Type-Safe Programming

Jonathan Müller

@foonathan

Easy to Use Correctly and Hard to Use Incorrectly

Users typically smart and motivated:

- Experienced with software.
- Willing to read a little documentation.
- *Want* to use your intefaces correctly.
- **If they don't, *it's your fault*.**

Hence:

- Action is possible ⇒ should almost always do what's expected.
- Action unlikely to do what's expected ⇒ should generally be impossible.

CONSISTENCY!

> If you pass a sequence of parameters which are not going to do the right thing, get the type system to reject it.
>
> Use the type system to encourage people to write calls which are almost certainly going to work.

Why types?

```
pythagoras(int, int):
    imul    edi, edi
    imul    esi, esi
    lea     eax, [rsi + rdi]
    ret
```

```
    mov     edi, dword ptr [rbp - 4]
    mov     esi, dword ptr [rbp - 8]
    call    pythagoras(int, int)
```

Why types?

```
void pythagoras(void* output, void* a, void* b)
{
    auto a_int = *static_cast<int*>(a);
    auto b_int = *static_cast<int*>(b);
    auto a_sqr = a_int * a_int;
    auto b_sqr = b_int * b_int;
    *static_cast<int*>(output) = a_sqr + b_sqr;
}
```

```
int a = 3;
int b = 4;
int res;
pythagoras(&res, &a, &b);
```

Why types?

```
void pythagoras(void* output, void* a, void* b)
{
    auto a_int = *static_cast<int*>(a);
    auto b_int = *static_cast<int*>(b);
    auto a_sqr = a_int * a_int;
    auto b_sqr = b_int * b_int;
    *static_cast<int*>(output) = a_sqr + b_sqr;
}
```

```
int a = 3;
int b = 4;
short res; // ups!
pythagoras(&res, &a, &b);
```

What is a *type*?

What is a *type*?

> [...] A data type [...] is a classification of data which tells the compiler [...] how the programmer intends to use the data. [https://en.wikipedia.org/wiki/Data_type]

> The main purpose of a type system is to reduce possibilities for bugs in computer programs. [https://en.wikipedia.org/wiki/Type_system]

Why types?

```
public int pythagoras(int a, int b)
{
    int a_sqr = a * a;
    int b_sqr = b * b;
    return a_sqr + b_sqr;
}
```

```
int a = 3;
int b = 4;
int res = pythagoras(a, b);
```

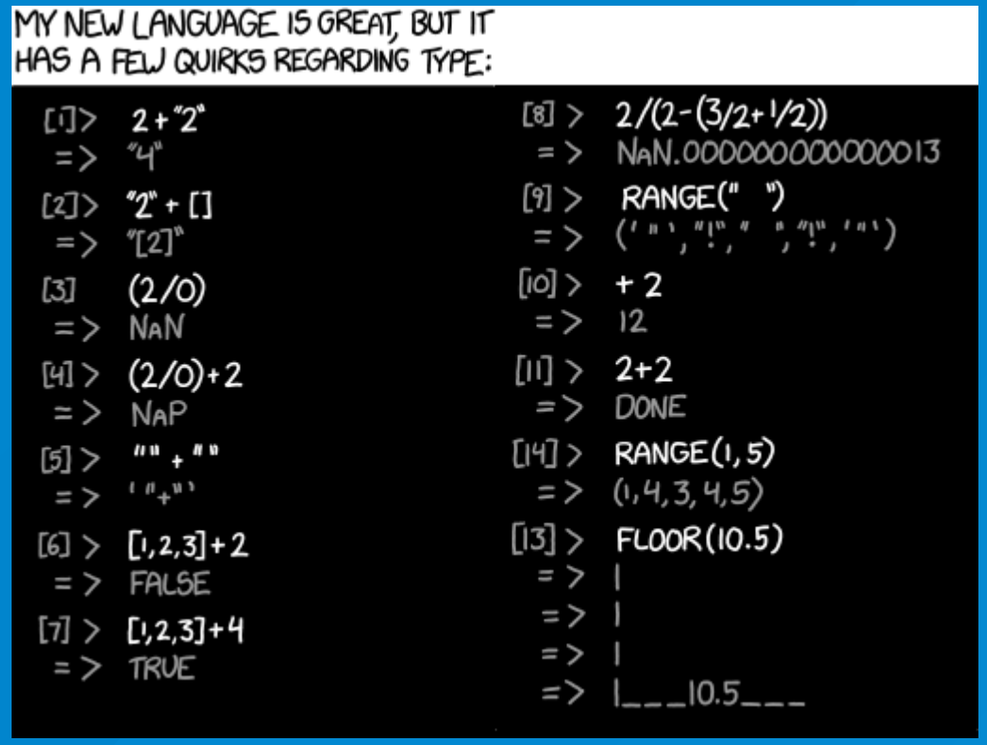
Why types?

```
public int pythagoras(int a, int b)
{
    int a_sqr = a * a;
    int b_sqr = b * b;
    return a_sqr + b_sqr;
}
```

```
int a = 3;
int b = 4;
short res = pythagoras(a, b); // compiler error!
```

This is *type safety*.

This isn't.



[XKCD 1537]

Type-safe Programming:

Trick the compiler into preventing API misuse by (ab-)using the type system.

Interlude:
Type safety in C++

What's the problem with this code?

```
int at_or(const std::vector<int> &vec, int index, int fallback)
{
    if (index < vec.size())
        return vec[index];
    return fallback;
}
```

```
std::vector<int> vec = {1, 2, 3, 4};
auto a = at_or(vec, 3, 42);   // 4
auto b = at_or(vec, 128, 42); // 42
```

What's the problem with this code?

```
int at_or(const std::vector<int> &vec, int index, int fallback)
{
    if (index < vec.size())
        return vec[index];
    return fallback;
}
```

```
std::vector<int> vec = {1, 2, 3, 4};
auto a = at_or(vec, 3, 42);   // 4
auto b = at_or(vec, 128, 42); // 42
auto c = at_or(vec, -1, 42);  // ???
```

What's the problem with this code?

```
int at_or(const std::vector<int> &vec, int index, int fallback)
{
    if (index < vec.size())
        return vec[index];
    return fallback;
}
```

```
std::vector<int> vec = {1, 2, 3, 4};
auto a = at_or(vec, 3, 42);   // 4
auto b = at_or(vec, 128, 42); // 42
auto c = at_or(vec, -1, 42);  // 42
```

What's the problem with this code?

```
int at_or(const std::vector<int> &vec, int index, int fallback)
{
    if (index < vec.size())
        return vec[index];
    return fallback;
}
```

```
std::vector<int> vec2(std::size_t(INT_MIN) + 1);
vec2.push_back(4);
auto d = at_or(vec2, INT_MIN, 42); // ???
```

What's the problem with this code?

```
int at_or(const std::vector<int> &vec, int index, int fallback)
{
    if (index < vec.size())
        return vec[index];
    return fallback;
}
```

```
std::vector<int> vec2(std::size_t(INT_MIN) + 1);
vec2.push_back(4);
auto d = at_or(vec2, INT_MIN, 42); // 4
```

Sign conversions are *fun*.

What's the problem with this code?

```
bool contains(const std::vector<int>& vec, int value)
{
    auto low = 0u; // prevent sign mismatch
    auto high = vec.size();
    while (low < high)
    {
        auto middle = low + (high - low) / 2;
        if (vec[middle] == value) return true;
        else if (vec[middle] > value)
            high = middle - 1u;
        else if (vec[middle] < value)
            low = middle + 1u;
    }

    return false;
}
```

~~Sign~~ Conversions are *fun*.

So, C++ isn't type-safe?

But Compilers should warn about that!

Call if call possible

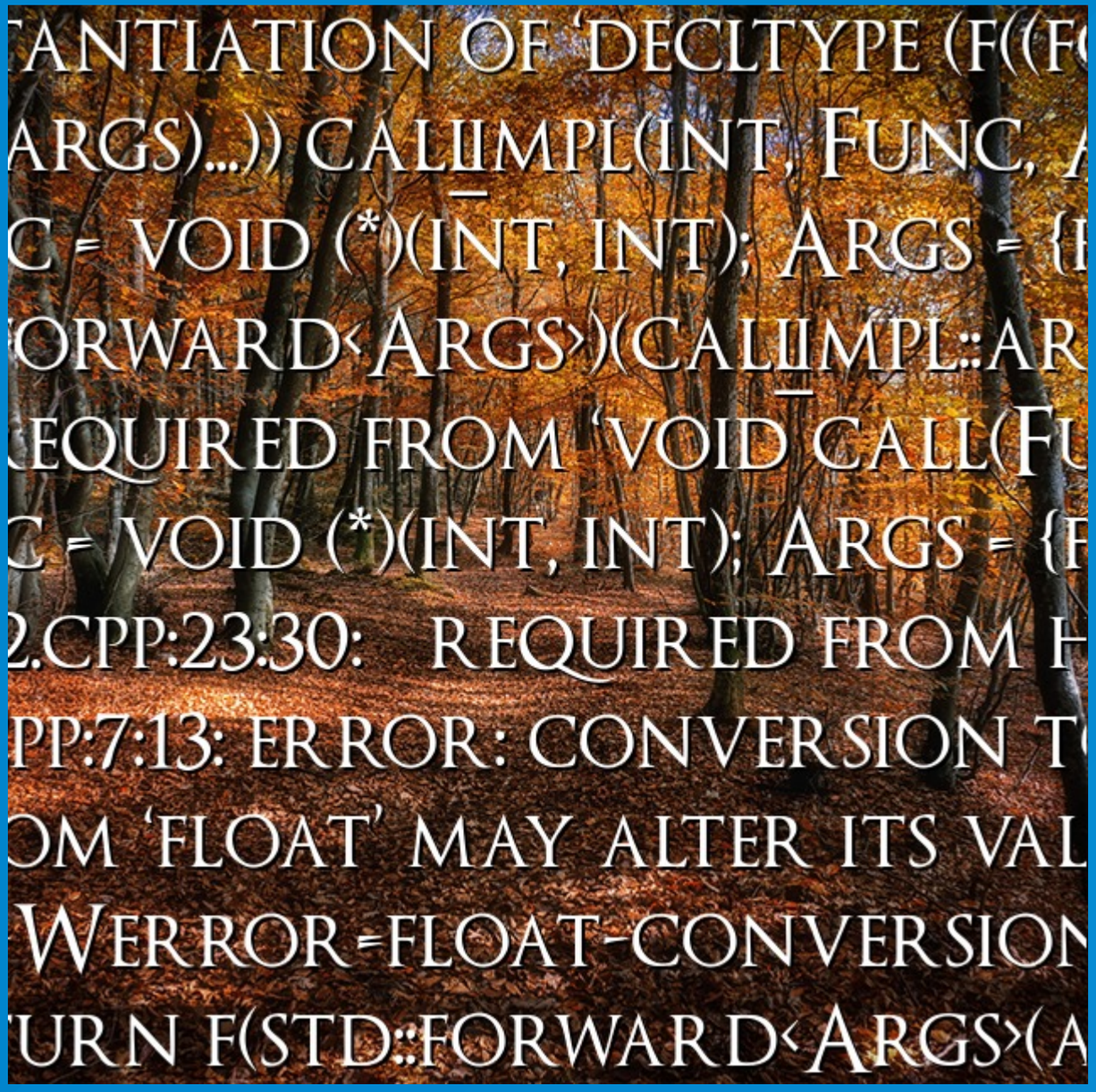
```
template <typename Func, typename ... Args>
auto call_impl(int, Func f, Args&&... args)
-> decltype(f(std::forward<Args>(args)...)
{
    return f(std::forward<Args>(args)...);
}

template <typename Func, typename ... Args>
void call_impl(short, Func, Args&&...) {}

template <typename Func, typename ... Args>
void call(Func f, Args&&... args)
{
    call_impl(0, f, std::forward<Args>(args)...);
}
```

Usage

```
void some_func(int a, int b);
…
call(some_func, 3.14, 42);
```


ANTIATION OF 'DECLTYPE (F((F
ARGS)...)) CALLIMPL(INT, FUNC, A
C = VOID (*)(INT, INT); ARGS = {F
ORWARD<ARGS>)(CALLIMPL::AR
EQUIRED FROM 'VOID CALL(FU
C = VOID (*)(INT, INT); ARGS = {F
2.CPP:23:30: REQUIRED FROM H
PP:7:13: ERROR: CONVERSION T
OM 'FLOAT' MAY ALTER ITS VAL
WERROR=FLOAT-CONVERSION
URN F(STD::FORWARD<ARGS>(A

SFINAW

Substitution Failure IgNores All Warnings

SFINAW - EWW

Substitution Failure IgNores All Warnings

-

Even With `-WError`

Let's make C++ type-safe!

A type-safe integer:

```
ts::integer<int> a;
ts::integer<unsigned> b;
ts::integer<short> c;

b = a; // error!
c = a; // error!
a = c; // okay

a < b; // error!
a < c; // okay

a + b; // error!
a + c; // okay, yields ts::integer<int>
```

Let's make C++ type-safe!

What's an integer anyway?

```
template <typename T>
constexpr bool is_integer = std::is_integral_v<T>;
```

Let's make C++ type-safe!

What's an integer anyway?

```
template <typename T>
constexpr bool is_integer = std::is_integral_v<T>
                && !std::is_same_v<T, bool>;
```

Let's make C++ type-safe!

What's an integer anyway?

```
template <typename T>
constexpr bool is_integer = std::is_integral_v<T>
                && !std::is_same_v<T, bool>
                && !std::is_same_v<T, char>;
```

Let's make C++ type-safe!

Safe operations:

```
template <typename A, typename B>
constexpr bool is_safe_integer_operation
    = is_integer<From> && is_integer<To>
      && std::is_signed_v<From> == std::is_signed_v<To>;
```

Let's make C++ type-safe!

Operation result:

```
template <typename A, typename B>
using bigger_type
    = std::conditional_t<sizeof(A) < sizeof(B), B, A>;

template <typename A, typename B>
using integer_result_type
    = std::enable_if_t<is_safe_integer_operation<A, B>, bigger_type<A, B>>;
```

Let's make C++ type-safe!

Safe conversion:

```
template <typename From, typename To>
constexpr bool is_safe_integer_conversion
    = is_safe_integer_operation<From, To>
      && sizeof(From) <= sizeof(To);
```

Let's make C++ type-safe!

A type-safe integer:

```
template <typename Integer>
class integer
{
    static_assert(is_integer<Integer>);

    Integer value_;

public:
    // overloads for all arithmetic operators
};
```

Let's make C++ type-safe!

Constructors:

```
integer() = delete;

template <typename T,
          typename = std::enable_if_t<is_safe_integer_conversion<T, Integer>>>
constexpr integer(const T& val) : value_(val) {}

template <typename T,
          typename = std::enable_if_t<is_safe_integer_conversion<T, Integer>>>
constexpr integer(const integer<T>& val) : value_(static_cast<T>(val)) {}
```

Let's make C++ type-safe!

Non-member operators:

```
template <typename IntA, typename IntB,
          typename = std::enable_if_t<is_safe_integer_operation<IntA, IntB>>
constexpr bool operator==(const integer<IntA>& a, const integer<IntB>& b)
{
    return …;
}

template <typename IntA, typename IntB>
constexpr auto operator+(const integer<IntA>& a, const integer<IntB>& b)
-> integer<integer_result_type<IntA, IntB>>
{
    return …;
}
```

Let's make C++ type-safe!

How to handle dangerous conversions?

`explicit` constructor?

Let's make C++ type-safe!

How to handle dangerous conversions?

- `make_signed()`
- `make_unsigned()`
- `abs()`
- `narrow_cast<Target>(source)`

Let's make C++ type-safe!

- `ts::integer`
- `ts::floating_point`
- `ts::boolean`

Let's make C++ type-safe!

Minimize implicit conversions:

- Use `explicit` constructors
- Use named conversion functions
- Only use implicit conversions when they are safe & cheap

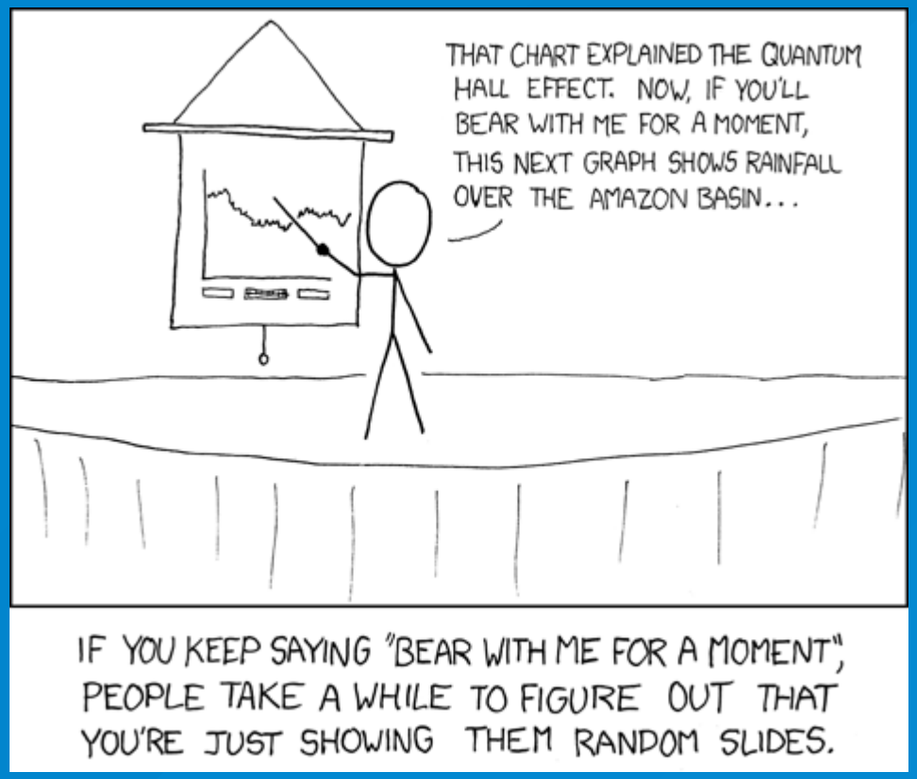
So, C++ is type-safe?


BACK
TO THE TYPE-SAFE
FUTURE

What's the problem with this code?

```
void generate_html(document& doc, std::string_view text)
{
    doc += "<p>";
    doc += text;
    doc += "</p>";
}
```

```
std::string text;
std::cin >> text;
generate_html(doc, text);
```


THAT CHART EXPLAINED THE QUANTUM HALL EFFECT. NOW, IF YOU'LL BEAR WITH ME FOR A MOMENT, THIS NEXT GRAPH SHOWS RAINFALL OVER THE AMAZON BASIN...
IF YOU KEEP SAYING "BEAR WITH ME FOR A MOMENT", PEOPLE TAKE A WHILE TO FIGURE OUT THAT YOU'RE JUST SHOWING THEM RANDOM SLIDES.

[XKCD 365]

Remember Hungarian Notation?

Express variable type in name:

```
int iVar;
```

Remember Hungarian Notation?

Express variable type in name:

```
int iVar;
int* piVar;
```

Remember Hungarian Notation?

Express variable type in name:

```
int iVar;
int* piVar;
int* a12piVar[12];
```

Remember Hungarian Notation?

Express variable type in name:

```
int iVar;
int* piVar;
int* a12piVar[12];

class CMyClass {};
```

Remember Hungarian Notation?

Express variable type in name:

```
int iVar;
int* piVar;
int* a12piVar[12];

class CMyClass {};
const CMyClass* const cpcCVar;
```

Remember Hungarian Notation?

Express variable type in name:

```
int iVar;
int* piVar;
int* a12piVar[12];

class CMyClass {};
const CMyClass* const cpcCVar;

using pvFvidCMyFunc = void(*)(int, double, CMyClass, ...);
```

Remember Hungarian Notation?

Express variable type in name:

```
int iVar;
int* piVar;
int* a12piVar[12];

class CMyClass {};
const CMyClass* const cpcCVar;

using pvFvidCMyFunc = void(*)(int, double, CMyClass, ...);
using a42pvFvidCMyFuncArray = pvFvidCMyFunc[42];
```

Remember Hungarian Notation?

Express variable type in name:

```
int iVar;
int* piVar;
int* a12piVar[12];

class CMyClass {};
const CMyClass* const cpcCVar;

using pvFvidCMyFunc = void(*)(int, double, CMyClass, ...);
using a42pvFvidCMyFuncArray = pvFvidCMyFunc[42];
using vpa42pvFvidCMyFuncArrayRef = volatile a42pvFvidCMyFuncArray*;
```

Remember Hungarian Notation?

`auto` to the rescue!

```
auto iVar = 42;
auto piVar = &iVar;
```

/s

Remember Hungarian Notation?

Express variable *type* in name:

```
void generate_html(document& doc, std::string_view sText) // sanitized text
{
    doc += "<p>";
    doc += sText;
    doc += "</p>";
}
```

```
std::string usText; // unsanitized text
std::cin >> usText;
generate_html(doc, usText); // wrong!
```

Remember Hungarian Notation?

Express variable type in *type*:

```
struct sanitized_string_view {…};
struct unsanitized_string {…};

void generate_html(document& doc, sanitized_string_view text)
{
    doc += "<p>";
    doc += text;
    doc += "</p>";
}
```

```
unsanitized_string text;
std::cin >> text;
generate_html(doc, text); // compilation error!
```

1. Semantic types

1. Semantic types

This doesn't work:

```
using second = int;
using millisceond = int;

second delta();
void sleep(millisecond time);
```

```
sleep(delta()); // no error!
```

(This slide intentionally left blank.)

1. Semantic types

Let's dream:

```
strong_typedef meter = int;
strong_typedef name = std::string;
```

What's the problem with this code?

```
std::string read_all(std::istream& file, int* no_lines = nullptr)
{
    // reset line count to 0
    if (no_lines)
        no_lines = 0;

    // read lines
    std::string result;
    for (char c; file >> c; result.push_back(c))
        if (c == '\n' && no_lines)
            // we had a newline, increment counter
            *no_lines++;

    return result;
}
```

Minimizing the interface

```
std::string read_all(std::istream& file, ts::optional_ref<int> no_lines)
{
    // reset line count to 0
    if (no_lines)
        no_lines.value() = 0;

    // read lines
    std::string result;
    for (char c; file >> c; result.push_back(c))
        if (c == '\n' && no_lines)
            // we had a newline, increment counter
            no_lines.value()++;

    return result;
}
```

Minimizing the interface

```
strong_typedef identifier = std::string;
strong_typedef handle = unsigned;

handle get_handle(identifier id);
```

```
identifier id(…);
id = id.substr(4);
id += "Hello World!";

auto h = get_handle(id);
h /= handle(2);
++h;
```

Strong typedefs, for real this time

```
template <class Tag, typename T>
class strong_typedef
{
    T value_;

public:
    strong_typedef() : value_() {}
    explicit strong_typedef(T value) : value_(std::move(value)) {}

    explicit operator       T&()       { return value_; }
    explicit operator const T&() const { return value_; }
};
```

Strong typedefs, for real this time

```
template <class StrongTypedef>
struct equality_comparision {};

template <class StrongTypedef>
bool operator==(const equality_comparison<StrongTypedef>& a,
                const equality_comparison<StrongTypedef>& b)
{
    auto& typedef_a = static_cast<const StrongTypedef&>(a);
    auto& typedef_b = static_cast<const StrongTypedef&>(b);
    return get(typedef_a) == get(typedef_b);
}

…
```

Strong typedefs, for real this time

```
struct handle
: ts::strong_typedef<handle, unsigned>,
  ts::strong_typedef_op::equality_comparison<handle>
{
    using handle::handle;
};
```

1. Semantic types

- Use different types for different domains

→ Avoid generic types like `int` or `std::string`

- Only expose the interface you need

→ Use something like `ts::strong_typedef`

What's the problem with this code?

```
void to_lower(std::string* str)
{
    for (auto& c : *str)
        c = std::tolower(c);
}
```

What's the problem with this code?

```
// pre: str != nullptr
void to_lower(std::string* str)
{
    for (auto& c : *str)
        c = std::tolower(c);
}
```

Semantic types!

```
void to_lower(gsl::non_null<std::string*> str)
{
    for (auto& c : *str)
        c = std::tolower(c);
}
```

Semantic types!

```
void to_lower(std::string& str)
{
    for (auto& c : str)
        c = std::tolower(c);
}
```

2. Constrained types

2. Constrained types

```
int get_some_int();

// pre: x ≥ 0
int ilog2(int x);
```

```
auto i      = get_some_int();
auto result = ilog2(i); // ups?
```

2. Constrained types

```
int get_some_int();


int ilog2(unsigned x);
```

```
auto i      = get_some_int();
auto result = ilog2(i); // ups?
```

2. Constrained types

```
int get_some_int();


int ilog2(ts::integer<unsigned> x);
```

```
auto i      = get_some_int();
auto result = ilog2(i); // compilation error!
```

2. Constrained types

Idea: general `constrained_type`

```
template <typename T, typename Constraint, typename Verifier>
class constrained_type
{
    T value_;
    Constraint predicate_;

    void verify()
    {
        Verifier::verify(value_, predicate_);
    }

    …
};
```

2. Constrained types

Idea: general `constrained_type`

```
template <typename T, typename Constraint, typename Verifier>
class constrained_type
{
public:
    explicit constrained_type(T value, Constraint predicate)
    : value_(std::move(value), predicate_(std::move(predicate)
    { verify(); }

    constrained_type(constrained_type&&) = delete;
    constrained_type& operator=(constrained_type&&) = delete;

    const T& operator* () const noexcept { return value_; }
    const T* operator->() const noexcept { return &value_; }
};
```

2. Constrained types

Idea: general `constrained_type`

```
class assertion_verifier;

class throwing_verifier;

class null_verifier;
```

2. Constrained types

```
template <typename T>
using non_null_pointer = ts::constrained_type<T*, ts::constraints::non_null>;

template <typename T>
non_null_pointer<T> alloc_or_throw()
{
    auto ptr = std::malloc(sizeof(T));
    if (!ptr) throw std::bad_alloc();
    return non_null_pointer<T>(static_cast<T*>(ptr));
}

template <typename T>
void destroy(non_null_pointer<T> ptr)
{
    (*ptr)->~T();
}
```

2. Constrained types

```
auto ptr = alloc_or_throw<int>();
…
destroy(ptr);
```

2. Constrained types

```
auto ptr = static_cast<int*>(std::malloc(sizeof(int));
…
destroy(ptr); // error!
```

2. Constrained types

```
auto ptr = static_cast<int*>(std::malloc(sizeof(int));
…
destroy(non_null_pointer<int>(ptr)); // possible constrained violation
```

2. Constrained types

```
auto ptr = static_cast<int*>(std::malloc(sizeof(int));
if (!ptr)
    my_fancy_error_handler();
auto constrained = non_null_pointer<int>(ptr);
…
destroy(constrained);
```

2. Constrained types

- `ts::bounded_type` + optionally `ts::clamping_verifier`
- `ts::tagged_type` (`ts::constrained_type` with `ts::null_verifier`)

2. Constrained types

```
using non_empty_string_ref
    = ts::constrained_type<std::string&, ts::constraints::non_empty>;

void pop_back(non_empty_str_ref str)
{
    str->pop_back(); // compilation error!
}
```

2. Constrained types

```
using non_empty_string_ref
    = ts::constrained_type<std::string&, ts::constraints::non_empty>;

void pop_back(non_empty_str_ref str)
{
    auto modifier = str.modify();
    modifier->pop_back();
    // dtor checks constraint
}
```

2. Constrained types

```
std::string str = get_string();
if (str.empty())
    return;
non_empty_string non_empty(std::move(str));
pop_back(non_empty);
…
// actually want to make it empty now
std::string maybe_empty = std::move(non_empty);
maybe_empty.pop_back();
…
if (!maybe_empty.empty())
    process(non_empty_string(std::move(maybe_empty)));
```

"But narrow contracts are good!"

"I know this precondition violation can't happen!™ "

"But... performance?!"

"How feasible is it really?"

What's the problem with this code?

```
// parse string until end of line
std::string content;
for (; *ptr != '\n'; ++ptr)
    content += *ptr;

// erase trailing whitespace
while (is_whitespace(content.back()))
    content.pop_back();
```

Using constrained types

```
char safe_back(non_empty_string str)
{
    return str.back();
}
```

Not really feasible.

Using a proper return type

```
ts::optional<char> safe_back(const std::string& str)
{
    return str.empty() ? ts::nullopt : str.back();
}
```

Using a proper return type

```
// erase trailing whitespace
while (is_whitespace(safe_back(content).value_or('\0')))
    content.pop_back();
```

"But this gives defined behavior to a precondition violation!"

ts::optional

```
// slide++
template <typename Func>
auto map(Func f)
{
    return has_value() ? make_optional(f(value())) : nullopt;
}
```

```
// erase trailing whitespace
while (safe_back(content).map(&is_whitespace).value_or(false))
    content.pop_back();
```

Bad

```
class cpp_entity
{
public:
    bool has_parent() const noexcept;

    // pre: has_parent()
    const cpp_entity& parent() const noexcept;
};
```

Good

```
class cpp_entity
{
public:
    ts::optional_ref<const cpp_entity> parent() const noexcept;
};
```

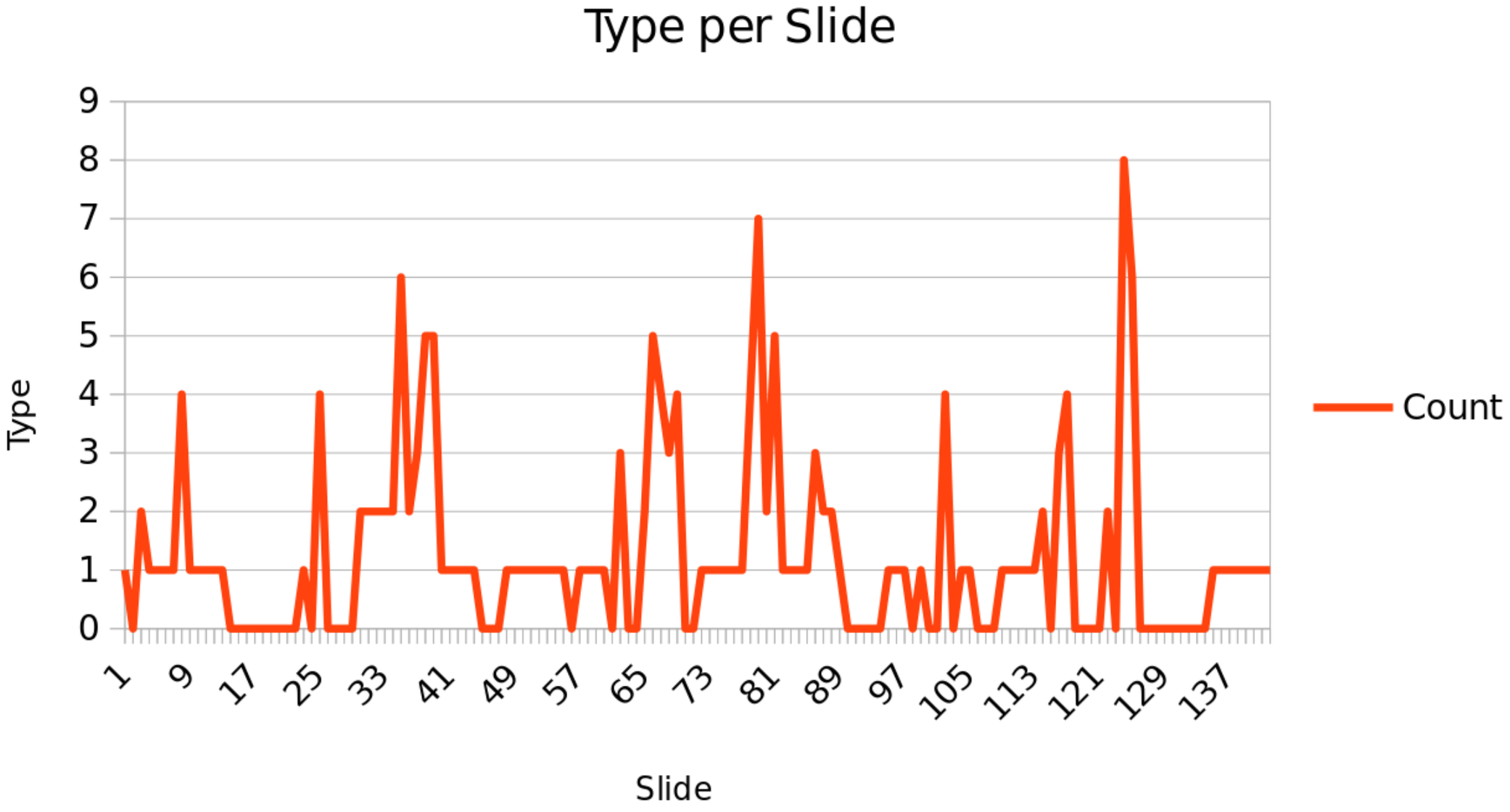
2. Constrained types

- Create semantic type where precondition can't occur - when possible

→ Use something like `ts::constrained_type`

- Consider `optional` return types when there isn't always a valid value

→ `has_xxx()`/`get_xxx()` pattern


Type per Slide
Type
Slide
Count
0
1
2
3
4
5
6
7
8
9
1
9
17
25
33
41
49
57
65
73
81
89
97
105
113
121
129
137

Handling valid ids

```
class id
{
public:
    explicit id(std::string str) : str_(std::move(str)) {}
    bool is_valid() const noexcept { return check_validity(str_); }
    const std::string& as_string() const noexcept { return str_; }

private:
    std::string str_;
};

id read_id(std::istream& file);

// pre: i.is_valid()
void register_id(const id& i);
```

Handling valid ids

```
class id {…};

id read_id(std::istream& file);

struct valid_id_constraint
{
    bool operator()(const id& i) { return i.is_valid(); }
};

using valid_id = ts::constrained_type<id, valid_id_constraint>;

void register_id(const valid_id& i);
```

Handling valid ids

```
// should be strong typedef
using id = std::string;

ts::optional<id> read_id(std::istream& file);

void register_id(const id& i);
```

What's the problem with this code?

```
enum class connection_state
{
    disconnected,
    connecting,
    connecting_failed,
    connected,
    connection_interrupted
};
```

```
struct connection
{
    connection_state state;
    ip_address server;
    …
};
```

What's the problem with this code?

```
// pre: con.state == connected
void send(connection& con, const char* data, std::size_t bytes)
{
    …
}
```

Using variant

```
struct connection
{
    struct disconnected      { … };
    struct connecting        { … };
    struct connecting_failed { … };
    struct connection        { … };

    …

    std::variant<disconnected, connecting,
                 connecting_failed, connection, … > state;
};
```

```
void send(connection::connection& con, const char* data, std::size_t bytes);
```

3. Multi-state types

3. Multi-state types

C++ isn't perfect:

1. `std::variant<Ts...>`: any of the `Ts` or none

3. Multi-state types

C++ isn't perfect:

1. `std::variant<Ts...>`: any of the `Ts` or none

→ Prevent throwing move

→ Create a fallback value

→ Embrace the empty state

3. Multi-state types

C++ isn't perfect:

1. `std::variant<Ts...>`: any of the `Ts` or none

→ Prevent throwing move

→ Create a fallback value

→ Embrace the empty state

↦ `ts::basic_variant`

3. Multi-state types

C++ isn't perfect:

2. Moved-from state.

Non-null `unique_ptr`

```
template <typename T>
class owning_ptr
{
public:
    template <typename ... Args>
    explicit owning_ptr(Args&&... args)
    : ptr_(new T(std::forward<Args>(args...))) {}

    ~owning_ptr() { delete ptr_; }

    owning_ptr(const owning_ptr&)             = delete;
    owning_ptr& operator=(const owning_ptr&) = delete;

    T& operator* () { return *ptr_; }
    T* operator->() { return  ptr_; }
};
```

Non-null (???) `unique_ptr`

```
owning_ptr(owning_ptr&& other)
: ptr_(other.ptr_)
{
    other.ptr_ = nullptr;
}
```

3. Multi-state types

- instead of `explicit operator bool()` consider using an optional type
- instead of `enum state` consider using a variant type
- be careful about moved-from states

Type-safe Programming:

Trick the compiler into preventing API misuse by (ab-)using the type system.

1. Semantic types
2. Constrained types
3. Multi-state types

But for that: Minimize implicit conversions!

Applications

`operator[]`

1. `T& operator[](std::size_t i)`
2. `T& operator[](int i)`

`operator[]`

1. `T& operator[](std::size_t i)`
2. `T& operator[](int i)`
3. `T& operator[](ts::integer<std::size_t> i)`

`operator[]`

~~1. `T& operator[](std::size_t i)`~~
~~2. `T& operator[](int i)`~~
~~3. `T& operator[](ts::integer<std::size_t> i)`~~

→ `T& operator[](ts::index_t i)`

`operator[]`:

Consider `RandomAccessIterator`:

```
RandomAccessIterator a, b;
difference_type d;
```

```
(a + d) -> RandomAccessIterator
(a - d) -> RandomAccessIterator
(a - b) -> difference_type

(a + b) -> nonsense
(a * b) -> nonsense
(a / b) -> nonsense
```

`operator[]`

Let's do the same for indices:

```
index_t      a, b; // unsigned
difference_t d; // signed
```

```
(a + d) -> index_t
(a - d) -> index_t
(a - b) -> difference_t

(a + b) -> nonsense
(a * b) -> nonsense
(a / b) -> nonsense
```

`operator[]`

The difference between two indices:

```
struct difference_t
: strong_typedef<difference_t, ptrdiff_t>,
  strong_typedef_op::equality_comparison<difference_t>,
  strong_typedef_op::relational_comparison<difference_t>,
  strong_typedef_op::unary_plus<difference_t>,
  strong_typedef_op::unary_minus<difference_t>,
  strong_typedef_op::addition<difference_t>,
  strong_typedef_op::subtraction<difference_t>
{
    using strong_typedef::strong_typedef;
};
```

operator[]

An index:

```
struct index_t
: strong_typedef<index_t, size_t>,
  strong_typedef_op::equality_comparison<index_t>,
  strong_typedef_op::relational_comparison<index_t>,
  strong_typedef_op::increment<index_t>,
  strong_typedef_op::decrement<index_t>,
  strong_typedef_op::unary_plus<index_t>
{
    index_t& operator+=(const distance_t& rhs) noexcept;
    index_t& operator-=(const distance_t& rhs) noexcept;
};
```

`operator[]`

An index:

```
index_t operator+(const index_t&    lhs, const distance_t& rhs);
index_t operator+(const distance_t& lhs, const index_t&    rhs);
index_t operator-(const index_t&    lhs, const distance_t& rhs);
index_t operator-(const distance_t& lhs, const index_t&    rhs);

distance_t operator-(const index_t& lhs, const index_t& rhs);
```

`operator[]`

Index strides:

```
index_t      a;
difference_t d;
stride_t     s;
```

```
(a + s) -> index_t
(a - s) -> index_t

(a + d * s) -> index_t
(a - d * s) -> index_t

(d * s) -> difference_t
```

But, urgh, `unsigned`...

What's the problem with this code?

```
for (ts::index_t i = size - 1u; i >= 0u; --i)
    process(array[i]);
```

If we simply used `int`...

```
for (int i = size - 1; i >= 0; --i)
    process(array[i]);
```

Let's copy iterators again

```
for (ts::index_t i = size; i != 0u; --i)
    process(array[i - 1u]);
```

Let's copy iterators again

```
for (reverse_index_t i = size; i != 0u; --i)
    process(at(array, i));
```

What's the problem with this code?

```
// true if the last processed character was a newl
auto was_newl = false;
for (auto x : …)
{
    if (x == '\n')
        was_newl = true;
    else if (was_newl)
        do_after_newline(c);
    else
        do_in_line(c);
}
```

What we want to express

```
auto was_newl = false; // this is some kind of flag
for (auto x : …)
{
    if (x == '\n')
        // **change** was_newl to true here
    else if (/* check & reset*/)
        do_after_newline(c);
    else
        do_in_line(c);
}
```

A `flag` type

```
class flag
{
    bool value_;

public:
    explicit flag(bool initial_value) noexcept
    : value_(initial_value) {}

    …
};
```

A `flag` type

```
void flag::change(bool new_value) noexcept
{
    assert(value_ != new_value);
    value_ = new_value;
}
```

A `flag` type

```
bool flag::try_reset() const noexcept
{
    auto changed = value_ == true;
    value_ = false;
    return changed;
}
```

A `flag` type

```
flag was_newl(false);
for (auto x : …)
{
    if (x == '\n')
        was_newl.change(true);
    else if (was_newl.try_reset())
        do_after_newline(c);
    else
        do_in_line(c);
}
```

A `flag` type

Should we add conversion to `bool`?

```
explicit operator bool() const noexcept
{
    return value_;
}
```

A `flag` type

Should we add conversion to `bool`?

No.

```
if (x == '\n')
    was_newl.change(true);
else if (was_newl)
    do_after_newline(c);
else
    do_in_line(c);
```

A `flag` type

Should we add conversion to `bool`?

Comparison:

```
if (some_flag == true)
    do_sth();
```

Type-Safe Programming:

Define. More. Types.

(And don't do implicit conversions.)

https://github.com/foonathan/type_safe

foonathan.net/cppnow2017.html

@foonathan