Building Robust Inter-Process Queues in C++

JODY HAGINS

CppCon The C++ Conference
2025 September 13 - 19

Building Robust Inter-Process Queues in C++

JODY HAGINS

Jody Hagins
jody.hagins@lseg.com
coachhagins@gmail.com

CppCon The C++ Conference
2025
September 13 - 19

The Message Queue

Not a task queue

The Message Queue

The Message Queue

Not a task queue

Pass messages between components

The Message Queue

Not a task queue

Pass messages between components

Object oriented programming

The Message Queue

Not a task queue

Pass messages between components

Object oriented programming

Across threads

The Message Queue

Not a task queue

Pass messages between components

Object oriented programming

Across threads

Across processes

The Message Queue

Not a task queue

Pass messages between components

Object oriented programming

Across threads

Across processes

Across hosts

Using Modern C++ to Revive an Old Design

JODY HAGINS

Cppcon
The C++ Conference

2022
September 12th-16th

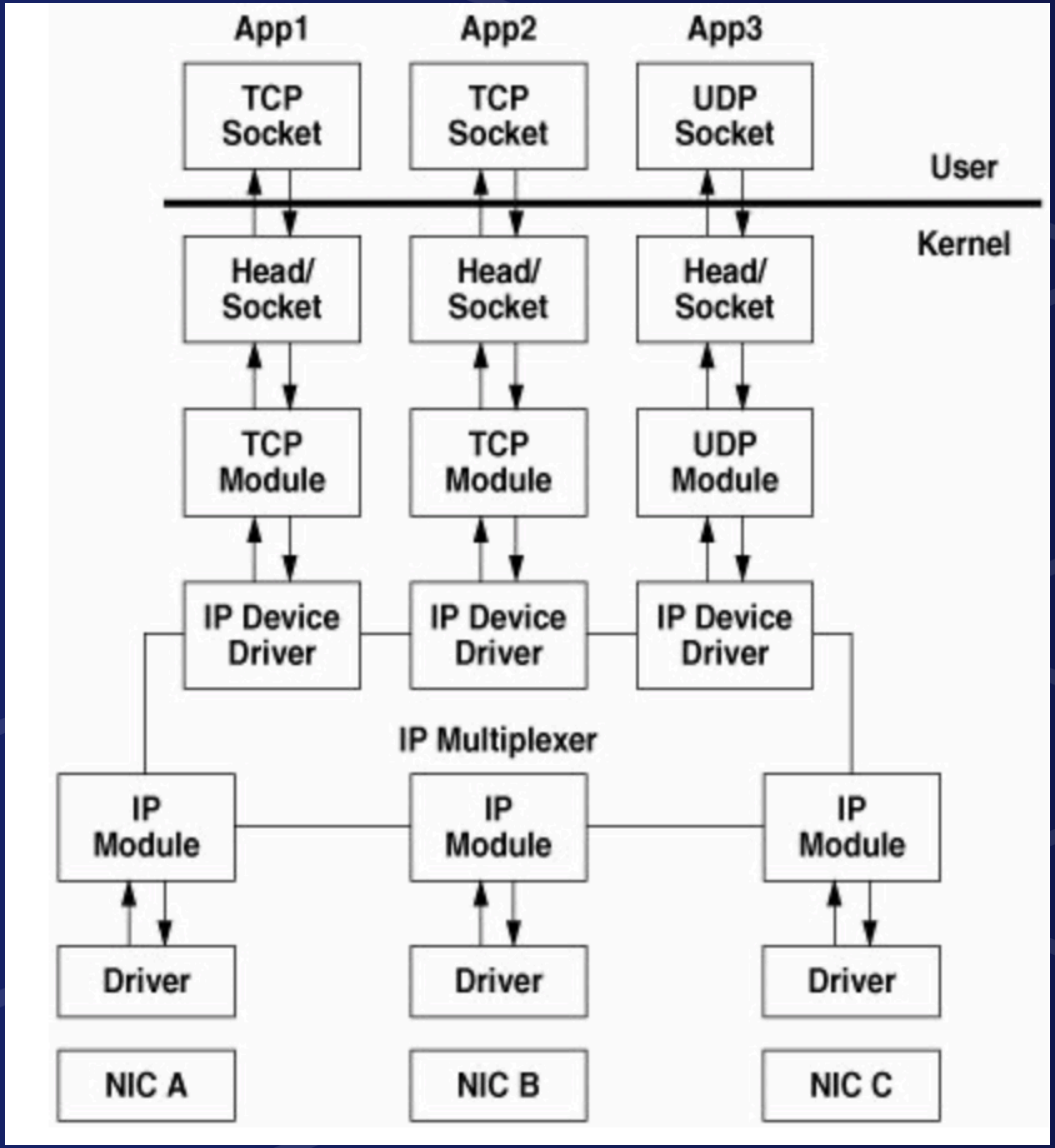
Coupling vs. Cohesion

"One goal of design is to minimize coupling between parts and to maximize cohesion within them."

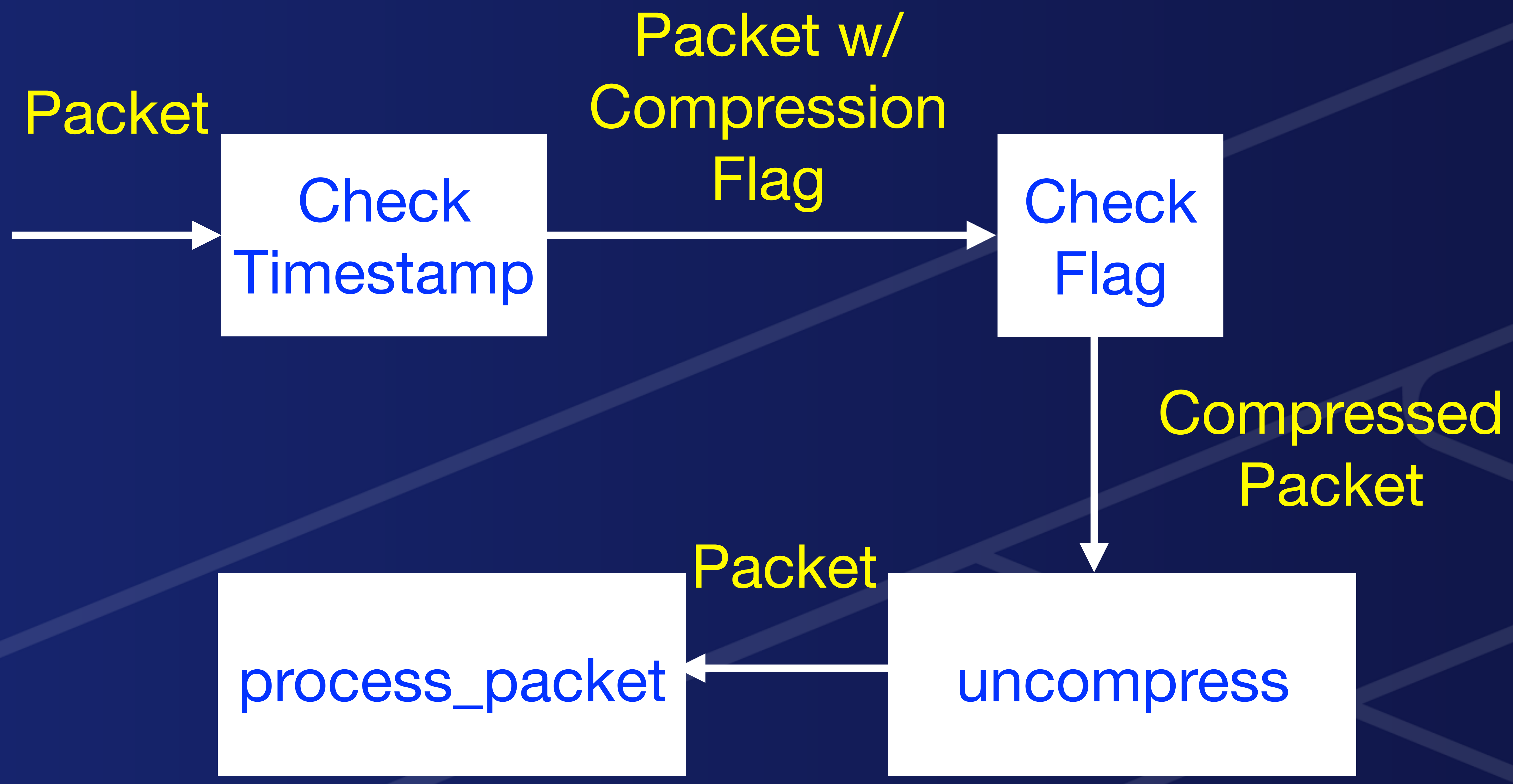
Multi-Paradigm Design for C++ *James Coplien*

An indication of the strength of interconnections between program units.

TCP/UDP in STREAMS



Coupling vs. Cohesion



What Are Inter-Process Queues

What Are Inter-Process Queues

- FIFO

What Are Inter-Process Queues

- FIFO
- Supports Multiple Message Types

What Are Inter-Process Queues

- FIFO
- Supports Multiple Message Types
- Between Unrelated Processes

What Are Inter-Process Queues

- FIFO
- Supports Multiple Message Types
- Between Unrelated Processes
- Low Latency

What Are Inter-Process Queues

- FIFO
- Supports Multiple Message Types
- Between Unrelated Processes
- Low Latency
- Resumable

What Are Inter-Process Queues

- FIFO
- Supports Multiple Message Types
- Between Unrelated Processes
- Low Latency
- Resumable
- Recoverable MCAST Queues

What Are Inter-Process Queues

- FIFO
- Supports Multiple Message Types
- Between Unrelated Processes
- Low Latency
- Resumable
- Recoverable MCAST Queues
- Meyers API

What Are Inter-Process Queues

- FIFO
- Supports Multiple Message Types
- Between Unrelated Processes
- Low Latency
- Resumable
- Recoverable MCAST Queues
- Meyers API
- Pollable

std::queue

std::queue

Just push and pop

std::queue

Just push and pop

And a constructor or two

std::queue

Just push and pop

And a constructor or two

std::queue<T,Container>::queue

`queue() : queue(Container()) {}`	(1)	(since C++11)
`explicit queue( const Container& cont = Container() );`	(2)	(until C++11)
`explicit queue( const Container& cont );`		(since C++11)
`explicit queue( Container&& cont );`	(3)	(since C++11)
`queue( const queue& other );`	(4)	(implicitly declared)
`queue( queue&& other );`	(5)	(since C++11) (implicitly declared)
`template< class InputIt >` `queue( InputIt first, InputIt last );`	(6)	(since C++23)
`template< class Alloc >` `explicit queue( const Alloc& alloc );`	(7)	(since C++11)
`template< class Alloc >` `queue( const Container& cont, const Alloc& alloc );`	(8)	(since C++11)
`template< class Alloc >` `queue( Container&& cont, const Alloc& alloc );`	(9)	(since C++11)
`template< class Alloc >` `queue( const queue& other, const Alloc& alloc );`	(10)	(since C++11)
`template< class Alloc >` `queue( queue&& other, const Alloc& alloc );`	(11)	(since C++11)
`template< class InputIt, class Alloc >` `queue( InputIt first, InputIt last, const Alloc& alloc );`	(12)	(since C++23)
`template< container-compatible-range<T> R>` `queue( std::from_range_t, R&& rg );`	(13)	(since C++23)
`template< container-compatible-range<T> R, class Alloc >` `queue( std::from_range_t, R&& rg, const Alloc& alloc );`	(14)	(since C++23)

std::queue

Just push and pop

And a constructor or two

And a destructor and assignment operators

std::queue

Just push and pop

And a constructor or two

And a destructor and assignment operators

And peek at the first and last???

Element access	
`front`	access the first element (public member function)
`back`	access the last element (public member function)

std::queue

Just push and pop

And a constructor or two

And a destructor and assignment operators

And peek at the first and last???

Query the capacity...

Capacity	
`empty`	checks whether the container adaptor is empty (public member function)
`size`	returns the number of elements (public member function)

std::queue

Just push and pop

And a constructor or two

And a destructor and assignment operators

And peek at the first and last???

Query the capacity...

More ways to push...

push_range (C++23)	inserts a range of elements at the end (public member function)
emplace (C++11)	constructs element in-place at the end (public member function)

std::queue

Just push and pop

And a constructor or two

And a destructor and assignment operators

And peek at the first and last???

Query the capacity...

More ways to push...

swap (C++11)	swaps the contents (public member function)

std::queue

Just push and pop

And a constructor or two

Non-member functions

`operator==`
`operator!=`
`operator<`
`operator<=`
`operator>`
`operator>=`
`operator<=>` (C++20)
	lexicographically compares the values of two queues
	(function template)

`std::swap(std::queue)` (C++11)	specializes the `std::swap` algorithm (function template)

std::queue

Just push and pop

And a constructor or two

And a destructor and assignment operators

Helper classes

std::uses_allocator<std::queue> (C++11)	specializes the std::uses_allocator type trait (class template specialization)
std::formatter<std::queue> (C++23)	formatting support for std::queue (class template specialization)

Deduction guides (since C++17)

std::queue

Template parameters

T - The type of the stored elements. The program is ill-formed if T is not the same type as `Container::value_type`.

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of *SequenceContainer*. Additionally, it must provide the following functions with the usual semantics:

- `back()`, e.g., `std::deque::back()`,
- `front()`, e.g. `std::list::front()`,
- `push_back()`, e.g., `std::deque::push_back()`,
- `pop_front()`, e.g., `std::list::pop_front()`.

The standard containers `std::deque` and `std::list` satisfy these requirements.

std::queue

Template parameters

T - The type of the stored elements. The program is ill-formed if T is not the same type as `Container::value_type`.

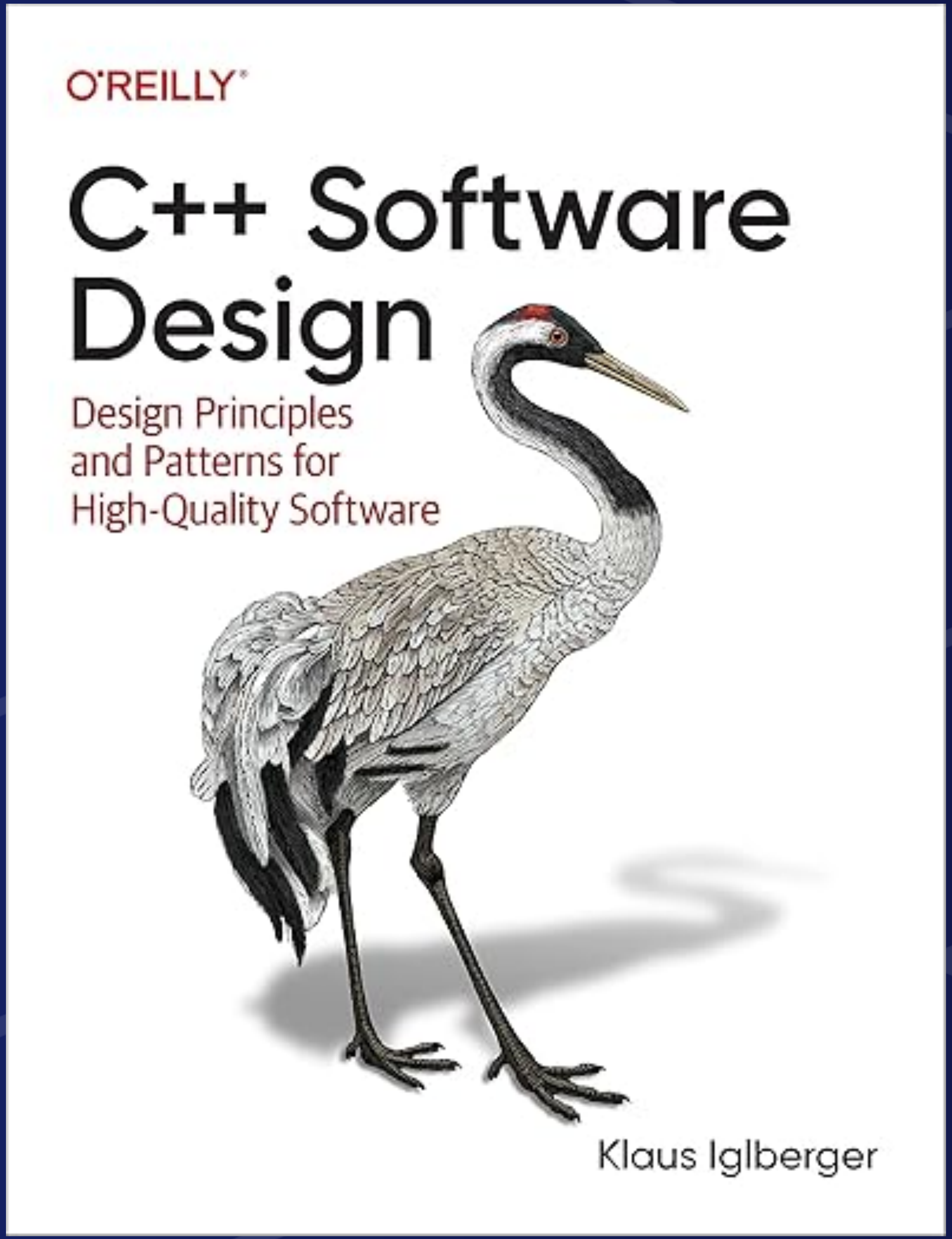
Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of *SequenceContainer*. Additionally, it must provide the following functions with the usual semantics:

- `back()`, e.g., `std::deque::back()`,
- `front()`, e.g. `std::list::front()`,
- `push_back()`, e.g., `std::deque::push_back()`,
- `pop_front()`, e.g., `std::list::pop_front()`.

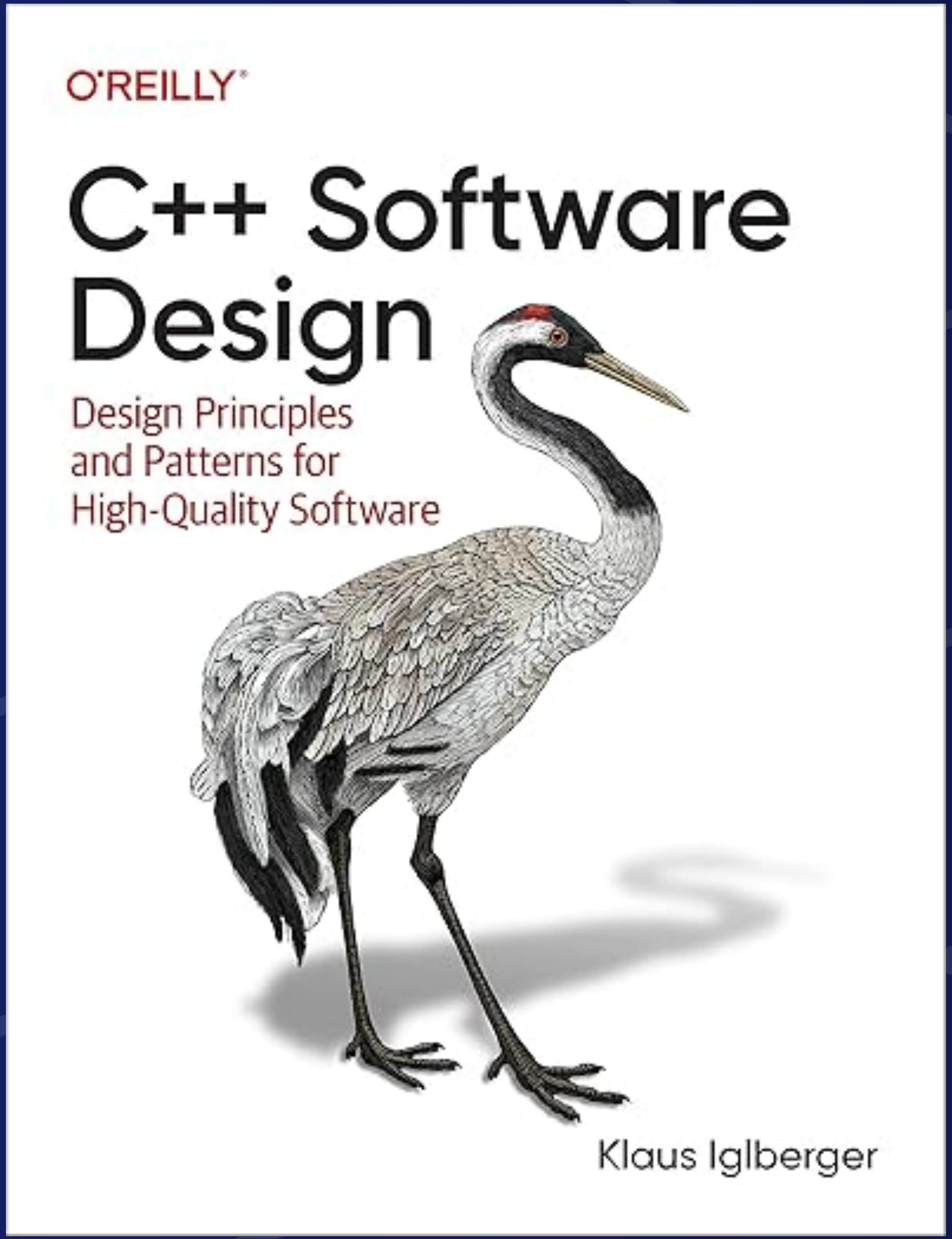
The standard containers `std::deque` and `std::list` satisfy these requirements.

Separation of Concerns

Separation of Concerns



Separation of Concerns



Separation of Concerns

- The Queue
  - Construction (metadata and buffer)
  - Synchronization (races with processes starting)
- The Producer
  - Must have an existing Queue
  - Highlander Rule
  - Can only write into queue
- The Consumer
  - Must have an existing Queue
  - Highlander Rule
  - Can only read from queue

Separation of Concerns

- The Queue
  - Construction (metadata and buffer)
  - Synchronization (races with processes starting)
- The Producer
  - Must have an existing Queue
  - Highlander Rule
  - Can only write into queue
- The Consumer
  - Must have an existing Queue
  - Highlander Rule
  - Can only read from queue

Separation of Concerns

- The Queue
  - Construction (metadata and buffer)
  - Synchronization (races with processes starting)
- The Producer
  - Must have an existing Queue
  - Highlander Rule
  - Can only write into queue
- The Consumer
  - Must have an existing Queue
  - Highlander Rule
  - Can only read from queue

Not Optional

Separation of Concerns

- The Queue
  - Construction (metadata and buffer)
  - Synchronization (races with processes starting)
- The Producer
  - Must have an existing Queue
  - Highlander Rule
  - Can only write into queue
- The Consumer
  - Must have an existing Queue
  - Highlander Rule
  - Can only read from queue

Advisory File Lock

```
struct AdvisoryFileLock
{
    struct Create { mode_t mode; };

    explicit AdvisoryFileLock(std::filesystem::path path);
    explicit AdvisoryFileLock(
        std::filesystem::path path, Create create);

    void swap(AdvisoryFileLock & that) noexcept;

    bool try_lock();
    void lock();
    void unlock() noexcept;

    std::filesystem::path const & path() const;
};
```

Characteristics

```
struct Characteristics
{
    struct Signature
    {
        std::array<char, 32> value_;

        std::string str() const
        {
            auto v = std::string_view(value_.data(), value_.size());
            return std::string(
                v.data(), std::min(v.size(), v.find('\0')));
        }

        constexpr auto operator <=> (Signature const &) const = default;
    };
```

Characteristics

```cpp
struct UUID
{
    std::array<std::byte, 16> value_;

    std::string str() const;

    static UUID make();

    constexpr auto operator <=> (UUID const &) const = default;
};

enum struct Capacity : std::uint64_t
{
};
```

Characteristics

```
enum struct MsgSize : std::uint32_t
{
};

enum struct MsgAlignment : std::uint32_t
{
};

enum struct CacheLineSize : std::uint16_t
{
};
```

Characteristics

```
enum Flags : std::uint16_t
{
    SingleProducer = 0x0001,
    MultiProducer = 0x0002,
    SingleProducerBlocks = 0x0004,
    MultiProducerBlocks = 0x0008,
    SingleConsumer = 0x0010,
    MultiConsumer = 0x0020,
    SingleConsumerBlocks = 0x0040,
    MultiConsumerBlocks = 0x0080,
    Interprocess = 0x0100,
    Multicast = 0x0200,
    Broadcast = Multicast,
    None = 0x0000,
    All = 0xffff,
};
```

MetaData

```
template <typename PidLockT>
requires ProcessIdLockC<PidLockT>
struct MetaData
{
    using PidLock = PidLockT;

    enum struct BufferOffset : std::size_t
    {
    };

    constexpr MetaData() = default;
    explicit MetaData(Characteristics const & c);
    explicit MetaData(
        Characteristics const & c,
        BufferOffset buffer_offset);
```

MetaData

```
std::unique_lock<PidLock> producer_lock()
{
    return std::unique_lock<PidLock>(producer_lock_);
}

std::unique_lock<PidLock> producer_lock(std::try_to_lock_t arg)
{
    return std::unique_lock<PidLock>(producer_lock_, arg);
}

std::unique_lock<PidLock> try_producer_lock()
{
    return producer_lock(std::try_to_lock);
}
```

MetaData

```
std::unique_lock<PidLock> consumer_lock()
{
    return std::unique_lock<PidLock>(consumer_lock_);
}

std::unique_lock<PidLock> consumer_lock(std::try_to_lock_t arg)
{
    return std::unique_lock<PidLock>(consumer_lock_, arg);
}

std::unique_lock<PidLock> try_consumer_lock()
{
    return consumer_lock(std::try_to_lock);
}
```

Process ID

```
using Value = std::conditional_t<
    std::atomic<__uint128_t>::is_always_lock_free,
    __uint128_t,
    std::uint64_t>;
Value value_;
```

Process ID

```
ProcessId
ProcessId::
current()
{
    static ProcessId id = [] {




        return ProcessId{::getpid()};
    }();
    return id;
}
```

Process ID

```
ProcessId
ProcessId::
current()
{
    static ProcessId id = [] {




        return ProcessId{::getpid()};
    }();
    return id;
}
```

Process ID

```
ProcessId
ProcessId::
current()
{
    static ProcessId id = [] {
        ::pthread_atfork(
            nullptr, // prepare
            nullptr, // parent
            [] { id = ProcessId{::getpid()}; });
        return ProcessId{::getpid()};
    }();
    return id;
}
```

Process ID

```
ProcessId
ProcessId::
current()
{
    static ProcessId id = [] {
        ::pthread_atfork(
            nullptr, // prepare
            nullptr, // parent
            [] { id = ProcessId{::getpid()}; });
        return ProcessId{::getpid()};
    }();
    return id;
}
```

Fork Handling

- Process Orchestration
  - Async-signal safe operations in the child
    - low level system calls
    - no malloc, printf, locks, stdio, most library code
    - mmap

- AtFork
  - Install and remove handlers
  - Prioritize execution

- Here be Dragons ; Real and Scary Dragons

Process ID Lock

```
template <typename BlockPolicyT>
struct TProcessIdLock
{
    using BlockPolicy = BlockPolicyT;

    TProcessIdLock() = default;

    explicit TProcessIdLock(std::in_place_t inplace);

    bool try_lock()
    {
        return pidlock_detail::try_lock_impl(
            data_.pid,
            ProcessId::current());
    }
```

Process ID Lock

```
void lock()
{
    auto const me = ProcessId::current();
    for (int i = 0; i < 10; ++i) {
        if (pidlock_detail::try_lock_impl(data_.pid, me)) {
            return;
        }
        std::this_thread::yield();
    }
    data_.block_policy().wait([&] {
        return pidlock_detail::try_lock_impl(data_.pid, me);
    });
}
```

Process ID Lock

```
void unlock()
{
   auto me = ProcessId::current();
    data_.pid.compare_exchange_strong(
        me,
        ProcessId::null());
    data_.block_policy().notify_one();
}
```

Process ID Lock

```
void unlock()
{
    auto me = ProcessId::current();
     data_.pid.compare_exchange_strong(
         me,
         ProcessId::null());
     data_.block_policy().notify_one();
}

ProcessId pid() const
{
     return data_.pid.load(std::memory_order_acquire);
}
```

Process ID Lock

```
bool try_lock_impl(wjh::Atomic<ProcessId> & pid, ProcessId const & me)
{
    auto expected = PID::null();
    if (exchange(pid, expected, me)) return true;
    if (expected != me) {
        if (auto p = PID::maybe(expected.pid());
            not p || expected != *p)
        {
            exchange(pid, expected, PID::null());
            expected = PID::null();
            if (exchange(pid, expected, me)) return true;
        }
    }
    return false;
}
```

Process ID Lock

```
bool try_lock_impl(wjh::Atomic<ProcessId> & pid, ProcessId const & me)
{
    auto expected = PID::null();
    if (exchange(pid, expected, me)) return true;
    if (expected != me) {
        if (auto p = PID::maybe(expected.pid());
            not p || expected != *p)
        {
            exchange(pid, expected, PID::null());
            expected = PID::null();
            if (exchange(pid, expected, me)) return true;
        }
    }
    return false;
}
```

Process ID Lock

```
bool try_lock_impl(wjh::Atomic<ProcessId> & pid, ProcessId const & me)
{
    auto expected = PID::null();
    if (exchange(pid, expected, me)) return true;
    if (expected != me) {
        if (auto p = PID::maybe(expected.pid());
            not p || expected != *p)
        {
            exchange(pid, expected, PID::null());
            expected = PID::null();
            if (exchange(pid, expected, me)) return true;
        }
    }
    return false;
}
```

Process ID Lock

```
bool try_lock_impl(wjh::Atomic<ProcessId> & pid, ProcessId const & me)
{
    auto expected = PID::null();
    if (exchange(pid, expected, me)) return true;
    if (expected != me) {
        if (auto p = PID::maybe(expected.pid());
            not p || expected != *p)
        {
            exchange(pid, expected, PID::null());
            expected = PID::null();
            if (exchange(pid, expected, me)) return true;
        }
    }
    return false;
}
```

Process ID Lock

```
bool try_lock_impl(wjh::Atomic<ProcessId> & pid, ProcessId const & me)
{
    auto expected = PID::null();
    if (exchange(pid, expected, me)) return true;
    if (expected != me) {
        if (auto p = PID::maybe(expected.pid());
            not p || expected != *p)
        {
            exchange(pid, expected, PID::null());
            expected = PID::null();
            if (exchange(pid, expected, me)) return true;
        }
    }
    return false;
}
```

Process ID Lock

```
bool try_lock_impl(wjh::Atomic<ProcessId> & pid, ProcessId const & me)
{
    auto expected = PID::null();
    if (exchange(pid, expected, me)) return true;
    if (expected != me) {
        if (auto p = PID::maybe(expected.pid());
            not p || expected != *p)
        {
            exchange(pid, expected, PID::null());
            expected = PID::null();
            if (exchange(pid, expected, me)) return true;
        }
    }
    return false;
}
```

Process ID Lock

```
bool try_lock_impl(wjh::Atomic<ProcessId> & pid, ProcessId const & me)
{
    auto expected = PID::null();
    if (exchange(pid, expected, me)) return true;
    if (expected != me) {
        if (auto p = PID::maybe(expected.pid());
            not p || expected != *p)
        {
            exchange(pid, expected, PID::null());
            expected = PID::null();
            if (exchange(pid, expected, me)) return true;
        }
    }
    return false;
}
```

Process ID Lock

```
bool try_lock_impl(wjh::Atomic<ProcessId> & pid, ProcessId const & me)
{
    auto expected = PID::null();
    if (exchange(pid, expected, me)) return true;
    if (expected != me) {
        if (auto p = PID::maybe(expected.pid());
            not p || expected != *p)
        {
            exchange(pid, expected, PID::null());
            expected = PID::null();
            if (exchange(pid, expected, me)) return true;
        }
    }
    return false;
}
```

Separation of Concerns

- The Queue
  - Construction (metadata and buffer)
  - Synchronization (races with processes starting)
- The Producer
  - Must have an existing Queue
  - Highlander Rule
  - Can only write into queue
- The Consumer
  - Must have and existing Queue
  - Highlander Rule
  - Can only read from queue

Producer

```
template <typename InvocableT, typename... ArgTs>
std::invoke_result_t<InvocableT, Writer &, ArgTs...>
write(InvocableT && invocable, ArgTs &&... args) noexcept(
    noexcept(std::invoke(
        std::declval<InvocableT>(),
        std::declval<Writer &>(),
        std::declval<ArgTs>()...)));
```

Producer

```
template <typename InvocableT, typename... ArgTs>
std::invoke_result_t<InvocableT, Writer &, ArgTs...>
write(InvocableT && invocable, ArgTs &&... args) noexcept(
    noexcept(std::invoke(
        std::declval<InvocableT>(),
        std::declval<Writer &>(),
        std::declval<ArgTs>()...)));
```

Producer

```
class Writer
{
    template <std::size_t N>
    void write(std::span<std::byte const, N> data) noexcept;

    template <TriviallyCopyableC T>
    T & emplace(auto &&... args) noexcept(
        std::is_nothrow_constructible_v<T, decltype(args)...>)
    requires std::constructible_from<T, decltype(args)...>;

    template <std::size_t Alignment = 1>
    void write(void const * data, std::size_t size) noexcept;
};
```

Producer

```
class Writer
{
    template <std::size_t N>
    void write(std::span<std::byte const, N> data) noexcept;

    template <TriviallyCopyableC T>
    T & emplace(auto &&... args) noexcept(
        std::is_nothrow_constructible_v<T, decltype(args)...>)
    requires std::constructible_from<T, decltype(args)...>;

    template <std::size_t Alignment = 1>
    void write(void const * data, std::size_t size) noexcept;
};
```

Producer

```
class Writer
{
    template <std::size_t N>
    void write(std::span<std::byte const, N> data) noexcept;

    template <TriviallyCopyableC T>
    T & emplace(auto &&... args) noexcept(
        std::is_nothrow_constructible_v<T, decltype(args)...>)
    requires std::constructible_from<T, decltype(args)...>;

    template <std::size_t Alignment = 1>
    void write(void const * data, std::size_t size) noexcept;
};
```

Producer

```
class Writer
{
    template <std::size_t N>
    void write(std::span<std::byte const, N> data) noexcept;

    template <TriviallyCopyableC T>
    T & emplace(auto &&... args) noexcept(
        std::is_nothrow_constructible_v<T, decltype(args)...>)
    requires std::constructible_from<T, decltype(args)...>;

    template <std::size_t Alignment = 1>
    void write(void const * data, std::size_t size) noexcept;
};
```

Separation of Concerns

- The Queue
  - Construction (metadata and buffer)
  - Synchronization (races with processes starting)
- The Producer
  - Must have an existing Queue
  - Highlander Rule
  - Can only write into queue
- The Consumer
  - Must have and existing Queue
  - Highlander Rule
  - Can only read from queue

Consumer

```
static constexpr std::uint64_t no_data = 0;
static constexpr std::uint64_t lapped = -1;

tl::expected<std::size_t, std::uint64_t> try_read(
    std::span<std::byte> data);
```

The Daugaard Queue

The Daugaard Queue

```
// Copyright 2018 Kaspar Daugaard. For educational purposes only.
// See http://daugaard.org/blog/writing-a-fast-and-versatile-spsc-ring-
buffer
```

https://www.daugaard.org/blog/

- How To Make Your SPSC Ring Buffer Do Nothing, More Efficiently
- Writing a Fast and Versatile SPSC Ring Buffer – Performance Results
- Writing a Fast and Versatile SPSC Ring Buffer

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
    SharedState m_ReaderShared;
```

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
    SharedState m_ReaderShared;
```

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
    SharedState m_ReaderShared;
```

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
    SharedState m_ReaderShared;
```

Daugaard: Local State

```
class RingBuffer { // ...
    // Writer and reader's local state.
    struct alignas(CACHE_LINE_SIZE) LocalState
    {
        LocalState()
            : buffer(nullptr), pos(0), end(0), base(0), size(0) {}

        char* buffer;
        size_t pos;
        size_t end;
        size_t base;
        size_t size;
    };

    LocalState m_Writer;
    LocalState m_Reader;
```

Daugaard: Local State

```
class RingBuffer { // ...
    // Writer and reader's local state.
    struct alignas(CACHE_LINE_SIZE) LocalState
    {
        LocalState()
            : buffer(nullptr), pos(0), end(0), base(0), size(0) {}

        char* buffer;
        size_t pos;
        size_t end;
        size_t base;
        size_t size;
    };

    LocalState m_Writer;
    LocalState m_Reader;
```

Daugaard: Local State

```
class RingBuffer { // ...
    // Writer and reader's local state.
    struct alignas(CACHE_LINE_SIZE) LocalState
    {
        LocalState()
            : buffer(nullptr), pos(0), end(0), base(0), size(0) {}

        char* buffer;
        size_t pos;
        size_t end;
        size_t base;
        size_t size;
    };

    LocalState m_Writer;
    LocalState m_Reader;
```

Daugaard: Local State

```
class RingBuffer { // ...
    // Writer and reader's local state.
    struct alignas(CACHE_LINE_SIZE) LocalState
    {
        LocalState()
            : buffer(nullptr), pos(0), end(0), base(0), size(0) {}

        char* buffer;
        size_t pos;
        size_t end;
        size_t base;
        size_t size;
    };

    LocalState m_Writer;
    LocalState m_Reader;
```

Daugaard: Local State

```
class RingBuffer { // ...
    // Writer and reader's local state.
    struct alignas(CACHE_LINE_SIZE) LocalState
    {
        LocalState()
            : buffer(nullptr), pos(0), end(0), base(0), size(0) {}

        char* buffer;
        size_t pos;
        size_t end;
        size_t base;
        size_t size;
    };

    LocalState m_Writer;
    LocalState m_Reader;
```

Daugaard: Local State

```
class RingBuffer { // ...
    // Writer and reader's local state.
    struct alignas(CACHE_LINE_SIZE) LocalState
    {
        LocalState()
            : buffer(nullptr), pos(0), end(0), base(0), size(0) {}

        char* buffer;
        size_t pos;
        size_t end;
        size_t base;
        size_t size;
    };

    LocalState m_Writer;
    LocalState m_Reader;
```

Daugaard: Write

```
// Send command to process array of items.
queue.Write(Command::ConsumeItems);
queue.Write(itemCount);
queue.WriteArray(items, itemCount);
queue.FinishWrite();
```

Daugaard: Read

```
Command cmd = queue.Read<Command>();
switch (cmd)
{
    case Command::ConsumeItems:
        int itemCount = queue.Read<int>();
        const Item* items = queue.ReadArray<Item>(itemCount);
        ConsumeItems(items, itemCount);
        queue.FinishRead();
        break;
}
```

Daugaard: Write

Daugaard: Write

```
// Send command to process array of items.
queue.Write(Command::ConsumeItems);
queue.Write(itemCount);
queue.WriteArray(items, itemCount);
queue.FinishWrite();
```

Daugaard: Write

```
// Send command to process array of items.
queue.Write(Command::ConsumeItems);
queue.Write(itemCount);
queue.WriteArray(items, itemCount);
queue.FinishWrite();
```

Daugaard: Write

```
// Send command to process array of items.
queue.Write(Command::ConsumeItems);
```

```
// Write an element to the buffer.
template <typename T>
FORCE_INLINE void Write(const T& value)
{
    void* dest = PrepareWrite(sizeof(T), alignof(T));
    new(dest) T(value);
}
```

Daugaard: Write

```
// Send command to process array of items.
queue.Write(Command::ConsumeItems);
```

```
// Write an element to the buffer.
template <typename T>
FORCE_INLINE void Write(const T& value)
{
    void* dest = PrepareWrite(sizeof(T), alignof(T));
    new(dest) T(value);
}
```

Daugaard: Write

```
void* RingBuffer::PrepareWrite(size_t size, size_t alignment)
{
    size_t pos = Align(m_Writer.pos, alignment);
    size_t end = pos + size;
    if (end > m_Writer.end)
        GetBufferSpaceToWriteTo(pos, end);
    m_Writer.pos = end;
    return m_Writer.buffer + pos;
}
```

Daugaard: Write

```
void* RingBuffer::PrepareWrite(size_t size, size_t alignment)
{
    size_t pos = Align(m_Writer.pos, alignment);
    size_t end = pos + size;
    if (end > m_Writer.end)
        GetBufferSpaceToWriteTo(pos, end);
    m_Writer.pos = end;
    return m_Writer.buffer + pos;
}
```

Daugaard: Write

```
void* RingBuffer::PrepareWrite(size_t size, size_t alignment)
{
    size_t pos = Align(m_Writer.pos, alignment);
    size_t end = pos + size;
    if (end > m_Writer.end)
        GetBufferSpaceToWriteTo(pos, end);
    m_Writer.pos = end;
    return m_Writer.buffer + pos;
}
```

Daugaard: Write

```
void* RingBuffer::PrepareWrite(size_t size, size_t alignment)
{
    size_t pos = Align(m_Writer.pos, alignment);
    size_t end = pos + size;
    if (end > m_Writer.end)
        GetBufferSpaceToWriteTo(pos, end);
    m_Writer.pos = end;
    return m_Writer.buffer + pos;
}
```

Daugaard: Write

```
void* RingBuffer::PrepareWrite(size_t size, size_t alignment)
{
    size_t pos = Align(m_Writer.pos, alignment);
    size_t end = pos + size;
    if (end > m_Writer.end)
        GetBufferSpaceToWriteTo(pos, end);
    m_Writer.pos = end;
    return m_Writer.buffer + pos;
}
```

Daugaard: Write

```
// Send command to process array of items.
queue.Write(Command::ConsumeItems);
queue.Write(itemCount);
queue.WriteArray(items, itemCount);
queue.FinishWrite();
```

Daugaard: Write

```
// Send command to process array of items.
queue.Write(Command::ConsumeItems);
queue.Write(itemCount);
queue.WriteArray(items, itemCount);
queue.FinishWrite();
```

Daugaard: Write

```
// Write an array of elements to the buffer.
template <typename T>
FORCE_INLINE void WriteArray(const T* values, size_t count)
{
    void* dest = PrepareWrite(sizeof(T) * count, alignof(T));
    for (size_t i = 0; i < count; i++)
        new(static_cast<T*>(dest) + i) T(values[i]);
}
```

```
queue.WriteArray(items, itemCount);

queue.FinishWrite();
```

Daugaard: Write

```
// Write an array of elements to the buffer.
template <typename T>
FORCE_INLINE void WriteArray(const T* values, size_t count)
{
    void* dest = PrepareWrite(sizeof(T) * count, alignof(T));
    for (size_t i = 0; i < count; i++)
        new(static_cast<T*>(dest) + i) T(values[i]);
}
```

```
queue.WriteArray(items, itemCount);

queue.FinishWrite();
```

Daugaard: Write

```
// Send command to process array of items.
queue.Write(Command::ConsumeItems);
queue.Write(itemCount);
queue.WriteArray(items, itemCount);
queue.FinishWrite();
```

Daugaard: Write

```
void RingBuffer::FinishWrite()
{
    m_WriterShared.pos.store(
        m_Writer.base + m_Writer.pos,
        std::memory_order_release);
}
```

Daugaard: Read

Daugaard: Read

```
Command cmd = queue.Read<Command>();
switch (cmd)
{
    case Command::ConsumeItems:
        int itemCount = queue.Read<int>();
        const Item* items = queue.ReadArray<Item>(itemCount);
        ConsumeItems(items, itemCount);
        queue.FinishRead();
        break;
}
```

Daugaard: Read

```
Command cmd = queue.Read<Command>();
switch (cmd)
{
    case Command::ConsumeItems:
        int itemCount = queue.Read<int>();
        const Item* items = queue.ReadArray<Item>(itemCount);
        ConsumeItems(items, itemCount);
        queue.FinishRead();
        break;
}
```

Daugaard: Read

```
Command cmd = queue.Read<Command>();
```

```
// Read an element from the buffer.
template <typename T>
FORCE_INLINE const T& Read()
{
    void* src = PrepareRead(sizeof(T), alignof(T));
    return *static_cast<T*>(src);
}
```

Daugaard: Read

```
Command cmd = queue.Read<Command>();
```

```
// Read an element from the buffer.
template <typename T>
FORCE_INLINE const T& Read()
{
    void* src = PrepareRead(sizeof(T), alignof(T));
    return *static_cast<T*>(src);
}
```

Daugaard: Read

```
Command cmd = queue.Read<Command>();
```

```
// Read an element from the buffer.
template <typename T>
FORCE_INLINE const T& Read()
{
    void* src = PrepareRead(sizeof(T), alignof(T));
    return *static_cast<T*>(src);
}
```

Daugaard: Read

```
Command cmd = queue.Read<Command>();
switch (cmd)
{
    case Command::ConsumeItems:
        int itemCount = queue.Read<int>();
        const Item* items = queue.ReadArray<Item>(itemCount);
        ConsumeItems(items, itemCount);
        queue.FinishRead();
        break;
}
```

Daugaard: Read

```
Command cmd = queue.Read<Command>();
switch (cmd)
{
    case Command::ConsumeItems:
        int itemCount = queue.Read<int>();
        const Item* items = queue.ReadArray<Item>(itemCount);
        ConsumeItems(items, itemCount);
        queue.FinishRead();
        break;
}
```

Daugaard: Read

```
Command cmd = queue.Read<Command>();
switch (cmd)
{
    case Command::ConsumeItems:
        int itemCount = queue.Read<int>();
        const Item* items = queue.ReadArray<Item>(itemCount);
```

```
template <typename T>
FORCE_INLINE const T* ReadArray(size_t count)
{
    void* src = PrepareRead(sizeof(T) * count, alignof(T));
    return static_cast<T*>(src);
}
```

Daugaard: Read

```
Command cmd = queue.Read<Command>();
switch (cmd)
{
    case Command::ConsumeItems:
        int itemCount = queue.Read<int>();
        const Item* items = queue.ReadArray<Item>(itemCount);
        ConsumeItems(items, itemCount);
        queue.FinishRead();
        break;
}
```

Daugaard: Initialization

Daugaard: Initialization

```
// Initialize. Buffer must have required alignment. Size must be a
// power of two.
void Initialize(void* buffer, size_t size)
{
    Reset();
    m_Reader.buffer = m_Writer.buffer = static_cast<char*>(buffer);
    m_Reader.size = m_Writer.size = m_Writer.end = size;
}

void Reset()
{
    m_Reader = m_Writer = LocalState();
    m_ReaderShared.pos = m_WriterShared.pos = 0;
}
```

Daugaard: Initialization

```
// Initialize. Buffer must have required alignment. Size must be a
// power of two.
void Initialize(void* buffer, size_t size)
{
    Reset();
    m_Reader.buffer = m_Writer.buffer = static_cast<char*>(buffer);
    m_Reader.size = m_Writer.size = m_Writer.end = size;
}

void Reset()
{
    m_Reader = m_Writer = LocalState();
    m_ReaderShared.pos = m_WriterShared.pos = 0;
}
```

Daugaard: Initialization

```
// Initialize. Buffer must have required alignment. Size must be a
// power of two.
void Initialize(void* buffer, size_t size)
{
    Reset();
    m_Reader.buffer = m_Writer.buffer = static_cast<char*>(buffer);
    m_Reader.size = m_Writer.size = m_Writer.end = size;
}

void Reset()
{
    m_Reader = m_Writer = LocalState();
    m_ReaderShared.pos = m_WriterShared.pos = 0;
}
```

Daugaard: Initialization

```
// Initialize. Buffer must have required alignment. Size must be a
// power of two.
void Initialize(void* buffer, size_t size)
{
    Reset();
    m_Reader.buffer = m_Writer.buffer = static_cast<char*>(buffer);
    m_Reader.size = m_Writer.size = m_Writer.end = size;
}

void Reset()
{
    m_Reader = m_Writer = LocalState();
    m_ReaderShared.pos = m_WriterShared.pos = 0;
}
```

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
    SharedState m_ReaderShared;
```

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
    SharedState m_ReaderShared;
```

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
    SharedState m_ReaderShared;
```

Implicit Lifetime

Implicit Lifetime

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

X * p = static_cast<X *>(std::malloc(sizeof(X)));
p->a = 1;
p->b = 2;
```

Implicit Lifetime

10 Some operations are described as *implicitly creating objects* within a specified region of storage. For each operation that is specified as implicitly creating objects, that operation implicitly creates and starts the lifetime of zero or more objects of implicit-lifetime types (6.8) in its specified region of storage if doing so would result in the program having defined behavior. If no such set of objects would give the program defined behavior, the behavior of the program is undefined. If multiple such sets of objects would give the program defined behavior, it is unspecified which such set of objects is created. [*Note*: Such operations do not start the lifetimes of subobjects of such objects that are not themselves of implicit-lifetime types. — *end note*]

Implicit Lifetime

10 Some operations are described as *implicitly creating objects* within a specified region of storage. For each operation that is specified as implicitly creating objects, that operation implicitly creates and starts the lifetime of zero or more objects of implicit-lifetime types (6.8) in its specified region of storage if doing so would result in the program having defined behavior. If no such set of objects would give the program defined behavior, the behavior of the program is undefined. If multiple such sets of objects would give the program defined behavior, it is unspecified which such set of objects is created. [*Note*: Such operations do not start the lifetimes of subobjects of such objects that are not themselves of implicit-lifetime types. — *end note*]

Implicit Lifetime

10 Some operations are described as *implicitly creating objects* within a specified region of storage. For each operation that is specified as implicitly creating objects, that operation implicitly creates and starts the lifetime of zero or more objects of implicit-lifetime types (6.8) in its specified region of storage if doing so would result in the program having defined behavior. If no such set of objects would give the program defined behavior, the behavior of the program is undefined. If multiple such sets of objects would give the program defined behavior, it is unspecified which such set of objects is created. [*Note*: Such operations do not start the lifetimes of subobjects of such objects that are not themselves of implicit-lifetime types. — *end note*]

Implicit Lifetime

12 [*Example:*

```
#include <cstdlib>
struct X { int a, b; };
X *make_x() {
  // The call to std::malloc implicitly creates an object of type X
  // and its subobjects a and b, and returns a pointer to that X object
  // (or an object that is pointer-interconvertible (6.8.2) with it),
  // in order to give the subsequent class member access operations
  // defined behavior.
  X *p = (X*)std::malloc(sizeof(struct X));
  p->a = 1;
  p->b = 2;
  return p;
}
```

— *end example*]

Implicit Lifetime

12 [*Example*:

```
#include <cstdlib>
struct X { int a, b; };
X *make_x() {
  // The call to std::malloc implicitly creates an object of type X
  // and its subobjects a and b, and returns a pointer to that X object
  // (or an object that is pointer-interconvertible (6.8.2) with it),
  // in order to give the subsequent class member access operations
  // defined behavior.
  X *p = (X*)std::malloc(sizeof(struct X));
  p->a = 1;
  p->b = 2;
  return p;
}
```

— *end example*]

Implicit Lifetime

9 Arithmetic types (6.8.1), enumeration types, pointer types, pointer-to-member types (6.8.2), `std::nullptr_t`, and cv-qualified (6.8.3) versions of these types are collectively called *scalar types*. Scalar types, trivially copyable class types (11.2), arrays of such types, and cv-qualified versions of these types are collectively called *trivially copyable types*. Scalar types, trivial class types (11.2), arrays of such types and cv-qualified versions of these types are collectively called *trivial types*. Scalar types, standard-layout class types (11.2), arrays of such types and cv-qualified versions of these types are collectively called *standard-layout types*. Scalar types, implicit-lifetime class types (11.2), array types, and cv-qualified versions of these types are collectively called *implicit-lifetime types*.

Implicit Lifetime

9 A class `S` is an *implicit-lifetime class* if it is an aggregate or has at least one trivial eligible constructor and a trivial, non-deleted destructor.

Implicit Lifetime

9 A class `S` is an *implicit-lifetime class* if it is an aggregate or has at least one trivial eligible constructor and a trivial, non-deleted destructor.

Implicit Lifetime

9 A class `S` is an *implicit-lifetime class* if it is an aggregate $\boxed{\text{or has at least one trivial eligible constructor}}$ and a trivial, non-deleted destructor.

Implicit Lifetime

9 A class `S` is an *implicit-lifetime class* if it is an aggregate or has at least one trivial eligible constructor and a trivial, non-deleted destructor.

```
class Foo
{
    // ...
    Foo(tag_t) = default;
};
```

Implicit Lifetime

9 A class `S` is an *implicit-lifetime class* if it is an aggregate or has at least one trivial eligible constructor and a trivial, non-deleted destructor.

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
    SharedState m_ReaderShared;
```

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
    SharedState m_ReaderShared;
```

5 [*Note*: Operations that are lock-free should also be address-free. That is, atomic operations on the same memory location via two different addresses will communicate atomically. The implementation should not depend on any per-process state. This restriction enables communication by memory that is mapped into a process more than once and by memory that is shared between two processes. — *end note*]

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
    SharedState m_ReaderShared;
```

5 [*Note*: Operations that are lock-free should also be address-free. That is, atomic operations on the same memory location via two different addresses will communicate atomically. The implementation should not depend on any per-process state. This restriction enables communication by memory that is mapped into a process more than once and by memory that is shared between two processes. — *end note*]

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
```

std::atomic<T>::atomic

`atomic() noexcept = default;`	(1)	(since C++11) (until C++20)
`constexpr atomic() noexcept(std::is_nothrow_default_constructible_v<T>);`		(since C++20)
`constexpr atomic( T desired ) noexcept;`	(2)	(since C++11)
`atomic( const atomic& ) = delete;`	(3)	(since C++11)

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
```

std::atomic<T>::atomic

`atomic() noexcept = default;`	(1)	(since C++11) (until C++20)
`constexpr atomic() noexcept(std::is_nothrow_default_constructible_v<T>);`	(1)	(since C++20)
`constexpr atomic( T desired ) noexcept;`	(2)	(since C++11)
`atomic( const atomic& ) = delete;`	(3)	(since C++11)

The Hinnant Rule

compiler implicitly declares

user declares

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Pop Quiz

```
struct Foo
{
    void operator = (Foo &&) = delete;
};
```

Pop Quiz

```
struct Foo
{
    void operator = (Foo &&) = delete;
};

static_assert(std::is_default_constructible_v<Foo>);
static_assert(std::is_copy_constructible_v<Foo>);
static_assert(std::is_move_constructible_v<Foo>);
static_assert(std::is_copy_assignable_v<Foo>);
static_assert(std::is_move_assignable_v<Foo>);
static_assert(std::is_destructible_v<Foo>);
```

The Hinnant Rule

compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        std::atomic<size_t> pos;
    };

    SharedState m_WriterShared;
```

std::atomic<T>::atomic

`atomic() noexcept = default;`	(1)	(since C++11) (until C++20)
`constexpr atomic() noexcept(std::is_nothrow_default_constructible_v<T>);`		(since C++20)
`constexpr atomic( T desired ) noexcept;`	(2)	(since C++11)
`atomic( const atomic& ) = delete;`	(3)	(since C++11)

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        size_t pos;
    };

    SharedState m_WriterShared;
    SharedState m_ReaderShared;
```

Daugaard: Shared State

```
class RingBuffer
{
    // Writer and reader's shared positions.
    struct alignas(CACHE_LINE_SIZE) SharedState
    {
        Atomic<size_t> pos;
    };

    SharedState m_WriterShared;
    SharedState m_ReaderShared;
```

Implementing Your Own Atomics

BEN SAKS

Cppcon
The C++ Conference

2025
September 13 - 19

Implicit Lifetime

- Messages must be implicit lifetime
- Must become comfortable with this for sharing data across processes

Cache Lines

```
#define CACHE_LINE_SIZE 64
```

Daugaard: Cache Lines

```
#if defined(CACHE_LINE_SIZE)
inline constexpr std::size_t cache_line_size = CACHE_LINE_SIZE;
#elif defined(__cpp_lib_hardware_interference_size) && \
    (__cpp_lib_hardware_interference_size >= 201703L)
inline constexpr std::size_t cache_line_size =
    std::hardware_destructive_interference_size;
#else
    #error "cache_line_size cannot be defined"
#endif
```

Daugaard: Cache Lines

Is std::hardware_destructive_interference_size right for Macs?

Asked 2 months ago Modified 2 months ago Viewed 124 times

5

Question, but also PSA.

The latest version of apple-clang has dropped: `Apple clang version 17.0.0 (clang-1700.0.13.3)`

It finally has `std::hardware_destructive_interference_size` and `std::hardware_constructive_interference_size`, but they seem wrong, at least on my architecture.

I'd expect 128, but they provide 64. Is my understanding correct? Is apple-clang wrong?

Is my understanding that the results for `std::hardware_destructive_interference_size

```
$ cat /tmp/a.cpp && g++ -std=c++20 /tmp/a.cpp && ./a.out && sysctl hw.cachelines
#include <iostream>
#include <new>

int main()
{
    std::cout << std::hardware_destructive_interference_size << '\n';
    std::cout << std::hardware_constructive_interference_size << '\n';
}
64
64
hw.cachelinesize: 128
```

Magic Buffer

1 MB

Magic Buffer

1 MB

```
struct Args
{
    void * addr = nullptr;
    size_type len = 0;
    int fd = -1;
    size_type offset = 0;
    bool magic = false;
    size_type page_size = 0;
};
```

Magic Buffer

2 MB

```
// Perform an anonymous mapping of twice the requested length.
auto const saved_fd = std::exchange(args.fd, -1);
auto const offset = std::exchange(args.offset, 0);
args.len *= 2;
args.magic = false;
auto mapped = do_mapping(args);
```

Magic Buffer

1 MB	1 MB

```
// Map the file over the first half of the anonymous mapping.
args.fd = saved_fd;
args.len /= 2;
args.offset = offset;
args.addr = mapped->address;
auto tmp = do_mapping(args);
tmp->release();
```

Magic Buffer


```
// Map the file over the second half of the anonymous mapping.
args.addr = static_cast<std::byte *>(mapped->address) + args.len;
tmp = do_mapping(args);
tmp->release();

addr_ = std::bit_cast<pointer>(mapped->address);
size_ = mapped->length / 2;
mapped_ = std::move(mapped);
```

Magic Buffer

1 MB 1 MB

```
static constexpr int prot = PROT_READ | PROT_WRITE;
int const flags = [&] {
    int f = MAP_SHARED;
    if (addr) f |= MAP_FIXED;
    if (fd < 0) f |= MAP_ANONYMOUS;
    return f;
}();
auto const mapped = ::mmap(
    addr, len, prot, flags, fd, off_t(offset));
```

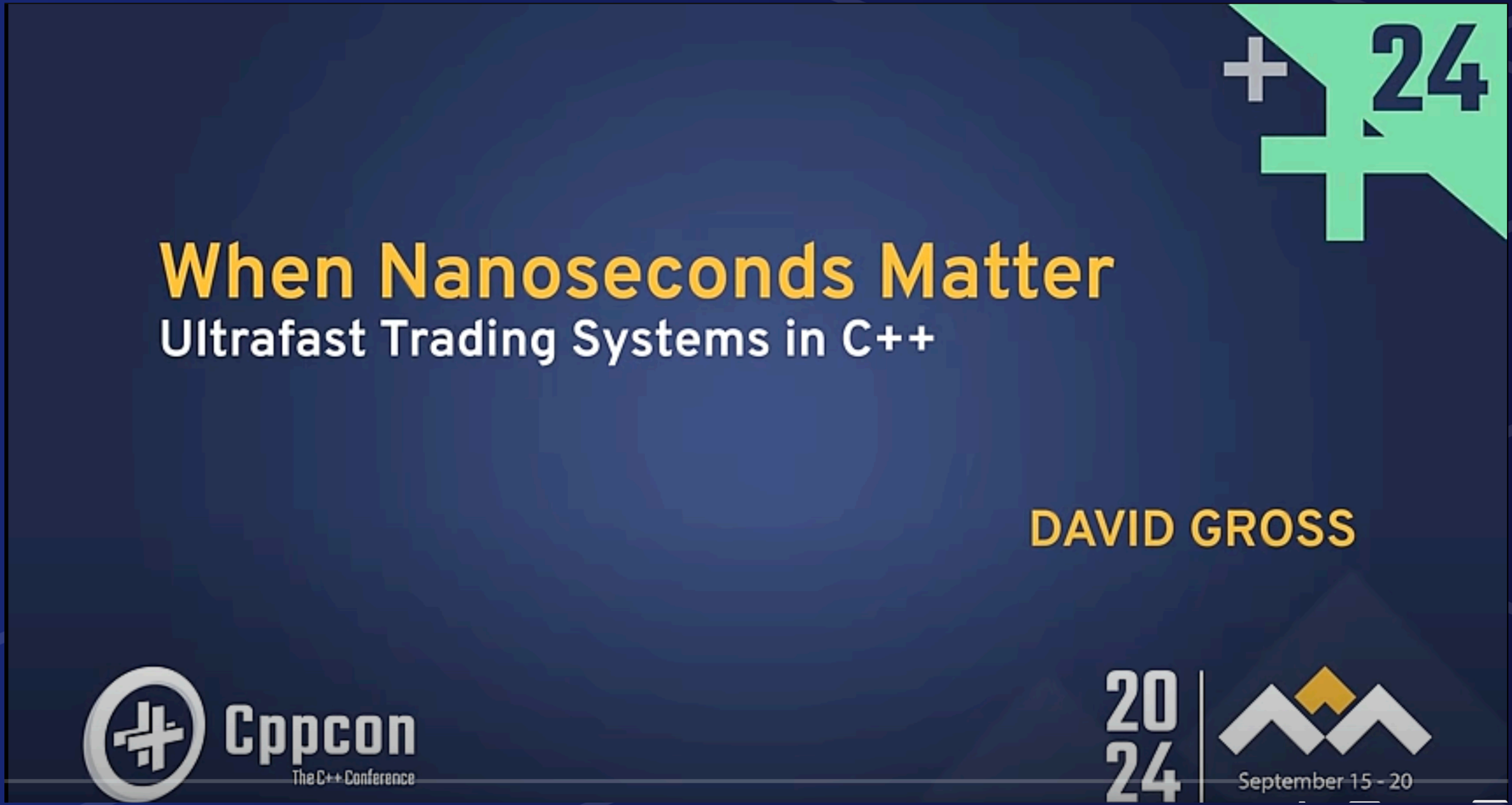
MCAST Queues

MCAST Queues

- Single Producer, Multiple Consumers
- Every consumer gets every message (mcast, broadcast)
- Producer never blocks
- Slow consumers may miss messages
- Most really fast ones have some kind of UB (seqlock, etc)

MCAST Queues

MCAST Queues



MCAST Queues

- Single Producer, Multiple Consumers
- Every consumer gets every message (mcast, broadcast)
- Producer never blocks
- Slow consumers may miss messages
- Most really fast ones have some kind of UB (seqlock, etc)

MCAST Queues

- The Queue
  - Lock for becoming the producer
  - If enabled, a lock for becoming the special consumer

MCAST Producer

```
template <typename InvocableT, typename... ArgTs>
std::invoke_result_t<InvocableT, Producer::Writer &, ArgTs...>
Producer::
write(InvocableT && invocable, ArgTs &&... args)
noexcept(noexcept(std::invoke(
    std::declval<InvocableT>(),
    std::declval<Writer &>(),
    std::declval<ArgTs>()...)))
{
    using R = std::invoke_result_t<InvocableT, Writer &, ArgTs...>;
    auto writer = Writer(info_);
```

MCAST Producer

```
template <typename InvocableT, typename... ArgTs>
std::invoke_result_t<InvocableT, Producer::Writer &, ArgTs...>
Producer::
write(InvocableT && invocable, ArgTs &&... args)
noexcept(noexcept(std::invoke(
    std::declval<InvocableT>(),
    std::declval<Writer &>(),
    std::declval<ArgTs>()...)))
{
    using R = std::invoke_result_t<InvocableT, Writer &, ArgTs...>;
    auto writer = Writer(info_);
```

MCAST Producer

```
    auto call = [&]() -> R {
        return std::invoke(
            std::forward<InvocableT>(invocable),
            writer,
            std::forward<ArgTs>(args)...);
    };
    if constexpr (std::is_same_v<R, void>) {
        call();
        info_.finish_write();
        return;
    } else {
        R result = call();
        info_.finish_write();
        return result;
    }
}
```

MCAST Producer

```
    auto call = [&]() -> R {
        return std::invoke(
            std::forward<InvocableT>(invocable),
            writer,
            std::forward<ArgTs>(args)...);
    };
    if constexpr (std::is_same_v<R, void>) {
        call();
        info_.finish_write();
        return;
    } else {
        R result = call();
        info_.finish_write();
        return result;
    }
}
```

MCAST Producer

```
template <std::size_t N>
void
Producer::Writer::
write(std::span<std::byte const, N> data) noexcept
{
    auto const size = std::uint32_t(data.size());
    assert(size == data.size());
    auto buffer = info_.prepare_buffer<8>(sizeof(std::uint64_t) + size);
    std::memcpy(buffer.data(), &size, sizeof(size));
    std::memcpy(
        buffer.data() + sizeof(std::uint64_t), data.data(), size);
    info_.local_counter += buffer.size();
}
```

MCAST Producer

```
template <std::size_t N>
void
Producer::Writer::
write(std::span<std::byte const, N> data) noexcept
{
    auto const size = std::uint32_t(data.size());
    assert(size == data.size());
    auto buffer = info_.prepare_buffer<8>(sizeof(std::uint64_t) + size);
    std::memcpy(buffer.data(), &size, sizeof(size));
    std::memcpy(
        buffer.data() + sizeof(std::uint64_t), data.data(), size);
    info_.local_counter += buffer.size();
}
```

MCAST Producer

```
template <std::size_t N>
void
Producer::Writer::
write(std::span<std::byte const, N> data) noexcept
{
    auto const size = std::uint32_t(data.size());
    assert(size == data.size());
    auto buffer = info_.prepare_buffer<8>(sizeof(std::uint64_t) + size);
    std::memcpy(buffer.data(), &size, sizeof(size));
    std::memcpy(
        buffer.data() + sizeof(std::uint64_t), data.data(), size);
    info_.local_counter += buffer.size();
}
```

MCAST Producer

```
template <std::size_t N>
void
Producer::Writer::
write(std::span<std::byte const, N> data) noexcept
{
    auto const size = std::uint32_t(data.size());
    assert(size == data.size());
    auto buffer = info_.prepare_buffer<8>(sizeof(std::uint64_t) + size);
    std::memcpy(buffer.data(), &size, sizeof(size));
    std::memcpy(
        buffer.data() + sizeof(std::uint64_t), data.data(), size);
    info_.local_counter += buffer.size();
}
```

MCAST Producer

```
template <std::size_t Alignment>
void
Producer::Writer::
write(void const * data, std::size_t size) noexcept
{
    auto buffer = info_.prepare_buffer<Alignment>(size);
    std::memcpy(buffer.data(), data, size);
    info_.local_counter += size;
}
```

MCAST Producer

```
template <std::size_t Alignment>
void
Producer::Writer::
write(void const * data, std::size_t size) noexcept
{
    auto buffer = info_.prepare_buffer<Alignment>(size);
    std::memcpy(buffer.data(), data, size);
    info_.local_counter += size;
}
```

MCAST Producer

```
template <std::size_t Alignment>
void
Producer::Writer::
write(void const * data, std::size_t size) noexcept
{
    auto buffer = info_.prepare_buffer<Alignment>(size);
    std::memcpy(buffer.data(), data, size);
    info_.local_counter += size;
}
```

MCAST Producer

```
template <std::size_t Alignment>
void
Producer::Writer::
write(void const * data, std::size_t size) noexcept
{
    auto buffer = info_.prepare_buffer<Alignment>(size);
    std::memcpy(buffer.data(), data, size);
    info_.local_counter += size;
}
```

MCAST Producer

```
template <TriviallyCopyableC T>
T &
Producer::Writer::
emplace(auto &&... args) noexcept(
    std::is_nothrow_constructible_v<T, decltype(args)...>)
requires std::constructible_from<T, decltype(args)...>
{
    auto buffer = info_.prepare_buffer<alignof(T)>(sizeof(T));
    auto t = new (static_cast<void *>(buffer.data()))
        T(std::forward<decltype(args)>(args)...);
    info_.local_counter += sizeof(T);
    return *t;
}
```

MCAST Producer

```
template <TriviallyCopyableC T>
T &
Producer::Writer::
emplace(auto &&... args) noexcept(
    std::is_nothrow_constructible_v<T, decltype(args)...>)
requires std::constructible_from<T, decltype(args)...>
{
    auto buffer = info_.prepare_buffer<alignof(T)>(sizeof(T));
    auto t = new (static_cast<void *>(buffer.data()))
        T(std::forward<decltype(args)>(args)...);
    info_.local_counter += sizeof(T);
    return *t;
}
```

MCAST Producer

```
template <TriviallyCopyableC T>
T &
Producer::Writer::
emplace(auto &&... args) noexcept(
    std::is_nothrow_constructible_v<T, decltype(args)...>)
requires std::constructible_from<T, decltype(args)...>
{
    auto buffer = info_.prepare_buffer<alignof(T)>(sizeof(T));
    auto t = new (static_cast<void *>(buffer.data()))
        T(std::forward<decltype(args)>(args)...);
    info_.local_counter += sizeof(T);
    return *t;
}
```

MCAST Producer

```
template <TriviallyCopyableC T>
T &
Producer::Writer::
emplace(auto &&... args) noexcept(
    std::is_nothrow_constructible_v<T, decltype(args)...>)
requires std::constructible_from<T, decltype(args)...>
{
    auto buffer = info_.prepare_buffer<alignof(T)>(sizeof(T));
    auto t = new (static_cast<void *>(buffer.data()))
        T(std::forward<decltype(args)>(args)...);
    info_.local_counter += sizeof(T);
    return *t;
}
```

MCAST Producer

```
inline Producer::Writer::
Writer(Info & x)
: info_(x)
{
    // Save the counter, so we can restore it later, if the write
    // operation was aborted for some reason including a user exception.
    info_.saved_counter = info_.local_counter;
}
```

MCAST Producer

```
inline Producer::Writer::
~Writer()
{
    // Restore the counter. Safe to do always because the tmp_counter is
    // set to the local_counter when finish_write is called.
    info_.local_counter = info_.saved_counter;
}
```

MCAST Producer

```
inline void
Producer::Info::
finish_write()
{
    shared->read_counter.store(
        local_counter, std::memory_order_release);

    // Update the saved_counter because the write operation has been
    // committed, and there is no longer a need to restore the original
    // value.
    saved_counter = local_counter;
}
```

MCAST Consumer

```
inline tl::expected<std::size_t, std::int_fast64_t>
Consumer::
try_read(std::span<std::byte> data)
{
    auto read_counter = aligned<Alignment::Int64>(info_.local_counter);
    std::uint32_t size;
    if (read_counter + sizeof(size) >= info_.cached_counter) {
        info_.cached_counter = info_.shared->read_counter.load(
            std::memory_order_acquire);
        if (read_counter + sizeof(size) > info_.cached_counter) {
            return tl::unexpected(no_data);
        }
    }
    auto const buf = info_.buffer + (read_counter & info_.mask);
    std::memcpy(&size, buf, sizeof(size));
```

MCAST Consumer

```
inline tl::expected<std::size_t, std::int_fast64_t>
Consumer::
try_read(std::span<std::byte> data)
{
    auto read_counter = aligned<Alignment::Int64>(info_.local_counter);
    std::uint32_t size;
    if (read_counter + sizeof(size) >= info_.cached_counter) {
        info_.cached_counter = info_.shared->read_counter.load(
            std::memory_order_acquire);
        if (read_counter + sizeof(size) > info_.cached_counter) {
            return tl::unexpected(no_data);
        }
    }
    auto const buf = info_.buffer + (read_counter & info_.mask);
    std::memcpy(&size, buf, sizeof(size));
```

MCAST Consumer

```
inline tl::expected<std::size_t, std::int_fast64_t>
Consumer::
try_read(std::span<std::byte> data)
{
    auto read_counter = aligned<Alignment::Int64>(info_.local_counter);
    std::uint32_t size;
    if (read_counter + sizeof(size) >= info_.cached_counter) {
        info_.cached_counter = info_.shared->read_counter.load(
            std::memory_order_acquire);
        if (read_counter + sizeof(size) > info_.cached_counter) {
            return tl::unexpected(no_data);
        }
    }
    auto const buf = info_.buffer + (read_counter & info_.mask);
    std::memcpy(&size, buf, sizeof(size));
```

MCAST Consumer

```
inline tl::expected<std::size_t, std::int_fast64_t>
Consumer::
try_read(std::span<std::byte> data)
{
    auto read_counter = aligned<Alignment::Int64>(info_.local_counter);
    std::uint32_t size;
    if (read_counter + sizeof(size) >= info_.cached_counter) {
        info_.cached_counter = info_.shared->read_counter.load(
            std::memory_order_acquire);
        if (read_counter + sizeof(size) > info_.cached_counter) {
            return tl::unexpected(no_data);
        }
    }
    auto const buf = info_.buffer + (read_counter & info_.mask);
    std::memcpy(&size, buf, sizeof(size));
```

MCAST Consumer

```
inline tl::expected<std::size_t, std::int_fast64_t>
Consumer::
try_read(std::span<std::byte> data)
{
    auto read_counter = aligned<Alignment::Int64>(info_.local_counter);
    std::uint32_t size;
    if (read_counter + sizeof(size) >= info_.cached_counter) {
        info_.cached_counter = info_.shared->read_counter.load(
            std::memory_order_acquire);
        if (read_counter + sizeof(size) > info_.cached_counter) {
            return tl::unexpected(no_data);
        }
    }
    auto const buf = info_.buffer + (read_counter & info_.mask);
    std::memcpy(&size, buf, sizeof(size));
```

MCAST Consumer

```
inline tl::expected<std::size_t, std::int_fast64_t>
Consumer::
try_read(std::span<std::byte> data)
{
    auto read_counter = aligned<Alignment::Int64>(info_.local_counter);
    std::uint32_t size;
    if (read_counter + sizeof(size) >= info_.cached_counter) {
        info_.cached_counter = info_.shared->read_counter.load(
            std::memory_order_acquire);
        if (read_counter + sizeof(size) > info_.cached_counter) {
            return tl::unexpected(no_data);
        }
    }
    auto const buf = info_.buffer + (read_counter & info_.mask);
    std::memcpy(&size, buf, sizeof(size));
```

MCAST Consumer

```
    if (size > data.size()) {
        return tl::unexpected(is_lapped() ? lapped :
                                            std::int_fast64_t(size));
    }
    read_counter += sizeof(std::uint64_t) + size;
    if (read_counter > info_.cached_counter) {
        return tl::unexpected(no_data);
    }
    std::memcpy(data.data(), buf + sizeof(std::uint64_t), size);
    if (is_lapped()) {
        return tl::unexpected(lapped);
    }
    info_.local_counter = read_counter;
    return size;
}
```

MCAST Consumer

```
    if (size > data.size()) {
        return tl::unexpected(is_lapped() ? lapped :
                                            std::int_fast64_t(size));
    }
    read_counter += sizeof(std::uint64_t) + size;
    if (read_counter > info_.cached_counter) {
        return tl::unexpected(no_data);
    }
    std::memcpy(data.data(), buf + sizeof(std::uint64_t), size);
    if (is_lapped()) {
        return tl::unexpected(lapped);
    }
    info_.local_counter = read_counter;
    return size;
}
```

MCAST Consumer

```
    if (size > data.size()) {
        return tl::unexpected(is_lapped() ? lapped :
                                           std::int_fast64_t(size));
    }
    read_counter += sizeof(std::uint64_t) + size;
    if (read_counter > info_.cached_counter) {
        return tl::unexpected(no_data);
    }
    std::memcpy(data.data(), buf + sizeof(std::uint64_t), size);
    if (is_lapped()) {
        return tl::unexpected(lapped);
    }
    info_.local_counter = read_counter;
    return size;
}
```

MCAST Consumer

```
    if (size > data.size()) {
        return tl::unexpected(is_lapped() ? lapped :
                                            std::int_fast64_t(size));
    }
    read_counter += sizeof(std::uint64_t) + size;
    if (read_counter > info_.cached_counter) {
        return tl::unexpected(no_data);
    }
    std::memcpy(data.data(), buf + sizeof(std::uint64_t), size);
    if (is_lapped()) {
        return tl::unexpected(lapped);
    }
    info_.local_counter = read_counter;
    return size;
}
```