/* The Code is Documentation Enough */

Tina Ulbrich

ROSEN Technology and Research GmbH

Meeting C++ 2025

ROSEN


P.1: Express ideas
Reason Compilers
programmers (c
Introduction

// Introduction

In.sec: Major sections

- In: Introduction
- P: Philosophy
- I: Interfaces
- F: Functions
- C: Classes and class hierarchies
- Enum: Enumerations
- R: Resource management
- ES: Expressions and statements
- Per: Performance
- CP: Concurrency and parallelism
- E: Error handling
- Con: Constants and immutability
- T: Templates and generic programming
- CPL: C-style programming
- SF: Source files
- SL: The Standard Library

Supporting sections:

- A: Architectural ideas
- NR: Non-Rules and myths
- RF: References
- Pro: Profiles
- GSL: Guidelines support library
- NL: Naming and layout suggestions
- FAQ: Answers to frequently asked questions
- Appendix A: Libraries
- Appendix B: Modernizing code
- Appendix C: Discussion
- Appendix D: Supporting tools
- Glossary
- To-do: Unclassified proto-rules

Source: https://isocpp.github.io/CppCoreGuidelines/

// Introduction

P.1: Express ideas directly in code

Reason

Compilers don’t read comments (or design documents) and neither do many programmers (consistently). What is expressed in code has defined semantics and can (in principle) be checked by compilers and other tools.

Example

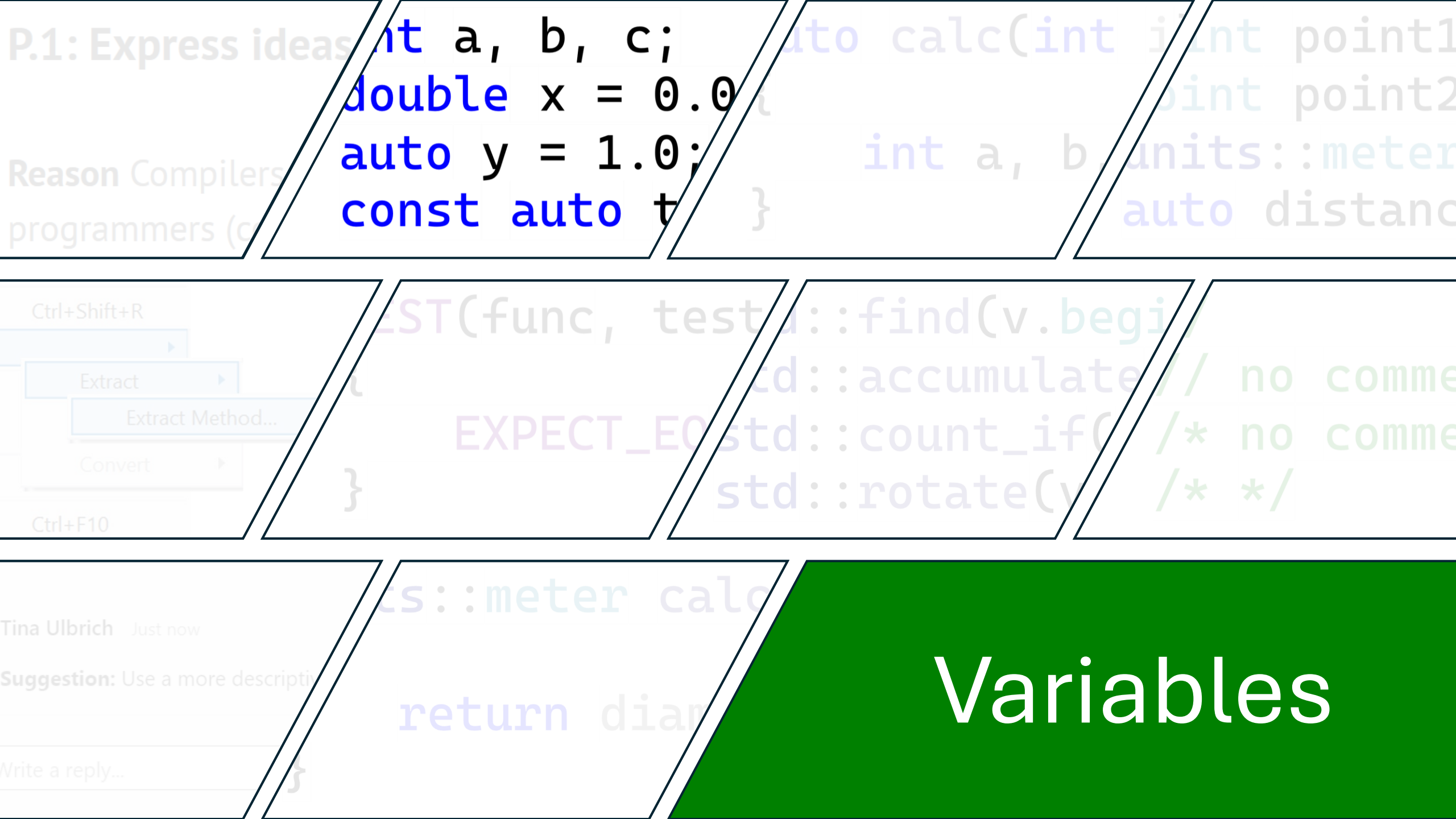
```
class Date {
public:
    Month month() const;  // do
    int month();          // don't
    // ...
};
```

// Introduction

P.2: Write in ISO Standard C++
P.3: Express intent
P.4: Ideally, a program should be statically type safe
P.5: Prefer compile-time checking to run-time checking
P.6: What cannot be checked at compile time should be checkable at run time
P.7: Catch run-time errors early
P.8: Don’t leak any resources
P.9: Don’t waste time or space
P.10: Prefer immutable data to mutable data
P.11: Encapsulate messy constructs, rather than spreading through the code
P.12: Use supporting tools as appropriate
P.13: Use support libraries as appropriate

// Introduction

- Variables
- Functions
- Strong Types
- Organising Code
- Unit Tests
- Language Features
- Comments
- Enforcement


int a, b, c;
double x = 0.0
auto y = 1.0;
const auto t
Variables

// Variables

```cpp
double calc(const std::vector<double>& a, const std::vector<double>& b)
{
    auto s1 = 0.0;
    auto s2 = 0.0;
    for (size_t i = 0; i < a.size(); ++i)
    {
        s1 += a[i] * b[i];
        s2 += b[i];
    }

    return s1 / s2;
}
```

// Variables

```cpp
double weighted_average(
    const std::vector<double>& values,
    const std::vector<double>& weights)
{
    auto sum_product = 0.0;
    auto sum_weights = 0.0;
    for (size_t i = 0; i < values.size(); ++i)
    {
        sum_product += values[i] * weights[i];
        sum_weights += weights[i];
    }

    return sum_product / sum_weights;
}
```

// Variables

```cpp
double weighted_average(
    const std::vector<double>& values,
    const std::vector<double>& weights)
{
    return std::ranges::fold_left(
            std::views::zip_transform(std::multiplies{}, values, weights),
        0.0, std::plus{}) /
        std::ranges::fold_left(weights, 0.0, std::plus{});
}
```

// Variables

```
double weighted_average(
    const std::vector<double>& values,
    const std::vector<double>& weights)
{
    const auto product_of_values_and_weights =
        std::views::zip_transform(std::multiplies{}, values, weights);

    const auto sum_product =
        std::ranges::fold_left(product_of_values_and_weights, 0.0, std::plus{});

    const auto sum_weights = std::ranges::fold_left(weights, 0.0, std::plus{});

    return sum_product / sum_weights;
}
```

// Variables

```
const auto result = std::views::zip(values, weights)
    | std::views::transform([](const auto& elem)
        {
            return std::get<0>(elem) * std::get<1>(elem);
        });
```

// Variables

```
const auto product_of_values_and_weights = std::views::zip(values, weights)
    | std::views::transform([](const auto& elem)
        {
            return std::get<0>(elem) * std::get<1>(elem);
        });
```

// Variables

```
const auto product_of_values_and_weights = std::views::zip(values, weights)
    | std::views::transform([](const auto& value_and_weight)
        {
            return std::get<0>(value_and_weight) * std::get<1>(value_and_weight);
        });
```

// Variables

```
const auto product_of_values_and_weights = std::views::zip(values, weights)
    | std::views::transform(std::multiplies{});
```

// Variables

Use words instead of abbreviations
`w` → `weights`

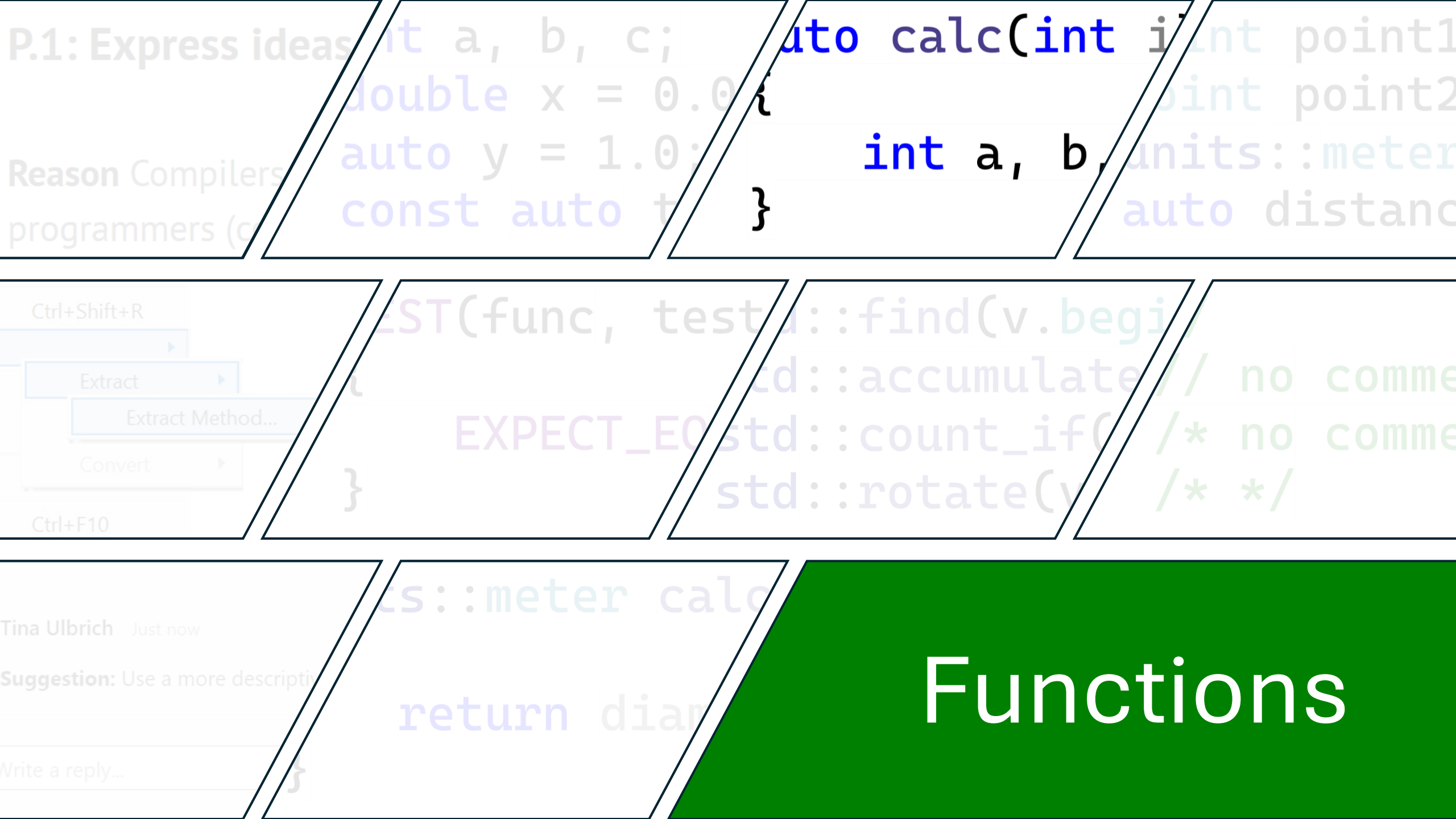
Consistent language and style
`snake_case`, `camelCase`, `PascalCase`

Clear context
`name` within `User` context
`user_name` outside of `User` context

Be specific
`data` → `customer_records`

No redundancies
`user.user_name` → `user.name`

As long as needed, as short as possible
`number_of_users_in_the_database` → `user_count`


auto calc(int i
{
int a, b,
}
Functions

// Functions

```
class Account
{
public:
    explicit Account(const std::string_view account_holder_name)
        : m_account_holder_name(account_holder_name)
    {}

    [[nodiscard]] long balance() const;
    void balance(const long balance);
    void balance(const Transaction& transaction);
    [[nodiscard]] bool valid() const;

private:
    std::string m_account_holder_name;
    long m_balance = 0;
    std::vector<Transaction> m_transactions;
};
```

// Functions

```
class Account
{
public:
    explicit Account(const std::string_view account_holder_name)
        : m_account_holder_name(account_holder_name)
    {}

    [[nodiscard]] long get_balance() const;
    void balance(const long balance);
    void balance(const Transaction& transaction);
    [[nodiscard]] bool valid() const;

private:
    std::string m_account_holder_name;
    long m_balance = 0;
    std::vector<Transaction> m_transactions;
};
```

// Functions

```
class Account
{
public:
    explicit Account(const std::string_view account_holder_name)
        : m_account_holder_name(account_holder_name)
    {}

    [[nodiscard]] long get_balance() const;
    void set_balance(const long balance);
    void balance(const Transaction& transaction);
    [[nodiscard]] bool valid() const;

private:
    std::string m_account_holder_name;
    long m_balance = 0;
    std::vector<Transaction> m_transactions;
};
```

// Functions

```
class Account
{
public:
    explicit Account(const std::string_view account_holder_name)
        : m_account_holder_name(account_holder_name)
    {}

    [[nodiscard]] long get_balance() const;
    void set_balance(const long balance);
    void update_balance(const Transaction& transaction);
    [[nodiscard]] bool valid() const;

private:
    std::string m_account_holder_name;
    long m_balance = 0;
    std::vector<Transaction> m_transactions;
};
```

// Functions

```
class Account
{
public:
    explicit Account(const std::string_view account_holder_name)
        : m_account_holder_name(account_holder_name)
    {}

    [[nodiscard]] long get_balance() const;
    void set_balance(const long balance);
    void update_balance(const Transaction& transaction);
    [[nodiscard]] bool is_balance_valid() const;

private:
    std::string m_account_holder_name;
    long m_balance = 0;
    std::vector<Transaction> m_transactions;
};
```

// Functions

```cpp
class Account
{
public:
    explicit Account(const std::string_view account_holder_name)
        : m_account_holder_name(account_holder_name)
    {}

    [[nodiscard]] long get_balance() const;
    void set_balance(const long balance);
    void update_balance(const Transaction& transaction);
    [[nodiscard]] bool is_balance_valid() const;

private:
    std::string m_account_holder_name;
    long m_balance = 0;
    std::vector<Transaction> m_transactions;
};
```

// Functions

Use verb + object
`calulate_user_id`, `get_user_name`

Boolean functions with prefixes like
`is_`, `has_`, `can_`, `should_`

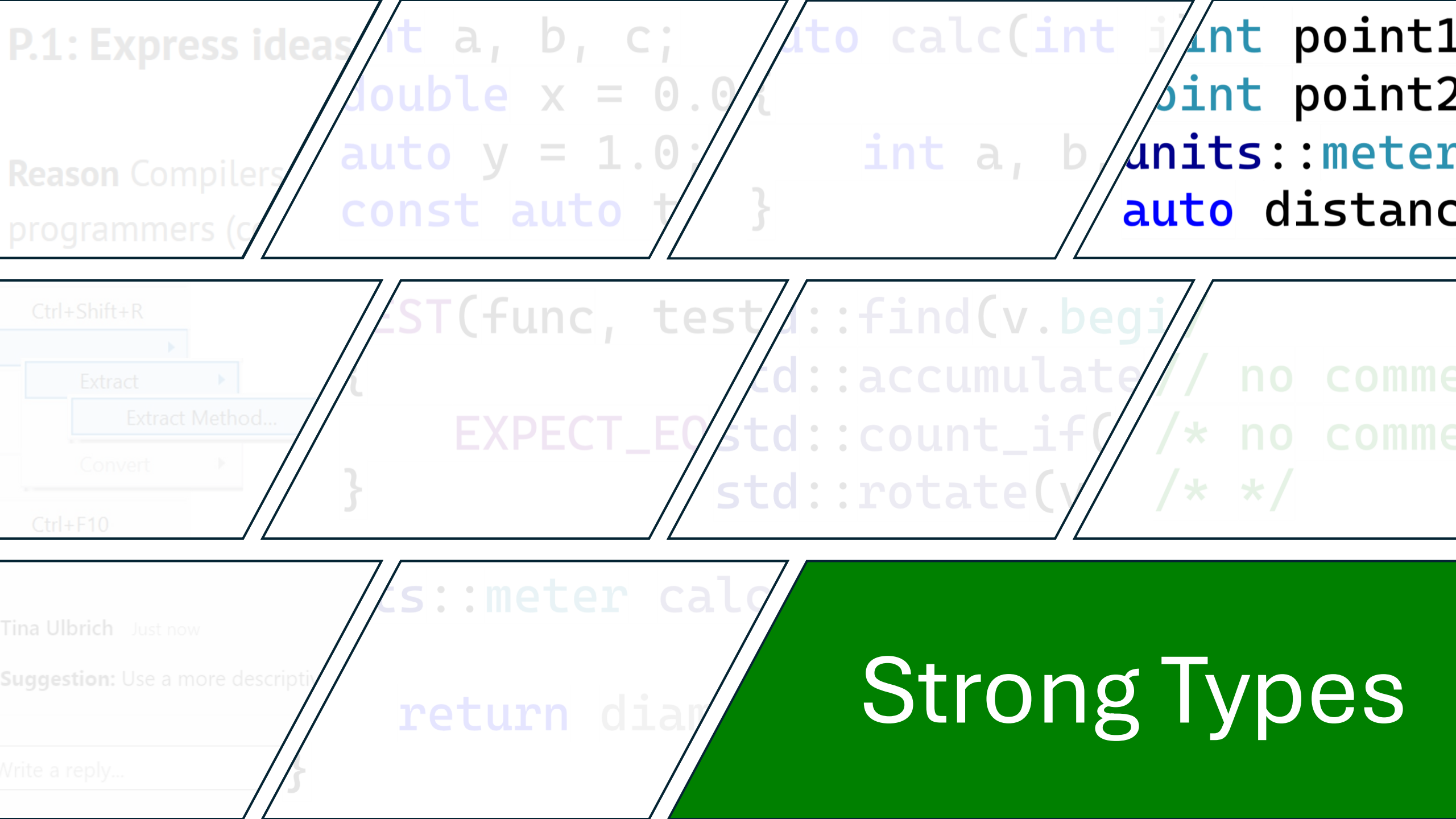
Name side effects
`update_` → changing something

Consistent wording
E.g. always use `get_` to get something

Consistent language and style
`snake_case`, `camelCase`, `PascalCase`

As long as needed, as short as possible
`do_calculate_function` → `calculate`

Use domain language
`adjust_total_price_by_factor` → `apply_discount`


P.1: Express ideas
Reason Compilers
programmers (c
Ctrl+Shift+R
Extract
Extract Method...
Convert
Ctrl+F10
Tina Ulbrich Just now
Suggestion: Use a more descripti
Write a reply...
int a, b, c;
double x = 0.0;
auto y = 1.0;
const auto t
auto calc(int i
int a, b
}
int point1
oint point2
units::meter
auto distanc
ST(func, test
{
EXPECT_EQ
}
std::find(v.begi
std::accumulate
std::count_if(
std::rotate(v
// no comme
/* no comme
/* */
ts::meter calc
return dian
}
Strong Types

// Strong Types

```
double calculate_speed(const double distance, const double time)
{
    return distance / time;
}
```

```
const double speed = calculate_speed(200.0, 10.0);
```

// Strong Types

```
double calculate_speed(const double distance, const double time)
{
    return distance / time;
}
```

```
const double speed_in_metre_per_second = calculate_speed(200.0, 10.0);
```

// Strong Types

```
double calculate_speed(const double distance, const double time)
{
    return distance / time;
}
```

```
const double speed_in_metre_per_second = calculate_speed(10.0, 200.0);
```

// Strong Types

```
struct DistanceAndTime
{
    double distance = 0.0;
    double time = 0.0;
};

double calculate_speed(const DistanceAndTime& distance_and_time)
{
    return distance_and_time.distance / distance_and_time.time;
}
```

// Strong Types

```
struct DistanceAndTime
{
    double distance = 0.0;
    double time = 0.0;
};

double calculate_speed(const DistanceAndTime& distance_and_time)
{
    const auto [distance, time] = distance_and_time;
    return distance / time;
}
```

```
const double speed = calculate_speed({ .distance = 100.0, .time = 10.0 });
```

// Strong Types

```
using metre = double;
using second = double;
using metres_per_second = double;

metres_per_second calculate_speed(const metre distance, const second time)
{
    return distance / time;
}
```

// Strong Types

```
using metre = double;
using second = double;
using metres_per_second = double;

metres_per_second calculate_speed(const metre distance, const second time)
{
    return distance / time;
}
```

```
const auto speed = calculate_speed(100.0, 10.0);
```

// Strong Types

```
using metre = double;
using second = double;
using metres_per_second = double;

metres_per_second calculate_speed(const metre distance, const second time)
{
    return distance / time;
}
```

```
const metres_per_second speed = calculate_speed(metre{ 100.0 }, second{ 10.0 });
```

// Strong Types

```
using metre = double;
using second = double;
using metres_per_second = double;

metres_per_second calculate_speed(const metre distance, const second time)
{
    return distance / time;
}
```

```
const metre speed = calculate_speed(second{ 100.0 }, metres_per_second{ 10.0 });
```

// Strong Types

```
struct metre { double value = 0.0; };
struct second { double value = 0.0; };
struct metres_per_second { double value = 0.0; };

metres_per_second calculate_speed(const metre distance, const second time)
{
    return distance / time;
}
```

```
const metres_per_second speed = calculate_speed(metre{ 100.0 }, second{ 10.0 });
```

// Strong Types

```
struct metre { double value = 0.0; };
struct second { double value = 0.0; };
struct metres_per_second { double value = 0.0; };

metres_per_second operator/(const metre distance, const second time)
{
    return metres_per_second{ distance.value / time.value };
}

metres_per_second calculate_speed(const metre distance, const second time)
{
    return distance / time;
}
```

```
const metres_per_second speed = calculate_speed(metre{ 100.0 }, second{ 10.0 });
```

// Strong Types

```cpp
#include <mp-units/systems/si/si.h>

using namespace mp_units;
using namespace mp_units::si::unit_symbols;

constexpr auto metres_per_second = si::metre / si::second;

quantity<metres_per_second> calculate_speed(
    const quantity<si::metre> distance, const quantity<si::second> time)
{
    return distance / time;
}
```

```cpp
const quantity<metres_per_second> speed = calculate_speed(100.0 * m, 10.0 * s);
```

// Strong Types

strong types:

boost strong typedef

type_safe by Jonathan Müller

NamedType by Jonathan Boccara

PSsst by Peter Sommerlad

strong_type by Björn Fahler

strong_typedef by Anthony Williams

units:

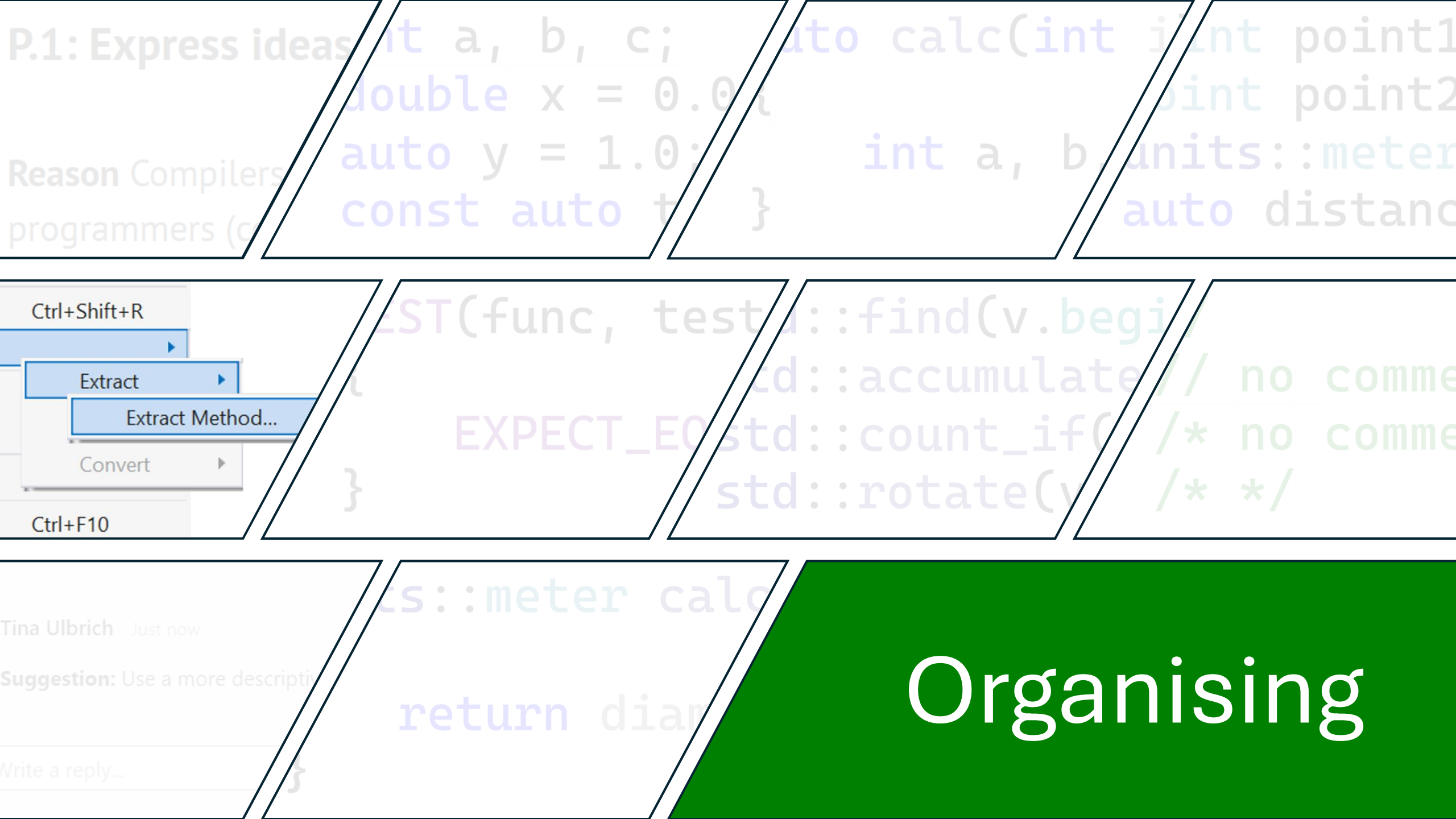
mp_units by Mateusz Pusz

boost units

Au units by Aurora

// Strong Types

- Prevent mix up of parameters
- Self documenting interfaces
- Checks at the type itself
- Less implicit conversions
- Libraries support available



Organising

// Organising

```
void analyze(const std::span<const double> nums)
{
    if (nums.empty()) return;

    double sum = 0;
    double min_val = nums[0];
    double max_val = nums[0];
    for (size_t i = 0; i < nums.size(); i++)
    {
        // calculate sum
        sum += nums[i];

        // calculate min
        if (nums[i] < min_val) min_val = nums[i];

        // calculate max
        if (nums[i] > max_val) max_val = nums[i];
    }

    // calculate mean
    const double mean = sum / nums.size();
...
```

```
...
    // calculate variance
    const auto variance_sum =
        std::ranges::fold_left(nums, 0.0,
        [mean](const double acc, const double x) {
            return acc + (x - mean) * (x - mean);
        });

    const double variance = variance_sum /
        (nums.size() - 1);

    // calculate stddev
    const double stddev = std::sqrt(variance);

    // print statistics
    std::print("Count: {}\n", nums.size());
    std::print("Sum: {}\n", sum);
    std::print("Mean: {}\n", mean);
    std::print("Variance: {}\n", variance);
    std::print("Stddev: {}\n", stddev);
    std::print("Min: {}\n", min_val);
    std::print("Max: {}\n", max_val);
}
```

// Organising

```cpp
double calculate_sum(
    const std::span<const double> nums) {
    return std::ranges::fold_left(
        nums, 0.0, std::plus{});
}

double calculate_mean(
    const std::span<const double> nums) {
    return calculate_sum(nums) / nums.size();
}

double calculate_variance(
    const std::span<const double> nums) {
    const double mean = calculate_mean(nums);
    const auto variance_sum =
        std::ranges::fold_left(nums, 0.0,
        [mean](const double acc, const double x) {
        return acc + (x - mean) * (x - mean);
        });
    return variance_sum / (nums.size() - 1);
}
```

```cpp
double calculate_stddev(
    const std::span<const double> nums) {
    return std::sqrt(calculate_variance(nums));
}

void print_statistics(
    const std::span<const double> nums) {
    std::print("Count: {}\n", nums.size());
    std::print("Sum: {}\n", calculate_sum(nums));
    std::print("Mean: {}\n", calculate_mean(nums));
    std::print("Variance: {}\n",
        calculate_variance(nums));
    std::print("Stddev: {}\n",
        calculate_stddev(nums));
    std::print("Min: {}\n", std::ranges::min(nums));
    std::print("Max: {}\n", std::ranges::max(nums));
}
```

// Organising

```
const double mean = calculate_mean(nums);

const double variance_sum = std::ranges::fold_left(nums, 0.0,
    [mean](const double acc, const double x) {
        return acc + (x - mean) * (x - mean);
        });

const double variance = variance_sum / (nums.size() - 1);

const double stddev = std::sqrt(variance);
```

// Organising

```
const double mean = calculate_mean(nums);
const double variance = calculate_variance(mean, nums);
const double stddev = std::sqrt(variance);
```

// Organising

```
void draw_line(const double x1, const double y1, const double x2, const double y2)
{
    ...
}

struct Point
{
    double x;
    double y;
};

void draw_line(const Point& point1, const Point& point2)
{
    ...
}
```

// Organising

```
start_game(true, true, false);
```

```
start_game(Mode::debug, Logging::enable, GodMode::disable);
```

// Organising

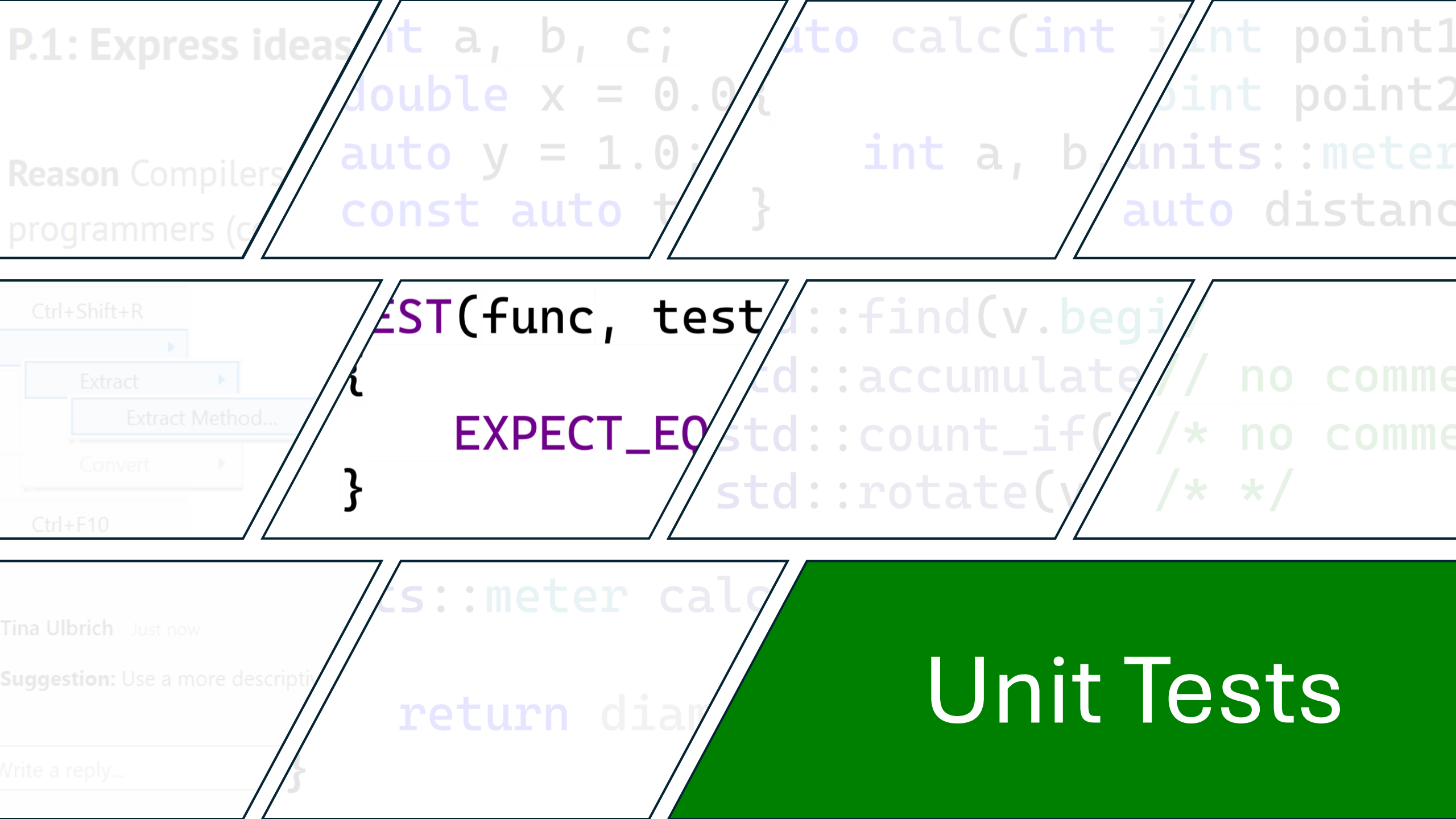
```
void process(const int x) {
    if (x > 0) {
        if (x < 100) {
            if (x % 2 == 0) {
                std::cout << "valid\n";
            }
        }
    }
}

void process(const int x) {
    if (x <= 0) return;
    if (x >= 100) return;
    if (x % 2 != 0) return;

    std::cout << "valid\n";
}
```

// Organising

- Limit the number of lines in functions
- Limit number of parameters to ~ 3
- Keep nesting depth to max 3
- Keep one layer of abstraction
- Use refactoring tools


EST(func, test
{
EXPECT_EQ
}
Unit Tests

Meeting C++ 2021

Program with GUTs

@KevlinHenney

// Unit Tests

- MethodName_StateUnderTest_ExpectedBehavior
  `CalculateMean_EmptyInput_ThrowsException`

- Given_When_Then
  `GivenEmptyInput_WhenCalculatingMean_ThenThrowsException`

- Should_ExpectedBehavior_When_State
  `ShouldThrowException_WhenInputIsEmpty`

- Descriptive Sentence
  `test_calculate_mean_throws_exception_for_empty_input`

// Unit Tests

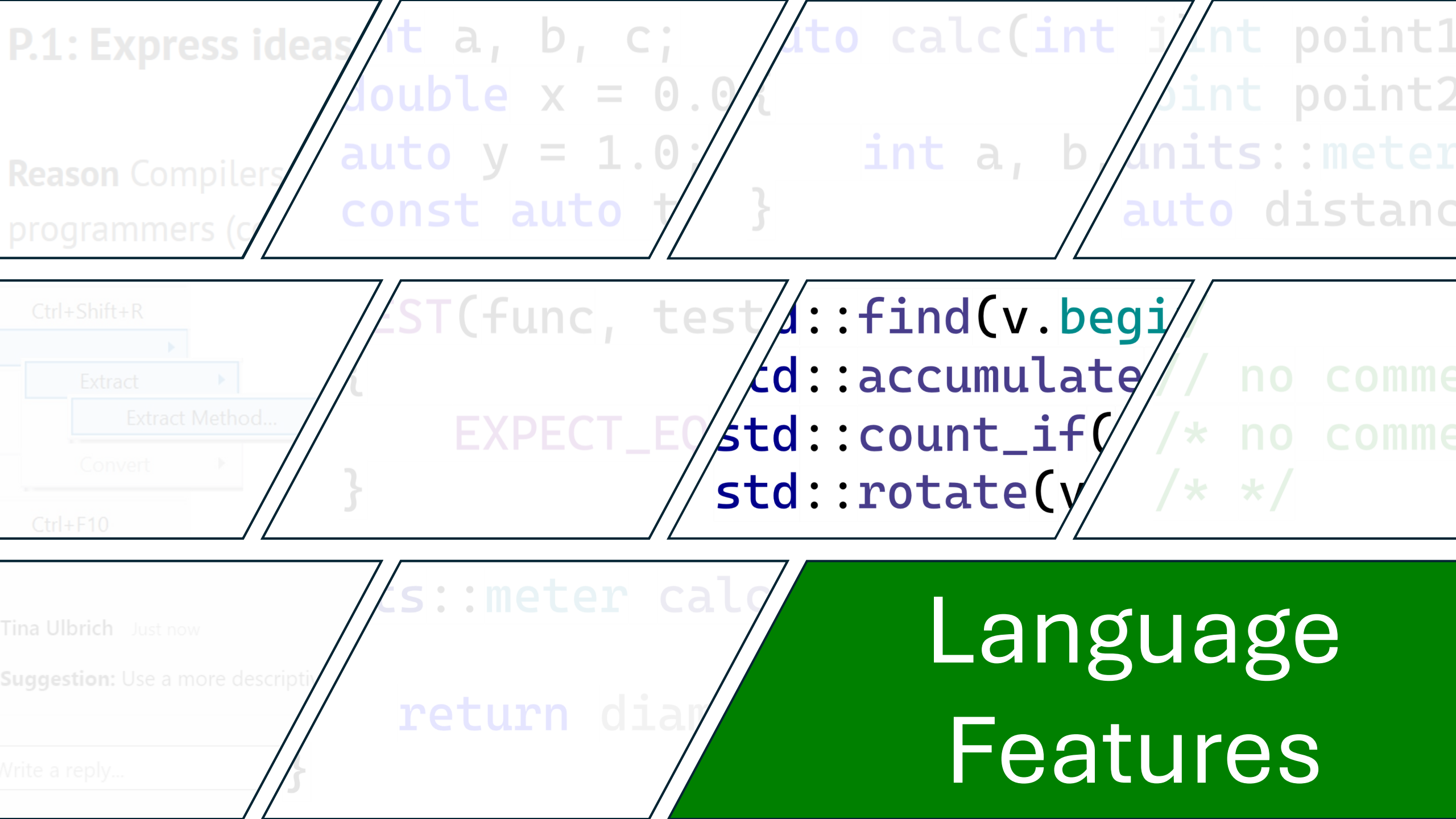
```
TEST(calculate_mean, returns_mean_when_input_is_not_empty)
{
    EXPECT_DOUBLE_EQ(calculate_mean({ 1., 2., 3., 4., 5. }), 3.);
}
```

// Unit Tests

```
TEST(calculate_mean, throws_exception_when_input_is_empty)
{
    EXPECT_THROW(calculate_mean({}), std::runtime_error);
}
```

```
TEST(calculate_mean, returns_the_unexpected_value_when_input_is_empty)
{
    EXPECT_EQ(calculate_mean({}).error().what(), "Input must not be empty");
}
```

```
TEST(calculate_mean, returns_zero_when_input_is_empty)
{
    EXPECT_EQ(calculate_mean({}), 0.0);
}
```


P.1: Express ideas
Reason Compilers
programmers (c
Ctrl+Shift+R
Extract
Extract Method...
Convert
Ctrl+F10
Tina Ulbrich Just now
Suggestion: Use a more descripti
Write a reply...
std::find(v.begi
std::accumulate
std::count_if(
std::rotate(v
Language
Features

// Language Features

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

Language
- Preprocessor – Comments
- ASCII chart
- Basic concepts
  - Keywords
  - Names (lookup)
  - Types (fundamental types)
  - The `main` function
  - Modules (C++20)
  - Contracts (C++26)
- Expressions
  - Value categories
  - Evaluation order
  - Operators (precedence)
  - Conversions – Literals
  - Constant expressions
- Statements
  - `if` – `switch`
  - `for` – range-`for` (C++11)
  - `while` – `do-while`
- Declarations – Initialization
- Functions – Overloading
- Coroutines (C++20)
- Classes (unions)
- Templates – Exceptions
- Freestanding implementations

Standard library (headers)

Named requirements

Language support library
- Program utilities
  - Signals – Non-local jumps
- Basic memory management
- Variadic functions
- `source_location` (C++20)
- Comparison utilities (C++20)
- Type support – `type_info`
- `numeric_limits` – `exception`
- `initializer_list` (C++11)
- Coroutine support (C++20)
- Contract support (C++26)

Concepts library (C++20)

Diagnostics library
- Assertions – System error (C++11)
- Exception types – Error numbers
- `basic_stacktrace` (C++23)
- Debugging support (C++26)

Memory management library
- Allocators – Smart pointers
- Memory resources (C++17)

Metaprogramming library (C++11)
- Type traits – `ratio`
- `integer_sequence` (C++14)

General utilities library
- Function objects – `hash` (C++11)
- Swap – Type operations (C++11)
- Integer comparison (C++20)
- `pair` – `tuple` (C++11)
- `optional` (C++17)
- `expected` (C++23)
- `variant` (C++17) – `any` (C++17)
- `bitset` – Bit manipulation (C++20)

Containers library
- `vector` – `deque` – `array` (C++11)
- `list` – `forward_list` (C++11)
- `inplace_vector` (C++26)
- `hive` (C++26)
- `map` – `multimap` – `set` – `multiset`
- `unordered_map` (C++11)
- `unordered_multimap` (C++11)
- `unordered_set` (C++11)
- `unordered_multiset` (C++11)
- Container adaptors
- `span` (C++20) – `mdspan` (C++23)

Iterators library

Ranges library (C++20)
- Range factories – Range adaptors
- `generator` (C++23)

Algorithms library
- Numeric algorithms
- Execution policies (C++17)
- Constrained algorithms (C++20)

Strings library
- `basic_string` – `char_traits`
- `basic_string_view` (C++17)

Text processing library
- Primitive numeric conversions (C++17)
- Formatting (C++20) – Localization
- `text_encoding` (C++26)
- Regular expressions (C++11)
  - `basic_regex` – Algorithms
  - Default regular expression grammar
- Null-terminated sequence utilities:
  - byte – multibyte – wide

Numerics library
- Common math functions
- Mathematical special functions (C++17)
- Mathematical constants (C++20)
- Basic linear algebra algorithms (C++26)
- Data-parallel types (SIMD) (C++26)
- Pseudo-random number generation
- Floating-point environment (C++11)
- `complex` – `valarray`

Date and time library
- Calendar (C++20) – Time zone (C++20)

Input/output library
- Print functions (C++23)
- Stream-based I/O – I/O manipulators
- `basic_istream` – `basic_ostream`
- Synchronized output (C++20)
- File systems (C++17)

Concurrency support library (C++11)
- `thread` – `jthread` (C++20)
- `atomic` – `atomic_flag`
- `atomic_ref` (C++20) – `memory_order`
- Mutual exclusion – Condition variables
- Futures – Semaphores (C++20)
- `latch` (C++20) – `barrier` (C++20)
- Safe Reclamation (C++26)

Execution support library (C++26)

Feature test macros (C++20)

Language – Standard library – Headers

Source: https://www.cppreference.com/

// Language Features

```
double calculate_variance(const std::span<const double> nums)
{
    const double mean = calculate_mean(nums);
    const double variance_sum =
        std::ranges::fold_left(nums, 0.0, [mean](const double acc, const double x) {
            return acc + (x - mean) * (x - mean);
        });

    return variance_sum / (nums.size() - 1);
}
```

// Language Features

```
template <typename T> requires std::ranges::random_access_range<T>
auto median(T& range)
{
    ...
}

template <std::ranges::random_access_range T>
auto median(T& range)
{
    ...
}

auto median(std::ranges::random_access_range auto& range)
{
    ...
}
```

// Language Features

```cpp
double calculate_mean(const std::span<const double> nums)
{
    if (nums.empty()) throw std::invalid_argument("Input must not be empty");

    const auto mean = calculate_sum(nums) / nums.size();

    if (mean >= std::ranges::min(nums)) std::terminate();

    return mean;
}
```

// Language Features

```
double calculate_mean(const std::span<const double> nums)
{
    Expects(!nums.empty());

    const auto mean = calculate_sum(nums) / nums.size();

    Ensures(mean >= std::ranges::min(nums));

    return mean;
}
```

// Language Features

```
double calculate_mean(const std::span<const double> nums)
    pre (!nums.empty());
    post (mean: mean >= std::ranges::min(nums))
```

Timur Doumler

Contracts for C++

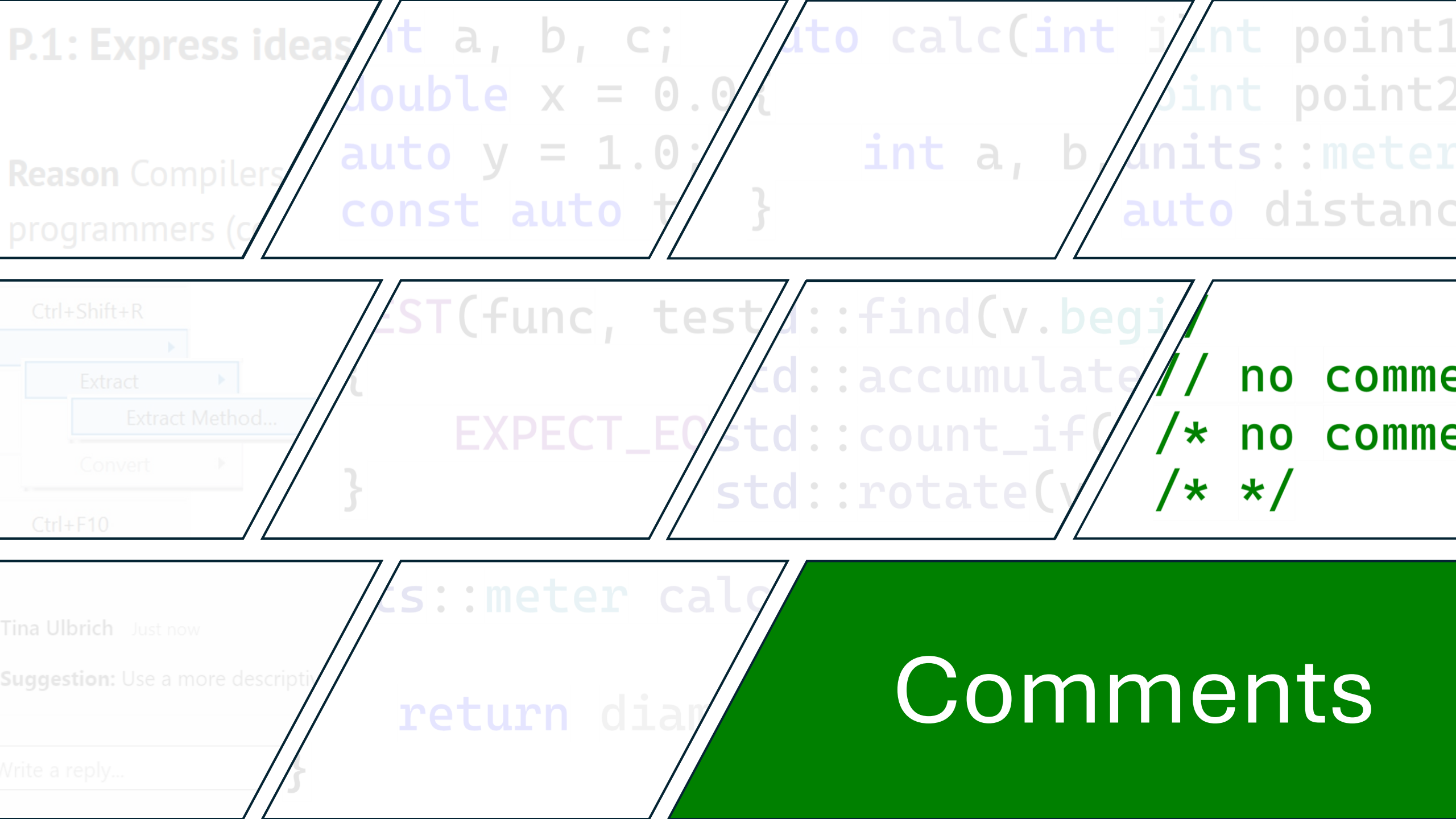
Version 2.0

Timur Doumler

ACCU
1 April 2025

// Language Features

- Standard libraries are part of the language
- (almost) always const
- Concepts for compile time validations of template parameters
- Contracts for pre and post conditions


// no comme
/* no comme
/* */
Comments

// Comments

```
// increment i by 1
++i;
```

Redundant documentation

```
// calculate the mean of the input
const auto median = calculate_median(input);
```

Outdated documentation

```
// TODO: fix later
workaround();
```

TODOs in code are often forgotten

```
// Here be dragons
calculate();
```

Jokes or insider comments

// Comments

```
// stable_sort to preserve the sorting of width for equal length
ranges::stable_sort(rectangles, less{}, &rectangle::length);
```

“Why” comment

```
// Workaround for GCC bug #12345: use explicit cast
const auto value = static_cast<int>(input);
```

Documentation of workarounds

```
// Premium customers get a discount
const auto new_price = is_premium ? price * 0.9 : price;
```

Non-obvious internal logic

```
// ReSharper disable once CppLocalVariableMayBeConst
auto short_generator = std::uniform_int_distribution<short>('a', 'z');
```

Disable static analysis warning

// Comments

```
def hankel(c, r=None):
    """
    Construct a Hankel matrix.

    The Hankel matrix has constant anti-diagonals, with `c` as its ...
```

Explain complex calculations

```
// Specialised for 3x3 matrix
auto solve(const auto a, const auto b);
```

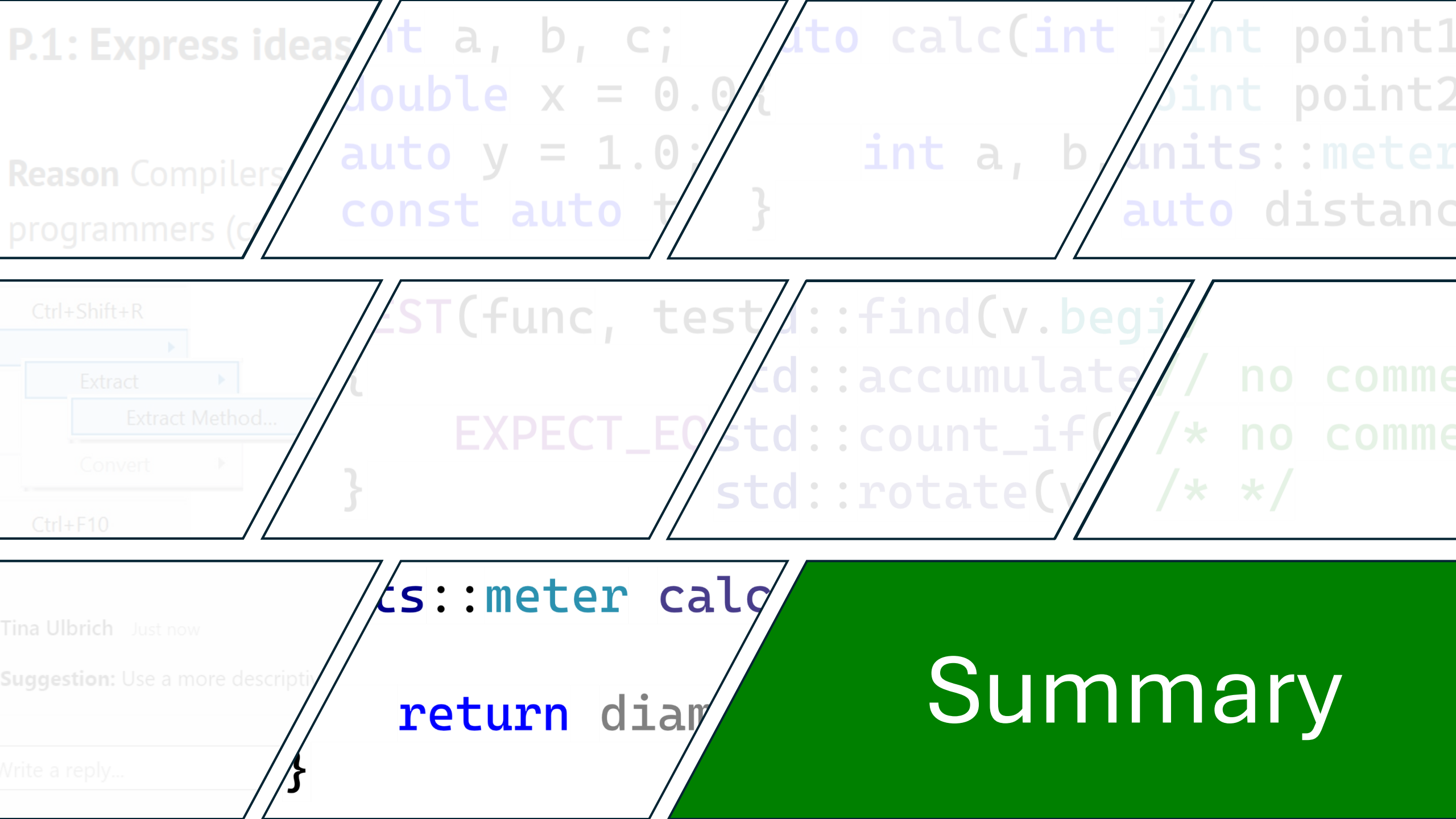
Specialised code

```
def round(a, decimals=0, out=None):
    """
    Evenly round to the given number of decimals.
    Parameters
    ----------
    a : array_like
        Input data. ...
```

Document external APIs


Tina Ulbrich Just now
Suggestion: Use a more descriptiv
Write a reply...
Enforcement

// Enforcement


Summary

// Summary

- Good and descriptive names for variables and functions
- Strong types for improved readability and safety
- Consistent code structure, supported by tools
- Unit tests are documentation too
- Language features and standard library improve readability
- Comment when clarification is needed
- Code reviews to ensure readability
- Be consistent but not too dogmatic

/* Thank you!
Questions? */

Tina Ulbrich


@tinaul@mastodon.social


@tinaul.bsky.social

tinaul

ROSEN