

`std::launder`, `std::start_lifetime_as`, and UB in `reinterpret_cast`

OR...type punning done right

Javier López-Gómez

using `std::cpp`, 2025-03-19



Javier Lopez-Gomez

🐙 jalopezg-git

🌐 <https://jalopezg.dev/>

✉️ javier.lopez.gomez AT proton.me

Summary of the last 5+ years (compilers-wise)...

- 2017–2020: PhD in Computer Science and Technology (ARCOS-UC3M)
 - Prototype implementation of C++ contracts (clang)
 - Research internship at CERN in 2019: Definition shadowing in the Cling C++ interpreter
- 2020–2023: Senior Fellow (SFT, CERN)
 - More Cling – but also RNTuple and general contributions to the ROOT project
- **2024–(currently): Senior Compiler Engineer (Zimperium, Inc.)**
 - Binary-to-binary obfuscation solution

Let's assume this type definition for `MyType`.

```
struct MyType {  
    uint32_t i;  
    float f;  
};  
static_assert(sizeof(MyType) == 8 && alignof(MyType) == 4);
```

Is this well-defined C++ or UB?

```
unsigned char b[]{0x00, 0x11, 0x22, 0x33, 0xff, 0x00, 0xff, 0x00};  
auto *p = reinterpret_cast<MyType *>(&b);  
some_fn_taking_int(p->i);
```

Let's assume this type definition for `MyType`.

```
struct MyType {
```

UNDEFINED BEHAVIOR



due to

- Violates strict aliasing
- Storage may not be suitably aligned for `MyType`
- `p` accesses `MyType` object out of its lifetime

```
some_fn_taking_int(p->i);
```

Given the previous definition for `MyType`, again

Is this well-defined C++ or UB?

```
MyType foo{123, 42.0f};  
auto *p = reinterpret_cast<unsigned char *>(&foo);  
fn_doing_something_on_char_array(p, sizeof(MyType));
```

Given the following C++ code snippet:

(MOSTLY) UNDEFINED BEHAVIOR



Is the

- See P1839R7; a defect in ISO C++ standard?

Given the previous definition for **MyType**, again

Is this well-defined C++ or UB?

```
alignas(MyType) unsigned char b[]{0x00, 0x11, 0x22, 0x33, 0xff, 0x00,  
    0xff, 0x00};  
auto o = std::bit_cast<MyType>(b);  
some_fn_taking_int(p->i);
```

Given the previous definition for `MvType` again

Is th

WELL-DEFINED

- Note, however, that result may vary depending on arch endianness

- 1 Introduction
- 2 Types, layout, and lifetime
- 3 Strict Aliasing
- 4 Type Punning done right
- 5 Practical Case: (de-)serialization
- 6 Conclusion

Introduction

Ill-formed [defns.ill.formed] [defns.well.formed]

3.26

[defns.ill.formed]

ill-formed program

program that is not well-formed (3.68)

3.68

[defns.well.formed]

well-formed program

C++ program constructed according to the syntax and semantic rules

Ill-formed, no diagnostic required: program is ill-formed, but no compiler diagnostic is required, e.g. different definitions for an inlined function.

Undefined behavior (UB) [defns.undefined]

3.65

[defns.undefined]

undefined behavior

behavior for which this document imposes no requirements

[*Note 1 to entry:* Undefined behavior may be expected when this document omits any explicit definition of behavior or when a program uses an incorrect construct or invalid data. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a *diagnostic message* (3.18)), to terminating a translation or execution (with the issuance of a diagnostic message). Many incorrect program constructs do not engender undefined behavior; they are required to be diagnosed. Evaluation of a constant expression (7.7) never exhibits behavior explicitly specified as undefined in Clause 4 through Clause 15. — *end note*]

Definitions: Type punning

- From Wikipedia¹: *“In computer science, a type punning is any programming technique that subverts or circumvents the type system of a programming language in order to achieve an effect that would be difficult or impossible to achieve within the bounds of the formal language.”*
 - I.e., accessing **underlying in-memory representation** of an object of type `Foo` as a different type, `Bar`, e.g.



...and accessing as `unsigned char[]`

- Why?** Useful for (de-)serialization, networking code, or calling legacy (C) library code.

¹Source: https://en.wikipedia.org/wiki/Type_punning.

As we saw, some `reinterpret_cast`s result in UB...

Q: Why, C++, why? 😞

A: TL;DR²: Can two pointers of different types really point to the same object?

Compiler may optimize based on that!

²There are more causes; but this is a nice summary.

As we saw, some `reinterpret_cast`s result in UB...

Q: Why, C++, why? 😞

A: TL;DR²: Can two pointers of **different types** really point to the **same object**?

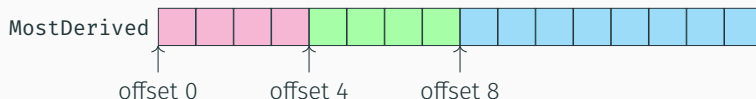
Compiler may optimize based on that!

²There are more causes; but this is a nice summary.

Types, layout, and lifetime

A quick reminder

- C++ has fundamental + compound types (arrays, pointers, classes...)
 - Class types have **base subobjects** and **member subobjects**



```
struct Base { int i; }  
struct Derived : Base { float f; }  
struct MostDerived : Derived { double d; }
```

- Types have size and **alignment**!

A quick reminder

- C++ has fundamental + compound types (arrays, pointers, classes...)

- Standard-layout [*class.prop*]: TL;DR
 - All non-static members defined in same class
 - All non-static members are also standard-layout
 - No **virtual**
- POD: standard-layout + trivial

- Types have size and alignment:

Storage Duration vs. Lifetime (1)

- *Storage Duration* \neq *Lifetime*!
- Storage Duration: `static`, `thread_local`, automatic, dynamic

Lifetime starts...

[intro.object], par. 1

6.7.2 Object model

[intro.object]

- ¹ The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is created by a definition (6.2), by a *new-expression* (7.6.2.8), by an operation that implicitly creates objects (see below), when implicitly changing the active member of a union (11.5), or when a temporary object is created (7.3.5, 6.7.7). An object occupies a region of storage in its period of construction (11.9.5), throughout its lifetime (6.7.4), and in its period of destruction (11.9.5).

Storage Duration vs. Lifetime (1)

- *Storage Duration* \neq *Lifetime*!
- Storage Duration: `static`, `thread_local`, automatic, dynamic

Lifetime ends...

[*basic.life*], par. 2.3–2.5

as described in 20.2.10.2. The lifetime of an object *o* of type *T* ends when:

- if *T* is a non-class type, the object is destroyed, or
- if *T* is a class type, the destructor call starts, or
- the storage which the object occupies is released, or is reused by an object that is not nested within *o* (6.7.2).

Storage Duration vs. Lifetime (2)

```
using T = std::vector<int>;

void *p = ::aligned_alloc(alignof(T), sizeof(T));
static_cast<T*>(p)->push_back(101); // UB!
```

```
auto v1 = new (p) T(); // -\
v1->push_back(42);      //   |- v1 vector lifetime
v1->~T();               //   _/
v1->push_back(102);     // UB!
```

```
auto v2 = new (p) T(); // -\
v2->push_back(123);     //   |- v2 vector lifetime
v2->~T();               //   _/
v2->push_back(103);     // UB!
```

```
::free(p);
```

Storage Duration vs. Lifetime (2)

```
using T = std::vector<int>;

void *p = ::aligned_alloc(alignof(T), sizeof(T));
static_cast<T*>(p)->push_back(101); // UB!
```

```
auto v1 = new (p) T(); // -\
v1->push_back(42);      //  |- v1 vector lifetime
v1->~T();               //  _/
v1->push_back(102);     // UB!
```

```
auto v2 = new (p) T(); // -\
v2->push_back(123);     //  |- v2 vector lifetime
v2->~T();               //  _/
v2->push_back(103);     // UB!
```

```
::free(p);
```

Storage Duration vs. Lifetime (2)

```
using T = std::vector<int>;

void *p = ::aligned_alloc(alignedof(T), sizeof(T));
static_cast<T*>(p)->push_back(101); // UB!

auto v1 = new (p) T(); // -\
v1->push_back(42);      //   |- v1 vector lifetime
v1->~T();               //   _/
v1->push_back(102);     // UB!

auto v2 = new (p) T(); // -\
v2->push_back(123);     //   |- v2 vector lifetime
v2->~T();               //   _/
v2->push_back(103);     // UB!

::free(p);
```

Storage Duration vs. Lifetime (2)

```
using T = std::vector<int>;
```

```
void *p = ::aligned_alloc(alignedof(T), sizeof(T));
```

Accessing an object out of its lifetime is UB...

...even for trivial types!

```
v2->push_back(103);    // UB!
```

```
::free(p);
```


Implicit Lifetime (C++20)

Per C++20, objects may be created implicitly if it avoids UB...

Listing 1: UB before C++20 (taken from P0593R6)

```
struct X { int a, b; };  
X *make_x() {  
    X *p = (X*)malloc(sizeof(struct X)); // C++20: implicitly create X  
    p->a = 1;  
    p->b = 2;  
    return p;  
}
```

Implicit Lifetime (C++20)

Per C++20, objects may be created implicitly if it avoids UB...

Listing 2: UB before C++20 (taken from P0593R6)

```
struct X { int a, b; };
```

```
X *make_x() {
```

```
    X *p = (X*)malloc(sizeof X);
```

```
    p->a = 1;
```

```
    p->b = 2;
```

```
    return p;
```

```
}
```



C++20: implicitly create X

Implicit Lifetime (C++20)

Per C++20, objects may be created implicitly if it avoids UB...

Happens in many places; remarkably:

- Memory allocation functions, incl. `malloc()`
- `memcpy()` and `memmove()`
- `std::bit_cast()`

X

Alignment Requirements

Alignment determines which base addresses are legal for a data type.

E.g. assuming `sizeof(int) == 4` and `alignof(int) == 4`

- An `int` may be at offset 0, 4, 8, etc.
- But not at offset 3

Align

E.g.

Important; underlying HW may be unable to do unaligned accesses.

In order to ensure alignment, compiler may insert padding bytes

```
struct X {  
    unsigned char c;  
    int i; ///  
    // <<<< Padding inserted before 'i'  
};
```

- Value of padding bytes is unspecified...
- **Packed structure:** do not include padding. Compiler-specific ❌

Strict Aliasing

Strict Aliasing rule

- **Aliasing:** more than one name (pointer / reference) refers to same address.
- **Strict Aliasing:** only pointers of the **same type** can point to the same object.

[basic.lval], par. 11

- ¹¹ An object of dynamic type T_{obj} is *type-accessible* through a glvalue of type T_{ref} if T_{ref} is similar (7.3.6) to:
- (11.1) — T_{obj} ,
 - (11.2) — a type that is the signed or unsigned type corresponding to T_{obj} , or
 - (11.3) — a `char`, `unsigned char`, or `std::byte` type.

If a program attempts to access (3.1) the stored value of an object through a glvalue through which it is not type-accessible, the behavior is undefined.⁴² If a program invokes a defaulted copy/move constructor or

A very obvious example

```
int fn(int *i, float *f) {
    (*i)++;
    *f = 3.1415f;
    return *i;
}
// Optimizer assumes value of '*i' can't be changed by assigning '*f'

int main(int argc, char *argv[]) {
    int a = 1;
    return fn(&a, reinterpret_cast<float *>(&a));
}
```

A very obvious example

```
int  
(  
*  
r  
}  
//  
int  
i  
r  
}
```

▶ See in Godbolt

```
fn(int*, float*):
```

```
    push    rbp  
    mov     rbp, rsp  
    mov     qword ptr [rbp - 8], rdi  
    mov     qword ptr [rbp - 16], rsi  
    mov     rax, qword ptr [rbp - 8]  
    mov     ecx, dword ptr [rax]  
    add     ecx, 1  
    mov     dword ptr [rax], ecx  
    mov     rax, qword ptr [rbp - 16]  
    movss   xmm0, dword ptr [rip + .LCPI0_0]  
    movss   dword ptr [rax], xmm0  
    mov     rax, qword ptr [rbp - 8]  
    mov     eax, dword ptr [rax]  
    pop     rbp  
    ret
```

(-00)

```
fn(int*, float*):
```

```
    mov     eax, dword ptr [rdi]  
    inc     eax  
    mov     dword ptr [rdi], eax  
    mov     dword ptr [rsi], 1078529622  
    ret
```

(-02)

A not-so-obvious example

BSD sockets: `sockaddr` vs. `sockaddr_in`

Q: Strict aliasing violation (Y/N)? 🧑

```
struct sockaddr_in addr_in;  
  
addr_in.sin_addr.s_addr = INADDR_ANY;  
addr_in.sin_port = htons(8080);  
// Violates strict aliasing ??  
bind(sockfd, reinterpret_cast<struct sockaddr *>(&addr_in), sizeof(  
    addr_in));
```

Type Punning done right

Possible (or not) approaches for type punning

People that type-pun, usually take one of the following approaches:

- Based on `union`
- Using `reinterpret_cast` (sometimes combined w/ `std::launder`)
- Use of `memcpy()`
- `std::bit_cast`
- `std::start_lifetime_as`

Some of them lead to **UB in C++** (but otherwise correct in C)

Possible (or not) approaches for type punning

People that type-pun, usually take one of the following approaches:

-
-
- You should also be aware of...
 - `sizeof(char)`, `sizeof(int)`, etc., are platform-dependent.
 - Use fixed-width integer types (e.g. `uint32_t`) if appropriate
 - ISO C++ requires `CHAR_BIT` to be ≥ 8 ; POSIX requires it to be `== 8`
-

Some of them lead to **UB in C++** (but otherwise correct in C)

union-based

From *[class.union.general]*: at most **one non-static member is active**; accessing non-active members of a **union** is UB!³.⁴

```
union {  
    unsigned char c[sizeof(int)];  
    int i;  
} u{ 0xff00ff00 }; // active member is 'i'  
  
u.c[0]; // UB!
```

IN GENERAL: DON'T! – It's legal in C, though.

³Accesses object whose lifetime has not begun.

⁴Exception on next slide.

union-based

From *[class.union.general]*: at most **one non-static member is active**; accessing non-active members of a **union** is UB!³.⁴

```
union {  
    unsigned char c[sizeof(int)];  
    int i;  
} u{ 0xff00ff00 }; // active member is 'i'  
  
u.c[0]; // UB!
```

IN GENERAL: DON'T! – It's legal in C, though.

³Accesses object whose lifetime has not begun.

⁴Exception on next slide.

union-based: THE ONLY EXCEPTION

BUT THERE'S ONE EXCEPTION!

[class.union.general], p. 2, Note 1 says...

[Note 1: One special guarantee is made in order to simplify the use of unions: If a standard-layout union contains several standard-layout structs that share a common initial sequence (11.4), and if a non-static data member of an object of this standard-layout union type is active and is one of the standard-layout structs, the common initial sequence of any of the standard-layout struct members can be inspected; see 11.4. — end note]

union-based: THE ONLY EXCEPTION

Again BSD sockets: `sockaddr` and `sockaddr_in` have a common prefix

BUT

```
union {  
    struct sockaddr addr;  
    struct sockaddr_in addr_in;  
};  
  
addr_in.sin_addr.s_addr = INADDR_ANY;  
addr_in.sin_port = htons(8080);  
// See also [class.mem.general], p. 29.  
bind(sockfd, &addr, sizeof(addr_in));
```

`reinterpret_cast<T>(obj)` is guaranteed to be safe if

- Pointer-interconvertible [*basic.compound*], p. 5:

5 Two objects *a* and *b* are *pointer-interconvertible* if

- (5.1) — they are the **same object**, or
- (5.2) — one is a **union object and the other is a non-static data member of that object** (11.5), or
- (5.3) — one is a **standard-layout class object and the other is the first non-static data member** of that object or any base class subobject of that object (11.4), or
- (5.4) — there exists an object *c* such that *a* and *c* are pointer-interconvertible, and *c* and *b* are pointer-interconvertible.

- Cast-to type is one of
 - `char`, `unsigned char` or `std::byte`
 - `decltype(obj)` (or its signed / unsigned type)

rei

- Don't do `reinterpret_cast<float *>(&an_int)` (violates strict aliasing)
- But even `reinterpret_cast<unsigned char *>(&something)` is not totally right
 - A `unsigned char []` object is not within its lifetime!^a
 - P1839R7 is supposed to fix that
 - What about `std::as_bytes` / `std::as_writable_bytes` then? 🧑

Key idea: pay attention to your `reinterpret_casts`; they are most likely UB (strict aliasing)!

^aMost compilers will do the right thing, but it's still UB!

object or

pointer-

std::launder is...

- According to cppreference.com:

Devirtualization fence with respect to `p`. Returns a pointer to an object at the same address that `p` represents, while the object can be a new base class subobject whose most derived class is different from that of the original `*p` object. Formally, given

- According to ISO C++ wording: a pointer optimization barrier

17.6.5 Pointer optimization barrier

[ptr.launder]

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

1 *Mandates:* `!is_function_v<T> && !is_void_v<T>` is true.

2 *Preconditions:* `p` represents the address *A* of a byte in memory. An object *X* that is within its lifetime (6.7.4) and whose type is similar (7.3.6) to *T* is located at the address *A*. All bytes of storage that would be reachable through (6.8.4) the result are reachable through `p`.

std::launder: w00t?

std::launder is...

- According to cppreference.com:

Devirtualization fence with respect to `p`. Returns a pointer to an object at the same address that `p` represents, while the object can be a new base class subobject whose most derived class is different from that of the original `*p` object. Formally, given

- According to ISO C++ wording: an inter-continuation barrier

17.6.5 Pointer optimization

[ptr.launder]

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

1 *Mandates:* `!is_function_v<T> && !is_void_v<T>` is true.

2 *Preconditions:* `p` represents the address *A* of a byte in memory. An object *X* that is within its lifetime (6.7.4) and whose type is similar (7.3.6) to *T* is located at the address *A*. All bytes of storage that would be reachable through (6.8.4) the result are reachable through `p`.

std::launder: pointer provenance

- *Pointer* \neq *MemoryAddress*; an **address** is just the **value of a pointer**.
- In C++, a **pointer points to an object**. The compiler can make assumptions on the pointee.

Instead, think of a pointer as a pair

$\langle \text{address}, \text{provenance} \rangle$

where *provenance* determines which values can be reached through the pointer.

- **Money laundering**: concealing the origin of money
- **Pointer laundering**: update the provenance of a pointer, i.e. remove any assumptions on the pointee

- *Pointer* \neq *MemoryAddress*; an **address** is just the **value of a pointer**.
- In C++, a **pointer points to an object**. The compiler can make assumptions on the pointee.

Instead, think of a pointer as a pair

$\langle \text{address}, \text{provenance} \rangle$

where *provenance* determines which values can be reached through the pointer.

- **Money laundering**: concealing the origin of money
- **Pointer laundering**: update the provenance of a pointer, i.e. remove any assumptions on the pointee

std::launder: pointer provenance

- *Pointer* \neq *MemoryAddress*; an **address** is just the **value of a pointer**.
- In C++, a **pointer points to an object**. The compiler can make assumptions on the

- **Pre-conditions:**

17.6.5 Pointer optimization barrier

[ptr.launder]

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

1 *Mandates:* `!is_function_v<T> && !is_void_v<T>` is true.

2 *Preconditions:* `p` represents the address *A* of a byte in memory. An object *X* that is within its lifetime (6.7.4) and whose type is similar (7.3.6) to *T* is located at the address *A*. All bytes of storage that would be reachable through (6.8.4) the result are reachable through `p`.

- Useful when replacing an object (reusing storage) that had `const` members or references
- **Pointer laundering:** update the provenance of a pointer, i.e. remove any assumptions on the pointee

std::launder: Example 1

```
uint32_t i = 42;
float *fp = reinterpret_cast<float *>(&i);

new (static_cast<void *>(&i)) float(12.34f);

// std::cout << *fp << std::endl; // UB! 'fp' doesn't point to a '
// float' object

std::cout << *std::launder(fp) << std::endl; // OK;
```

std::launder: Example 2

```
struct MyStruct {  
    const float f;  
};  
  
MyStruct obj{12.34f};  
  
MyStruct *p = &obj;  
std::cout << "f=" << p->f << std::endl;  
  
new (static_cast<void*>(p)) MyStruct{3.1415f};  
  
std::cout << p->f << std::endl; // ???  
std::cout << std::launder(p)->f << std::endl; // OK  
}
```

memcpy() over (representation of) different object

```
uint32_t i = 42;  
  
float f;  
memcpy(&f, &i, sizeof(uint32_t)); // OK  
  
unsigned char c[sizeof(uint32_t)];  
memcpy(c, &i, sizeof(uint32_t)); // Also OK
```

Strict-aliasing [✓]

Alignment reqs. [✓]

Lifetime [✓]

memcpy() over (representation of) different object

```
uint32_t i = 42;
```

- `memcpy()` is optimized out where possible by major compilers

▶ See in Godbolt

std::bit_cast [bit.cast], p. 1-2

22.11.3 Function template bit_cast

[bit.cast]

```
template<class To, class From>  
constexpr To bit_cast(const From& from) noexcept;
```

Constraints:

- `sizeof(To) == sizeof(From)` is true;
- `is_trivially_copyable_v<To>` is true; and
- `is_trivially_copyable_v<From>` is true.

Returns: An object of type `To`. Implicitly creates objects nested within the result (6.7.2). Each bit of the value representation of the result is equal to the corresponding bit in the object representation of `from`. Padding bits of the result are unspecified. For the result and each object created within

std::bit_cast: Example

```
float f = 42.0f;  
  
auto c = std::bit_cast<std::array<unsigned char, sizeof(float)>>(f);  
// Use the 'c' array
```

Strict-aliasing [✓]

Alignment reqs. [✓]

Lifetime [✓]

- (Very) roughly speaking: ISO-CPP-blessed for the previous `memcpy()`
- It's `constexpr`; optimized out where possible [▶ See in Godbolt](#)
- **Note:** `std::bit_cast<SomeType *>()` is simply **WRONG!** (P0476R1)

std::start_lifetime_as (C++23)

- Explicit lifetime management (C++23)
- Starts lifetime of an object of the given type at the given address
 - Underlying object representation is preserved

```
uint32_t i = 42;  
float *f = std::start_lifetime_as<float>(&i);  
  
std::cout << *f << std::endl; // OK
```

std::start_lifetime_as (C++23)

- Recall the strict aliasing rule! No pair of objects of different type can be at same address!
 - Thus, accessing `i` becomes UB!

```
uint32_t i = 42;
float *f = std::start_lifetime_as<float>(&i);

std::cout << *f << std::endl; // OK
std::cout << i << std::endl; // UB!
std::start_lifetime_as<int>(&i);

std::cout << i << std::endl; // Still UB!
std::cout << *std::launder(&i) << std::endl; // OK
```

std::start_lifetime_as (C++23)

- Recall the strict aliasing rule! No pair of objects of different type can be at same address!
 - Thus, accessing `i` becomes UB!

```
uint32_t i = 42;
float *f = std::start_lifetime_as<float>(&i);

std::cout << *f << std::endl; // OK
std::cout << i << std::endl; // UB!
std::start_lifetime_as<int>(&i);

std::cout << i << std::endl; // Still UB!
std::cout << *std::launder(&i) << std::endl; // OK
```

Punning via ... is ...

- Based on `union` ❌ (note the exception)
- Using `reinterpret_cast`
 - Pointer-interconvertible: ✅
 - To `char`-like type: [Cast ✅] [Deref ?]
 - Rest: ❌
- Using `std::as_bytes / std::as_writable_bytes` (C++20) ✅ ?
- Use of `memcpy()` ✅
- `std::bit_cast` ✅
- `std::start_lifetime_as` ✅

What if $C++ \leq xx$

If $C++ \leq 23$...

- Taking into account [P0593R6](#), it may be implemented by using the special properties of `memmove()`
- The compiler infers the type of the object whose lifetime starts

If $C++ \leq 20$...

- For `std::bit_cast`: consider possible impl. described in cppreference.com
- Or...just use `memcpy()`

Practical Case: (de-)serialization

Introduction to serialization (1)

Serialize: lay out an in-memory **data structure** as a **sequence of bytes**.

Note that...

- A type can have nested pointers / references
- `sizeof(int)` , `sizeof(long)` , etc. can vary depending on target^a

```
static_cast(sizeof(char) == 1); // ??  
static_cast(sizeof(int) == 4); // ??
```

- Differences in machine endianness
- Types require alignment (+ maybe padding)

^aSee, e.g. ILP32, LLP64 models.

Can be just `memcpy()` if...

- Data structure is trivially copiable
- No pointers / references to other data structures
- Uses fixed-width types, e.g. `uint32_t`
- Machine boundary is never crossed (e.g. IPC)
- Alignment is satisfied on de-serialization

Can be just `memcpy()` if

-
-
-
-
-

Else, implement proper data serialization!

- Encode numbers using a common endianness, e.g. big-endian
- Flatten data structure to byte buffer → follow pointers / references

Let's consider this simple data structure

```
struct Foobar {  
    uint64_t l64;  
    uint32_t i32;  
  
    static size_t Serialize(const Foobar& obj, unsigned char *buf);  
    static Foobar Deserialize(const unsigned char *buf);  
};
```

Example: machine boundary NOT crossed: '(de-)serialize'

```
size_t Foobar::Serialize(const Foobar& obj, unsigned char *buf) {  
    memcpy(buf, &obj, sizeof(Foobar));  
}  
  
Foobar Foobar::Deserialize(const unsigned char *buf) {  
    Foobar ret;  
    memcpy(&ret, buf, sizeof(Foobar));  
    return ret;  
}
```

Example: machine boundary NOT crossed: '(de-)serialize'

Padding bytes are **unspecified**. If you care, avoid this!.

Example: machine boundary crossed: (de-)serialize

```
size_t Foobar::Serialize(const Foobar& obj, unsigned char *buf) {
    buf += SerializeUInt64(obj.l64, buf);
    buf += SerializeUInt32(obj.i32, buf);
}

Foobar Foobar::Deserialize(const unsigned char *buf) {
    Foobar ret;
    buf += DeserializeUInt64(buf, obj.l64);
    buf += DeserializeUInt32(buf, obj.i32);
    return ret;
}
```

Example: machine boundary crossed: (de-)serialize

Example of UInt32 little-endian (de-)serialization

```
size_t SerializeUInt32(uint32_t i, unsigned char *buf) {  
    bytes[0] = (i & 0x000000ff);  
    bytes[1] = (i & 0x0000ff00) >> 8;  
    bytes[2] = (i & 0x00ff0000) >> 16;  
    bytes[3] = (i & 0xff000000) >> 24;  
    return 4;  
}  
  
size_t DeserializeUInt32(unsigned char *buf, uint32_t& i) {  
    i = std::uint32_t(buf[0]) + (std::uint32_t(buf[1]) << 8)  
        + (std::uint32_t(buf[2]) << 16) + (std::uint32_t(buf[3]) <<  
        24);  
    return 4;  
}
```

As a rule of thumb

- *standard-layout* / *POD* + machine boundary not crossed (ever) → copy + start lifetime on receiving end
- Rest: implement proper serialization!

See also: [▶ Boost Serialization](#)

Conclusion

About `reinterpret_cast`

- Most of its uses are UB; don't do it unless you are absolutely sure!
- Even `reinterpret_cast<char *>` is not well-defined, but it works in practice
- Use `std::bit_cast` or `std::start_lifetime_as` if available

Type punning is only okay if...

- It doesn't break the strict aliasing rule
 - Only accesses bytes reachable through the original pointer
 - Complies to the alignment requirements of the type
 - The accessed object is within its lifetime
 - **NO unions**, please!
-
- Implement proper serialization where appropriate!

Other Resources

1. The ISO C++ standard (working draft): <https://github.com/cplusplus/draft>
2. JTC1/SC22/WG21 P0593R6:
<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0593r6.html>
3. JTC1/SC22/WG21 P0476R1:
<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0476r1.html>
4. JTC1/SC22/WG21 P3292R0:
<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3292r0.html>
5. JTC1/SC22/WG21 P1839R7:
<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p1839r7.html>
6. https://en.cppreference.com/w/cpp/numeric/bit_cast
7. https://en.wikipedia.org/wiki/64-bit_computing#64-bit_data_models
8. <https://www.kernel.org/doc/html/latest/core-api/unaligned-memory-access.html>

Thank you!



[Link to slides used in this presentation]

-
1. Paragraphs from the ISO C++ standard have been cited verbatim from the working draft.

Backup

A very obvious example

```
int fn(int *i, float *f) {
```

```
(
```

```
*
```

```
r
```

```
}
```

```
//
```

```
int
```

```
i
```

```
r
```

```
}
```

I did not say this – but it's here just for completeness

- `-fno-strict-aliasing` option (GCC / clang) may help
- Useful for legacy codebases or when you **really (really!)** know what you are doing
 - Linux kernel uses it, e.g. in `arch/arm64/kernel/vdso32/Makefile`

Type-based Alias Analysis (TBAA)

Alias analysis based on the type system of a high level language.

LLVM-specific, from `▶ llvm/lib/Analysis/TypeBasedAliasAnalysis.cpp`:

- **Scalar TBAA:** alias analysis on fundamental datatype, **regardless of where it appears**.
- **Struct-path aware TBAA:** analysis takes into account **base type** (struct), **access type** (leaf, fundamental type), and its **offset**.

If you are interested, read the very clear 100+ code comment in there!