Compile-Time Validation

ALON WOLF

Software Validation

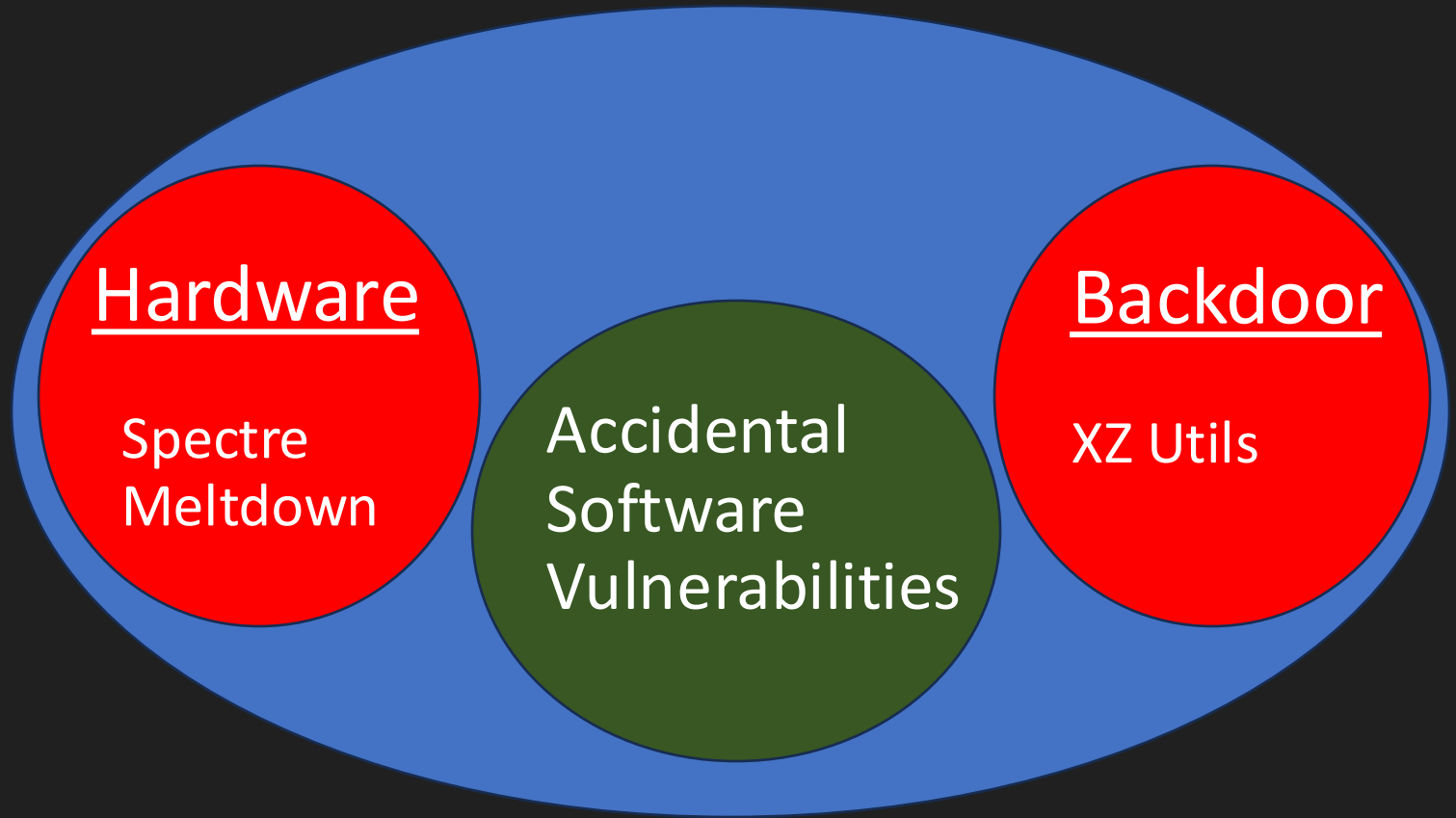
"Confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled"

- ISO/IEC 23643:2020

Security

"Resistance to intentional, unauthorized act(s) designed to cause harm or damage to a system"

- ISO/IEC 23643:2020



Memory Safety

"Memory safety is the state of being protected from various software bugs and security vulnerabilities when dealing with memory access, such as buffer overflows and dangling pointers"

- Wikipedia

Memory Safety - Invalidation

Invalidation occurs when an object or resource is modified or deallocated, causing references, pointers, or iterators that were pointing to it to become invalid

```
void foo(){
    vector<int> vec = { 0, 1, 2, /*...*/ };
    auto& ref = vec[0];

    vec.push_back(42);

    cout << ref; // ref may be invalid
}
```

Memory Safety - Out of Bounds

Accessing (read/write) memory out of bounds of an allocated buffer or container

```
void foo(){
    int index, value;
    cin >> index >> value;

    vector<int> vec = { 0, 1, /* ... */ };

    vec[index] = value;
}
```

Vulnerability - Injection

"An injection flaw is a vulnerability which allows an attacker to relay malicious code through an application to another system"
- OWASP

```
void foo(){
    string str;
    cin >> str;

    db.run("SELECT * FROM Users WHERE name = " + str);
}
```

Software Safety

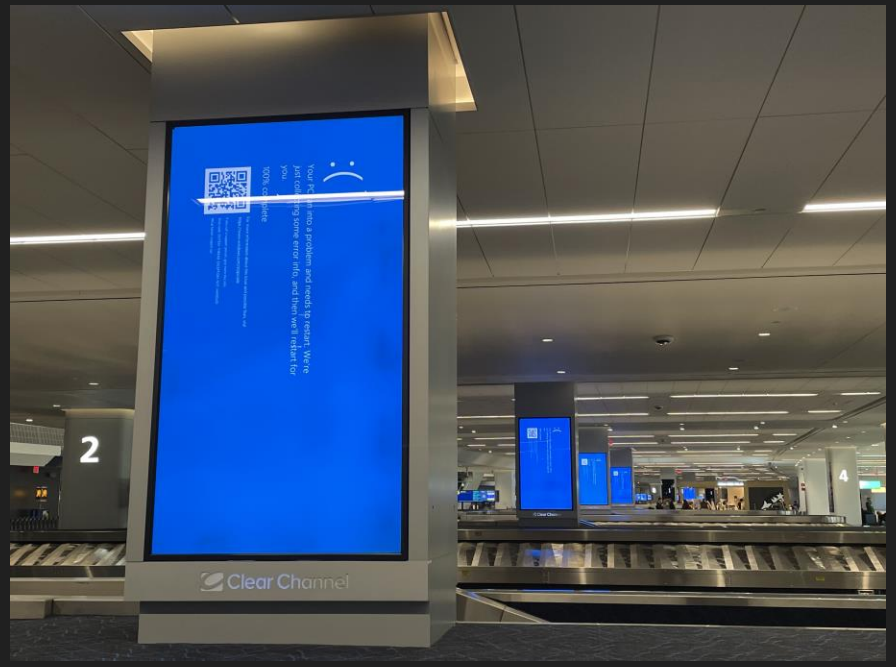
"Ability of software to be free from unacceptable risk.

... It is the ability of software to resist failure and malfunctions that can lead to death or serious injury to people, loss or severe damage to property, or severe environmental harm."

- ISO/IEC 23643:2020

Software Safety - Crowdstrike

Faulty updated crashed roughly 8.5 million systems, affecting airports, banks, hotels, hospitals and many other businesses and industries



Performance

Many applications have different performance requirements:

- Realtime applications latency (usually milliseconds)
- High frequency trading (usually nanoseconds)
- Mobile applications battery usage
- Cloud or servers electricity consumption

Other Validations

- Business logic
- Error handling
- Edge cases
- Communication with other devices

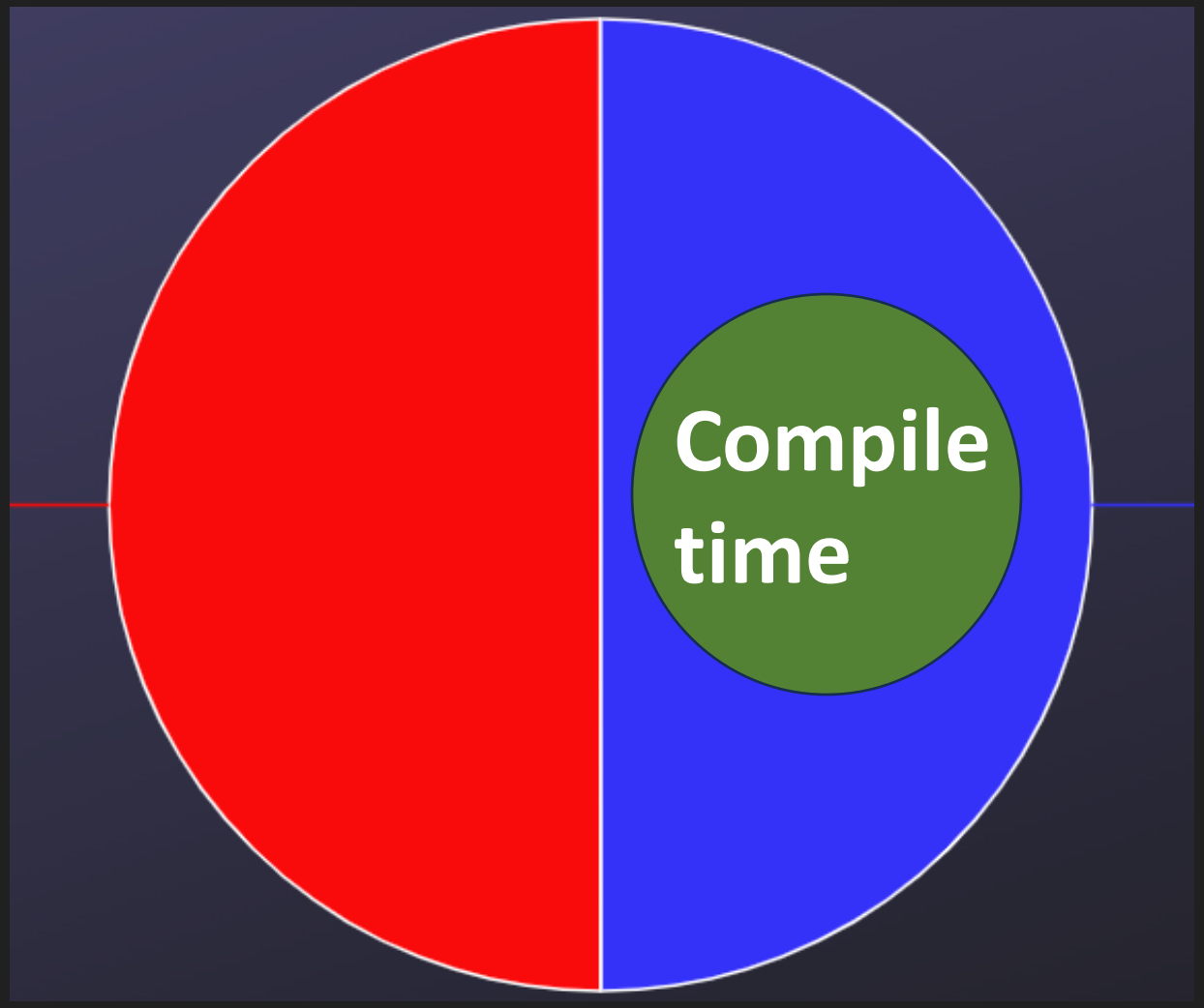
Static Vs Runtime

Static: Examines code without running it to catch potential issues early.

Runtime: Observes the program during execution to identify performance and behavior issues.

Both analyses complement each other and are used together to ensure thorough software quality and reliability.

Static Vs Runtime



Static vs Runtime – Bounds Check

Potential out of bounds write

```
void foo(){
    std::vector<int> vec = get_vec();
    size_t index = get_index();

    vec[index] = 42;
}
```

Static vs Runtime – Bounds Check

Bounds check at runtime with vector::at.

The existence of the bounds check be statically validated.

```
void foo(){
    std::vector<int> vec = get_vec();
    size_t index = get_index();

    vec.at(index) = 42;
}
```

Runtime Performance Validation

Detecting performance issues at runtime

```
void must_be_fast() {
    using namespace std::chrono;
    auto start = high_resolution_clock::now();
    /*...*/
    auto end = high_resolution_clock::now();

    validate_performance(start, end);
}
```

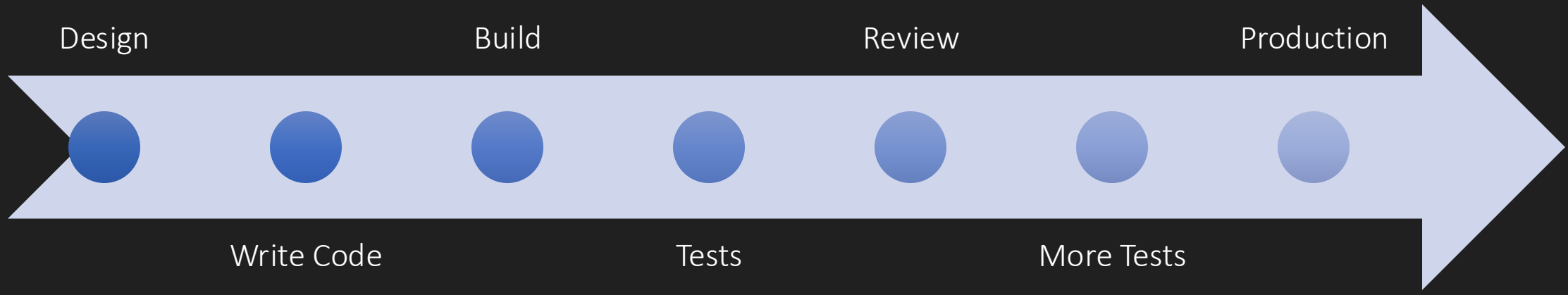
Static Performance Validation

Static detection of performance issues

```
void must_be_fast() {
    /*...*/
    can_slowly_read_huge_file();
}
```

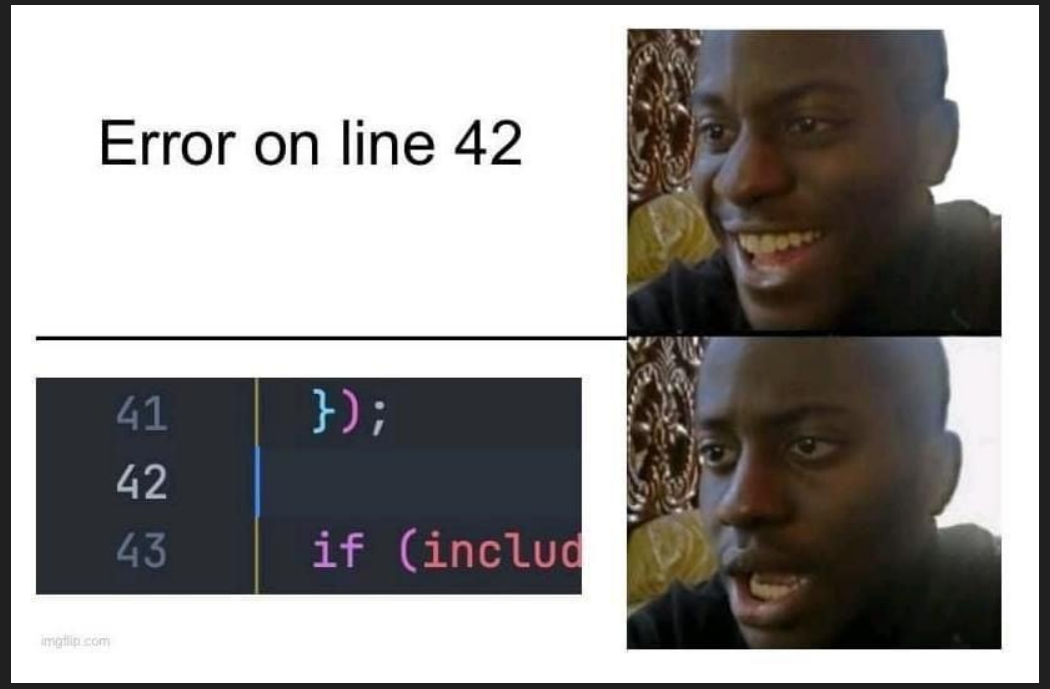
Software Development

Detecting errors early in the development pipeline reduces costs, saves time, minimizes risk, and improves efficiency.



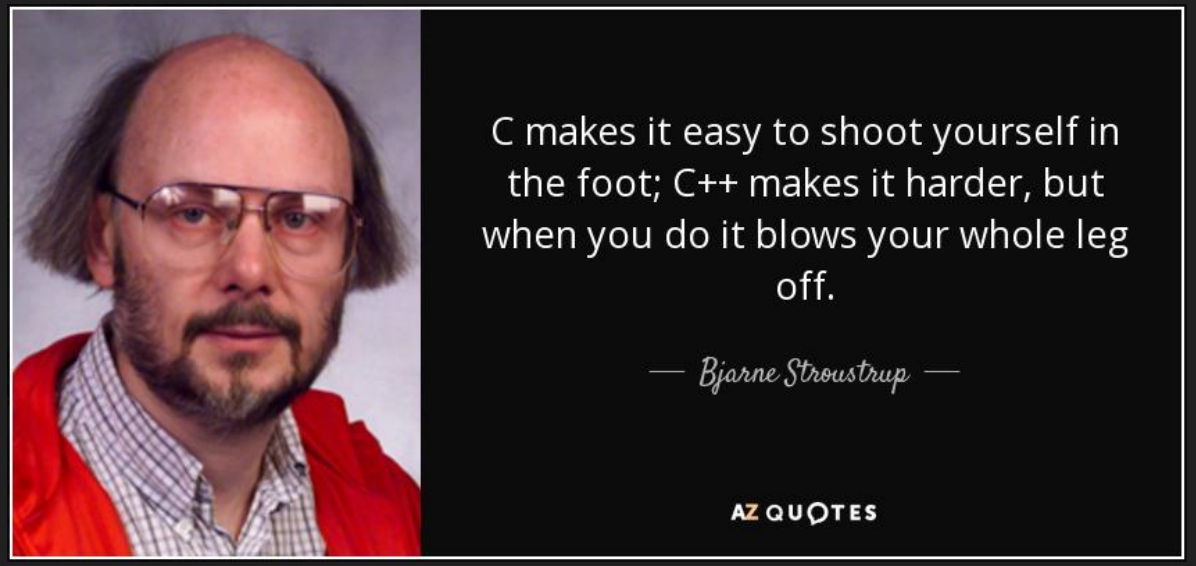
Error Reporting - Goals

Error messages should be clear, informative, and point to source of the error so that it can be fixed quickly and easily

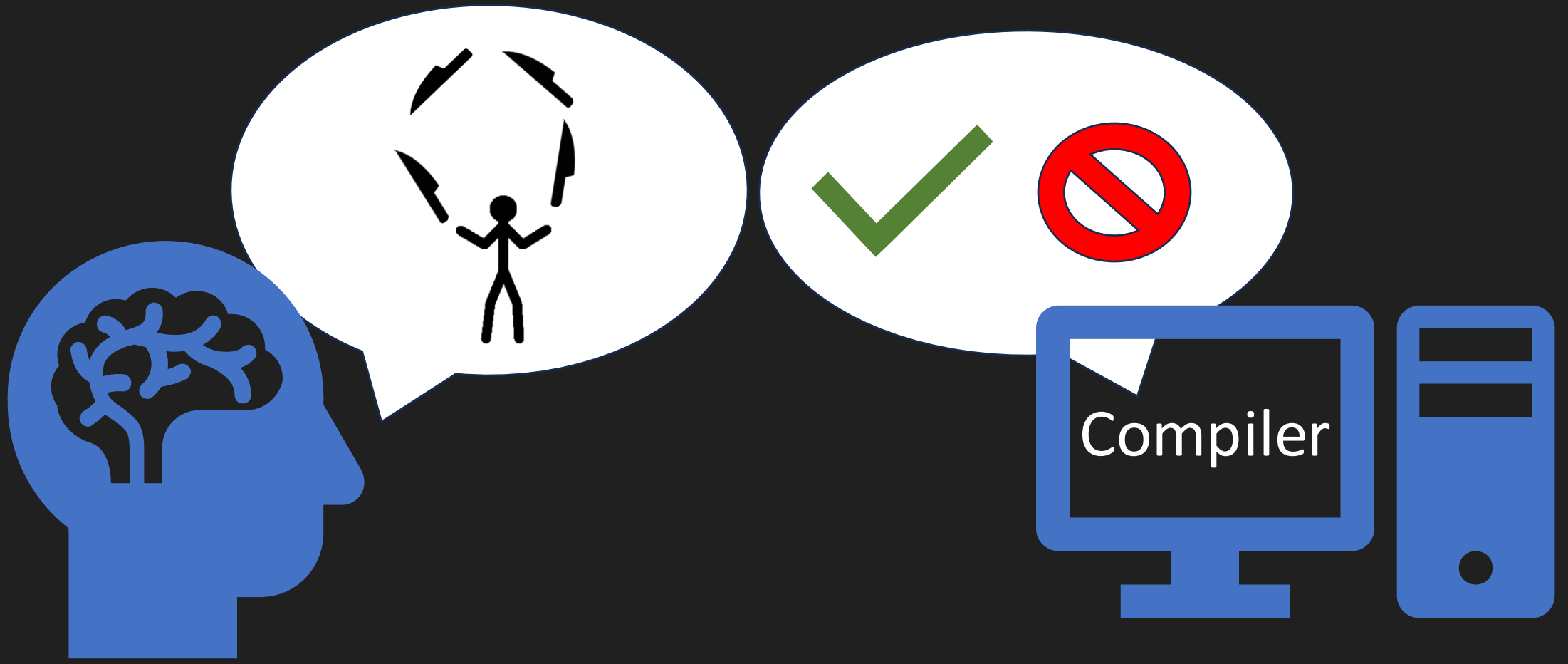


C++ Dangers

C++ includes some dangerous features that can introduce bugs and security risks if used improperly

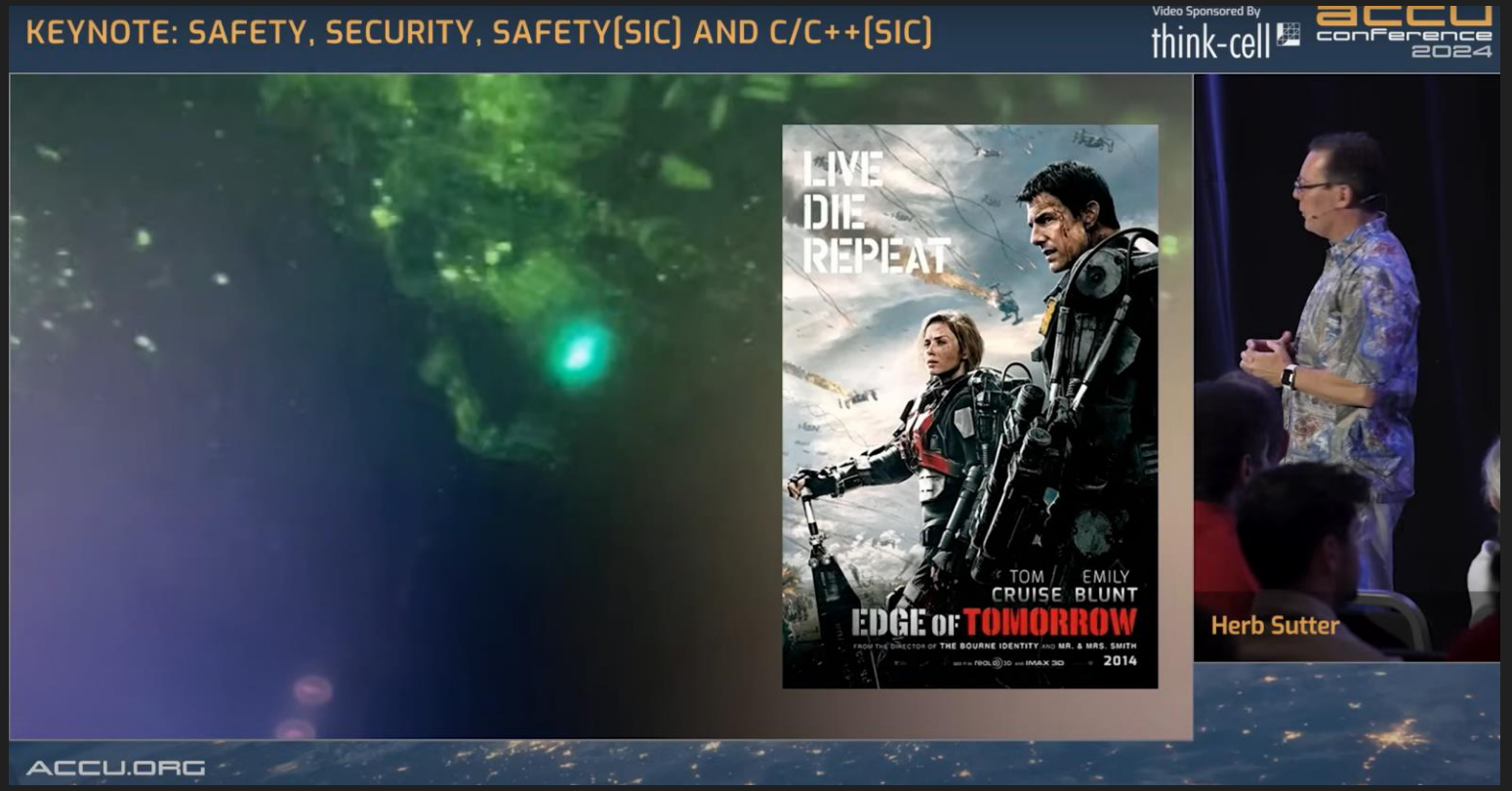


Compile-time Validation



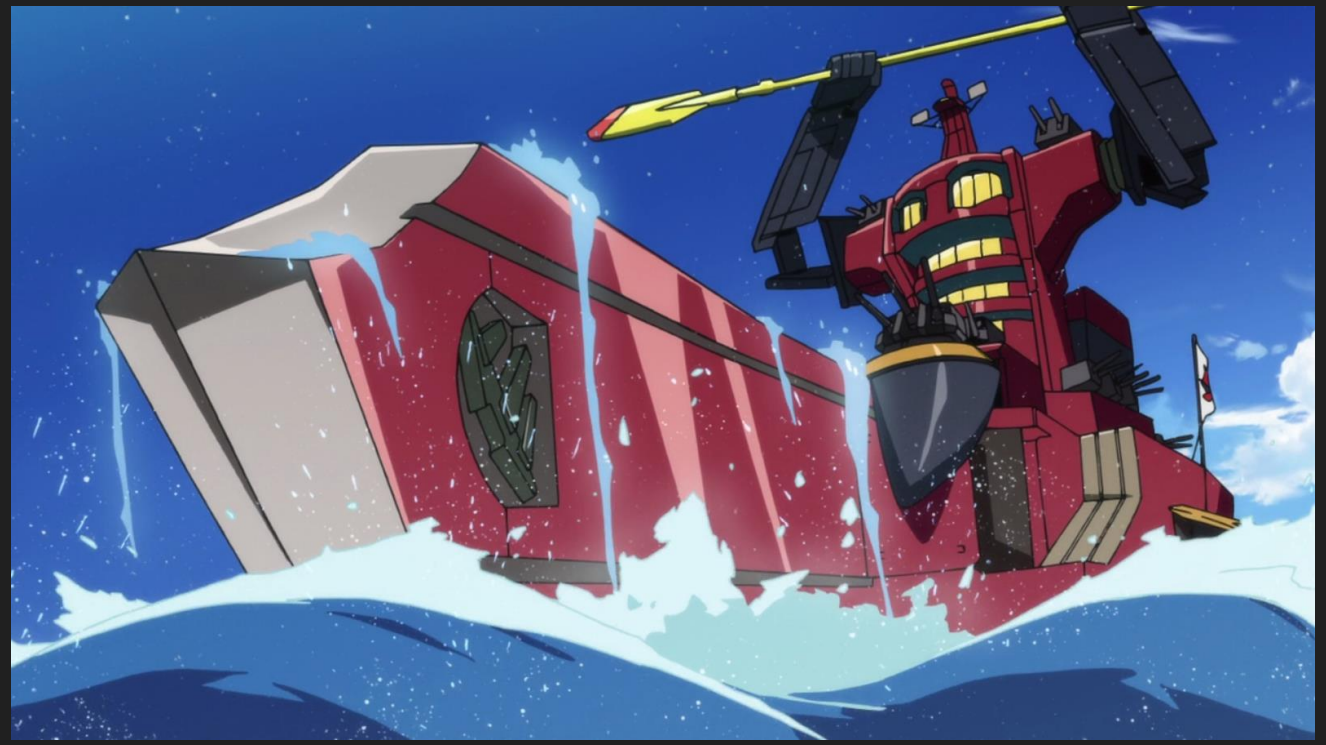
Mech Suit or Swim?

Choose between safety and flexibility



Mech Suit that Swims

Safety and flexibility at the same time



Source: Gurren Lagan

Runtime Error Reporting

Runtime error reporting offers various options for displaying error messages, including printing to a console, writing to a file, or transmitting over a network.

```
void foo(){
    auto error = detect_error();
    if (error) {
        report_error(error);
    }
}
```

Runtime Error Reporting

Runtime error reporting can be used with compile-time error detection

```
void foo(){
    constexpr auto error = detect_error();
    if constexpr (error) {
        report_error(error);
    }
}
```

Error Reporting – static_assert

Error message must be a string literal which limits the information that can be provided

```
void foo() {
    constexpr auto error = detect_error();
    static_assert(!error, "error message");
}
```

Error Message

We want to generate custom error messages at compile-time to provide better diagnostics

```
struct custom_error {};

void foo() {
    constexpr auto error = std::optional(custom_error{});
    if constexpr (error) {
        report_error<*error>();
    }
}
```

Error Message

```
template<auto error>
constexpr auto report_error() {
    static_assert(sizeof(error) == 0);
}
```

```
'constexpr auto report_error() [with auto error = custom_error()]':
<source>:15:33: error: static assertion failed
   15 |     static_assert(sizeof(error) == 0);
      |                   ~~~~~~~~~~~~~~^~~~
<source>:15:33: note: the comparison reduces to '(1 == 0)'
Compiler returned: 1
```

Error Message

```
template<auto error>
inline constexpr auto always_false = sizeof(error) == 0;

template<auto error>
constexpr auto report_error(){
    static_assert(always_false<error>);
}
```

```
<source>:16:19: error: static assertion failed
   16 |     static_assert(always_false<error>);
      |                   ^~~~~~~~~~~~~~~~~~~
<source>:16:19: note: 'always_false<custom_error()>' evaluates to false
```

Error Message - Parameters

Using data members to include parameters in error message

```
struct invalid_index {
    int index;
};

report_error<invalid_index{42}>();
```

```
:20:19: note: 'always_false<invalid_index{42}>' evaluates to false
```

fixed_str

A wrapper around a fixed size array that represents a compile-time string

```
template<int N>
struct fixed_str {
    constexpr fixed_str(const char(&str)[N]) {
        std::copy(str, str + N, data);
    }

    char data[N] = {};
};
```

Error Message – fixed_str

```
report_error<fixed_str("Hello Cppcon :)")>();
```

Clang, GCC: Prints the string error message

```
note: 'always_false<fixed_str<16>{"Hello Cppcon :)"}>'
```

MSVC: Prints the string characters as ascii numbers

```
'auto report_error<fixed_str<16>{char72,101,108,108,111,32,67,112,
```

User Generated Error Messages

C++26 enables user generated error messages in static_assert:

message.size() is implicitly convertible to std::size_t.

message.data() is implicitly convertible to const char*.

```
template<auto error>
constexpr auto report_error(){
    static_assert(always_false<error>, error.message());
}
```

Compile-time Unit Tests

Unit tests are automated tests written to validate that individual components of a program function as expected.

Some C++ computations run at compile-time by using constexpr, consteval, or template metaprogramming.

These compile-time components can also be tested at compile-time.

Unit Tests Setup

The outer static_assert enables placing this pattern in a function, class, or namespace without polluting the scope

```
static_assert(([]{
    static_assert(foo() == 42, "test failed");
    // more unit tests
}(), true));
```

Unit Tests Example – Qlibs++

Qlibs++ is a set of compile-time libraries which uses compile-time unit tests to validate its components and features

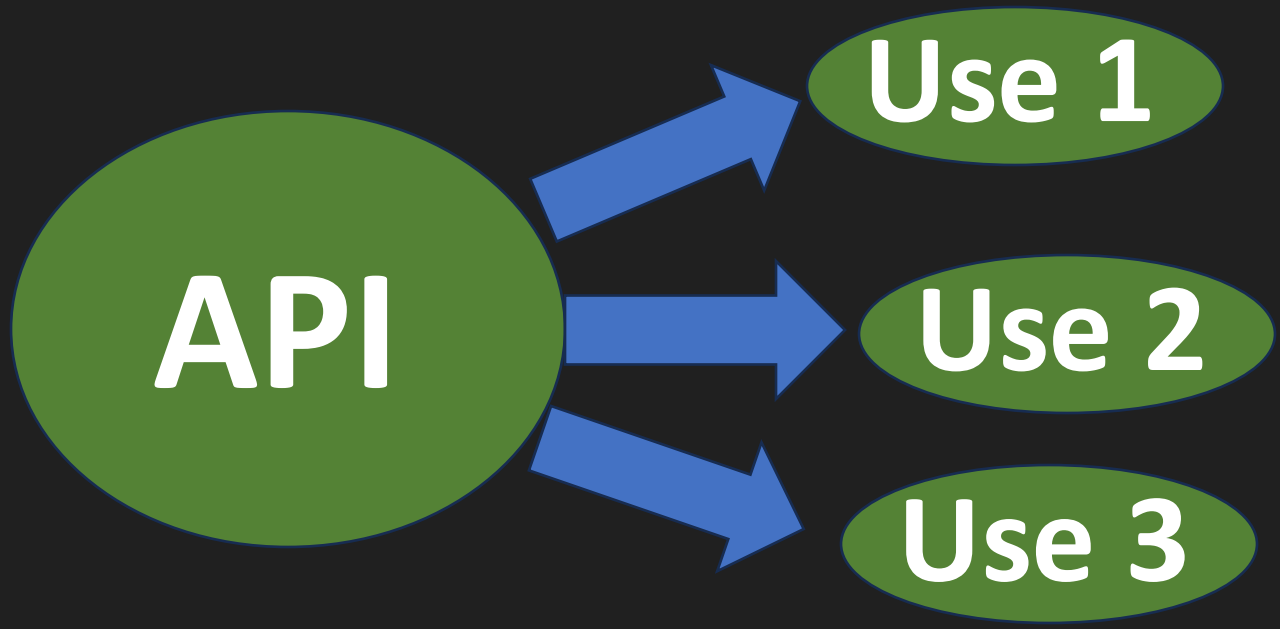
```
constexpr mp::vector v{mp::meta<int>};
static_assert(is_same_v<int, mp::type_of<v[0]>>);

struct empty {};
static_assert(diagnose_member_name<empty, "any">()
        == "`empty` has no data members.");
```

Qlibs++ by Kris Jusiak

diagnose_member_name contributed by Patrick Roberts

Assert Consistency

Changes in one place at the code can require changes in other places in the code base. These dependent places can be detected at compile-time.



Example - Enum

Enums are often used in a switch-case statements to handle different scenarios

```
enum class action {
    jump,
    fly
};

void on_action(action user_action) {
    switch(user_action) {
        case action::jump:
            jump(); break;
        case action::fly:
            fly();  break;
    }
}
```

Example - Enum

```
enum class action {
    jump,
    fly,
    swim
};

void on_action(action user_action) {
    switch(user_action) {
        case action::jump:
            jump(); break;
        case action::fly:
            fly();  break;
        // forgot to handle action::swim
    }
}
```

Example - Enum

Count the number of enum values with Magic Enum

```
enum class action {
    jump,
    fly
};

void on_action(action user_action) {
    static_assert(magic_enum::enum_count<action>() == 2);
    switch(user_action){
        /* ... */
    }
}
```

Example - Enum

Use compile-time switch and assert all cases are handled

```
void on_action(action user_action) {
    magic_enum::enum_switch([] (auto value) {
        if constexpr(value == action::jump){
            jump();
        }
        else if constexpr(value == action::fly){
            fly();
        }
        else {
            static_assert(sizeof(value) == 0);
        }
    }, user_action);
}
```

Reflection for C++26 (P2996R1)

Proposal to add static reflection to C++.

- Reflection operator ^T that returns the meta information of a type
- Metafunctions are functions that takes and returns meta information: members_of(^T), name_of(^T)
- Splice operator [:value:] converts meta information into a compile-time value and injects it into the source code.

Enum Switch with Reflection

Convert runtime enum value into a compile-time value with reflection

```
template<class E>
constexpr auto enum_switch(auto callback, E arg) {
    return [: expand(enumerators_of(^E)) :] >> [&]<auto value>() {
        if (arg == [:value:]) {
            return callback.template operator()<[:value:]>();
        }
    };
}
```

Compiler explorer:
https://godbolt.org/z/vMfo4n8aT

Functional Programming

- Immutability: Data is immutable, meaning once created, it cannot be changed. Instead of modifying existing data, you create new data structures with the desired changes.
- Function Composition: Combining simple functions to build more complex functions. This is often done using function composition operators.
- Monads: Encapsulates computations with context, allowing for the chaining of operations while managing side effects or state through a standardized interface.

Composition - Return Values

Compose two functions (f1, f2) into a new function.

The return value of the previous function is passed to the next function.

```
auto compose(auto f1, auto f2) {
    return [=] {
        return f2(f1());
    };
}
```

Composition - Continuation

Compose two functions (f1, f2) into a new function.

Each function takes a callback to next function as argument.

```
auto compose(auto f1, auto f2) {
    return [=] (auto next) {
        return f1([=] () {
            return f2(next);
        });
    };
}
```

Composition - Continuation

- Internal iteration:

```
for(int i = 0; i < n; i++){
    next(i);
}
```

- Prevents dangling references:

```
std::vector<int> vec = { /* ... */ };

next(std::span(vec));
```

Function Properties Composition

The properties of a composed function can be directly derived from the properties of the individual functions used in its composition.

```
auto f1 = foo | bar;
auto f2 = f1 | baz;

// pseudocode
properties of f2 == compose(
    properties of f1,
    properties of baz
)
```

Function Composition

```
struct fn_props {
    perf performance;
    bool is_memory_safe;
    bool can_terminate;
};

constexpr fn_props compose(fn_props fn1, fn_props fn2) {
    static_assert(
        perf::slow < perf::fast &&
        magic_enum::enum_count<perf>() == 2);

    return fn_props {
        .performance = std::min(fn1.performance, fn2.performance),
        .is_memory_safe = fn1.is_memory_safe && fn2.is_memory_safe,
        .can_terminate = fn1.can_terminate || fn2.can_terminate,
    };
}
```

Value Wrapper

Pass compile-time values as function arguments.

Can overload functions based on these compile-time values.

```
template <auto V>
struct value_wrapper {
    constexpr auto get() const { return V; }
};

template <auto V>
inline constexpr auto vw = value_wrapper<V>{};

auto foo(value_wrapper<1>) { /*...*/ }
auto foo(value_wrapper<2>) { /*...*/ }
```

Composable Function

Contains a callable function object and its properties.

```
template <
    fn_props Props,
    std::default_initializable Func
>
struct fn : Func {
    constexpr fn(value_wrapper<Props>, Func) {}

    constexpr auto props() const { return Props; }
};
```

Composition - Function and Properties

Create a new function by composing the properties and function objects of existing ones.

```
template<fn_props Props1, class F1, fn_props Props2, class F2>
constexpr auto operator|(fn<Props1, F1>, fn<Props2, F2>) {
    return fn(
        vw<compose(Props1, Props2)>,
        []<class Next>(Next, auto&&... args) -> decltype(auto) {
            constexpr auto invoke_f2 = [](auto&&... args) -> decltype(auto) {
                return F2{}(Next{}, FWD(args)...);
            };
            return F1{}(invoke_f2, FWD(args)...);
    });
}
```

Index Example - Unsafe

Memory unsafe - accessing a vector element without bounds check

```
constexpr auto unsafe_index = fn(vw<fn_props{
    .performance = perf::fast,
    .is_memory_safe = false,
    .can_terminate = true,
}> ,[](auto next, /* ...*/ ) {
    return next(vec[index], /* ...*/ );
});
```

Index Example - Bounds Check

Memory safe - accessing a vector element with bounds check

```
constexpr auto safe_index = fn(vw<fn_props{
    .performance = perf::fast,
    .is_memory_safe = true,
    .can_terminate = true,
}> ,[](auto next, /* ...*/ ) {
    return next(vec.at(index), /* ...*/ );
});
```

Assert Function Validity

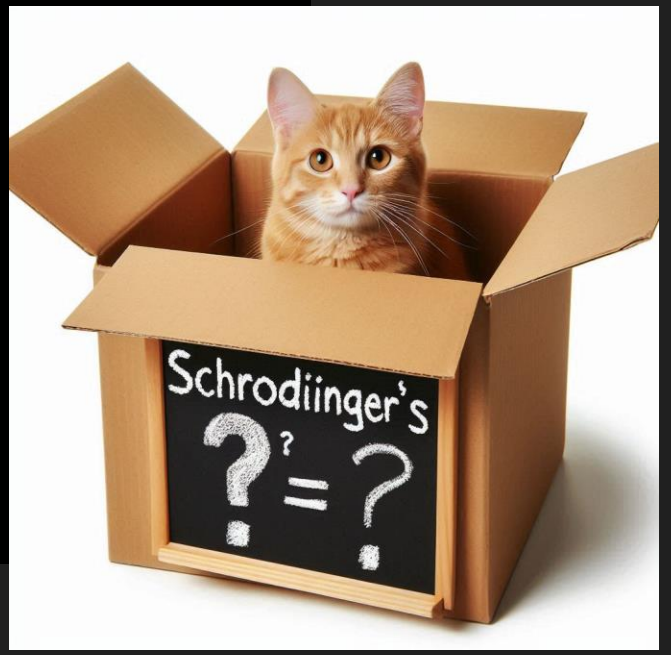
Validate the function properties at compile-time

```
constexpr auto foo = baz | bar;
static_assert(foo.props().is_memory_safe);

foo();
```

std::expected

Represents either a successful result value or an error

```
std::expected<int, error> result = foo();

if(result) {
    std::cout << result.value();
}

// unchecked access can throw
result.value();
result.error();
```



Monadic Operations

Safely access the wrapped result value.

The context type can change between monadic operations.

```
std::expected<int, error> result = foo();

auto transformed = result.transform([](const int& r){
    return sqrt(float(r));
}).transform([](float r){
    return r * 42.f;
});
```

Context Variable

Contains a reference to a variable and an apply function to change the reference type

```
template<class T>
struct context_var {
    static constexpr auto id = T::id;

    template<class Action>
    constexpr auto apply(Action) const {
        constexpr auto next_type = value.apply(Action{});
        using next = decltype(fwd_like<decltype(value)>(next_type));
        return context_var<remove_ref<next>>{ reinterpret_cast<next&>(value) };
    }

    T& value;
};
```

Context

A tuple of context variables. An action is applied to all variables.

```
template<class... Ts> struct context {
    tuple<Ts...> data;
    context(tuple<Ts...> data = std::tuple{}) : data(data) {}

    template<class T>
    auto add(T& var) const {
        return context<Ts..., context_var<T>>(
            std::tuple_cat(data, tuple(context_var{ var })));
    }

    template<class Action> auto apply(Action) const {
        return std::apply(/*...*/, data);
    }

    template<auto ID> auto& get() const { /*...*/ }
    template<auto ID> auto remove() const { /*...*/ }
}
```

Optional

```
template<auto ID> struct destroy_action {};

enum class opt_exists {
    yes,
    no,
    maybe
};

template<class T, auto ID, opt_exists State>
class opt {
    std::optional<T> data;
public:
    static constexpr auto id = ID;

    constexpr auto apply(auto) const {
        return std::type_identity<opt>{};
    }
```

Optional

Apply action when optional is destroyed

```
constexpr auto apply(destroy_action<ID>) const {
    return std::type_identity<opt<T, ID, opt_exists::no>>();
}

auto reset(auto context) {
    data.reset();
    return context.apply(destroy_action<ID>{});
}

auto& get(this auto& self) {
    static_assert(State == opt_exists::yes);
    return *self.data;
}
```

Reference Invalidation

Mark invalid when the object it points to is destroyed

```
template<class T, auto ID, auto OtherID, bool Valid>
struct ref : base<ID> {
    ref(T& v): value(v){}

    auto& get() const {
        static_assert(Valid, "reference is invalid");
        return value;
    }

    constexpr auto apply(destroy_action<OtherID>) const {
        return std::type_identity<ref<T, ID, OtherID, false>>();
    }
private:
    T& value;
};
```

Example

The new context type is passed to the next function

```
constexpr int opt_id = 1;
constexpr int ref_id = 2;

constexpr auto foo = fn(props, [](auto next) {
    opt<int, opt_id, opt_exists::yes> my_opt = foo();
    ref<int, ref_id, my_opt.get_id(), true> my_ref (my_opt.get());
    next(context{}.add(my_opt).add(my_ref));
}) |
fn(props, [](auto next, auto ctx) {
    next(ctx.get<opt_id>().reset(ctx));
}) |
fn(props, [](auto next, auto ctx) {
    ctx.get<ref_id>().get() = 42; // static_assert error
});
```

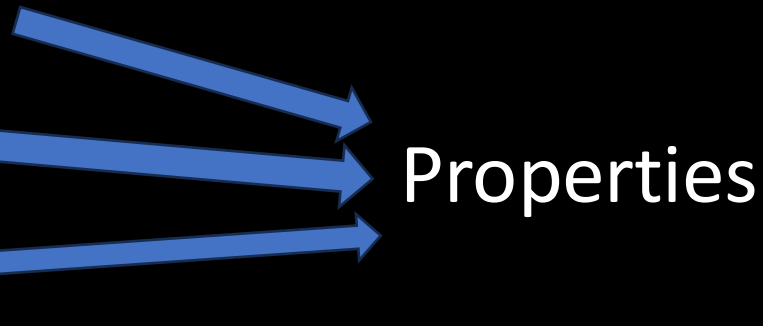
Cons

- Using functional programming should be a design decision not a requirement for compile-time validation.
- Relies on compiler optimizations such as inlining, TCO, and removing references.
- Requires wrapping variables in a context and passing it to the next function after every action.

Goals - Properties

Compute function properties from all the function calls in its body

```
void foo() {
    vector<int> vec = { /* ... */ };

    auto itr = find_itr(vec);

    vec.clear();
}
```



Goals - Invalidation

Change a variable type in the middle of the function

```
void foo() {
    vector<int> vec = { /* ... */ };
    auto itr = find_itr(vec);

    *itr = 1; // valid

    vec.push_back(2);
    *itr = 2; // invalid

    itr = find_itr(vec);
    *itr = 3; // valid
}
```

Goals – Custom Types

Validate custom types and logic

```
void foo() {
    my_vector<int> vec = { /* ... */ };
    auto index = find_index(vec);

    vec.grow_by(4);
    vec[index] = 0; // valid

    vec.clear();
    vec[index] = 2; // invalid
}
```

Goals – Control Flow

Validate based of control flow for: if/else, for loops, etc...

```
void foo() {
    my_vector<int> vec = { /* ... */ };
    auto index = find_index(vec);

    if(rand() % 2){
        vec.clear();
    }
    else {
        vec[index] = 0; // valid
    }

    vec[index] = 2; // invalid
}
```

Issue: Compile-time Constants

Types and compile-time values are constants and can't change

```
constexpr int f() { return 42; }

void foo() {
    vector<int> vec = { /* ... */ };
    auto itr = find_itr(vec);

    using T1 = decltype(*itr);
    if(rand() % 2){
        vec.clear();
    }
    using T2 = decltype(*itr);

    static_assert(is_same_v<T1, T2>);
    static_assert(f() == f());
}
```

Compiler Loophole – Stateful Metaprogramming

- A friend function exists outside of the class it is declared in.
- When a template class is instantiated it also instantiates all the friend functions that it defines.
- Existence of a function can be detected at compile-time.

```
template<class T>
struct loophole {
    friend auto detect_me(T&) {}
};
```

State

Represents a type at a given point during the compilation.

Whenever the type change the count is incremented.

```
template<int Count, class T>
struct state_t {
    static constexpr int count = Count;
    using type = T;
};
```

Reader

Detect whether the existence of a friend function with a give count value.

Tag is compile-time ID of the changeable type.

```
template<int Count, auto Tag>
struct reader {
    friend auto inject_state_func(reader<Count, Tag>);
};
```

Get Last State

Find and return the latest state

```
template<auto Tag, auto eval, int Count = 0>
consteval auto get_last_state() {
    constexpr bool exists = requires(reader<Count, Tag> r) {
        inject_state_func(r);
    };

    if constexpr (exists) {
        return get_last_state<Tag, eval, Count + 1>();
    }
    else {
        constexpr reader<Count - 1, Tag> read;
        return decltype(inject_state_func(read)){};
    }
}
```

Setter

Set the type of a state by instantiating next counter

```
template<int Count, class T, auto Tag>
struct setter {
    static constexpr state_t<N, T> state{};

    friend auto inject_state_func(reader<Count, Tag>) {
        return state;
    }
};

template<class T, auto Tag, auto eval>
consteval auto set_state() {
    using last_state = decltype(get_last_state<Tag, eval>()),
    using next = setter<last_state::count + 1, T, Tag>;
    return next{}.state;
}
```

Meta State

Represents a type that can change during compilation

```
template<class Init, auto Tag = [] {} >
class meta_state {
public:
    template <typename T, auto eval = [] {}>
    using set = decltype(set_state<T, Tag, eval>());

    template <auto eval = [] {} >
    using get = typename get_state<Tag, eval>::type;
private:
    static constexpr setter<0, Init, Tag> init = {};
};
```

Meta State - Example

```
// initialize a state to store the int type
using state = meta_state<int>;

state::get<> my_int = 42;
static_assert(std::is_same_v<decltype(my_int), int>, "");

using T = state::set<std::string>;

state::get<> my_str = "this is amazing!";
static_assert(std::is_same_v<decltype(my_str), std::string>, "");
```

Template Instantiation

Each template specialization is instantiated once.

The current meta state type is computed during instantiation.

```
using state = meta_state<int>;

constexpr auto eval = 1;

using T1 = state::get<eval>;
static_assert(std::is_same_v<T1, int>);

using T2 = state::set<const char*>;
using T3 = state::get<eval>; // not updated
static_assert(std::is_same_v<T3, int>);
```

Eval – Force Template Instantiation

Force new template instantiation for each call by using an anonymous lambda as template parameter

```
template<auto eval = []{}>
constexpr auto foo() {}

// different syntax but they all do the same thing
foo();
foo<>();
foo<[]{}>();
```

Accessing Meta State

Inside template parameters declaration:

```
template<
    class state,
    auto eval = [] {},
    class T = typename state::template set<int, eval>
>
```

Inside a function with auto return type:

```
template<class state, auto eval = [] {}>
auto foo(){
    using T = typename state::template set<int>;
}
```

Validation – Meta State Everything

1. Attach a meta state to every function, scope, and variable.
2. Validate and update all relevant meta states at every function call.

Compiler Explorer: https://godbolt.org/z/vs7Efshx1

Scaling Issues

- Validation requires global reasoning which isn't available at a specific function call.
- Difficult to update and validate many meta states.
- Difficult to combine multiple validation constraints.

```
template<
    auto eval = []{},
    TODO: detect and invalidate all iterators
>
void grow_by(size_t size) { /* ... */ }
```

Function Validation

A function is valid if all the states that can be reached during the execution of the function are valid

```
auto foo() {
    state 1
    state 2
    if cond {
        state 3
    }
    state 4
}
```

Function Validation

A state can be computed from all the actions that came before it

```
auto foo() {
    action 1
    action 2
    if cond {
        action 3
    }
    action 4
}
```

```
state 1 = compute_state(action 1)
state 2 = compute_state(action 1, action 2)
state 3 = compute_state(action 1, action 2, action 3)
state N = compute_state(action 1, ..., action N)
```

Counter

A compile-time incrementing counter.

Used to assign unique IDs.

```
template<auto Eval = [] {} >
struct counter {
    using state = meta_state<value_wrapper<0>, Eval>;

    template<
        auto eval = [] {},
        auto current = state::get<eval>::value,
        class v = state::set<value_wrapper<current + 1>, eval>
    >
    static constexpr auto next() {
        return current;
    }
};
```

Value List

List of compile-time values

```
template<auto... values>
struct value_list {
    template<auto V>
    using push_back = value_list<values..., V>;
};
```

Recorder

Record and store all the actions in the function

```
template<auto Eval = [] {} >
struct recorder {
    using state = meta_state<value_list<none{}>, Eval>;

    template<
        auto action,
        auto eval = [] {},
        class new_list = state::get<eval>::push_back<action>,
        class v = state::set<new_list, eval>;
    >
    static constexpr auto add() { return true; }
};
```

Unpack Actions

Convert a value list into an array of variants

```
template<auto... values>
constexpr auto unpack_actions(value_list<values...>) {
    using types = type_list<decltype(values)...>;
    using element_type = decltype(unique(types{}))::
                            template rename<std::variant>;

    return std::array<element_type, sizeof...(values)>{
        element_type(values)...
    };
}
```

Rule

Validate compile-time constraints:

- Mixin – Records actions to the meta state
- Validate – Validate that the function is correct from the actions

```
struct my_rule {
    template<class State>
    struct mixin {
        template<auto eval = []{}, /* ... */>
        auto record(){}
    };

    static constexpr std::optional<error> validate(auto actions) {
        /* ... */
    }
};
```

Profile State

Contains a counter to generate sequential unique IDs and recorder to store the actions.

```
template<auto eval = [] {}>
struct profile_state {
    using counter = ::counter<[]{}>;
    using recorder = ::recorder<[]{}>;
};
```

Profile – Composition of rules

Contains an inner type that inherits from all the rules mixins instantiated with a state

```
template<class... Rules>
struct profile {
    template<auto Eval = []{}, class State = profile_state<Eval>>
    struct validator : get_mixin<Rules, State>... {
        template<
            auto eval = [] {},
            class list = state::recorder::get<eval>
        >
        static constexpr auto validate() { /* ... */ }
    }
}
```

Profile Validator – Validate

```
constexpr auto actions = unpack_actions(list{});

return [actions](this auto self, auto rule, auto... rest) {
    constexpr auto error = decltype(rule)::validate(actions);
    if constexpr (error) {
        return error;
    }
    else if(sizeof...(rest) != 0){
        return self(rest...);
    }
    else {
        return std::optional<none>(std::nullopt);
    }
}(Rules{}...);
```

Profile - Run

Use a profile to run and validate a function

```
template<
    class Profile,
    auto eval = [] {}
>
decltype(auto) run(Profile, auto&& func) {
    using validator = Profile::validator<eval>;
    return func(validator{}); // run and record

    constexpr auto error = validator::validate();
    if constexpr (error) {
        report_error<error->message>();
    }
}
```

Validation Error

Report a compile-time error message at a given action index

```
template<class T>
struct validation_error {
    int action_index;
    T message;
};
```

Recorder - Validation Error

When recording an action compare it's index to the error index and report the error if it does

```
template<validation_error error, auto Eval = [] {} >
struct recorder { /*...*/
    template </*...*/
        class new_list = state::get<eval>::push_back<value>,
    /*...*/>
    static constexpr auto add() {
        if constexpr (new_list::size - 1 == error.action_index) {
            report_error<error.message>();
        }
    }
};
```

Run Profile - Validation Error

Record actions and validate without running the function by using decltype

```
template<class Profile, auto eval = [] {}>
decltype(auto) run(Profile, auto&& func){
    using validator = Profile::validator<default_error>;
    using T = decltype(func(validator{}));

    constexpr auto error = validator::validate( ).value_or(default_error);

    return func(Profile::validator<error>{});
    /*...*/
}
```

Reference Borrowing

Track read/read-write access to a variable.

Will be used to prevent invalidation of references and pointers.

```
enum class borrow_type {
    ref,
    mut
};

struct borrow_action {
    borrow_type type;
    int id;
};
```

Borrowable

```
template<class T, class recorder, int ID>
class borrowable {
    /*...*/
    template<auto eval = [] {},
        class v = recorder::add<borrow_action{ borrow_type::ref, ID },
    eval>()>
    const T& ref() const { return value; }

    template<auto eval = [] {},
        class v = recorder::add<borrow_action{ borrow_type::mut, ID },
    eval>()>
    T& mut() { return value; }
private:
    T value;
};
```

Borrow Rule

Mixin passes the recorder and a unique ID when a new borrowable is created

```
struct borrow_rule {
    template<class state>
    struct mixin {
        template</*...*/>
        static auto borrow(/*...*/) {
            constexpr auto id = state::counter::next<>();
            return borrowable(vw<id>, state::recorder, /*...*/);
        }
    }

    constexpr auto validate(auto actions){ /*...*/ }
}
```

Borrow State

Can have single writable reference or multiple read only references

```
struct borrow_state {
    int ref_count = 0;
    int mut_count = 0;

    consteval bool is_invalid(borrow_type type) {
        if(type == borrow_type::ref)
            ref_count++;
        else if(type == borrow_type::mut)
            mut_count++;
        else
            std::unreachable();

        return (mut_count == 1 && ref_count > 1) || (mut_count > 1);
    }
};
```

Borrow Rule - Validate

Track and validate borrow states with the recorded actions

```
vector<borrow_state> states;
/*...*/
if(action->id >= states.size()) {
    states.resize(action->id + 1);
}

if(states[action->id].is_invalid(action->type)){
    return validation_error{ i, states[action->id] };
}
```

Borrow Rule - Example

```
using safe_profile = profile<borrow_rule>;

run(safe_profile{}, []<class M>(M){
    auto num = M::borrow(42); // ref - 0, mut - 0
    auto& ref1 = num.ref();   // ref - 1, mut - 0
    auto& mut1 = num.mut(); // static assert error {1,1}
});
```

Control Flow

Record and validate control flow

```
enum class control_flow {
    if_,
    else_if,
    for_,
    return_,
    /*...*/
};
```

```
if (cond) { M::add<control_flow::if_>;

} M::add<control_flow::end_scope>;
```

Macro – Text to String

A macro can convert text into a string literal:

- All comments will be removed
- White spaces will collapse into a single space

```
#define TO_STRING(...) #__VA_ARGS__

static_assert(std::string_view("if ( cond ) { }") == TO_STRING(if  (
cond   ) { // comments won't be stringified
  /* white spaces will collapse to a single space */
}));
```

Control Flow - Parsing

Parse string of code into a control flow enum value

```
constexpr control_flow parse_control(std::string_view str) {
    if (str == "{")
        return control_flow::start_scope;
    if (str == "}")
        return control_flow::end_scope;
    if (str.starts_with("if (") && str.ends_with(") {"))
        return control_flow::if_;
    if (str.starts_with("for (") && str.ends_with(") {"))
        return control_flow::for_;
    if (str == "return;" || (str.starts_with("return ") &&
str.ends_with(";")))
        return control_flow::return_;
    /*...*/
    std::unreachable();
}
```

Control Flow Parse Rule

Parse a string into a control flow and record it

```
template<class state>
struct mixin {
    template<
        fixed_str str,
        auto eval = [] {},
        auto control = parse_control(str.view()),
        class v = state::recorder::add<control, eval>()
    >
    using parse = v;
};
```

Control Flow - Example

```
#define M_(...) \
sizeof(typename M::template parse<#__VA_ARGS__>); \
__VA_ARGS__
```

```
for (auto& point : vec) {
    if (point.x == n) {
        return point.y;
    }
}
```

Control Flow - Example

```
#define M_(...) \
sizeof(typename M::template parse<#__VA_ARGS__>); \
__VA_ARGS__
```

```
M_(for (auto& point : vec) {)
    M_(if (point.x == n) {)
        M_(return point.y;)
    M_(})
M_(})
```

Borrow Rule - Control Flow

Track borrow states within each scope

```
vector<vector<borrow_state>> states = { {} };
/*...*/
if (auto action = try_get<control_flow>(actions[i]); action) {
    if (is_end_scope(action)) {
        states.pop_back();
    }
    if (is_start_scope(action)) {
        auto last = states.back();
        states.emplace_back(std::move(last));
    }
}
```

Validate Pointers

A pointer to a variable that goes out of scope will be dangling

```
{
    int v1 = 1;
    int* outer = &v1;
    {
        int v2 = 2;
        outer = &v2;
    }
    *outer = 42; // invalid
}
```

Validate Pointers

Record initialization and assignment of pointers

```
enum class ptr_command {
    init,
    assign
};

struct ptr_action {
    ptr_command command;
    int self_id;
    int other_id;
};
```

Pointer

```
template<class T, class Recorder, int ID>
class ptr {
    /*...*/
    template<
        int OtherId,
        class eval = [] {},
        class v = recorder::template
            add<ptr_action{ ptr_command::assign, ID, OtherId }, eval>()
    >
    void assign(ptr<T, Recorder, OtherId> other) {
        this->data = other.data;
    }
private:
    T* data;
};
```

Pointer Rule - Validation

Track how many nested scopes exists at any given moment

```
int depth = 1;
/*...*/
if (auto action = try_get<control_flow>(actions[i]); action) {
    if(is_end_scope(action))
        depth--;
    if(is_start_scope(action))
        depth++;
}
```

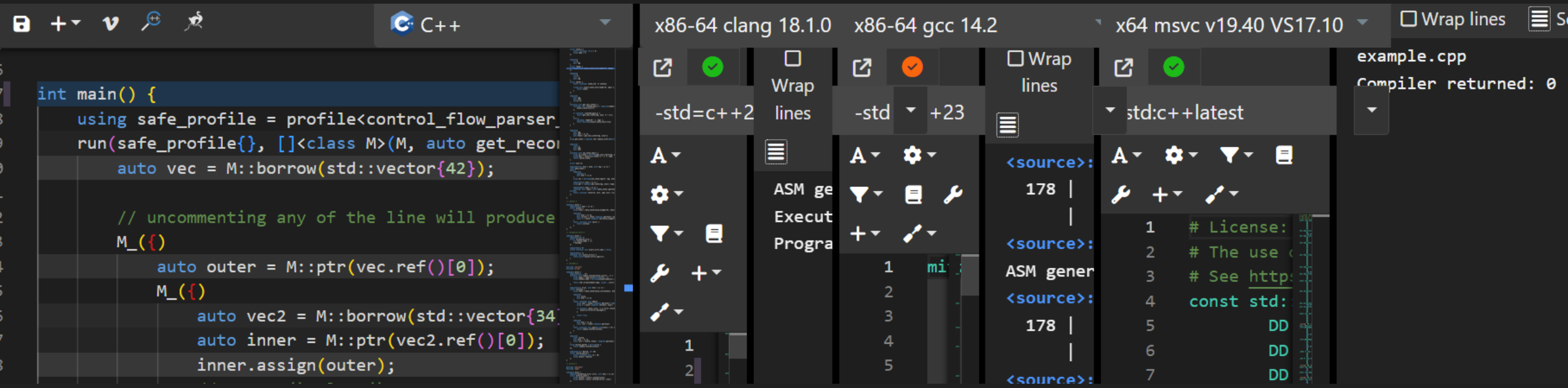
Pointer Rule - Validation

Validate that a pointer in an outer scope isn't pointing to a value in an inner scope

```
vector<int> ptrs_depth;
/*...*/
if (action.command == ptr_command::init) {
    ptrs_depth.resize(max(action.self_id + 1, ptrs_depth.size()), 0);
    ptrs_depth[action.self_id] = depth;
}
else if (action.command == ptr_command::assign) {
   if (ptrs_depth[action.self_id] < ptrs_depth[action.other_id]) {
        return validation_error(i, lifetime_too_short{});
    }
}
```

Demo

Compiler Explorer: https://godbolt.org/z/xn7vb3Ebv



Multiple Control Flows

Validate each control flow path separately

```
template<class Action>
struct control_flow_path {
    int action_index;
    std::vector<Action> actions;
};
```

Find Control Flows

Depth first search to find control flow paths in a function

```
while (path.action_index != actions.size()) {
    auto action = actions[path.action_index];
    if (is_branch(action)) {
        auto other = path;
        other.next_index = skip_branch(actions, other.action_index);
        paths.push_back(move(other));
    }
    path.action.push_back(action);
    path.action_index++;
}
```

Async Cleanup

Validate that async cleanup function is called once before the object is destroyed in every control flow

if (false)

```
auto resource = get_resource();
/*...*/
if (condition) {
    /*...*/
    co_await resource.cleanup();
    return;
}
/*...*/
co_await resource.cleanup();
```

if (true)

Performance Validation

- General Allocator – Slow general purpose allocator (malloc)
- Frame Allocator – Fast linear allocator that deallocates everything at the end of the frame

```
{
    auto buf = buffer(general_allocator, size);

    buf.use_in_this_frame();
}
```

Performance Validation

Pass the recorded actions to the function

```
decltype(auto) run(Profile, auto&& func) {
    /*...*/
    return func(validator{}, []{
        return actions;
    });
}
```

Performance Validation

Choose an allocator based on how the buffer is used

```
buffer(
    size_t size,
    auto& general_allocator,
    auto& frame_allocator,
    auto get_actions) {

    constexpr auto usage = get_usage(ID, get_actions());
    if constexpr (usage == buffer_usage::frame) {
        data = frame_allocator.allocate(size);
    }
    else {
        data = general_allocator.allocate(size);
    }
}
```

Pros

- Validate custom constraints at compile-time
- Separation between action recording and validation
- Choose which validation rules to enforce for each function
- Introspect a function's actions to optimize runtime
- Report a custom error message at compile-time from the location that caused it

Cons

- Can only validate based on the recorded actions
- Requires proxy types to record operations
- Uses macros to record control flow
- Uses compiler loophole for stateful metaprogramming

C++26 Reflection

Use std::meta::info to record generic actions.

Query std::meta::info to access the compile-time information for validation.

```
template<int N>
struct action {
    meta::info func;
    array<meta::info, N> args;
};
```

```
recorder::add<action{
    ^func,
    array{ (^args)... }
}>;
```

```
identifier_of(info r) -> string_view;
source_location_of(info r) -> source_location;
type_of(info r) -> info;
value_of(info r) -> info;
template_of(info r) -> info;
template_arguments_of(info r) -> vector<info>;
members_of(info type_class) -> vector<info>;
enumerators_of(info type_enum) -> vector<info>;
/*...*/
```

C++26 Reflection – Proxy Types

Generate a proxy type with define_class

```
template<class T>
consteval auto make_proxy() -> info {
  vector<meta::info> old_members = members_of(^T);
  vector<meta::info> new_members;
  /*...*/
  return define_class(
    substitute(^proxy_impl, { ^T }),
    new_members);
}
```

C++26 Reflection – Stateful Metaprogramming

Detect existing template instantiation with is_complete_type

```
class Counter {
  template<int N> struct Helper;
public:
  static consteval int next() {
    int k = 0;
    meta::info r;
    while (is_complete_type(r = substitute(
            ^Helper, { meta::reflect_value(k) }))){
      ++k;
    }

    define_class(r, {});
    return k;
  }
};
```

Circle – Lifetime Safety

Circle is a C++ compiler by Sean Baxter with many language extensions.

Many operations are prohibited in safe functions.

```
#feature on safety

int main() safe {
    int index = 42;

    unsafe {
        int& ref = &index;
        int* ptr = &index;
        printf("%d",  *ptr);
    }
}
```

Circle – Lifetime Safety

Adds lifetime annotation:

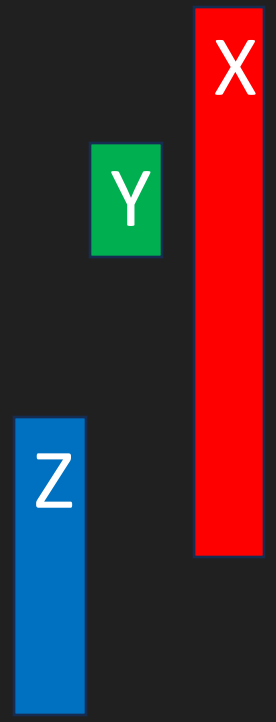
T^ is a borrow reference.

/a is a lifetime annotation.

```
auto get_max/(a)(int^/a x, int^/a y) safe -> int^/a {
  return (*x > *y) ? x : y;
}
```

Circle – Lifetime Safety

Lifetime validation and destructive move




```
int main() safe {
  int^ ref;
  int x = 1;
  {
    int y = 2;
    ref = get_max(^x, ^y);
  }
  int val1 = *ref;  // error
  int z = 3;
  ref = get_max(^x, ^z);
  rel x; // destructive move

  int val2 = *ref; // error
}
```

C++ Safety Progress

Profiles - Bjarne Stroustrup:

https://github.com/BjarneStroustrup/profiles/tree/main

Circle lifetime safety – Sean Baxter:

https://www.circle-lang.org/

Erroneous behaviour for uninitialized reads - Thomas Köppe:

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2795r5.html

Towards memory safety in C++ - Thomas Neumann:

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2771r0.html

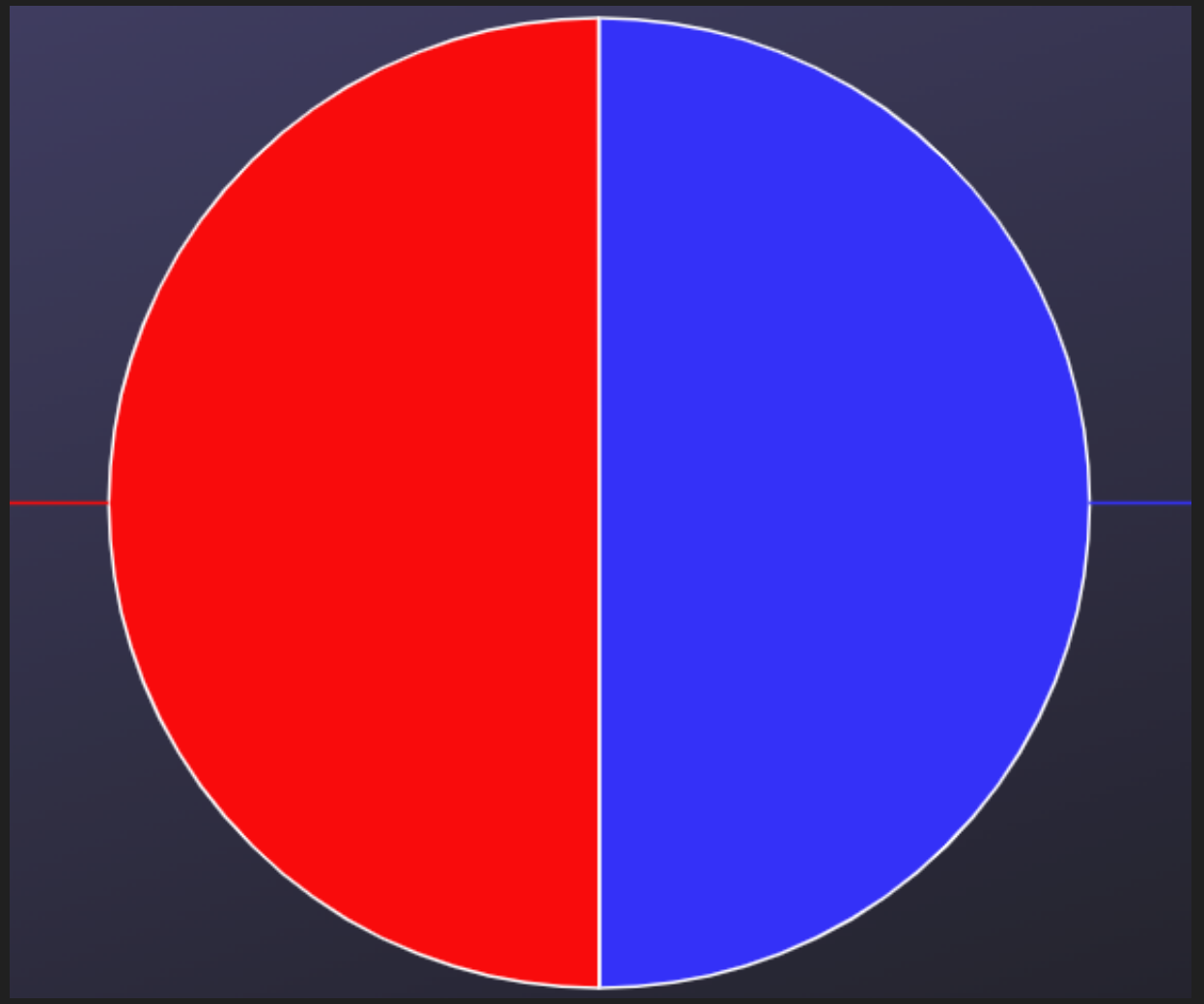
Clang lifetime inference and analysis – Google:

https://github.com/google/crubit/blob/main/docs/design/lifetimes_static_analysis.md

C++ Discussion

C++ Discussion

Unsafe code



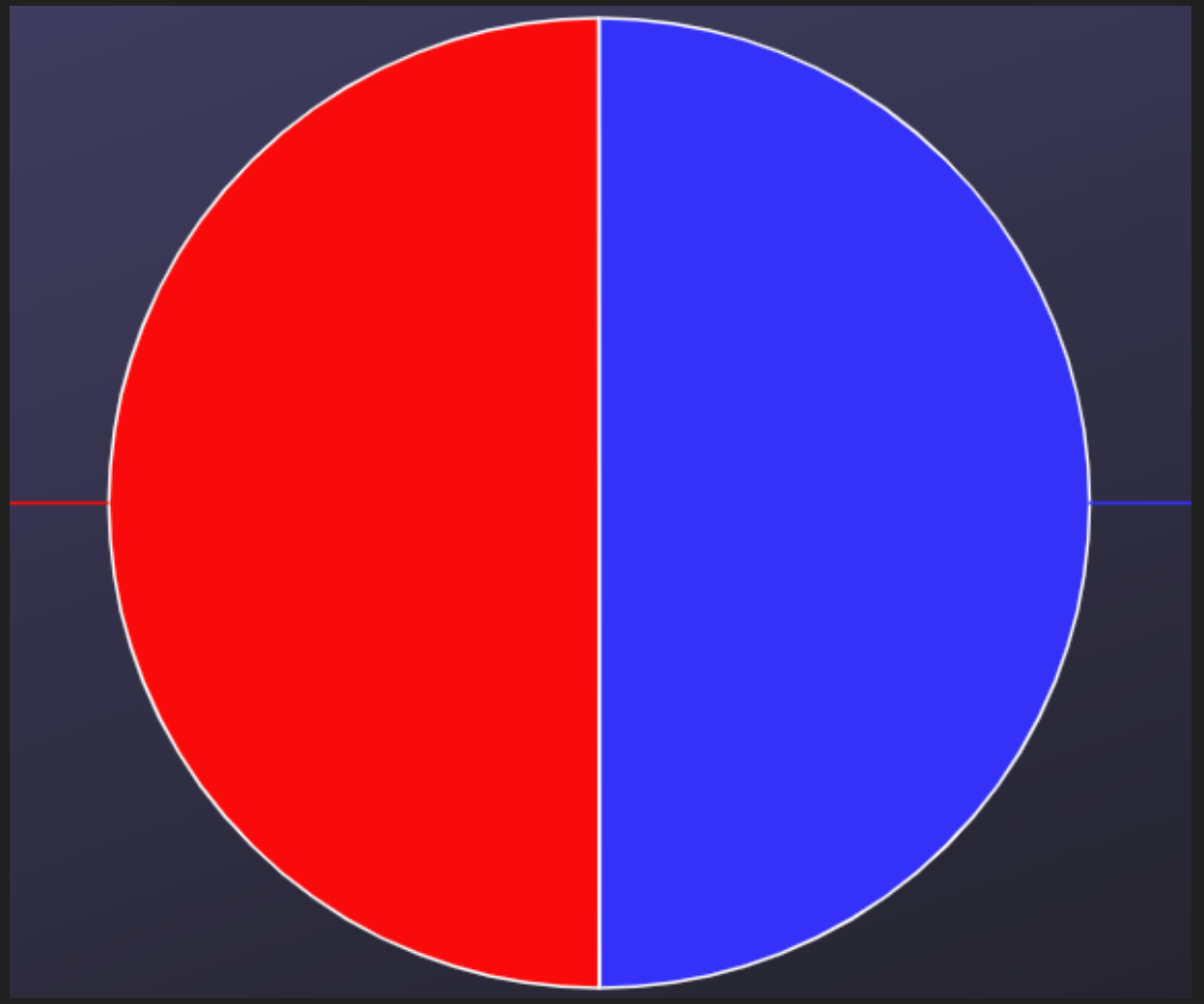
Safe code

C++ Discussion

```
auto vec = get_vec();
auto& ref = vec.at(i);

vec.pop_back();
vec.emplace_back();

std::cout << ref;
```

C++ Discussion

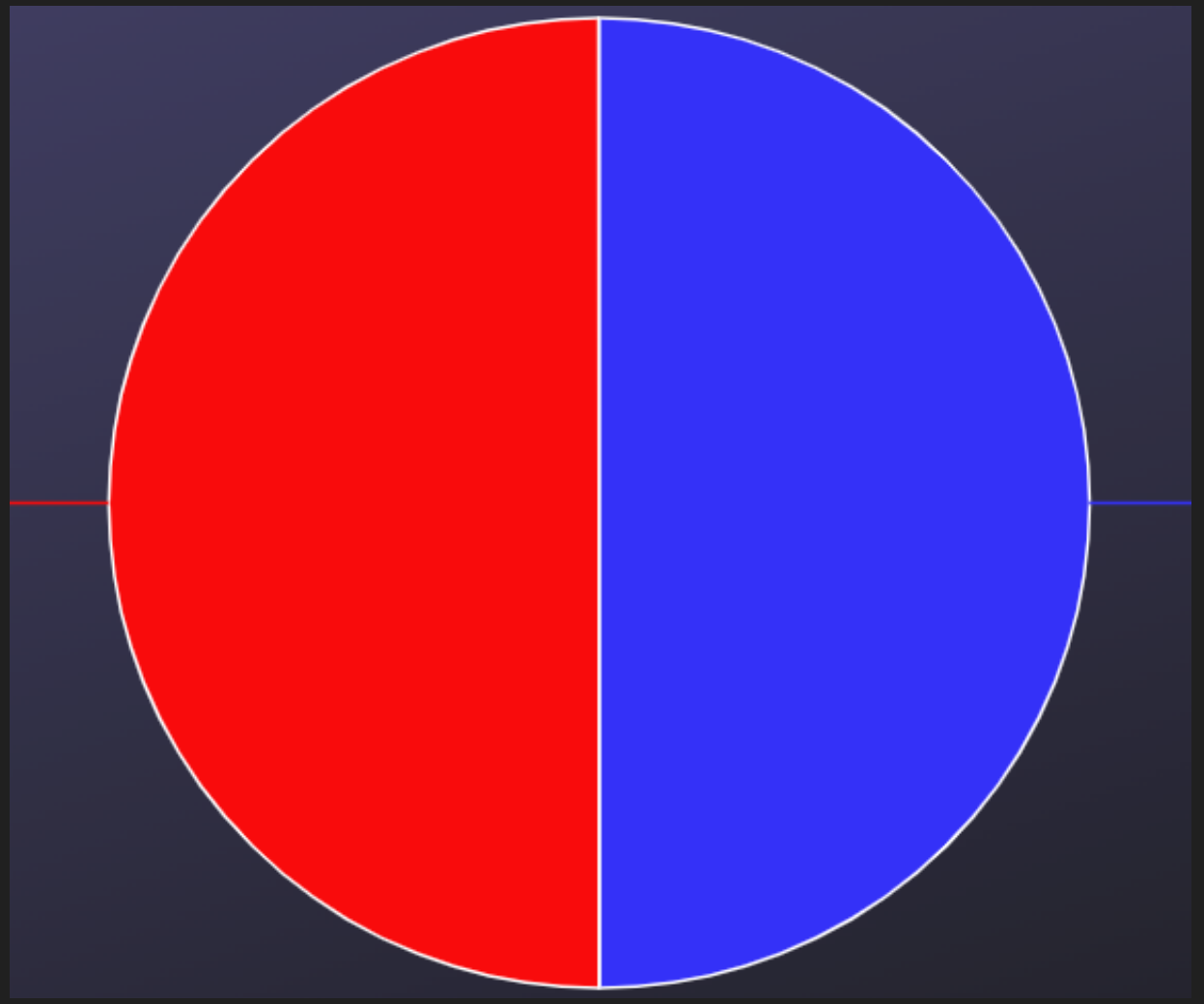
Code that we can't express why it is safe



Safe code

C++ Discussion

Code that we can't express why it is valid

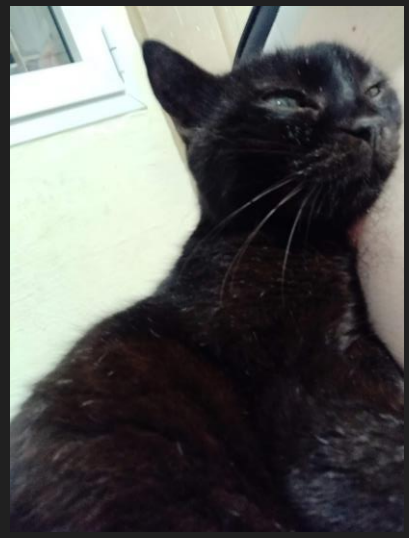


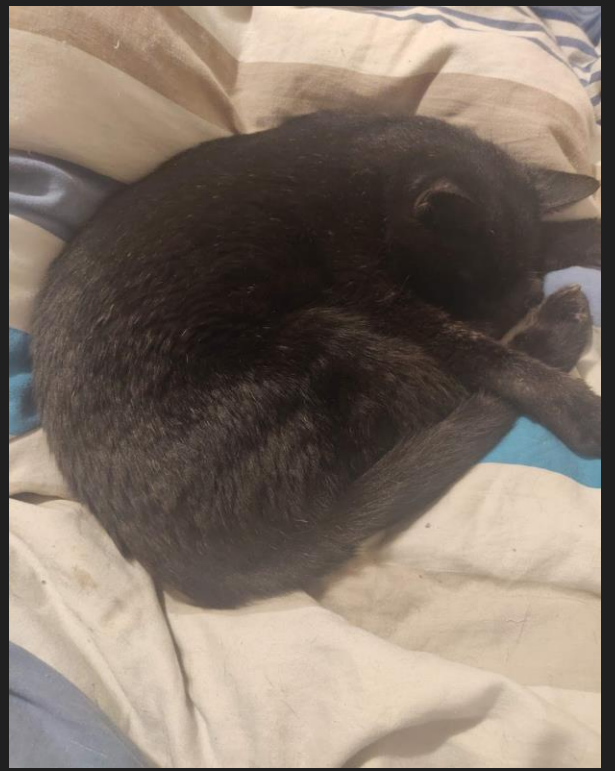
Valid code

Mitzi

- A header only library for creating and validating custom constraints at compile-time.
- Uses C++23 and plans to use C++26 reflection.
- In early development – not ready for production yet.
- Github: https://github.com/a10nw01f/Mitzi
- Compiler Explorer: https://godbolt.org/z/9roK8qdnh

Mitzi





Thank You

Q&A