

C++ now

# An Adventure in Modern Library Design

Robert Leahy

2024

FGBL SI 20240606 PS

1. Build a time series database engine
- 2.
- 3.

1. Build a time series database engine
2. Insert market data into time series database
- 3.

1. Build a time series database engine
2. Insert market data into time series database
3. Query time series database

# Time Series Database?

In general

- Stores data in temporal order

- Efficiently retrieves data between arbitrary times

Our time series database

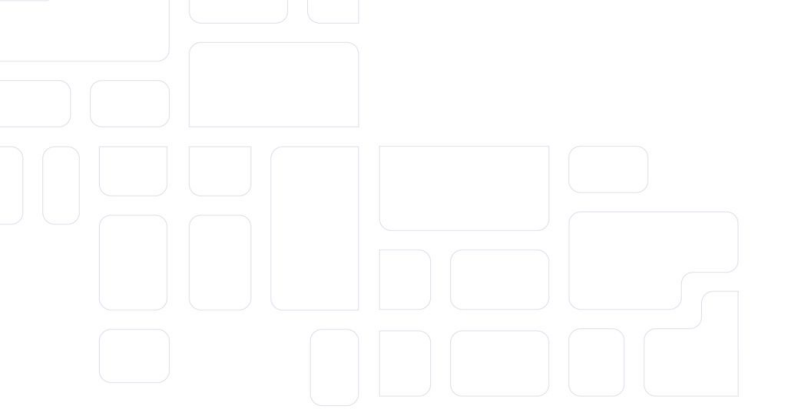
- Two levels of keying in addition to temporal key

  - File

  - Symbol (each symbol in each file is its own time series)

- Live/intra-day insertion and querying supported

```
#pragma pack(push)
#pragma pack(4)
struct erased_update {
    using timestamp_type = std::uint64_t;
    using length_type = std::uint32_t;
    timestamp_type timestamp;
    length_type length : 24;
    length_type kind : 8;
};
#pragma pack(pop)
```



```
struct symbol {  
    char name[24];  
};
```



```
enum class update_status {  
    none_for_now,  
    complete,  
    processed  
};
```

```
struct open_query {  
    using timestamp_type = erased_update::timestamp_type;  
    const timestamp_type begin;  
    const timestamp_type end;  
    const symbol instrument;  
    explicit open_query(symbol, timestamp_type, timestamp_type);  
    result<update_status> run() noexcept;  
    const erased_update* last_update() const noexcept;  
};
```

# Query Cardinality

One `open_query` retrieves updates for

- One symbol within
- One file across
- One time range

# Query Cardinality

One `open_query` retrieves updates for

- One symbol within

- One file across

- One time range

Useful queries may involve

- Multiple symbols within

- Multiple files across

- Multiple time ranges (less common)

# Streaming merge sort



Trade @  $t_2$

Trade @  $t_0$

⋮

Order @  $t_3$

Order @  $t_1$



Trade @  $t_4$

Trade @  $t_2$

Trade @  $t_0$

Order @  $t_3$

Order @  $t_1$



Trade @  $t_4$

Trade @  $t_2$

⋮

Order @  $t_1$

Trade @  $t_0$

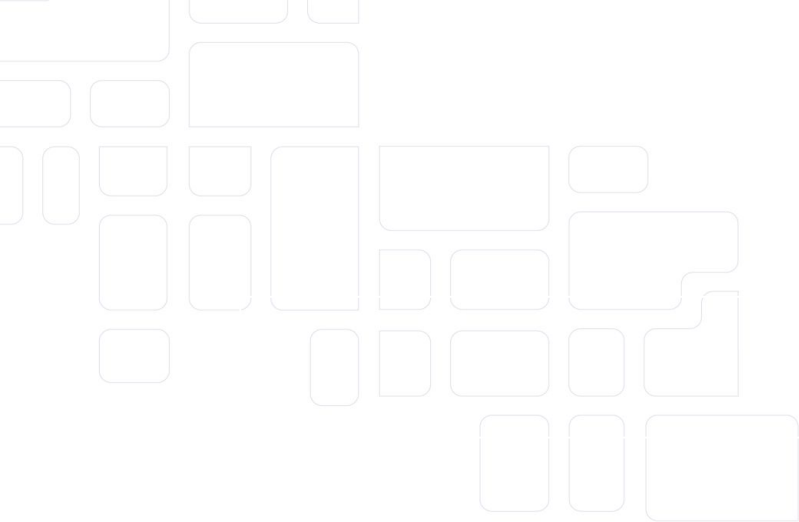
⋮

Order @  $t_5$

Order @  $t_3$



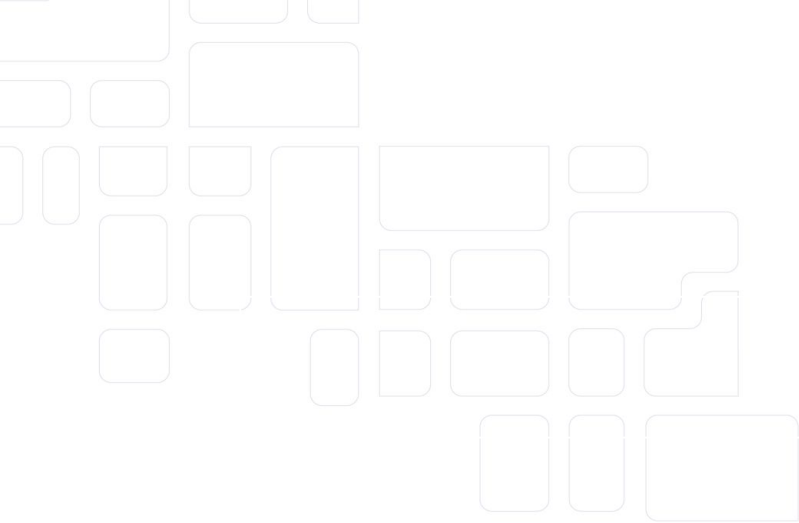




⋮

Order @  $t_1$

Order @  $t_0$



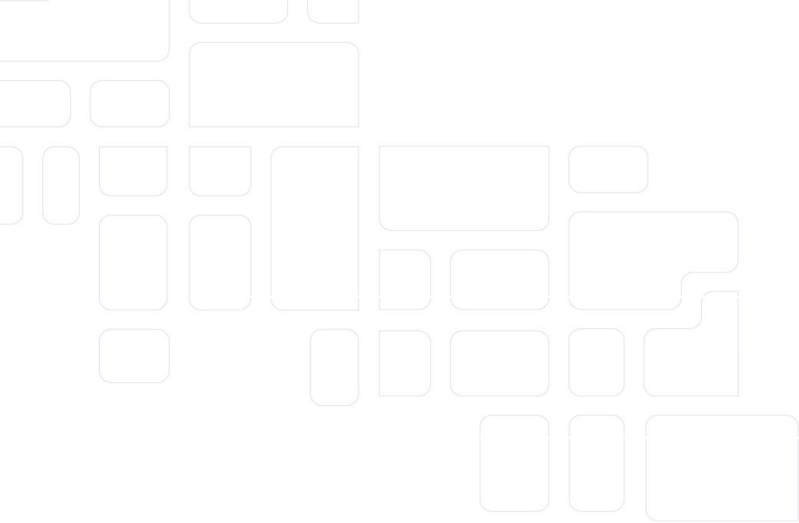
⋮

Order @  $t_2$

Order @  $t_1$

Order @  $t_0$

A black arrow pointing from the text "Order @  $t_1$ " down and to the left towards the text "Order @  $t_0$ ".



⋮

Order @  $t_3$

Order @  $t_2$



Order @  $t_1$

Order @  $t_0$

```
struct merge_sort {  
    void push_back(open_query& query);  
    struct run_type {  
        update_status status;  
        open_query* processed;  
    };  
    result<run_type> run();  
};
```



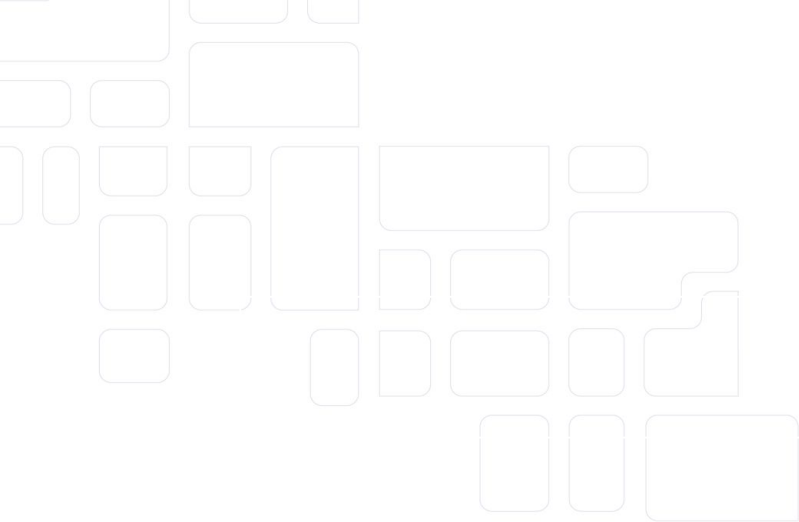
none\_for\_now

⋮

Order @  $t_1$

Order @  $t_0$

```
class open_query {  
    using timestamp_type = erased_update::timestamp_type;  
    const timestamp_type begin;  
    const timestamp_type end;  
    const symbol instrument;  
  
    timestamp_type most_recent_known_timestamp;  
  
    constexpr open_query(symbol, timestamp_type, timestamp_type);  
    result<update_status> run() noexcept;  
    const erased_update* last_update() const noexcept;  
};
```

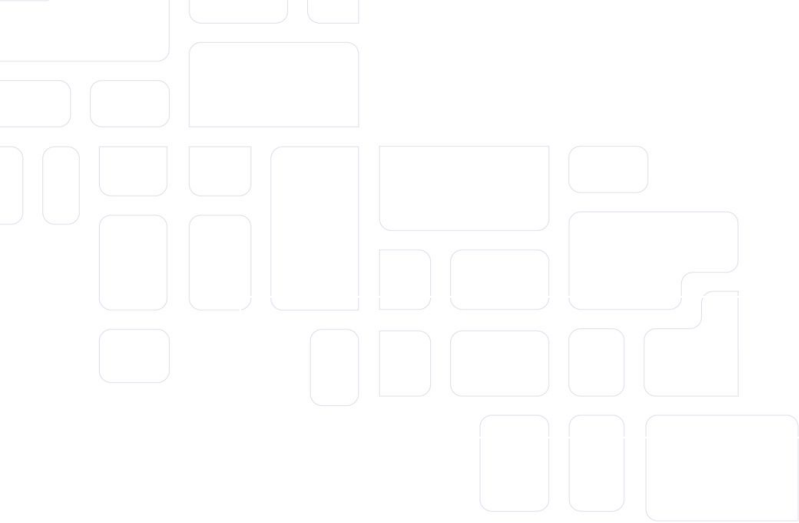


none\_for\_now @  $t_2$

⋮

Order @  $t_1$

Order @  $t_0$



none\_for\_now @  $t_2$

Order @  $t_0$

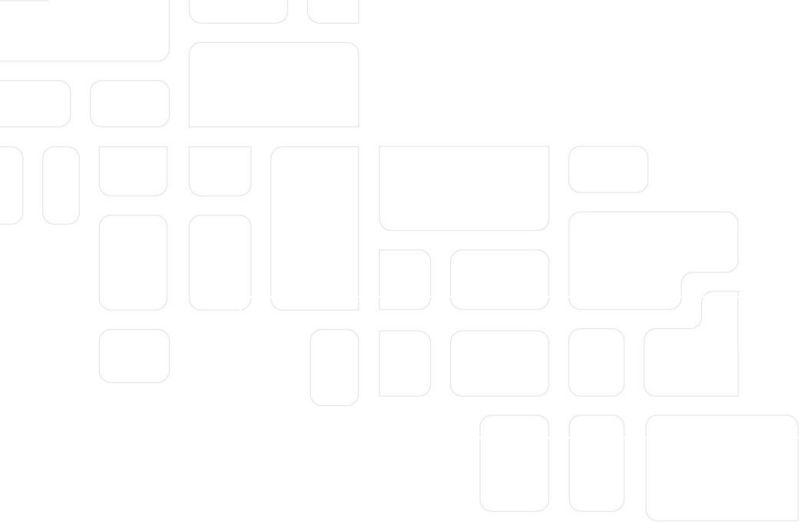


Order @  $t_1$

Order @  $t_3$

⋮





none\_for\_now @  $t_2$

Order @  $t_1$

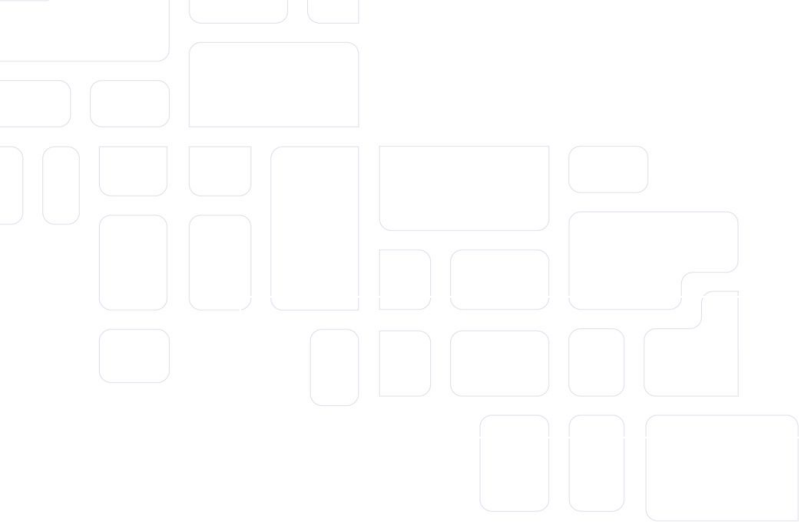
Order @  $t_0$

⋮

Order @  $t_4$

Order @  $t_3$





none\_for\_now @  $t_5$

Order @  $t_3$

Order @  $t_1$

Order @  $t_0$

⋮

Order @  $t_6$

Order @  $t_4$



# Update Heterogeneity

`erased_update` provides common header for entries in the time series

Trades and orders (for example) store different data

Trade and order updates are therefore schematically distinct

Need a technique for dealing with streams of heterogeneous updates



```
template<typename As>  
result<const As*> update_as(const erased_update* u) noexcept;
```

# Taking a Byte Out of C++

Avoiding Punning by Starting Lifetimes

ROBERT LEAHY



20  
22



```
template<typename Update>
struct update_as_query : open_query {
    using update_type = Update;
    using open_query::open_query;
    result<update_status> run() noexcept;
    const Update* last_update() const noexcept;
};
```

# Failing Successfully: Reporting and Handling Errors

ROBERT LEAHY

```
template<typename Update>
struct update_as_query : open_query {
    using update_type = Update;
    using open_query::open_query;
    result<update_status> run() noexcept;
    const Update* last_update() const noexcept;

    const erased_update* raw_last_update() const noexcept;

};
```



```
template<typename Query>
    requires requires(const Query& q) { q.raw_last_update(); }
decltype(auto) raw_last_update(const Query& q) noexcept(
    noexcept(q.raw_last_update()))
{
    return q.raw_last_update();
}
```

```
template<typename Query>
    requires requires(const Query& q) { q.raw_last_update(); }
decltype(auto) raw_last_update(const Query& q) noexcept(
    noexcept(q.raw_last_update()))
{
    return q.raw_last_update();
}
```

```
template<typename Query>
decltype(auto) raw_last_update(const Query& q) noexcept(
    noexcept(q.last_update()))
{
    return q.last_update();
}
```

# Incremental Computation

Databases should be able to handle many queries simultaneously

Should not allow a single query to consume all resources

Queries should periodically yield the CPU

+ 21

# Deploying the Networking TS

ROBERT LEAHY



20  
21



# Incremental Computation

Databases should be able to handle many queries simultaneously

Should not allow a single query to consume all resources

Queries should periodically yield the CPU

`open_query::run` does this by returning after every result

# Incremental Computation

Databases should be able to handle many queries simultaneously

Should not allow a single query to consume all resources

Queries should periodically yield the CPU

`open_query::run` does this by returning after every result

What about filtering?

```
const std::vector<int> v{6, 5, 4, 3, 4, 5, 6};  
const auto pred = [](const int i) noexcept {  
    return i > 4;  
};  
for (auto& i : v | std::ranges::views::filter(pred)) {  
    std::cout << i << std::endl;  
}
```

# Filtering

[0]	[1]	[2]	[3]	[4]	[5]	[6]
6	5	4	3	4	5	6

Stepping forward from index 0 requires one increment

Stepping from index 1 requires four increments

Range model requires that a step produce a value

- “Step” is `operator++`

- “Produce a value” implies validity of `operator*`

Filtering large data sets with very particular predicates leads to hogging CPU

- Because `operator++` doesn't return after processing one result

- Because `operator*` needs to be valid



# Solution

Allow `last_update` to return `nullptr`

How can we sort when `nullptr` is returned?

Add `last_position` member

Similar to adding `raw_last_update` member

Also requires `last_position` free function

Similar to requirement for `raw_last_update` free function

# Dynamically-Expanding Queries

Market data is sent on multiple channels

- So the database has a manifest of channels

Market data is not necessarily sent keyed on instrument name

- So the database contains a map of name to ID

Each query has three tiers

- For each channel

- Lookup the instrument, and if it's found

- Query the actual market data of interest

# Throw away the throwaway prototype

# New Model

Obtain stream of results incrementally by performing single step at a time

Single step of work ends in one of four ways:

1. Error
2. Result
3. Position
  - Yielded without generating a result (can try again immediately)
  - Results unavailable at this time (try again later)
4. Completion

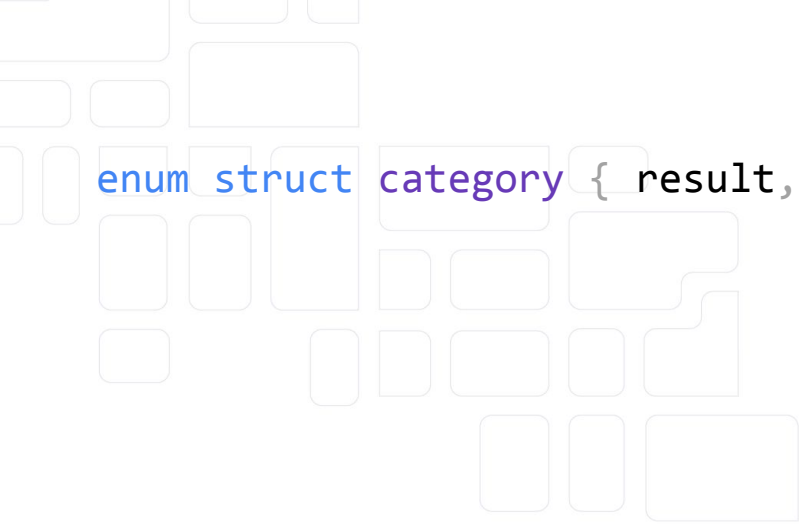
# New Model

Single step yields result of the form

```
outcome::result<std::variant<Args...>>
```

Error in `outcome::result<...>`

Result, position, and completion in `std::variant<Args...>`



```
enum struct category { result, position, completion };
```

```
enum struct category { result, position, completion };  
  
template<typename>  
inline constexpr auto category_of = category::result;
```

```
enum struct category { result, position, completion };

template<typename>
inline constexpr auto category_of = category::result;

template<typename T, category Category>
concept category_of_c = category_of<std::remove_cvref_t<T>> == Category;
```



```
enum struct category { result, position, completion };

template<typename>
inline constexpr auto category_of = category::result;

template<typename T, category Category>
concept category_of_c = category_of<std::remove_cvref_t<T>> == Category;

template<typename T>
concept result = category_of_c<T, category::result>;
template<typename T>
concept position = category_of_c<T, category::position>;
template<typename T>
concept completion = category_of_c<T, category::completion>;
```

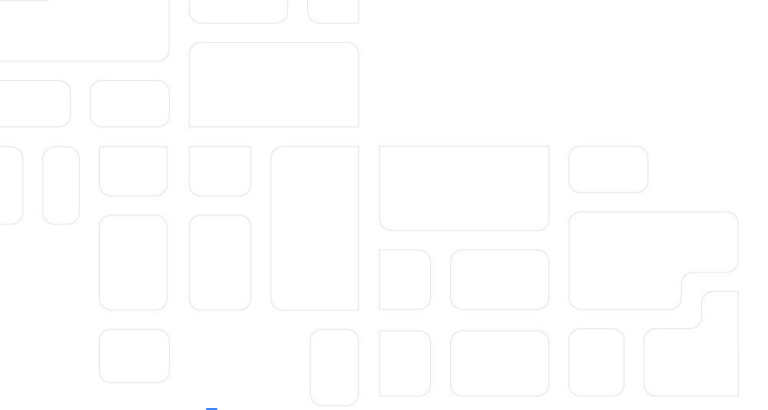
A subexpression

$s$

denotes a stream if the expression

$\text{run}(s)$

is well-formed



```
template<typename T>
concept stream = requires(T&& t) {
    run(std::forward<T>(t));
};
```

The name `run` denotes a customization point object.

Let *do-run*(*s*) evaluate the expression:

- `s.run()` if that expression is well-formed and yields an object type, otherwise
- `run(s)` if that expression is well-formed and yields an object type where the meaning of `run` is established as-if by performing argument-dependent lookup only.

Otherwise, *do-run*(*s*) is ill-formed.

```
namespace detail::run {
```

```
void run();
```

```
}
```

```
namespace detail::run {
```

```
void run();
```

```
template<typename T>
```

```
concept run_result = std::is_object_v<T>;
```

```
}
```

```
namespace detail::run {
```

```
void run();
```

```
template<typename T>
```

```
concept run_result = std::is_object_v<T>;
```

```
template<typename T>
```

```
concept member = requires(T&& t) { { std::forward<T>(t).run() } -> run_result; };
```

```
}
```

```

namespace detail::run {
    void run();

    template<typename T>
    concept run_result = std::is_object_v<T>;
    template<typename T>
    concept member = requires(T&& t) { { std::forward<T>(t).run() } -> run_result; };
    template<typename T>
    concept free_function = !member<T> && requires(T&& t) {
        { run(std::forward<T>(t)) } -> run_result;
    };
}

```



```

namespace detail::run {
    void run();

    template<typename T>
    concept run_result = std::is_object_v<T>;
    template<typename T>
    concept member = requires(T&& t) { { std::forward<T>(t).run() } -> run_result; };
    template<typename T>
    concept free_function = !member<T> && requires(T&& t) {
        { run(std::forward<T>(t)) } -> run_result;
    };
    template<typename T>
    concept c = member<T> || free_function<T>;
}

```

```

namespace detail::run {
    void run();

    template<typename T>
    concept run_result = std::is_object_v<T>;
    template<typename T>
    concept member = requires(T&& t) { { std::forward<T>(t).run() } -> run_result; };
    template<typename T>
    concept free_function = !member<T> && requires(T&& t) {
        { run(std::forward<T>(t)) } -> run_result;
    };
    template<typename T>
    concept c = member<T> || free_function<T>;

    template<member T>
    constexpr decltype(auto) do_run(T&& t) noexcept(noexcept(std::forward<T>(t).run())) {
        return std::forward<T>(t).run();
    }
}

```

```

namespace detail::run {
    void run();

    template<typename T>
    concept run_result = std::is_object_v<T>;
    template<typename T>
    concept member = requires(T&& t) { { std::forward<T>(t).run() } -> run_result; };
    template<typename T>
    concept free_function = !member<T> && requires(T&& t) {
        { run(std::forward<T>(t)) } -> run_result;
    };
    template<typename T>
    concept c = member<T> || free_function<T>;

    template<member T>
    constexpr decltype(auto) do_run(T&& t) noexcept(noexcept(std::forward<T>(t).run())) {
        return std::forward<T>(t).run();
    }
    template<free_function T>
    constexpr decltype(auto) do_run(T&& t) noexcept(noexcept(run(std::forward<T>(t)))) {
        return run(std::forward<T>(t));
    }
}

```

Document	P1895R0
Date	2019-10-07
Reply To	Lewis Baker < <a href="mailto:lbaker@fb.com">lbaker@fb.com</a> > Eric Niebler < <a href="mailto:eniebler@fb.com">eniebler@fb.com</a> > Kirk Shoop < <a href="mailto:kirkshoop@fb.com">kirkshoop@fb.com</a> >
Audience	LEWG
Target	C++23

# tag\_invoke: A general pattern for supporting customisable functions

## Abstract

Modern customization point objects ([`customization.point.object`]) were a step forward over raw ADL for making libraries customizable. However, there are a couple of problems they leave unsolved:

1. Each one internally dispatches via ADL to a free function of the same name, which has the effect

# Member customization points for Senders and Receivers

## Abstract

There have been various suggestions that Senders and Receivers need a new language feature for customization points, to avoid the complexity of ADL `tag_invoke`.

This paper makes the case that C++ already has such a language facility, and it works just fine for the purposes of Senders and Receivers.

That language facility is member functions.

In a nutshell, the approach in this paper is relatively straightforward; for all non-query customization points, ADL `tag_invoke` overloads become member functions. Query customization points become query member functions that take the query tag as an argument.

Let  $e$  be the expression  $do-run(s)$ , and the type  $E$  be  $decltype((e))$ . Then  $run(s)$  is expression-equivalent to:

- $e$ , if that expression is well-formed and  $E$  is of the form  $outcome::result<std::variant<Args...>>$ , otherwise
- $outcome::result<std::variant<Args...>>(e)$  if that expression is well-formed and  $E$  is of the form  $std::variant<Args...>$ , otherwise
- $outcome::result<std::variant<T>>(e.assume\_value())$  if that expression is well-formed,  $E$  is of the form  $outcome::result<T>$ , and  $bool(e)$  is true, otherwise
- $outcome::result<std::variant<T>>(e.as\_failure())$  if that expression is well-formed, and  $E$  is of the form  $outcome::result<T>$ , otherwise
- $outcome::result<std::variant<T>>(e)$  if that expression is well-formed.

Otherwise,  $run(s)$  is ill-formed.

```
namespace detail::run {
```

```
template<typename... Args>
```

```
void wrap(outcome::result<std::variant<Args...>&&) = delete;
```

```
}
```

```
namespace detail::run {  
  
template<typename... Args>  
void wrap(outcome::result<std::variant<Args...>>&&) = delete;  
template<typename... Args>  
constexpr outcome::result<std::variant<Args...>> wrap(std::variant<Args...>&& v)  
    noexcept((std::is_nothrow_move_constructible_v<Args> && ...))  
{  
    return outcome::success(std::move(v));  
}  
  
}
```



```

namespace detail::run {
    template<typename... Args>
    void wrap(outcome::result<std::variant<Args...>>&&) = delete;
    template<typename... Args>
    constexpr outcome::result<std::variant<Args...>> wrap(std::variant<Args...>&& v)
        noexcept((std::is_nothrow_move_constructible_v<Args> && ...))
    {
        return outcome::success(std::move(v));
    }
    template<typename T>
    constexpr outcome::result<std::variant<T>> wrap(outcome::result<T>&& r) noexcept(
        std::is_nothrow_move_constructible_v<T>)
    {
        if (r) return outcome::success(std::move(r).assume_value());
        return std::move(r).as_failure();
    }
}

```

```

namespace detail::run {

template<typename... Args>
void wrap(outcome::result<std::variant<Args...>>&&) = delete;

template<typename... Args>
constexpr outcome::result<std::variant<Args...>> wrap(std::variant<Args...>&& v)
    noexcept((std::is_nothrow_move_constructible_v<Args> && ...))
{
    return outcome::success(std::move(v));
}

template<typename T>
constexpr outcome::result<std::variant<T>> wrap(outcome::result<T>&& r) noexcept(
    std::is_nothrow_move_constructible_v<T>)
{
    if (r) return outcome::success(std::move(r).assume_value());
    return std::move(r).as_failure();
}

template<typename T>
constexpr outcome::result<std::variant<T>> wrap(T&& t) noexcept(
    std::is_nothrow_move_constructible_v<T>)
{
    return outcome::success(std::move(t));
}

}

```

```
namespace detail::run {  
  
template<typename T>  
concept almost_wrappable = c<T> && requires(T&& t) {  
    run::wrap(run::do_run(std::forward<T>(t)));  
};  
  
}
```

```
namespace detail::run {  
  
template<typename T>  
concept almost_wrappable = c<T> && requires(T&& t) {  
    run::wrap(run::do_run(std::forward<T>(t)));  
};  
  
template<typename T>  
concept wrappable = almost_wrappable<T> && requires(T&& t) {  
    { run::wrap(run::do_run(std::forward<T>(t))) } -> std::move_constructible;  
};  
  
}
```

A decorative background in the top-left corner consisting of a grid of rounded rectangles in various sizes and shades of gray, some overlapping.

```
inline constexpr struct {
```

```
} run{};
```

```
inline constexpr struct {  
    template<detail::run::c T>  
    static constexpr auto operator()(T&& t) noexcept(  
        noexcept(detail::run::do_run(std::forward<T>(t)))  
    )  
    {  
        return detail::run::do_run(std::forward<T>(t));  
    }  
}
```

```
} run{};
```

```
inline constexpr struct {
    template<detail::run::c T>
    static constexpr auto operator()(T&& t) noexcept(
        noexcept(detail::run::do_run(std::forward<T>(t))))
    {
        return detail::run::do_run(std::forward<T>(t));
    }
    template<detail::run::wrappable T>
    static constexpr auto operator()(T&& t) noexcept(
        noexcept(detail::run::wrap(detail::run::do_run(std::forward<T>(t)))))
    {
        return detail::run::wrap(detail::run::do_run(std::forward<T>(t)));
    }
} run{};
```

```
inline constexpr struct {
    template<detail::run::c T>
    static constexpr auto operator()(T&& t) noexcept(
        noexcept(detail::run::do_run(std::forward<T>(t))))
    {
        return detail::run::do_run(std::forward<T>(t));
    }
    template<detail::run::wrappable T>
    static constexpr auto operator()(T&& t) noexcept(
        noexcept(detail::run::wrap(detail::run::do_run(std::forward<T>(t)))))
    {
        return detail::run::wrap(detail::run::do_run(std::forward<T>(t)));
    }
    template<detail::run::almost_wrappable T>
    static void operator()(T&&) = delete;
} run{};
```



Does this actually solve any of our problems?

# Provoking Problem

Snapshots previously supported at

- Receipt timestamp, or
- Exchange timestamp

Some exchanges have concept of transaction

- Multiple updates which must be reckoned atomically

Obtain snapshot automatically at beginning of enclosing transaction

# Solution With Streams

Given the ability to create a streams which retrieve

- Channel metadata

- Symbol metadata

- Closest transaction begin

- Snapshot stream

Combine them with “branch” primitive such that

- Channel metadata branches to

- Symbol metadata which branches to

- Closest transaction begin which branches to

- Snapshot

Let's filter



```
template<typename T>  
using run_result_t = decltype(run(std::declval<T>()));
```



```
template<typename T>  
using run_result_t = decltype(run(std::declval<T>()));
```

```
template<typename T>  
using run_result_value_type_t = typename run_result_t<T>::value_type;
```

```
template<category, typename T>
struct as : T {
    using T::T;
};
```

```
template<category Category, typename Base>
inline constexpr auto category_of<as<Category, Base>> = Category;
```

# Filtering

Inner stream yields `outcome::result<std::variant<Args...>>`

What should filtering stream yield?



# Filtering

Inner stream yields `outcome::result<std::variant<Args...>>`

What should filtering stream yield?

```
template<typename>  
struct filter_result;
```

```
template<typename... Args>  
struct filter_result<outcome::result<std::variant<Args...>>>  
    : std::type_identity<outcome::result<std::variant<  
        Args...,  
        as<category::position, Args>...>>>  
{};
```

```
struct foo {};  
struct bar {};  
struct baz {};  
  
template<>  
constexpr inline auto category_of<bar> = category::position;  
template<>  
constexpr inline auto category_of<baz> = category::completion;  
  
struct example {  
    std::variant<foo, as<category::position, foo>, bar, baz> run();  
};
```

```
std::variant<  
    foo,  
    as<category::position, foo>,  
    bar,  
    baz,  
    as<category::position, foo>,  
    as<category::position, as<category::position, foo>>,  
    as<category::position, bar>,  
    as<category::position, baz>>;
```

```
template<category Category>
struct make_matching_predicate {
    template<typename T>
    using fn = std::bool_constant<category_of<T> != Category>;
};
```



MAYSTREET

# Higher order metafunction



```
category category_of(const std::string_view sv) noexcept;
```

```
category category_of(const std::string_view sv) noexcept;

auto make_matching_predicate(const category c) noexcept {
    return [c](const std::string_view sv) noexcept {
        return category_of(sv) == c;
    };
}
```

```

category category_of(const std::string_view sv) noexcept;

auto make_matching_predicate(const category c) noexcept {
    return [c](const std::string_view sv) noexcept {
        return category_of(sv) == c;
    };
}


int main() {
    for (auto&& result :
        std::vector<std::string>{"foo", "as<category::position, foo>", "bar", "baz"} |
        std::ranges::views::filter(make_matching_predicate(category::result)))
    {
        std::cout << result << std::endl;
    }
}

```




```
namespace detail::results_of {  
  
template<category Category>  
struct make_matching_predicate {  
    template<typename T>  
        using fn = std::bool_constant<streams::category_of<T> != Category>;  
};  
  
}
```

```
namespace detail::results_of {  
  
template<category Category>  
struct make_matching_predicate {  
    template<typename T>  
    using fn = std::bool_constant<streams::category_of<T> != Category>;  
};  
  
}  
  
template<category Category, typename Variant>  
using results_of_t = ::boost::mp11::mp_remove_if<  
    Variant,  
    detail::results_of::make_matching_predicate<Category>::template fn>;
```



```
template<category Category, typename Variant>
using results_of_t = ::boost::mp11::mp_remove_if<
    Variant,
    detail::results_of::make_matching_predicate<Category>::template fn>;
```



```
template<category Category, typename Variant>
using results_of_t = ::boost::mp11::mp_remove_if_q<
    Variant,
    detail::results_of::make_matching_predicate<Category>>>;
```

```
template<category Category>
struct make_as {
    template<typename T>
    using fn = as<Category, T>;
};
```

```
template<typename Inner, typename Predicate>
class filter {
    Inner inner_;
    Predicate pred_;
    using inner_result_type_ = run_result_value_type_t<Inner>;
```

```
    // ...
};
```

```
template<typename Inner, typename Predicate>
class filter {
    Inner inner_;
    Predicate pred_;
    using inner_result_type_ = run_result_value_type_t<Inner>;
    using inner_results_type_ = results_of_t<
        category::result,
        inner_result_type_>;
```

```
    // ...
};
```

```
template<typename Inner, typename Predicate>
class filter {
    Inner inner_;
    Predicate pred_;
    using inner_result_type_ = run_result_value_type_t<Inner>;
    using inner_results_type_ = results_of_t<
        category::result,
        inner_result_type_>;
    using filtered_types_ = ::boost::mp11::mp_transform_q<
        make_as<category::position>,
        inner_results_type_>;

    // ...
};
```



```
template<typename Inner, typename Predicate>
class filter {
    Inner inner_;
    Predicate pred_;
    using inner_result_type_ = run_result_value_type_t<Inner>;
    using inner_results_type_ = results_of_t<
        category::result,
        inner_result_type_>;
    using filtered_types_ = ::boost::mp11::mp_transform_q<
        make_as<category::position>,
        inner_results_type_>;
    using result_type_ = ::boost::mp11::mp_unique<
        ::boost::mp11::mp_append<
            inner_result_type_,
            filtered_types_>>;
    // ...
};
```

```
template<typename Inner, typename Predicate>
class filter {
    // ...
public:
    constexpr outcome::result<result_type_> run() noexcept(/* ... */) {
        OUTCOME_TRY(auto res, streams::run(inner_));

    }
    // ...
};
```

```

template<typename Inner, typename Predicate>
class filter {
    // ...
public:
    constexpr outcome::result<result_type_> run() noexcept(/* ... */) {
        OUTCOME_TRY(auto res, streams::run(inner_));
        return std::visit(

            , std::move(res));
    }
    // ...
};

```

Document number:	P0051R3=yy-nnnn
Date:	2018-02-12
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá < <a href="mailto:vicente.botet@nokia.com">vicente.botet@nokia.com</a> >

## C++ generic overload function (Revision 3)

Experimental overload function for C++. This paper proposes one function that allow to overload lambdas or function objects, but also member and non-member functions.

There will be another proposal to take care of grouping lambdas or function objects, member and non-member functions so that the first viable match is selected when a call is done.

```
template<typename... Functions>
struct overload : private Functions... {
    overload() = default;
    template<typename... Args>
        requires std::conjunction_v<std::is_constructible<Functions, Args&&>...>
        constexpr explicit overload(Args&&... args) noexcept(
            std::conjunction_v<std::is_nothrow_constructible<Functions, Args&&>...>)
            : Functions(std::forward<Args>(args))...
    {}
    using Functions::operator()...;
};

template<typename... Args>
explicit overload(Args...) -> overload<Args...>;
```

```

template<typename Inner, typename Predicate>
class filter {
    // ...
public:
    constexpr outcome::result<result_type_> run() noexcept(/* ... */) {
        OUTCOME_TRY(auto res, streams::run(inner_));
        return std::visit(overload(

            ), std::move(res));
    }
    // ...
};

```

```

template<typename Inner, typename Predicate>
class filter {
    // ...
public:
    constexpr outcome::result<result_type_> run() noexcept(/* ... */) {
        OUTCOME_TRY(auto res, streams::run(inner_));
        return std::visit(overload(
            [](auto&& r) constexpr noexcept(/* ... */) -> outcome::result<result_type_> {
                return result_type_(std::forward<decltype(r)>(r));
            },
            std::move(res));
    }
    // ...
};

```

```

template<typename Inner, typename Predicate>
class filter {
    // ...
public:
    constexpr outcome::result<result_type_> run() noexcept(/* ... */) {
        OUTCOME_TRY(auto res, streams::run(inner_));
        return std::visit(overload(
            [](auto&& r) constexpr noexcept(/* ... */) -> outcome::result<result_type_> {
                return result_type_(std::forward<decltype(r)>(r));
            },
            [&](result auto&& r) constexpr noexcept(/* ... */) ->
                outcome::result<result_type_>
            {

                })), std::move(res));
    }
    // ...
};

```



```

template<typename Inner, typename Predicate>
class filter {
    // ...
public:
    constexpr outcome::result<result_type_> run() noexcept(/* ... */) {
        OUTCOME_TRY(auto res, streams::run(inner_));
        return std::visit(overload(
            [](auto&& r) constexpr noexcept(/* ... */) -> outcome::result<result_type_> {
                return result_type_(std::forward<decltype(r)>(r));
            },
            [&](result auto&& r) constexpr noexcept(/* ... */) ->
                outcome::result<result_type_>
            {
                if (std::invoke(pred_, std::as_const(r))) {
                    return result_type_(std::forward<decltype(r)>(r));
                }

                return result_type_{};
            }
        ), std::move(res));
    }
    // ...
};


```

```

template<typename Inner, typename Predicate>
class filter {
    // ...
public:
    constexpr outcome::result<result_type_> run() noexcept(/* ... */) {
        OUTCOME_TRY(auto res, streams::run(inner_));
        return std::visit(overload(
            [](auto&& r) constexpr noexcept(/* ... */) -> outcome::result<result_type_> {
                return result_type_(std::forward<decltype(r)>(r));
            },
            [&](result auto&& r) constexpr noexcept(/* ... */) ->
                outcome::result<result_type_>
            {
                if (std::invoke(pred_, std::as_const(r))) {
                    return result_type_(std::forward<decltype(r)>(r));
                }
                return as<category::position, std::remove_cvref_t<decltype(r)>>>(
                    std::forward<decltype(r)>(r));
            })), std::move(res));
    }
    // ...
};


```

```
template<typename Inner, typename Predicate>
class filter {
    // ...
    template<typename T>
        requires std::constructible_from<Inner, T&&>
    explicit constexpr filter(T&& t, Predicate pred) noexcept(/* ... */)
        : inner_(std::forward<T>(t)),
          pred_(std::move(pred))
    {}
};
```



```
template<typename Inner, typename Predicate>  
explicit filter(Inner, Predicate) -> filter<Inner, Predicate>;
```

# What about immovable types?



```
template<typename T>
    requires std::constructible_from<Inner, T&&>
explicit constexpr filter(T&& t, Predicate pred) noexcept(/* ... */);
```

What about immovable types that aren't unary constructible?

```
template<typename Query>
struct query {
    template<typename... Args>
        requires std::constructible_from<Query, Args&&...>
    explicit query(Args&&... args) noexcept(
        std::is_nothrow_constructible_v<Query, Args&&...>);
    // Because our results contain a reference to the
    // inner query it must remain stable and therefore
    // we cannot be movable
    query(const query&) = delete;
    query& operator=(const query&) = delete;
    // `...
};
```



```
const auto pred = [](const auto& r) {  
    // ...  
};  
filter<query<open_query>, decltype(pred)> f(  
    symbol{},  
    1714483800000000000ULL,  
    1714509000000000000ULL,  
    pred);
```

# What about immovable types and guaranteed RVO?



```
query<open_query> create_query();
```

```
query<open_query> create_query();  
// ...  
const auto pred = [](const auto& r) {  
    // ...  
};  
filter<query<open_query>, decltype(pred)> f(create_query(), pred);
```

Document number: P0927R2

Date: 2018-10-05

Reply-to: James Dennett <[jdennett@google.com](mailto:jdennett@google.com)><sup>1</sup>  
Geoff Romer <[gromer@google.com](mailto:gromer@google.com)>

Audience: Evolution Working Group

# Towards A (Lazy) Forwarding Mechanism for C++

## Synopsis

By allowing function declarations to specify that certain arguments are to be passed lazily, we can reduce the need for the preprocessor and simplify and improve generic code that forwards

# Lazy Forwarding Workaround

Need to...

- ...materialize prvalue directly into storage

- ...satisfy `std::constructible_from<Inner, T&&>`

# Lazy Forwarding Workaround

Need to...

...materialize prvalue directly into storage

...satisfy `std::constructible_from<Inner, T&&>`

## Implicit conversion operator

```
query<open_query> create_query();  
// ...  
struct elider {  
    operator query<open_query>() const {  
        return create_query();  
    }  
};  
const auto pred = [] (const auto& r) {  
    // ...  
};  
filter<query<open_query>, decltype(pred)> f(elider{}, pred);
```



```
template<typename Function>
struct elide {
    explicit constexpr elide(Function f) noexcept(
        std::is_nothrow_move_constructible_v<Function>)
        : f_(std::move(f))
    {}
    template<typename Self>
    constexpr operator decltype(std::declval<Self>().f_())(this Self&& self)
        noexcept(noexcept(std::forward<Self>(self).f_()))
    {
        return std::forward<Self>(self).f_();
    }
private:
    Function f_;
};
```

```
query<open_query> create_query();  
// ...  
const auto pred = [] (const auto& r) {  
    // ...  
};  
filter<query<open_query>, decltype(pred)> f(  
    elide([] { return create_query(); }),  
    pred);
```

```
#define ELIDE(x) ::elide([&] { return x; })
query<open_query> create_query();
// ...
const auto pred = [] (const auto& r) {
    // ...
};
filter<query<open_query>, decltype(pred)> f(
    ELIDE(create_query()),
    pred);
```

```
#define ELIDE(x) ::elide([&] { return x; })
query<open_query> create_query();
// ...
const auto pred = [] (const auto& r) {
    // ...
};
filter f(
    ELIDE(create_query()),
    pred);
```

```
template<typename T>  
struct deduce : std::remove_cvref<T> {};
```

```
template<typename T>
struct deduce : std::remove_cvref<T> {};

namespace detail::deduce {

template<typename>
struct extract;
template<typename T>
struct extract<elide<T>> : std::type_identity<T> {};

}
```

```

template<typename T>
struct deduce : std::remove_cvref<T> {};

namespace detail::deduce {

template<typename>
struct extract;
template<typename T>
struct extract<elide<T>> : std::type_identity<T> {};

}

template<typename T>
    requires requires {
        typename detail::deduce::extract<std::remove_cvref_t<T>>::type;
    }
struct deduce<T> : std::type_identity<
    decltype(std::forward_like<T>(std::declval<
        typename detail::deduce::extract<std::remove_cvref_t<T>>::type>()))> {};

```

```

template<typename T>
struct deduce : std::remove_cvref<T> {};

namespace detail::deduce {

template<typename>
struct extract;
template<typename T>
struct extract<elide<T>> : std::type_identity<T> {};

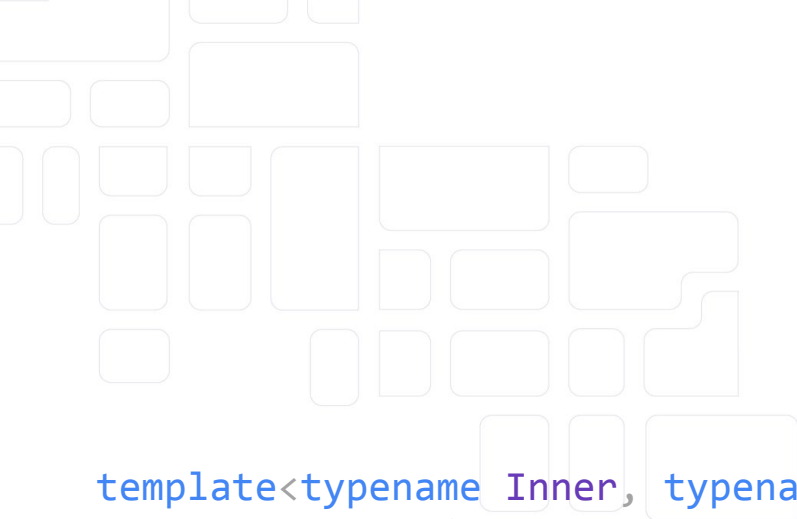
}

template<typename T>
    requires requires {
        typename detail::deduce::extract<std::remove_cvref_t<T>>::type;
    }
struct deduce<T> : std::type_identity<
    decltype(std::forward_like<T>(std::declval<
        typename detail::deduce::extract<std::remove_cvref_t<T>>::type>()))> {};

template<typename T>
using deduce_t = typename deduce<T>::type;

```





```
template<typename Inner, typename Predicate>  
explicit filter(Inner&&, Predicate) -> filter<deduce_t<Inner>, Predicate>;
```

# Summary

Think in terms of operations, not types

Ask yourself what you're doing generally, not particularly

Learn about the problem domain as you go

Throwaway the throwaway prototype

# Thank you

Robert Leahy  
Lead Software Engineer  
[rleahy@rleahy.ca](mailto:rleahy@rleahy.ca)



MAYSTREET

# Suggested Questions

Wasn't this all just reinventing...

- ...ranges?

- ...input iterators?

- ...coroutines?

Is there a performance overhead?

How are the compile times?