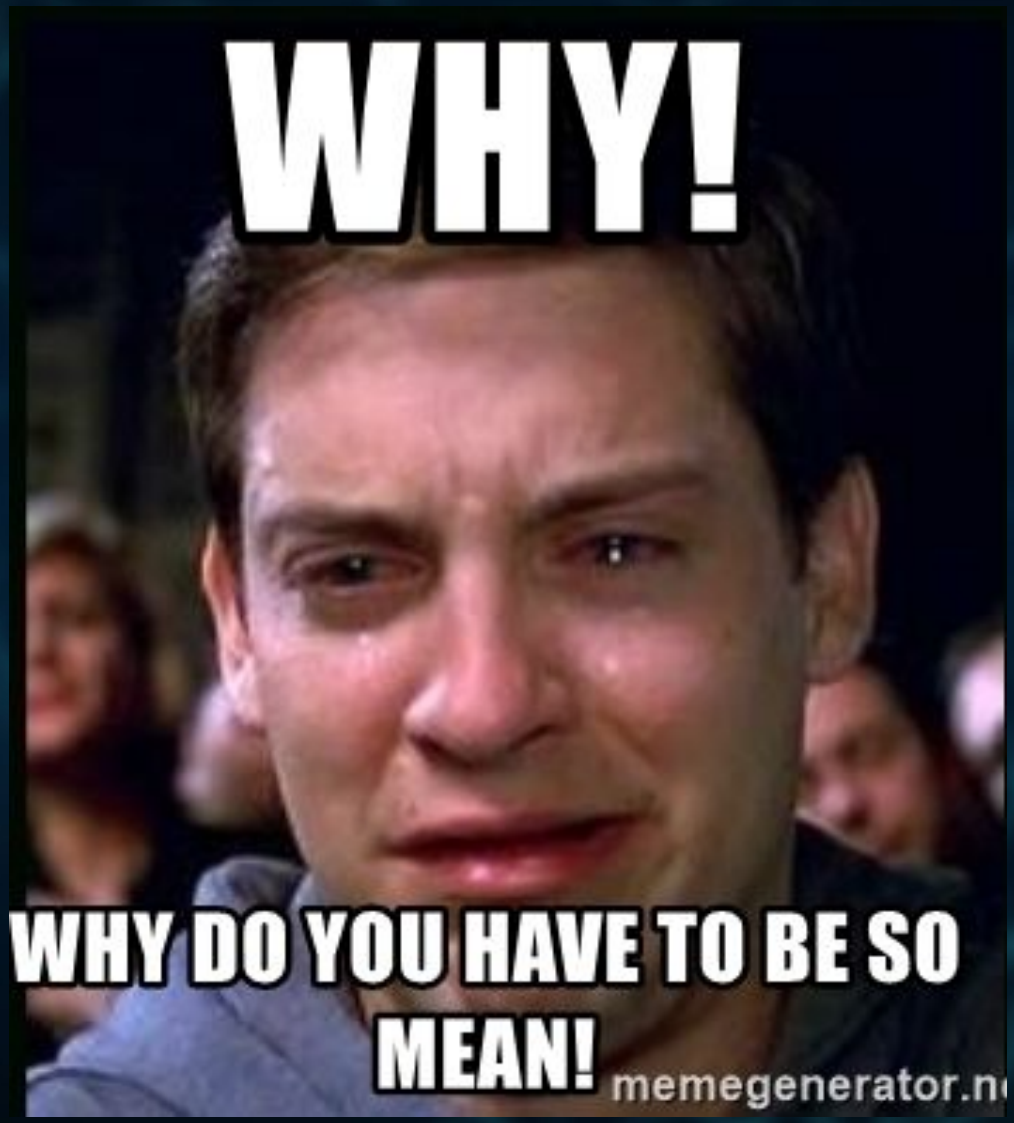
# Casting to Larger Size

```
43    auto* c = reinterpret_cast<char*>(&x);
44    auto  d = *reinterpret_cast<double*>(c);
45    fmt::print("x:{} d:{}\n", x, d);
```

```
x:10 d:5e-323
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Why oh Why?




Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# UB Again 💥

INT:

| byte 3 | byte 2 | byte 1 | byte 0 |
|---|---|---|---|
| 0x0 | 0x0 | 0x0 | 0xa |

INT EXT:

| byte 7 | byte 6 | byte 5 | byte 4 | byte 3 | byte 2 | byte1 | byte 0 |
|---|---|---|---|---|---|---|---|
| 0xc | 0xf | 0xf | 0xd | 0x0 | 0x0 | 0x0 | 0xa |

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# But the Actual Issue

INT:

| byte 3 | byte 2 | byte 1 | byte 0 |
|---|---|---|---|
| 0x0 | 0x0 | 0x0 | 0xa |

INT EXT:

| byte 7 | byte 6 | byte 5 | byte 4 | byte 3 | byte 2 | byte1 | byte 0 |
|---|---|---|---|---|---|---|---|
| 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0xa |

DBL:

| byte 7 | byte 6 | byte 5 | byte 4 | byte 3 | byte 2 | byte1 | byte 0 |
|---|---|---|---|---|---|---|---|
| 0x40 | 0x24 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 |

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Even Ints Can Misbehave

```cpp
auto z32 = 0xfeedbeef;

auto z64 = *reinterpret_cast<uint64_t*>(&z32);

fmt::print("z32: {:04x} z64: {:08x}\n", z32, z64);
```

```
z32: feedbeef z64: feedbeeff
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Even Ints Can Misbehave

```cpp
uint64_t z64 = 0xaabbccddeeff;

auto z32 = *reinterpret_cast<uint32_t*>(&z64);

fmt::print("z64: {:08x} z32: {:04x}", z64, z32);
```

```
z64: aabbccddeeff z32: ccddeeff
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# bit_cast

```
37      auto f = std::bit_cast<float>(x);
```

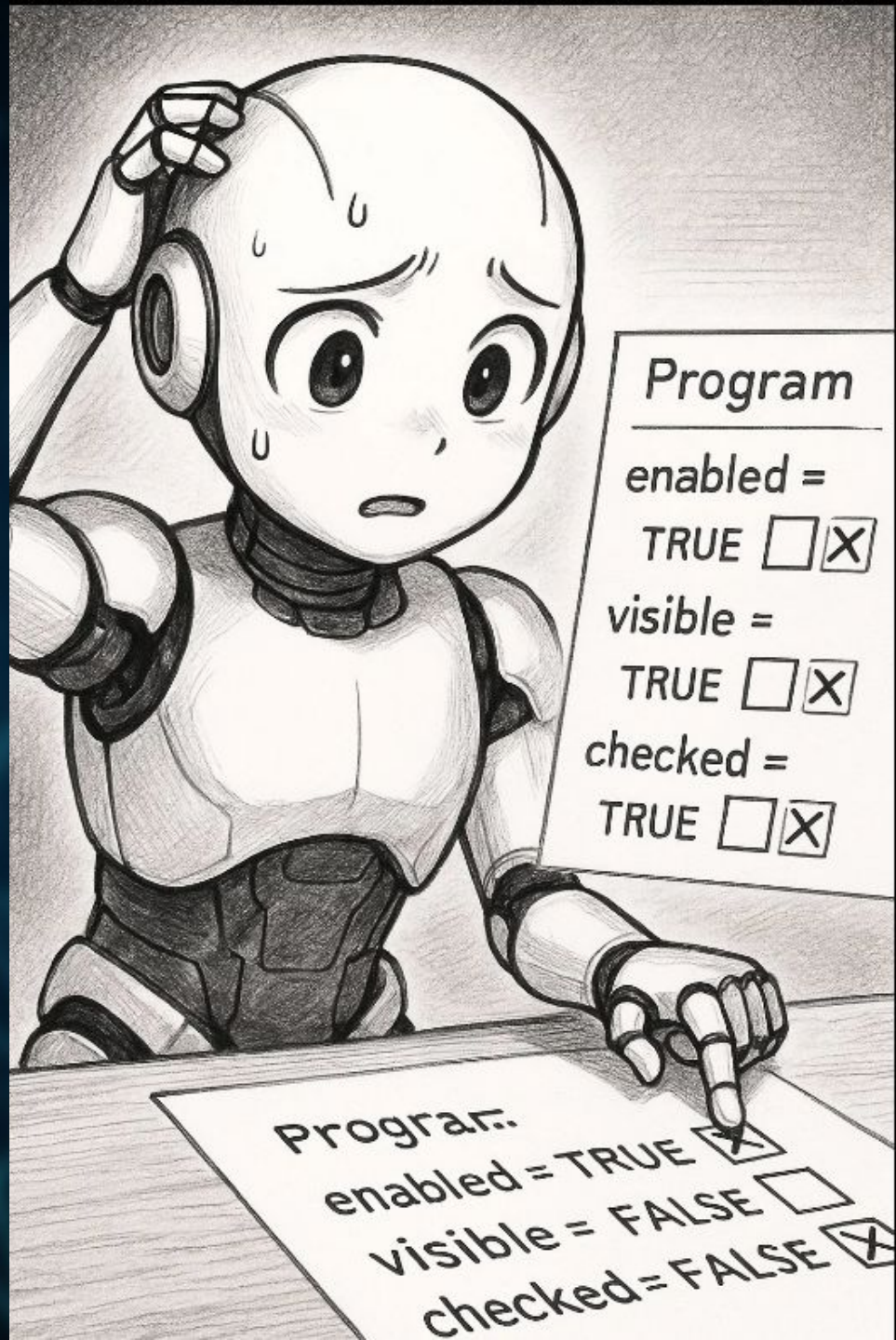
Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# bit_cast, the Catch

```
39      auto d = std::bit_cast<double>(x);
'sizeof(double) == sizeof(int)' (8 == 4) evaluated to false
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Fun With Booleans




Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Fun With Booleans : Clang

```
53    uint32_t x{0xf6};
54    auto b = *reinterpret_cast<bool*>(&x);
55    fmt::print("x={:b} b={}\n", x, b);
```

```
x=11110110 b=false
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

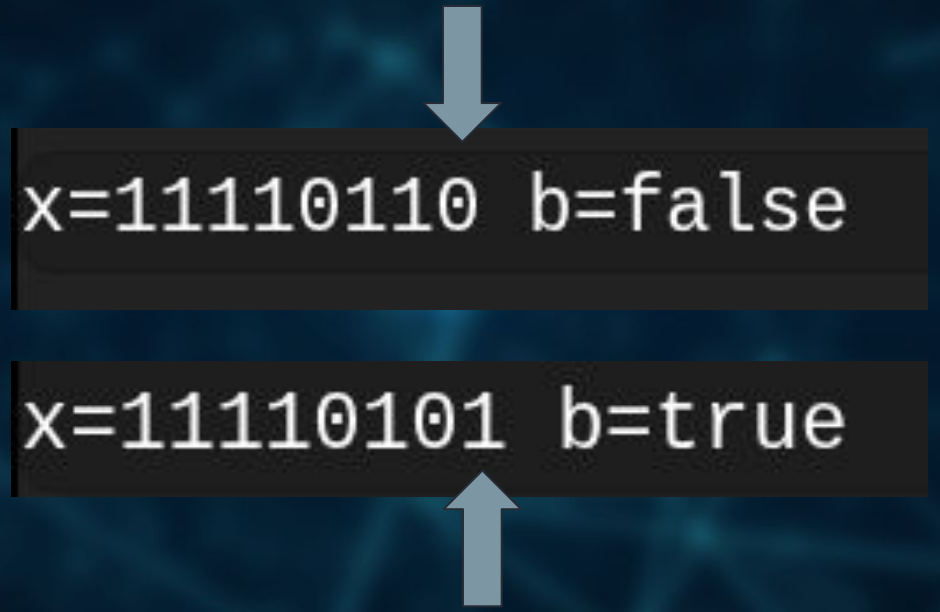
# Fun With Booleans : Clang

```
57    uint32_t x1{0xf5};
58    auto b1 = *reinterpret_cast<bool*>(&x1);
59    fmt::print("x={:b} b={}\n", x1, b1);
```

```
x=11110101 b=true
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Fun With Booleans : Clang




Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Fun With Booleans : GCC

```
x=11110110 b=true
x=11110101 b=true
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Fun With Booleans : GCC

```
uint32_t tmp = 0xffffff00;
fmt::print("{:04x}, bool:{}", tmp,
*reinterpret_cast<bool*>(&tmp));
```

```
ffffff00, bool:false
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Unaligned Memory Access


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# What is an Unaligned Memory Access

- Reading N bytes from an address that is not aligned to N , a % N != 0
  - Performance cost
  - May raise processor exception

# Unaligned Memory Access: Example

```
62      std::array<char, 4> a{'a', 'b', 'c', 'd'};
63
64      auto* ptr1 = &a[1];
65
66      auto* ptr2 = reinterpret_cast<uint16_t*>(&a[1]);
67
68      fmt::print("ptr1->{} ptr2->{}\n", *ptr1, *ptr2);
```

This is Fine

unaligned access

```
ptr1->b ptr2->25442
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Unaligned Memory Access: Sanitizers are Friends

- Running with `-fsanitize=undefined`

```
runtime error: reference binding to misaligned address
0x7ffe54d0fb6d for type 'uint16_t' (aka 'unsigned short'),
which requires 2 byte alignment
0x7ffe54d0fb6d: note: pointer points here
00 00 00 61 62 63 64  00 00 00 00 00 00 00 00  00 00 00 00
00 00 00 00  00 00 00 00 00 00 00 00  00
```

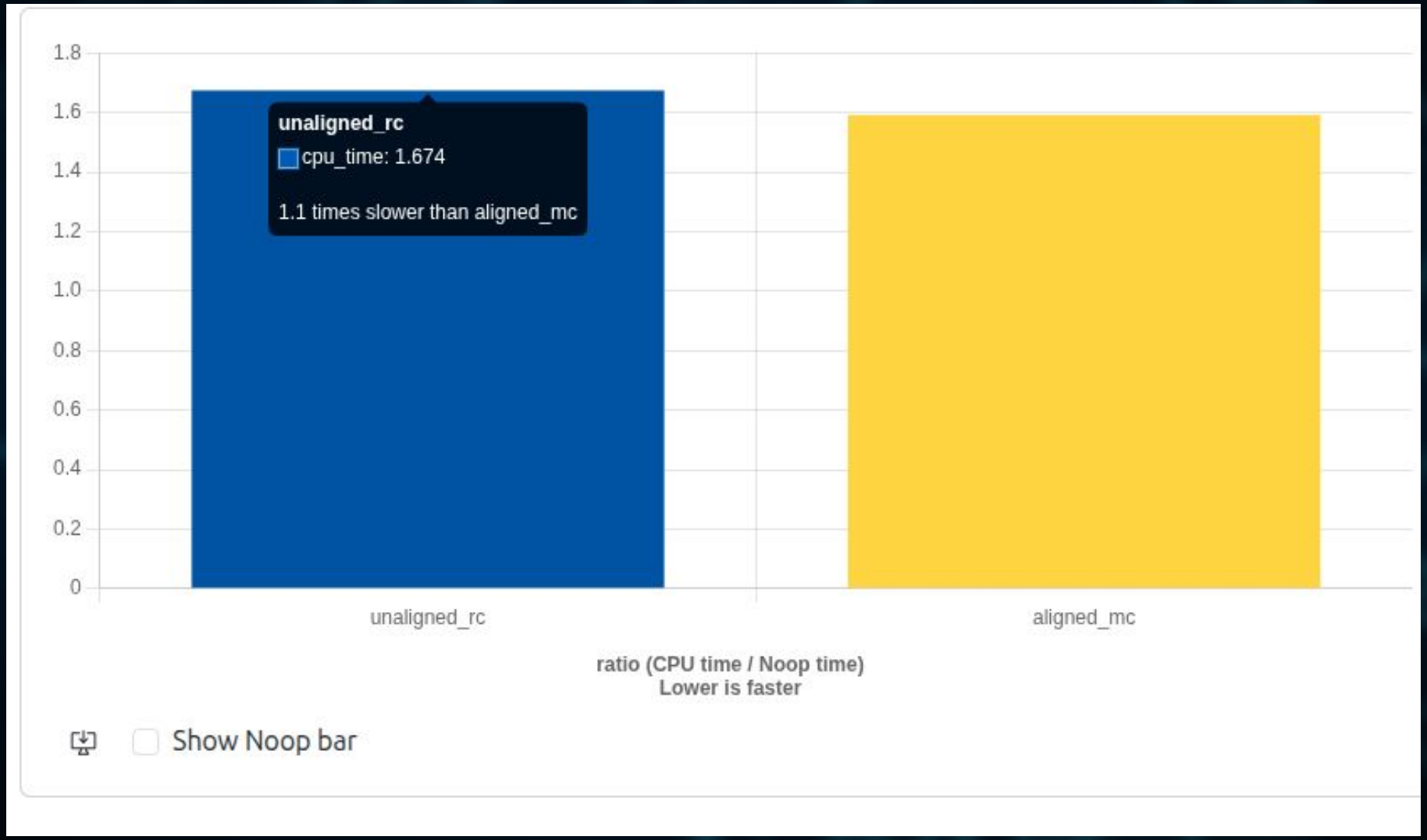
Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Unaligned Memory Access: So What Should We Do

```
70      uint16_t u16{};
71      std::memcpy(&u16, &a[1], 2);
72      fmt::print("ptr1->{} u16{}=\n", *ptr1, u16);
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Unaligned Memory Access: So What Should We Do



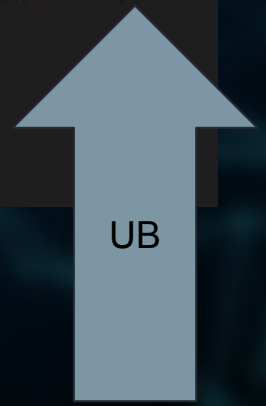

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Creating Objects from Bytes

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Bytes to Objects

```
4   struct X {int y;};
5
6   int main(){
7       alignas(X) std::byte buff[sizeof(X)];
8       auto* p = new(&buff) X{10};
9       int y = reinterpret_cast<X*>(&buff)->y;
10      return y;
11  }
```




Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Bytes to Objects

```
 5   struct X {int y;};
 6
 7   int main(){
 8       alignas(X) std::byte buff[sizeof(X)];
 9       auto* p = new(&buff) X{10};
10       int y = std::launder(reinterpret_cast<X*>(&buff))->y;
11       return y;
12   }
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Bytes to Objects

## std::launder

Defined in header <new>

```
template< class T >
constexpr T* launder( T* p ) noexcept;
```

(since C++17)

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# More Surprises With Basic Usage

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# More Surprises: Example

```
 6   struct X {
 7       std::array<int, 5> x;
 8       std::array<float, 5> y;
 9   };
10
11   void process_network(std::byte* buff){
12       decltype(auto) d = std::launder(reinterpret_cast<X*>(buff));
13       d->x[1] = 10;
14       d->y[2] = 1.1f;
15   }
```

# More Surprises: Example


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# More Surprises: Example

```
 6   struct X {
 7       std::array<int, 5> x;
 8       std::array<float, 5> y;
 9   };
10
11   void process_network(std::byte* buff){
12       decltype(auto) d = std::launder(reinterpret_cast<X*>(buff));
13       d->x[1] = 10;
14       d->y[2] = 1.1f;
15   }
```

# More Surprises: Lets Fix It

```
18  void safe_network_process(std::byte* buff){
19      X d;
20      std::memcpy(&d, buff, sizeof(X));
21      d.x[1] = 10;
22      d.y[2] = 1.1f;
23      std::memcpy(buff, &d, sizeof(X));
24  }
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# More Surprises: What About Performance

```
1   process_network(std::byte*):
2           mov     dword ptr [rdi + 4], 10
3           mov     dword ptr [rdi + 28], 1066192077
4           ret
5
6   safe_network_process(std::byte*):
7           mov     dword ptr [rdi + 4], 10
8           mov     dword ptr [rdi + 28], 1066192077
9           ret
```

# More Surprises: Sometimes Compilers May Miss

```
26  void safe_network_process_better(std::byte* buff){
27      decltype(auto) d = std::launder(
28          reinterpret_cast<X*>(
29              std::memmove(buff, buff, sizeof(X))));
30      d->x[1] = 10;
31      d->y[2] = 1.1f;
32  }
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# More Surprises: Sometimes Compilers May Miss

```
1   process_network(std::byte*):
2           mov     dword ptr [rdi + 4], 10
3           mov     dword ptr [rdi + 28], 1066192077
4           ret
5
6   safe_network_process(std::byte*):
7           mov     dword ptr [rdi + 4], 10
8           mov     dword ptr [rdi + 28], 1066192077
9           ret
10
11  safe_network_process_better(std::byte*):
12          mov     dword ptr [rdi + 4], 10
13          mov     dword ptr [rdi + 28], 1066192077
14          ret
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# More Surprises: Better C++23 solution

```
35  void safe_network_process_better2(std::byte* buff){
36      decltype(auto) d = std::start_lifetime_as<X>(buff);
37      d->x[1] = 10;
38      d->y[2] = 1.1f;
39  }
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# More Surprises: Better C++23 solution

| C++23 feature | Paper(s) | GCC libstdc++ | Clang libc++ | MSVC STL | Apple Clang* |
|---|---|---|---|---|---|
| Explicit lifetime management (`std::start_lifetime_as`) (FTM)* | P2590R2<br>P2679R2 | | | | |

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

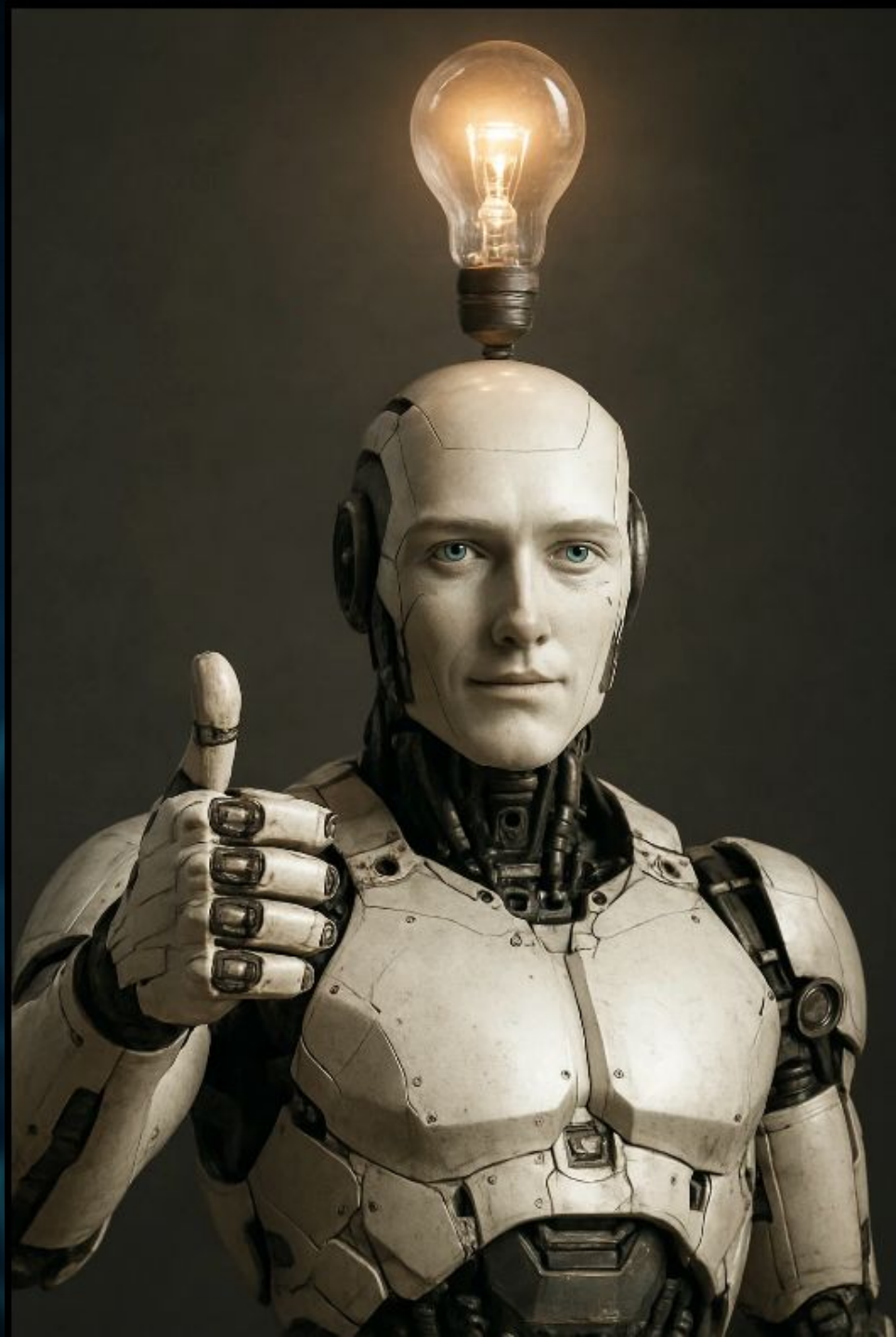
# The Moral Of The Story

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# The Moral Of The Story

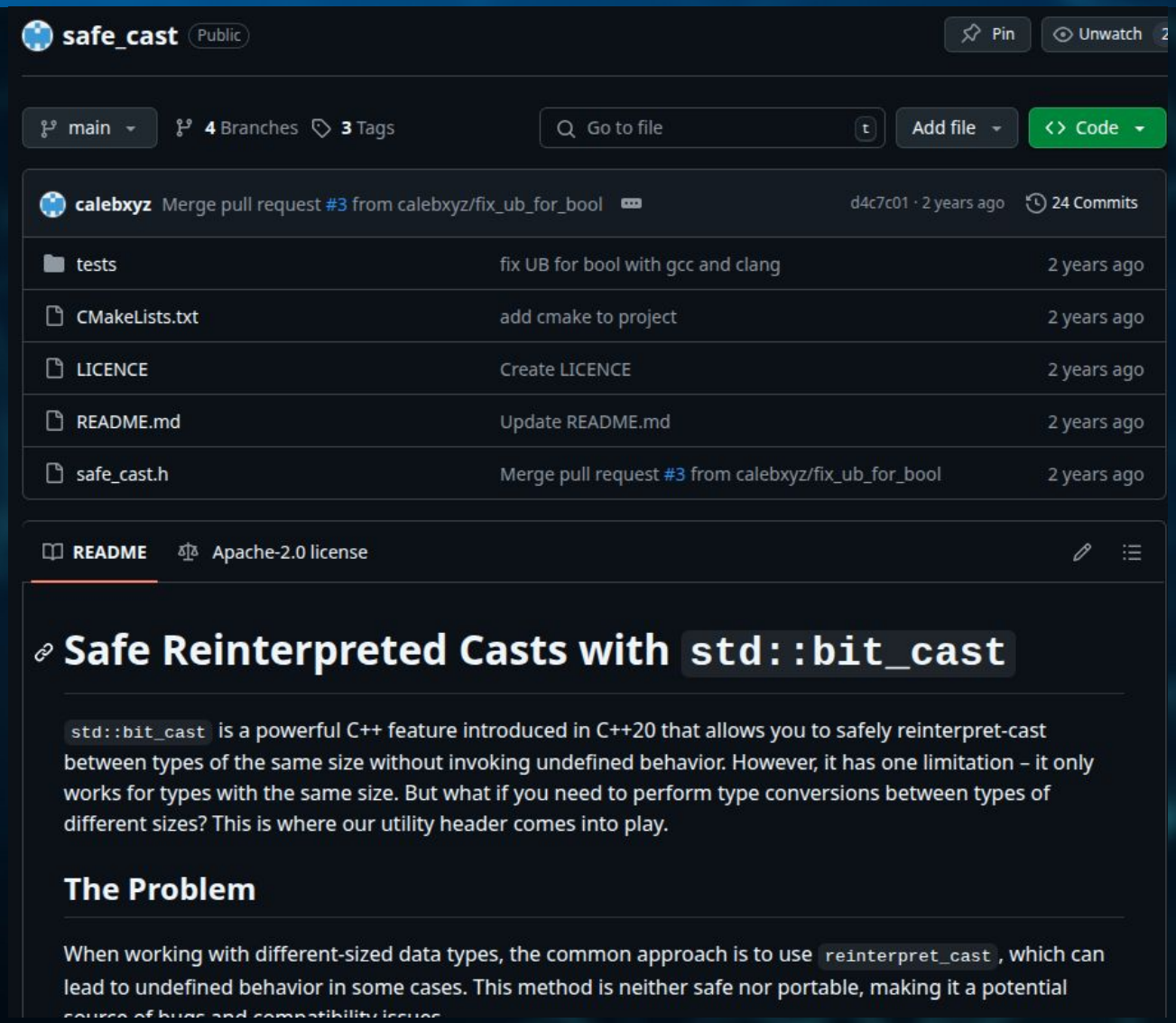
- With great tools comes great responsibility
  - Understand what the tool does
  - Know the quirks and pitfalls
  - Hide it in libraries if possible to protect users
  - Sanitizers are our friends
    - -fsanitize=undefined
  - UB is bad we should avoid it

# I’m Here To Help



Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib




Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Example

```
uint64_t to_uint64(float f){
    return *std::launder(reinterpret_cast<uint64_t*>(
        std::launder(reinterpret_cast<std::byte*>(&f))));
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Example

```
template <typename TO>
TO to_type(float f){
    return *std::launder(reinterpret_cast<TO*>(
        std::launder(reinterpret_cast<std::byte*>(&f))));
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Example

```
int main(){
    auto f = 256.2f;
    fmt::print("uint32_t value: {}\n",
    to_type<uint32_t>(f));
    fmt::print("uint64_t value: {}\n",
    to_type<uint64_t>(f));
    return 0;
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Example

```
uint32_t value: 1132468634
uint64_t value: 16164930548655987098
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Example

```
int main(){
    auto f = 256.2f;
    fmt::print("uint32_t value: {}\n",
    to_type<uint32_t>(f));
    fmt::print("uint32_t value byets 0x{:08X}\n",
    to_type<uint32_t>(f));
    fmt::print("uint64_t value: {}\n",
    to_type<uint64_t>(f));
    fmt::print("uint64_t value byets 0x{:016X}\n",
    to_type<uint64_t>(f));
    return 0;
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Example

```
uint32_t value: 1132468634
uint32_t value bytes 0x4380199A
uint64_t value: 12126180866728663450
uint64_t value bytes 0xA848D9404380199A
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Example

```
fmt::print("uint64_t value: {}\n",
static_cast<uint64_t>(f));
```

```
uint64_t value: 256
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Example

```cpp
template <typename TO>
TO to_type(float f){
    return *std::launder(reinterpret_cast<TO*>(
        std::launder(reinterpret_cast<std::byte*>(&f))));
}

uint64_t to_uint64(float f){
    return static_cast<uint64_t>(
        to_type<uint32_t>(f));
}
```

```
uint64_t value: 1132468634
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Example

```
fmt::print("uint64_t value: {}\n",
bits::bit_cast<uint64_t>(f));
```

```
uint64_t value: 1132468634
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Quick Peek Behind the Curtain

```
template <typename T1, typename T2>
concept larger_size = sizeof(T1) > sizeof(T2);
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Quick Peek Behind the Curtain

```
template <typename T>
concept integral_not_bool = std::integral<T> and (not std::same_as<T, bool>);
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Quick Peek Behind the Curtain

```
template <integral_not_bool To, std::floating_point From>
constexpr To bit_cast_size(From from)
  requires larger_size<From, To>
{
  if constexpr (std::same_as<From, float>){
    return std::bit_cast<To>(static_cast<std::make_unsigned_t<To>>(std::bit_cast<uint32_t>(from)));
  } else {
    return std::bit_cast<To>(static_cast<std::make_unsigned_t<To>>(std::bit_cast<uint64_t>(from)));
  }
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Quick Peek Behind the Curtain

```cpp
template <integral_not_bool To, integral_not_bool From>
constexpr To bit_cast_size(From from) requires (not equal_size<To, From>)
{
  //if both are integral all we want is to get the bits we need extend zeroext them or just cut them
  //we will never want to sign extend because that will change the bits
  return std::bit_cast<To>(static_cast<std::make_unsigned_t<To>>(static_cast<std::make_unsigned_t<From>>(from)));
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Turning to Byte Array

```
auto to_buffer(auto val){
    return std::bit_cast<std::array<
    std::byte, sizeof(decltype(val))>>(val);
}
```

```
auto buff = to_buffer(0xdeadbeef);
```

0xDEADBEEF

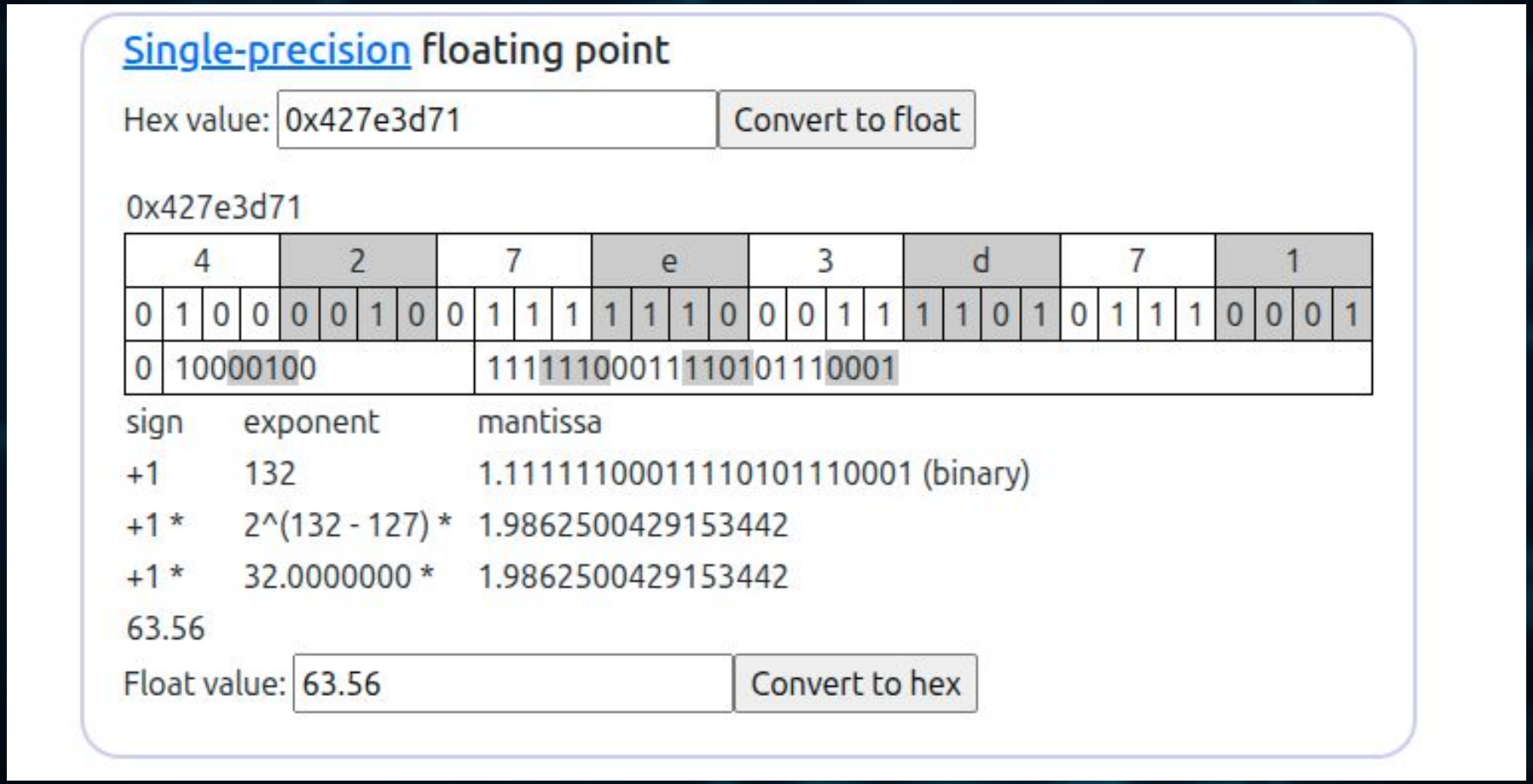
Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Receiving Bytes from Network

```
float from_network(std::byte* bytes){
    return bits::bit_cast<float, 0, std::byte, 4>(
        bytes);
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# Safe_Cast Lib: Receiving Bytes From Network



Single-precision floating point

Hex value: 0x427e3d71 Convert to float

0x427e3d71

| 4 | | | | 2 | | | | 7 | | | | e | | | | 3 | | | | d | | | | 7 | | | | 1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 10000100 | | | | | | | 11111100011110101110001 | | | | | | | | | | | | | | | | | | | | | | | |

| sign | exponent | mantissa |
|---|---|---|
| +1 | 132 | 1.11111100011110101110001 (binary) |
| +1 * | 2^(132 - 127) * | 1.9862500429153442 |
| +1 * | 32.0000000 * | 1.9862500429153442 |

63.56

Float value: 63.56 Convert to hex

# Safe_Cast Lib: Receiving Bytes from Network

```
std::byte fl[] = {
    std::byte(0x71), std::byte(0x3d),
    std::byte(0x7e), std::byte(0x42)};
fmt::print("{}\n", from_network(fl));
```

```
63.56
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

- ✅ Try safe_cast
- ✅ Use `std::bit_cast`
- ✅ Align memory
- ✅ Know when you break aliasing
- ✅ Use sanitizers
- ✅ Code should be warning free
- ❌ Don’t cast to unrelated types
- ❌ Don’t trust UB to “just work”


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# QUESTIONS

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

# THANK YOU FOR LISTENING

**ALEX DATSKOVSKY**

ALEX.DATSKOVSKY@SPEEDATA.IO

WWW.LINKEDIN.COM/IN/ALEXDATHSKOVSKY