Taking a Byte Out of C++

Avoiding Punning by Starting Lifetimes

ROBERT LEAHY

Cppcon
The C++ Conference
2022
September 12th-16th

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

<table>
<tr><td rowspan="4">a</td><td>02</td><td rowspan="4">2</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td></tr>
<tr><td rowspan="4">b</td><td>03</td><td rowspan="4">3</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td></tr>
</table>

02
00
00
00
03
00
00
00

```
std::byte[8]
```

02
00
00
00
03
00
00
00

std::byte[8]

std::uint16_t[4]

02
00
00
00
03
00
00
00

std::byte[8]

std::uint16_t[4]

std::uint32_t[2]

02
00
00
00
03
00
00
00

std::byte[8]

std::uint16_t[4]

std::uint32_t[2]

std::uint64_t

02
00
00
00
03
00
00
00

`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;
}
```

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

<table>
<tr><td rowspan="8">12’884’901’890</td><td>02</td><td rowspan="4">2</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td></tr>
<tr><td>03</td><td rowspan="4">3</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td></tr>
</table>

<table>
<tr><td rowspan="8">4</td><td>04</td><td rowspan="4">4</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td><td rowspan="4">0</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td></tr>
<tr><td>00</td></tr>
</table>

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

```asm
main:
        mov     eax, 2
        ret
```

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

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

static_assert(sizeof(foo) == sizeof(std::uint64_t));

std::uint32_t bar(std::uint6432_t& i, const foo& f) noexcept {
    if (f.a == 2) {
        i = 4;
    }
    if (f.a == 2) {
        return f.a;
    }
    return f.b;
}
```

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

```cpp
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;
}
```

```cpp
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

Trivial types still have lifetimes

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

```cpp
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));
}
```

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

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

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

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

```cpp
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);
}
```

```asm
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;
```

```cpp
template<typename To, typename From>
    requires (sizeof(To) == sizeof(From))     &&
             std::is_trivially_copyable_v<To> &&
             std::is_trivially_copyable_v<From>
To bit_cast(const From& from) noexcept {
    alignas(To) std::byte buffer[sizeof(To)];
    const auto ptr = std::memcpy(buffer, std::addressof(from), sizeof(To));
    return *reinterpret_cast<To*>(ptr);
}
```

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

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

```cpp
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*):
```

```cpp
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));
}
```

```asm
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
```

```cpp
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);
}
```

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

```cpp
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

```cpp
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;
}
```

```cpp
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&)
```

```cpp
#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)
```

```cpp
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;
```

```cpp
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)) {
        const auto ec = make_error_code(update_as_error::too_short);
        return std::unexpected(ec);
    }
    return std::start_lifetime_as<T>(&u);
}
```

```cpp
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;
```

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

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
  - Pseudo-destructor call

    ```
    t.~T()
    ptr->~T()
    ```
  - 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;
```

```cpp
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;
std::cout << "Sequence number " << u->sequence_number << std::endl;
```

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

```cpp
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};
}
```

```cpp
#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)
```

```cpp
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};
}
```

```cpp
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>;
```

```cpp
template<update_as_concept T>
update_as_result<T> update_as(const erased_update& u) noexcept {
    if (u.length < sizeof(T)) {
        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

MAYSTREET