

# Taking a Byte Out of C++

**Avoiding Punning by Starting Lifetimes** 

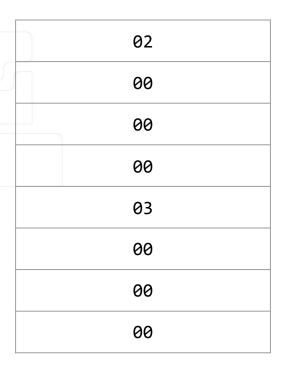
**ROBERT LEAHY** 





```
struct foo {
    std::uint32 t a;
    std::uint32_t b;
foo f{2, 3};
```

a	02	2
	00	
	00	
	00	
b	03	3
	00	
	00	
	00	



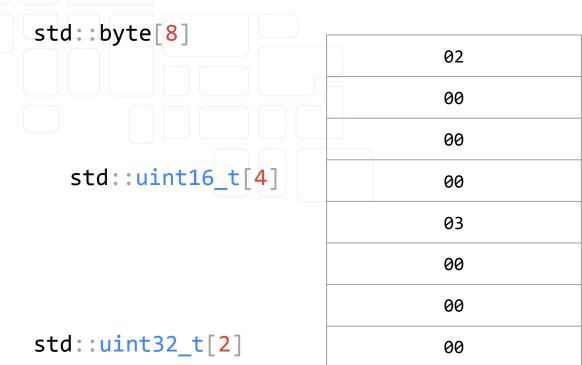
#### std::byte[<mark>8</mark>]

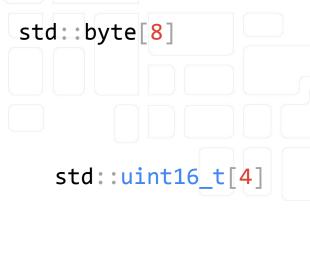
02
00
00
00
03
00
00
00



std::uint16\_t[4]

02
00
00
00
03
00
00
00





std::uint32_t[2]
------------------

ı	
	02
/	00
	00
	00
	03
	00
	00
	00

std::uint64\_t

```
std::byte[8]
  std::uint16_t[4]
```

std::uint32\_t[2]

```
02
00
00
00
03
00
00
00
```

```
std::uint64_t

struct foo {
    std::uint32_t a;
    std::uint32_t b;
```

```
struct foo {
   std::uint32 t a;
    std::uint32_t b;
static assert(sizeof(foo) == sizeof(std::uint64 t));
std::uint32 t bar(std::uint64 t& i, const foo& f) noexcept {
   if (f.a == 2) {
       i = 4:
   if (f.a == 2) {
        return f.a;
    return f.b;
```

```
int main() {
    foo f{2, 3};
    return bar((std::uint64_t&)f, f);
}
```

	02	
	00	2
	00	2
	00	
12'884'901'890	03	
	00	
	00	3
	00	

	04	
	00	4
	00	4
	00	
4	00	
	00	
	00	0
	00	

```
int main() {
    foo f{2, 3};
    return bar((std::uint64_t&)f, f);
}
```

```
main:
```

mov eax, 2 ret

```
bar(unsigned long&, foo const&):
                eax, dword ptr [rsi]
        mov
                eax, 2
        cmp
        je
                .LBB0 1
                eax, 2
        cmp
        jne
                .LBB0_3
.LBB0_4:
        ret
.LBB0_1:
                qword ptr [rdi], 4
        mov
                eax, 2
        cmp
        je
                .LBB0 4
.LBB0_3:
                eax, dword ptr [rsi + 4]
        mov
        ret
```

### No Type Punning

An object within its lifetime may only be accessed in certain ways

Through a reference to its type (addition of cv qualification allowed)

Through a reference to its signed or unsigned equivalent

Through a reference to char, unsigned char, or std::byte

Any other access modality is undefined behavior

```
struct foo {
    std::uint32 t a;
    std::uint32_t b;
static assert(sizeof(foo) == sizeof(std::uint64 t));
std::uint32_t bar(std::uint<mark>6432</mark>_t& i, const foo& f) noexcept {
    if (f.a == 2) {
        i = 4:
    if (f.a == 2) {
        return f.a;
    return f.b;
```

```
bar(unsigned longint 4, foo const 4):
                eax, dword ptr [rsi]
        mov
                eax, 2
        cmp
        je
                 .LBB0_1
                eax, 2
        cmp
                 .LBB0_3
        jne
.LBB0_4:
        ret
.LBB0_1:
                gdword ptr [rdi], 4
        mov
                eax, dword ptr [rsi]
        mov
                eax, 2
        cmp
        je
                 .LBB0_4
.LBB0_3:
                eax, dword ptr [rsi + 4]
        mov
        ret
```

### C++ Has an Object Model

Bytes supply storage for objects

Objects have lifetimes

Duration of storage is not necessarily the same as object lifetime

Accessing object outside lifetime is undefined behavior

```
const auto ptr = (int*)std::malloc(sizeof(int) * 4);
if (!ptr) {
   throw std::bad alloc();
for (int i = 0; i < 4; ++i) {
   ptr[i] = i;
```

```
const auto ptr = (std::string*)std::malloc(sizeof(std::string) * 4);
if (!ptr) {
    throw std::bad alloc();
for (int i = 0; i < 4; ++i) {
   ptr[i] = std::to_string(i);
```

## C++ Types May Have Invariants

One of the core value propositions of C++

Invariants are established by constructors

Invariants are maintained by members

Some types don't have such strict requirements

Contain basic values

Don't maintain complicated (or any) invariants

Such types are trivial types



# Implicit-Lifetime Types (C++20)

Certain types are "implicit-lifetime"

Aggregate types
At least one trivial constructor and trivial destructor

Certain operations implicitly create objects of implicit-lifetime type

std::malloc et al.

std::memcpy and ::memmove

Starting lifetime of array of char, unsigned char, or std::byte

operator new and operator new[]

See P0593

```
const auto ptr = (int*)std::malloc(sizeof(int) * 4);
if (!ptr) {
   throw std::bad alloc();
for (int i = 0; i < 4; ++i) {
   ptr[i] = i;
```

```
const auto ptr = (std::string*)std::malloc(sizeof(std::string) * 4);
if (!ptr) {
    throw std::bad alloc();
for (int i = 0; i < 4; ++i) {
   ptr[i] = std::to_string(i);
```

```
const auto ptr = (std::string*)std::malloc(sizeof(std::string) * 4);
if (!ptr) {
    throw std::bad alloc();
for (int i = 0; i < 4; ++i) {
    new(ptr + i) std::string(std::to_string(i));
```

```
int baz(const void* ptr) noexcept {
    return *static_cast<const int*>(ptr);
}
```

```
baz(void const*):
        mov eax, dword ptr [rdi]
        ret
```

```
int baz(const void* ptr) noexcept {
  int retr;
  std::memcpy(&retr, ptr, sizeof(int));
  return retr;
}
```

```
baz(void const*):
    mov eax, dword ptr [rdi]
    ret
```

```
int baz(const void* ptr) noexcept {
   alignas(int) std::byte buffer[sizeof(int)];
   const auto retr = std::memcpy(buffer, ptr, sizeof(int));
   return *reinterpret_cast<int*>(retr);
}
```

```
baz(void const*):
    mov eax, dword ptr [rdi]
    ret
```

#### std::bit\_cast

Creates an object whose value representation is that of another object

Objects must be the same size

Objects must both be trivially copyable

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

```
void corge(const int&) noexcept;
void quux(const void* ptr) noexcept {
    corge(*static_cast<const int*>(ptr));
}
```

```
quux(void const*):
    jmp corge(int const&)
```

```
void corge(const int&) noexcept;
void quux(const void* ptr) noexcept {
   alignas(int) std::byte buffer[sizeof(int)];
   const auto retr = std::memcpy(buffer, ptr, sizeof(int));
   corge(*reinterpret_cast<int*>(retr));
}
```

quux(void const\*):

```
void corge(const int&) noexcept;
void quux(const void* ptr) noexcept {
   alignas(int) std::byte buffer[sizeof(int)];
   const auto retr = std::memcpy(buffer, ptr, sizeof(int));
   corge(*reinterpret_cast<int*>(retr));
}
```

```
quux(void const*):
    sub    rsp, 24
    mov    eax, DWORD PTR [rdi]
    lea    rdi, [rsp+12]
    mov    DWORD PTR [rsp+12], eax
    call    corge(int const&)
    add    rsp, 24
    ret
```

```
void corge(const int&) noexcept;
void quux(const void* ptr) noexcept {
    const auto mutable_ptr = const_cast<void*>(ptr);
    const auto byte_ptr = new(mutable_ptr) std::byte[sizeof(int)];
    const auto int_ptr = reinterpret_cast<const int*>(byte_ptr);
    corge(*int_ptr);
}
```

```
quux(void const*):
    jmp corge(int const&)
```

```
template<class T>
T* start lifetime as(void* p) noexcept;
template < class T>
const T* start_lifetime_as(const void* p) noexcept;
template<class T>
volatile T* start lifetime as(volatile void* p) noexcept;
template<class T>
const volatile T* start lifetime as(const volatile void* p) noexcept;
template<class T>
T* start lifetime as array(void* p, size_t n) noexcept;
template<class T>
const T* start lifetime as_array(const void* p, size_t n) noexcept;
template<class T>
volatile T* start_lifetime_as_array(volatile void* p, size_t n) noexcept;
template<class T>
const volatile T* start lifetime as array(const volatile void* p, size t n) noexcept;
```

See P2590

```
template<typename T>
const T* start_lifetime_as(const void* p) noexcept {
   const auto mp = const_cast<void*>(p);
   const auto bytes = new(mp) std::byte[sizeof(T)];
   const auto ptr = reinterpret_cast<const T*>(bytes);
   (void)*ptr;
   return ptr;
}
```

```
void corge(const int&) noexcept;
void quux(const void* ptr) noexcept {
    corge(*std::start_lifetime_as<int>(ptr));
}
```

```
quux(void const*):
    jmp corge(int const&)
```

```
#pragma pack(push)
#pragma pack(4)
struct erased_update
    std::uint64_t raw_timestamp;
    std::uint32 t length;
    std::uint64 t timestamp() const noexcept {
        alignas(std::uint64_t) std::byte buffer[sizeof(std::uint64_t)];
        const auto ptr = std::memcpy(buffer, &raw timestamp, sizeof(std::uint64 t));
        return *reinterpret cast<std::uint64 t*>(ptr);
#pragma pack(pop)
```

```
struct open_query {
    const erased_update* last_update() const noexcept;
```

```
#pragma pack(push)
#pragma pack(4)
struct update : erased_update {
    std::uint32_t sequence_number;
};
#pragma pack(pop)
```

```
open_query q(/* ... */);
// ...
const auto ptr = q.last_update();
if (ptr->length < sizeof(update)) {
    throw std::runtime_error("Update too short!");
}
const auto u = std::start_lifetime_as<update>(ptr);
std::cout << "Sequence number " << u->sequence_number << std::endl;</pre>
```

```
enum class update as error { success = 0, too short };
std::error code make error code(update as error) noexcept;
template<typename T>
using update as result = std::expected<const T*, std::error code>;
template<typename T>
update as result<T> update as(const erased update& u) noexcept {
    if (u.length < sizeof(T)) {</pre>
        const auto ec = make error code(update as error::too short);
        return std::unexpected(ec);
    return std::start lifetime as<T>(&u);
```

```
open_query q(/* ... */);
const auto ptr = q.last_update();
const auto u = update_as<update>(*ptr).value();
std::cout << "Sequence number " << u->sequence_number << std::endl;</pre>
```

```
open_query q(/* ... */);
const auto ptr = q.last_update();
const auto u = update as<update>(*ptr).value();
std::cout << "Timestamp " << ptr->timestamp() << std::endl;</pre>
std::cout << "Sequence number " << u->sequence number << std::endl;</pre>
```

## Ending an Object's Lifetime

Lifetime can end in the usual ways

Object with automatic storage duration goes out of scope

Object with dynamic storage duration is deleted

Can also end in other ways

Reuse of backing storage

## std::launder

Reusing storage invalidates pointers and references to the old object

Unless the old and new objects are "transparently replaceable"

Pointers point to the storage but no longer to the object

std::launder obtains a pointer to the object from a pointer to the storage

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

```
open_query q(/* ... */);
auto ptr = q.last update();
const auto u = update as<update>(*ptr).value();
ptr = std::launder(ptr);
std::cout << "Timestamp " << ptr->timestamp() << std::endl;</pre>
std::cout << "Sequence number " << u->sequence number << std::endl;</pre>
```

```
#pragma pack(push)
#pragma pack(4)
struct update : erased_update {
    std::uint32_t sequence_number;
    std::string_view name() const noexcept;
};
#pragma pack(pop)
```

```
std::string_view_update::name() const noexcept {
    const auto size = length - sizeof(update);
   if (!size) {
       return {};
   const auto ptr = reinterpret_cast<const std::byte*>(this) + sizeof(*this);
    const auto str = std::start lifetime as array<char>(ptr, size);
   return {str, size};
```

```
#pragma pack(push)
#pragma pack(4)
struct update : erased update {
    std::uint32 t sequence number;
    struct leg type {
        std::uint32 t id;
        std::uint16 t ratio quantity;
        std::uint8 t buy;
        std::uint8 t reserved;
    std::span<const leg type> legs() const noexcept;
#pragma pack(pop)
```

```
auto update::legs() const noexcept -> std::span<const leg type> {
    const auto size = (length - sizeof(update)) / sizeof(leg type);
   if (!size) {
       return {};
   const auto ptr = reinterpret cast<const std::byte*>(this) + sizeof(*this);
    const auto arr = std::start lifetime as array<leg type>(ptr, size);
   return {arr, size};
```

```
enum class update validate error { success = 0, size, buy };
std::error code make error code(update validate error) noexcept;
std::error code update::validate() const noexcept {
    const auto remaining = length - sizeof(update);
    if (remaining % sizeof(leg_type)) {
        return make error code(update validate error::size);
   for (auto&& leg : legs()) {
        if (leg.buy > 1) {
            return make error code(update validate error::buy);
    return {};
```

```
template<typename T>
concept update_as_good_validate = requires(const T u) {
   { u.validate() } noexcept -> std::same as<std::error code>;
template<typename T>
concept update as has validate = requires(const T u) {
   u.validate();
};
template<typename T>
concept update as concept = !update as has validate<T> | |
   update as good validate<T>;
```

```
template<update as concept T>
update as result<T> update as(const erased update& u) noexcept {
    if (u.length < sizeof(T)) {</pre>
        const auto ec = make_error_code(update_as_error::too_short);
        return std::unexpected(ec);
    const auto retr = std::start lifetime as<T>(&u);
    if constexpr (update as good validate<T>) {
        const auto ec = retr->validate();
        if (ec) {
            return std::unexpected(ec);
    return retr;
template<typename T>
update as result<T> update as(const erased update&) = delete;
```

## Summary

Bytes which constitute an object reside in storage

Meaningfulness of the concept of an object not necessarily related to storage

All objects have lifetimes regardless of how trivial they are

Implicit lifetime rules enable zero copy techniques with well-defined behavior

As with all low level techniques care must be taken

Potentially-dangerous operations can and should be factored out and isolated

Remainder of code is clean, correct, efficient, and well-defined

## Questions?

Robert Leahy Lead Software Engineer rleahy@rleahy.ca

