



Constrain yourself

Andrzej Krzemieński

Andrzej's C++ blog
akrzemi1.wordpress.com



Agenda

- Overview
- Case study 1 – Short codes
- Case study 2 – Magic values
- Case study 3 – Hidden semantics
- Conclusion

Compilers detecting bugs

Compilers help already:

```
return i.substr(str, 2);  
// intent: str.substr(i, 2);
```



Grey area



No compiler can help:

```
return v < max;  
// intent: max < v
```

Compilers detecting bugs

Compilers help already:

```
return i.substr(str, 2);  
// intent: str.substr(i, 2);
```



Compiler may be able to help

```
// but we may need  
// to help the compiler
```



No compiler can help:

```
return v < max;  
// intent: max < v
```

Constraints

- A number of (too) versatile types:
 - `int`
 - `double`
 - `string`
 - `T*`
- What you can use `int` for:
 - mathematical integer
 - index
 - UID
 - enumeration
 - bitmask
 - ...
- Do you need all of this at once?

Constraints



```
std::bitset<8> flags;  
float f = flags; // err  
flags * 3;      // err  
if (flags.any()) // ok
```



```
unsigned flags;  
float f = flags; // ok  
flags * 3;      // ok  
if (flags)      // ok
```

Constraints

- Focus on what you *cannot* do with the object
 - You will understand your program better
- Clearly encode it in the type system
 - Your intentions are communicated clearly
 - Machines can make use of the information
- Proceed to coding
 - Compiler will protest when you *accidentally* do what you should not

Example: unique_ptr

Single object

```
unique_ptr<int> p = /**/;  
  
p - p; // err  
*p;    // ok  
p[0];  // err
```

Array of objects

```
unique_ptr<int[]> p = /**/;  
  
p - p; // err  
*p;    // err  
p[0];  // ok
```

- Change in type → change in allowed operations
- Single object? – no need for `operator[]`
- Array of objects? – no need for `operator*`
- Bad usages detected at compile-time

Example: unique_ptr

```
unique_ptr<int> p = /**/;  
p.get() - p.get();  
p.get()++;  
p.get()[0];
```

- Still possible to do unsafe things
- Safety by default, but disabled on demand



Case study 1 » Short codes



Short codes

- Human-friendly IDs.
- Type `string` can represent any short code (and more).
- Most of `string`'s interface is useless for short codes.
- A short code *is not* a `string` with fixed size.

We need:

- (Fast) comparisons
- Input/Output ops
- Regular ops

We don't need:

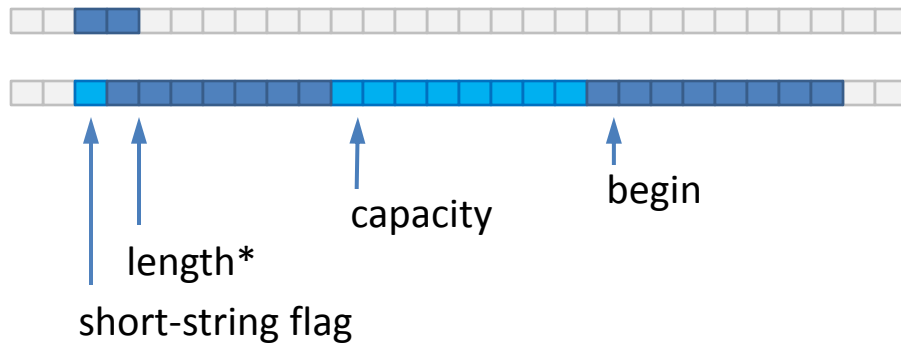
- `find_first_of()`
- `append()`
- `at()`, `front()`
- `substr()`

Optimization opportunities

- How many codes fit into a CPU cache line?
- How fast is the comparison?

`char[2]`

`string` (with SSO)



String with SSO:

- An `if` check
- `sizeof(s) == 3 * 8`

Correctness opportunities

```
map.associate_code("PL", "Poland");  
// ...  
assert (map.match_code("PL", "Poland")); // fails
```

Why does it fail?

Correctness opportunities

```
map.associate_code("PL", "Poland");  
// ...  
assert (map.match_code("PL", "Poland")); // fails
```

Why does it fail?

```
void associate_code(string code, string country);  
bool match_code(string country, string code);
```

- Two *strings* are interchangeable.

Correctness opportunities

```
map.associate_code("PL", "Poland");  
// ...  
assert (map.match_code("PL", "Poland")); // fails
```

Why does it fail?

```
void associate_code(string code, string country);  
bool match_code(string country, string code);
```

- Two strings are interchangeable.
- A string and a Code are not.

```
bool match_code(Code code, string country);  
bool match_code(string country, Code code);
```

Correctness opportunities

```
bool match(string const& aux_code, record const& rec)
{
    return aux_code == rec.auxiliary_resident_state; // bug
}
```

But the intent was:

```
bool match(string const& aux_code, record const& rec)
{
    return aux_code == rec.auxiliary_code;
}
```

- Type `record` has both (and more) members.

Type design

```
template <int N>
class Code
{
    array<char, N> data_;
    friend bool operator==(Code l, Code r) { return l.data_ == r.data_; }
    friend bool operator< (Code l, Code r) { return l.data_ < r.data_; }
    // and !=, >, <=, >=
};
```

- Code size is part of type.
- A `Code<2>` and a `Code<3>` are not interchangeable.

Type design » Default construction

- Confusion: what value?
- Danger: someone may use this value.
- Ideally, `Code` should not be default-constructible.
- Needed for default-construct-and-fill idiom:

```
int i [[uninitialized]]; // no such attribute yet
std::cin >> i;
```

- Most XML and DB frameworks require default constructor

Type design » Default construction

Decision: use all zeros to represent the artificial state

- Now we have a *special* state.
- It is *not* singular: all operations can be safely executed.
- Comparisons work: special value \neq any normal value.
- We now need function `is_initialized()`.
- Objects not always hold a genuine code.

Type design » Default construction

We can emulate the attribute

```
class uninitialized_code_t {};  
constexpr uninitialized_code_t uninitialized_code {};
```

- A *tag* class
- Purpose: alter the type system

```
template <int N>  
Code<N>::Code(uninitialized_code_t) : data_{};
```

Usage:

```
Code<2> c = uninitialized_code;  
Code<2> fun() { return uninitialized_code; }
```

Type design » From string

- The unsafe part: how do you handle bad size?

```
bool Code<N>::from_string(string_view s)
{
    if (BOOST_LIKELY(len == N)) // optimization hint
        return std::memcpy(data_, s.data(), s.size()), true;
    else
        return false;
}
```

- You need an object before conversion
- Bad size is legal: you can use the function for validation
- Alternative: use precondition

Type design » To string

- C-like:

```
const char* Code<N>::data() const { return data_.data(); }  
unsigned Code<N>::size() const { return N; }
```

- C++-like:

```
string Code<N>::to_string() const  
{  
    if (BOOST_LIKELY(is_initialized()))  
        return string(data(), size());  
    else  
        return string(); // so that it is empty  
}
```

Type design » Literals

Our goal:

```
Code<2> c1 {"PL"}; // ok  
Code<2> c2 {"PLN"}; // compile-time error
```

Implementation:

```
explicit Code<N>::Code( const char (&literal)[N + 1] );
```

- No implicit conversion, even if the size is right
- Error message: "no appropriate constructor found"

Type design » Literals

Alternate implementation:

```
template <int M>
explicit Code<N>::Code( const char (&literal)[M] )
{
    static_assert (M <= N + 1, "too long literal");
    static_assert (M >= N + 1, "too short literal");
    // copy
}
```


- Error message "literal constructor with bad length".
- Error message quality vs allowing other overloads.

Type design » Adding semantics

```
bool in_country(Airport ac, Country cc);
```

Strings

```
using Country = std::string  
using Airport = std::string  
  
Airport port {"LHR"};  
Country ctry {"GB"};  
  
in_country(ctry, port);
```



- Compiles
- Does the wrong thing

Codes

```
using Country = Code<2>;  
using Airport = Code<3>;  
  
Airport port {"LHR"};  
Country ctry {"GB"};  
  
in_country(ctry, port);
```

- Compile-time error

Type design » Adding semantics

```
using Currency = Code<3>;  
using Airport  = Code<3>; // same type!
```

Solution: tags

```
template <typename /*tag*/, int N>  
class Code { /*...*/ };  
  
using Currency = Code<class Currency_tag, 3>; // different  
using Airport  = Code<class Airport_tag,  3>; // types!
```

- A tag, to generate different types
- Opaque typedef emulation

Type design » Adding semantics

```
using Currency = Code<class Currency_tag, 3>;  
using Airport  = Code<class Airport_tag,  3>;  
  
bool accepts(Airport ac, Currency cc);
```

If literal constructor was implicit:

```
accepts("EUR", "FRA"); // compiles!
```

If explicit ctors were allowed in type-deduced list-initialization:

```
accepts({"EUR"}, {"FRA"}); // compiles!
```

Current state:

```
accepts(Currency{"EUR"}, Airport{"FRA"}); // fails
```

Testing

Not being able to convert is a feature.

Features need to be tested.

```
using Currency = Code<class Currency_tag, 3>;
using Airport  = Code<class Airport_tag,  3>;

int main()
{
    Currency{"PLN"} == Airport{"WAW"}; // expected failure
}
```

- The compiler should return 1 for this program.
- May not be testable inside the program.
- Not even with SFINAE.

Code bloat prevention

```
template <int N> class code_base
{
    const char* data() const;
    unsigned size() const;
    // ...
};

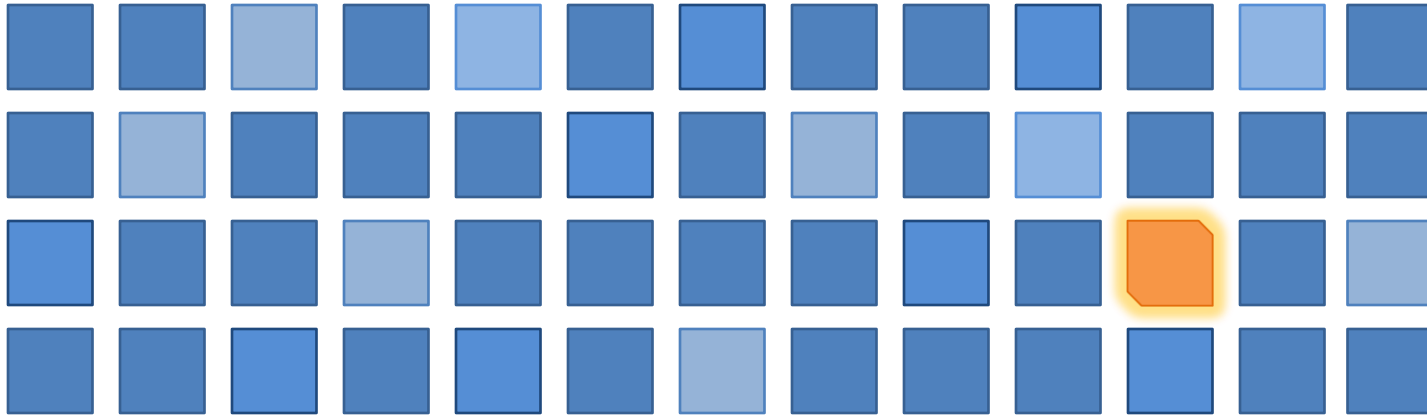
template <typename, int N> class Code : public code_base<N>
{
    friend bool operator==(code l, code r);
    friend bool operator< (code l, code r);
    // ...
};
```

Variable-length codes

```
template <typename, int Lo, int Hi>
class Code
{
    template <int S>
    explicit Code( const char (&literal)[S] )
    {
        static_assert(S >= Lo + 1, "too short literal");
        static_assert(S <= Hi + 1, "too long literal");
        // copy
    }
};
```

- Fill array with zeros
- Need to change `size()` to search for zeros

Case study 2 » Magic values



The problem

```
std::string nested_name(std::string path)
{
    return path.substr(path.find("::") + 2);
}

assert (nested_name("std::data") == "data");
assert (nested_name("::data") == "data");
```

- Can you see the bug?

The problem

```
std::string nested_name(std::string path)
{
    return path.substr(path.find("::") + 2);
}

assert (nested_name("std::data") == "data");
assert (nested_name("::data") == "data");

return nested_name("") == ""; // crashes!
```

- The npos value!

The problem

```
double Flight_plan::weight()
{
    if (Impl* impl = get_impl())
        return impl->compute_weight();
    else
        return BOGUS_WEIGHT; // -1000.0;
}
```

- Can you see the bug?

The problem

```
double Flight_plan::weight()
{
    if (Impl* impl = get_impl())
        return impl->compute_weight();
    else
        return BOGUS_WEIGHT; // -1000.0;
}
```

```
bool too_heavy(Flight_plan const& p)
{
    return p.weight() > p.aircraft().max_weight();
}
```

- if `impl` is null, aircraft is not too heavy!

Hidden semantics

- Two “kinds” of returned value:
 - Either the proper value
 - Or no-value
- Expected usage:

```
auto v = get_val();  
if (is_proper(v))  
    use (v);  
else  
    do_something_else();
```

Boost.Optional?

```
auto v = get_val();

if (is_proper(v))
    use (extract(v));
else
    do_something_else();
```

```
optional<T> v = get_val();
use(v); // error

if (v)
    use (*v);
else
    do_something_else();
```

- Exactly this use-pattern
- Value semantics
- `operator*` — no inadvertent use

Boost.Optional – memory layout

double



optional<double>



↑
has-value flag

↑
padding

↑
storage for double

`sizeof(optional<double>) == 2 * sizeof(double)`

Boost.Optional – type-safety hole

```
optional<double> Flight_plan::weight();  
double Aircraft::max_weight();
```

```
bool too_heavy(Flight_plan const& p)  
{  
    return p.weight() > p.aircraft().max_weight();  
}
```

Still compiles!

- `optional<T>` is less-than comparable.
- `T` converts to `optional<T>`!
- This implies 'mixed' less-than comparison.
- `Non-T < any T`!

Custom wrapper — design criteria

Boost.Optional doesn't satisfy our criteria?

Create a type that does!

Our criteria:

- Different type than `T`
- No spatial overhead
- Safety over convenience

Boost.Optional criteria:

- Different type than `T`
- Works for any `T`
- No `T` created in empty optional
- Convenience over safety

Custom wrapper

```
class opt_int
{
    int _val;
public:
    opt_int() : _val{-1} {}
    explicit opt_int(int i) : _val{i} {}
    bool has_value() const { return _val != -1; }
    int value() const { return _val; } // precondition: has_value()
};
```

- No implicit conversions – no surprises
- `sizeof(opt_int) == sizeof(int)`
- In some places -1 will not do

Custom wrapper

```
template <typename T, T Ev> // requires Regular<T>
class compact_optional
{
public:
    compact_optional() : _val{Ev} {}
    explicit compact_optional(T v) : _val{v} {}
    bool has_value() const { return _val != Ev; }
    T value() const { return _val; } // precondition: has_value()
};
```

- Type-safety: different empty-values render different types
- What about non-int types?
- What about types with no equality?

Empty-value policy

```
template <typename T, typename EVP>
class compact_optional
{
public:
    compact_optional() : _val{EVP::empty()} {}
    bool has_value() const { return !EVP::is_empty(_val); }
    // ...
};
```

```
struct evp_empty_double
{
    static double empty() { return -1000.0; }
    static bool is_empty(double v) { return v == empty(); }
};
```

Empty-value policy

```
struct evp_string
{
    static string empty()           // may allocate
    {
        return string("\0\0", 2);
    }

    static bool is_empty(string const& s) // no alloc
    {
        return s.compare(0, s.npos, "\0\0", 2) == 0;
    }
};
```

Empty-value policy

A set of predefined policies:

- `evp_int<Int, v>` – integral type with `v` as empty.
- `evp_fp_nan<FPT>` – floating-point-types with NaN as empty.
- `evp_value_init<T>` – value-initialized `T` as empty.
- `evp_stl_empty<T>` – empty STL container as empty.
- `evp_bool` – empty `bool` encoded in 1 byte.
- empty state as special bit pattern in raw storage
- ...

Policy implies the type

```
template <typename EVP> // no type T  
class compact_optional  
{  
    typedef decltype(EVP::empty()) value_type;  
    // ...  
};
```

- I can deduce T from the policy.
- No need to provide it.

Too little semantics

```
using count = compact_optional<evp_int<int, -1>>;  
using flat  = compact_optional<evp_int<int, -1000>>;
```

- Different types

```
using count  = compact_optional<evp_int<int, -1>>;  
using number = compact_optional<evp_int<int, -1>>;
```

- Same type
- Even though `count` and `number` represent something different

Adding semantics

```
template <typename Ev, typename /*tag*/>
class compact_optional
{
    // ...
};
```

- A tag parameter: only for differentiating types

```
compact_optional<evp_int<int, -1>, class tag_A>;
compact_optional<evp_int<int, -1>, class tag_B>;
```

- Different types
- Remember about the code bloat

Performance over safety

How to 'dump' `compact_optional`:

```
template <typename EVP, typename Tag>
size_t dump(compact_optional<EVP, Tag> const& v)
{
    return dump(v.unsafe_raw_value());
}
```

- Fast, but type-unsafe.
- It must be as fast as dumping raw value.
- Performance first.

Case study 3 » Hidden semantics



Symptoms

Time of day in format HHMM

Actual output

Expected output

045

045

100

140

1010

1650

1225

2025

The root cause

```
struct flight_data
{
    // ...
    date arrival_date;
    int arrival_time; // minutes since midnight
};
```

- Can you see the bug?

The root cause

```
struct flight_data
{
    // ...
    date arrival_date;
    int arrival_time; // minutes since midnight
};
```

```
void display(const flight_data& f)
{
    // ...
    cout << f.arrival_date;
    cout << f.arrival_time;
}
```

- 1:40 is 100 minutes

Hotfixes

```
void display(const flight_data& f)
{
    // ...
    cout << f.arrival_date;
    cout << minutes_as_HHMM(f.arrival_time);
}
```

- Did you cover all the places?
- What about the future additions?

Hotfixes

```
struct flight_data
{
    // ...
    date arrival_date;
    int arrival_time_as_minutes_since_midnight;
};
```

- The name sends a clear message.
- If you rename, compiler will show you all usages.

Hotfixes

```
void display(const flight_data& flt, bool abbreviate, const supplementary_data& suppl)
{
    bool do_vca = suppl.do_any_tla;
    bool do_strict = suppl.force_strict || !suppl.aux_vec.empty();

    if (!abbreviate || suppl.traffic && suppl.traffic->do_tse)
        do_strict = true;
    do_vca |= is_interline(flt);

    cout << boost::format("%1%;%2% %3%")
        % boost::str(boost::format("%1% %2%")
            % (do_vca ? flt.status : extend_status(flt.status)) % suppl.alteration_str)
        % boost::str(boost::format("from %1% %2% %3%")
            % flt.departure_apr % flt.departure_date
            % (do_strict ? flt.arrival_time_as_minutes_since_midnight : flt.est_arrival_time))
        % boost::str(boost::format("to %1% %2% %3%")
            % flt.arrival_apr % (flt.departure_date + flt.dep_offset)
            % (do_strict ? flt.departure_time : flt.est_departure_time));
}
```

- Will you remember to fix it?

Hotfixes

```
void display(const flight_data& flt, bool abbreviate, const supplementary_data& suppl)
{
    bool do_vca = suppl.do_any_tla;
    bool do_strict = suppl.force_strict || !suppl.aux_vec.empty();

    if (!abbreviate || suppl.traffic && suppl.traffic->do_tse)
        do_strict = true;
    do_vca |= is_interline(flt);

    cout << boost::format("%1%;%2% %3%")
        % boost::str(boost::format("%1% %2%")
            % (do_vca ? flt.status : extend_status(flt.status)) % suppl.alteration_str)
        % boost::str(boost::format("from %1% %2% %3%")
            % flt.departure_apr % flt.departure_date
            % (do_strict ? flt.arrival_time_as_minutes_since_midnight : flt.est_arrival_time))
        % boost::str(boost::format("to %1% %2% %3%")
            % flt.arrival_apr % (flt.departure_date + flt.dep_offset)
            % (do_strict ? flt.departure_time : flt.est_departure_time));
}
```

- Will you remember to fix it?

Hotfixes

```
void display(const flight_data& flt, bool abbreviate, const supplementary_data& suppl)
{
    bool do_vca = suppl.do_any_tla;
    bool do_strict = suppl.force_strict || !suppl.aux_vec.empty();

    if (!abbreviate || suppl.traffic && suppl.traffic->do_tse)
        do_strict = true;
    do_vca |= is_interline(flt);

    cout << boost::format("%1%;%2% %3%")
        % boost::str(boost::format("%1;%2%")
            % (do_vca ? flt.status : extend_status(flt.status)) % suppl.alteration_str)
        % boost::str(boost::format("from %1% %2% %3%")
            % flt.departure_apr % flt.departure_date
            % (do_strict ? flt.arrival_time_as_minutes_since_midnight : flt.est_arrival_time))
        % boost::str(boost::format("to %1% %2% %3%")
            % flt.arrival_apr % (flt.departure_date + flt.dep_offset)
            % (do_strict ? flt.departure_time : flt.est_departure_time));
}
```

- Will you remember to fix it?

The solution

```
class Minutes_since_midnight
{
    int minutes_;
    // invariant: minutes_ >= 0 && minutes_ < 24 * 60

public:
    explicit Minutes_since_midnight(int m) : minutes_(m) {}
    // requires: m >= 0 && m < 24 * 60

    int as_int() const { return minutes_; }

    // and relational operators ...
};
```

The solution

No accidental operations on `int` apply:

```
struct flight_data
{
    // ...
    Minutes_since_midnight arrival_time;
};

void display(const flight_data& f)
{
    // ...
    cout << f.arrival_time; // compile-time error!
}
```

- Compiler will tell you where to convert.
- Even in future.

The solution

Still an `int`, if you need one:

```
int minutes_as_HHMM(Minutes_since_midnight t)
{
    return t.as_int() / 60 * 100 + t.as_int() % 60;
}
```

```
Minutes_since_midnight t;
static_assert(sizeof(t) == sizeof(int), "compactness");
```

- Reality: no abstraction will cover all use cases.
- No loss compared with a raw `int`.

The solution

Ability to define the *invariant*:

```
bool invariant() const
{
    return minutes_ >= 0 && minutes_ < 24 * 60;
}

explicit Minutes_since_midnight(int m) : minutes_(m)
{
    assert (invariant());
}
```

```
Minutes_since_midnight t {1440}; // assertion failure
```

- Not an ideal, but some additional safety measure
- Clear statement of intent

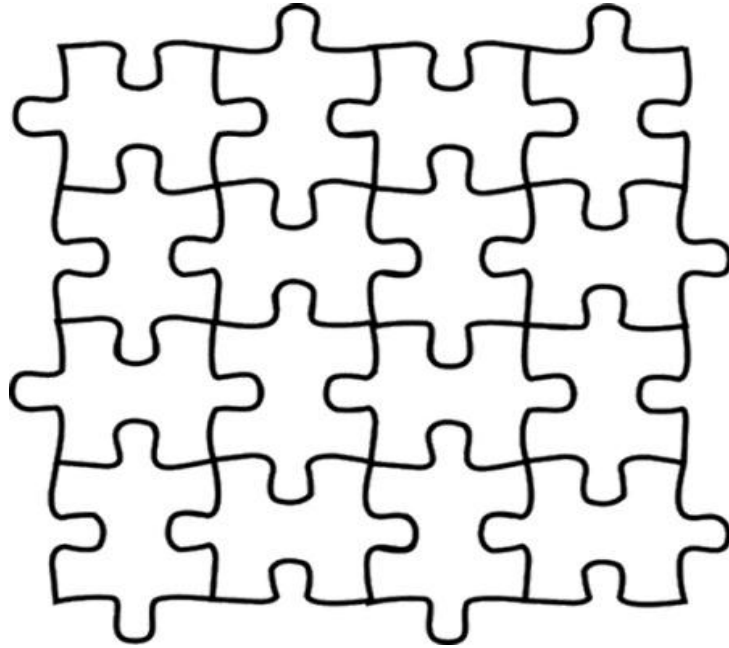
Is solution worth the trouble?

- Good abstraction is likely reused

```
struct flight_data
{
    // ...
    Minutes_since_midnight departure_time;
    Minutes_since_midnight arrival_time;
};
```

- Are bug fixes and patch releases more attractive?

Conclusion



Summary

- Type system performs (limited) static analysis.
- We get the same binary; but more compile-time checks.
- Zero run-time cost.
- Unsafe interface on demand.
- Reveal hidden semantics.
- Sometimes adding constraints comes with optimizations.
- Tag template parameter: opaque typedef for free.

Criteria for adding type wrapper

- Do you suspect how people can accidentally misuse the type?
- Do all operations of a given type make sense for our usage?
- We want some operations never to be called in our program.
- Additional semantics (with identical interface).
- Do we have an invariant?
- Can we get other benefits?

Boost.Units

```
time      t1, t2;
distance  d1, d2;
velocity  v1, v2; // type different but related

v1 = v1 + v2;      // ok
t1 = v1 + v2;      // error (at compile-time)
t2 = t1 + d2;      // error (at compile-time)
v1 = d1 / t1;      // ok
d1 = d1 / t1;      // error (at compile-time)
```

- Dimensional analysis performed by type system.

More than safety

- Good abstractions.
- Intentions stated clearly.
- Improved communication.
- Even less bugs.



Overview

Case study 1

Case study 2

Case study 3

Conclusion

