Avoiding Disasters with Strongly Typed C++

Arno Lepisk

arno@lepisk.se / arno.lepisk@hiq.se

As time before finding bugs increase,
the cost of fixing them rises.

Who am I?

Arno Lepisk

Software Engineering consultant

When do we want bugs to be found

When do we want bugs to be found

- Production

When do we want bugs to be found

- Production
- Q&A

When do we want bugs to be found

- Production
- Q&A
- System testing

When do we want bugs to be found

- Production
- Q&A
- System testing
- Unit testing

When do we want bugs to be found

- Production
- Q&A
- System testing
- Unit testing
- Compile time

Type safety

Type safety

Is C++ type safe?

Type safety in C++

```
void setNodeName(int nodeId,
                 const std::string & nodeName);
```

```
int myId = 3;
std::string myName("foo");

setNodeName(myId, myName); // OK
setNodeName(myName, myId); // Error
```

Type safety in C++ (cont)

```
void setNodeSize(int nodeId,
                 int nodeSize);
```

```
int myId = 3;
int mySize = 7;

setNodeSize(myId, mySize); // OK
setNodeSize(mySize, myId); // !!!
```

Remedy

```
using id_type = int;
using size_type = int;
void setNodeSize(id_type nodeId,
                 size_type nodeSize);
```

```
id_type myId = 3;
size_type mySize = 7;

setNodeSize(myId, mySize); // OK
setNodeSize(mySize, myId); // !!!
```

struct

```
struct id_type { int val; }
struct size_type { int val; }

void setNodeSize(id_type nodeId,
                 size_type nodeSize);
// uses nodeId.val and nodeSize.val
```

```
id_type myId{3};
size_type mySize{7};

setNodeSize(myId, mySize); // OK
setNodeSize(mySize, myId); // Error
```

Add constructor

```
struct id_type {
  id_type(int v) : val(v) {}
  int val;
};
struct size_type {
  size_type(int v) : val(v) {}
  int val;
};
```

Put into template

```
template<typename Tag>
class strongint {
    int val;
public:
    int & get() { return val; }
    const int & get() const { return val; }
    explicit strongint(int v) : val(v) {}
};
```

Put into template

```
template<typename Tag>
class strongint {
    int val;
public:
    int & get() { return val; }
    const int & get() const { return val; }
    explicit strongint(int v) : val(v) {}
};
```

```
using id_type = strongint<struct id_tag>;
using size_type = strongint<struct size_tag>;
```

Performance 1

```cpp
void foo(int);
void foo(id_type);

void bar_int() {
    foo(3);
}

void bar_id() {
    foo(id_type(3));
}
```

x86-64 clang (trunk) (Editor #1, Compiler #1) C++

x86-64 clang (trunk) | -std=c++11 -O3

A | 11010 | .LX0: | .text | // | \s+ | Intel | Demangle | Libraries | Add new...

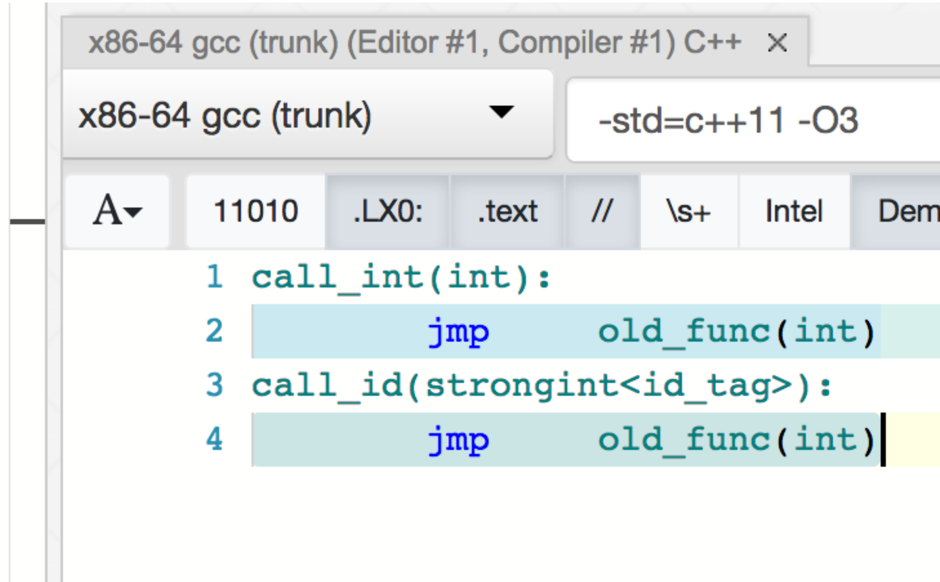
```asm
bar_int():                               # @bar_int()
        movl    $3, %edi
        jmp     foo(int)                 # TAILCALL
bar_id():                                # @bar_id()
        movl    $3, %edi
        jmp     foo(strongint<id_tag>) # TAILCALL
```

Performance 2

```cpp
void old_func(int id);

void call_int(int id) {
    old_func(id);
}

void call_id(id_type id) {
    old_func(id.get());
}
```



But we want an `int`!

But we want an `int`!

```
template<typename Tag>
class arithmeticstrongint {
...
  arithmeticstrongint & operator+(...);
  arithmeticstrongint & operator-(...);
};
```

Hasn't someone else already thouhgt about this?

What is out there?

Jonathan Boccara - `NamedType`

`https://github.com/joboccara/NamedType`

Jonathan Müller - `type_safe`

`https://github.com/foonathan/type_safe`

NamedType

```
#include <named_type.hpp>
using namespace fluent;

using id_type = NamedType<int, struct id_tag,
                          Comparable>;
```

type_safe

```cpp
#include <type_safe/strong_typedef.hpp>
using namespace type_safe;
using namespace type_safe::strong_typedef_op;

struct id_type
    : public strong_typedef<id_type, int>
    , public equality_comparison<id_type>
{
    using strong_typedef::strong_typedef;
};
```

Lets make an `int`!

```
using my_int =
  NamedType<int, struct my_int_tag,
            Printable, Addable, Subtractable,
            Multiplicable, Comparable>;
```

```
struct my_int
  : public strong_typedef<my_int, int>
  , public output_operator<my_int>
  , public integer_arithmetic<my_int>
  , public equality_comparison<my_int>
  , public relational_comparison<my_int>
{
    using strong_typedef::strong_typedef;
};
```

Using `my_int`

```
my_int i1(2);
my_int i2(3);

auto i3 = i1 + i2;
static_assert(
  std::is_same_v<decltype(i3),my_int>
);

std::cout << i3 << std::endl;

// compile time error
// auto i4 = i1 + 1;
```

Get the underlying value

```
my_int mi;
...
// NamedType
auto i = mi.get();

// type_safe
auto i = type_safe::get(mi);
```

Combining types

```
price_t calc_price(price_t price,
                   amount_t amount) {
  return amount * price;
}
```

What do we need?

```
price + price -> price     amount + amount -> amount
price - price -> price     amount - amount -> amount

price * price ??           amount * amount ??

price * amount -> price    amount * price -> price
```

Implementing in type_safe

```
struct amount
  : public strong_typedef<amount, int>
  , public addition<amount>
  , public subtraction<amount>
{ using strong_typedef::strong_typedef; };

struct price
  : public strong_typedef<price, int>
  , public addition<price>
  , public subtraction<price>
  , public mixed_multiplication<price, amount>
{ using strong_typedef::strong_typedef; };
```

Implementing in NamedType

```
using amount =
  NamedType<int, struct amount_tag,
            Addable, Subtractable>;

using price =
  NamedType<int, struct price_tag,
            Addable, Subtractable,
            Multiply??
            >;
```

Our own skill

```
template<typename M>
struct MixedMultiplicable {
  template<typename T>
  struct type : public ::fluent::crtp<T, type> {
    friend T operator*(type const & self,
                       M const & other) {
      return T(self.underlying().get() *
               getValue(other));
    }
    ...
  };
};
```

Helpers

```
template<typename T, typename Parameter,
         template<typename> class... Skills>
const T &
getValue(const
  ::fluent::NamedType<T, Parameter, Skills...> & nt)
noexcept(noexcept(nt.get()))
{ return nt.get(); }

template<typename T>
const T &
getValue(const T & t)
noexcept
{ return t; }
```

Use the skill

```
using amount =
  NamedType<int, struct amount_tag,
            Addable, Subtractable>;

using price =
  NamedType<int, struct price_tag,
            Addable, Subtractable,
            MixedMultiplicable<amount>::type>;
```

Next

```
offset + offset -> offset
offset - offset -> offset
offset * int -> offset

position + offset -> position

position - position -> offset
```

position - position in NamedType

```
template<typename M>
struct SubtractToType {
  template<typename T>
  struct type : public ::fluent::crtp<T, type> {
    friend M operator-(const type & t1,
                       const type & t2) {
      return M(t1.underlying().get() -
               t2.underlying().get());
    }
  };
};
```

offset/position in NamedType

```
using offset =
  NamedType<int, struct offset_tag,
            Addable, Subtractable,
            MixedMultiplicable<int>::type>;
using position =
  NamedType<int, struct position_tag,
            MixedAddable<offset>::type,
            SubtractToType<offset>::type>;
```

type_safe

```
template<typename StrongTypeArg, typename RType>
struct subtract_to_type {
  friend constexpr RType
  operator-(const StrongTypeArg & a1,
            const StrongTypeArg & a2)
  noexcept(noexcept( [...] ))
  {
    return RType(::type_safe::get(a1)-
                 ::type_safe::get(a2));
  }
};
```

offset/position in type_safe

```
struct offset
  : public strong_typedef<offset, int>
  , public addition<offset>
  , public subtraction<offset>
  , public mixed_multiplication<offset, int>
{ using strong_typedef::strong_typedef; };

struct position
  : public strong_typedef<position, int>
  , public mixed_addition<position, offset>
  , public mixed_subtraction<position, offset>
  , public subtract_to_type<position, offset>
{ using strong_typedef::strong_typedef; };
```

use of position/offset

```
position p1(7);
auto o = p1 - position(2);
auto p3 = p1 + o;

static_assert(std::is_same_v<decltype(o),
                             offset>);
static_assert(std::is_same_v<decltype(p3),
                             position>);

// Compile error
// auto pe = position(3) + position(4);
static_assert(!has_op_v<std::plus<>,
                        position, position>);
```

Testing with `static_assert`

```
template<typename, typename, typename,
         typename = std::void_t<>>
struct has_op : std::false_type {};

template<typename OP, typename T1, typename T2>
struct has_op<OP, T1, T2,
  std::void_t<decltype(OP()(std::declval<T1>(),
                            std::declval<T2>()))>>
  : std::true_type {};

template<typename OP, typename T1, typename T2>
inline constexpr bool has_op_v = has_op<OP,T1,T2>::value;
```

Length

```
void setLength(length_t l);
```

std::chrono::duration

```
using namespace std::chrono_literals;

constexpr auto s = 1s;
constexpr std::chrono::milliseconds ms = s;

static_assert(s.count() == 1);
static_assert(ms.count() == 1000);
```

No!

No!

How does chrono work?

```
namespace std {
typedef duration<int64_t> seconds;
typedef duration<int64_t,
                 milli>   milliseconds;
}
```

std::ratio

```
milli = std::ratio<1, 1000>;
```

std::ratio

```
milli = std::ratio<1, 1000>;
```

```
std::ratio<18,4>::type -> std::ratio<9,2>

std::ratio_add<ratio<1,2>, ratio<1,5>>
  -> std::ratio<7,10>;
std::ratio_subtract<ratio<1,2>, ratio<1,5>>
  -> std::ratio<3,10>;
std::ratio_multiply<ratio<1,2>, ratio<1,5>>
  -> std::ratio<1,10>;
std::ratio_divide<ratio<1,2>, ratio<1,5>>
  -> std::ratio<5,2>;
```

Our own length

```
template<typename T,
         typename Scale = std::ratio<1>>
class Length {
  // Construction, addition, subtraction etc
};
```

Some convenience defines

```
template<typename T> using Meter =
  Length<T>;
template<typename T> using Millimeter =
  Length<T, std::milli>;
template<typename T> using Kilometer =
  Length<T, std::kilo>;

template<typename T> using Inch =
  Length<T, std::ratio<254,10000>::type>;
template<typename T> using Foot =
  Length<T, std::ratio<3048,10000>::type>;
template<typename T> using Mile =
  Length<T, std::ratio<1609344,1000>::type>;
```

Add operations

```
constexpr Meter<int> m1(1);
constexpr Meter<int> m2(2);

static_assert(m1 + m1 == m2);
```

Add operations

```
constexpr Meter<int> m1(1);
constexpr Meter<int> m2(2);

static_assert(m1 + m1 == m2);
```

```
length<int, ratio<<1>>(l1) +
length<int, ratio<<1>>(l2)
 -> length<int, ratio<1>>(l1+l2)
```

Different underlying types

```
Meter<int> + Meter<double> -> Meter<???>
```

Different underlying types

```
Meter<int> + Meter<double> -> Meter<???>
```

```
Meter<int>(li) + Meter<double>(ld)
  -> Meter<decltype(li+ld)>
```

Different ratios

```
l1 * (n1/d1) + l2 * (n2/d2)
```

Different ratios

```
l1 * (n1/d1) + l2 * (n2/d2)

gcd(n1/d1, n2/d2) = gcd(n1,n2) / lcm(d1,d2) = nc/dc
```

Different ratios

`l1 * (n1/d1) + l2 * (n2/d2)`

`gcd(n1/d1, n2/d2) = gcd(n1,n2) / lcm(d1,d2) = nc/dc`

```
Meter<int>(2) + Foot<int>(3) ->
2 * 1/1 + 3 * 381/1250 ->
2 * 1250 * 1/1250 + 3 * 381 * 1/1250 ->
3643 * 1/1250
```

Use of length

```
constexpr Foot<int> f1(1);
constexpr Inch<int> i12(12);
static_assert(f1 == i12);
```

Use of length

```
constexpr Foot<int> f1(1);
constexpr Inch<int> i12(12);
static_assert(f1 == i12);
```

```
constexpr auto c =
  Inch<int>(5) + Centimeter<int>(8);
static_assert(c == Millimeter<int>(207));
```

Add some sugar...

```
static_assert(1_ft == 12_in);
static_assert(1_m == 100_cm);
static_assert((5_in + 8_cm) == 207_mm);
```

Dimensions

- length * length -> area
- length * length * length -> volume

Dimensions

- length * length -> area
- length * length * length -> volume
- length / length -> scalar

Dimension aware "length"

```
template<typename T, int Dim,
         typename S>
class Length {
...
};
```

Operations on dimension-aware length

```
length<T, D, S> + length<T, D, S> ->
  length<T, D, S>

length<T, D1, S1> * length<T, D2, S2> ->
  length<T, D1+D2, S1*S2>

length<T, D1, S1> / length<T, D2, S2> ->
  length<T, D1-D2, S1/S2>
```

Use of dimension aware length

```
auto area = Foot<int>(3) * Foot<int>(4);
std::cout << area << std::endl;
std::cout << SquareMeter<double>(area)
          << std::endl;
```

```
12 ft²
1.11484 m²
```

Physical quantities

- length * length -> area
- length / time -> velocity

- velocity / time -> acceleration

- acceleration * mass -> force

- force * length -> energy
- energy / time -> power

Unit

```
template<typename U,
         int L, int M, int T,
         typename S>
class unit {
...
};
```

Operations

```
unit<U, L, M, T, S> + unit<U, L, M, T, S> ->
  unit<U, L, M, T, S>

unit<U, L1, M1, T1, S> * unit<U, L2, M2, T2, S> ->
  unit<U, L1+L2, M1+M2, T1+T2, S>

unit<U, L1, M1, T1, S> / unit<U, L2, M2, T2, S> ->
  unit<U, L1-L2, M1-M2, T1-T2, S>
```

Example multiplication

```
Length = unit<int, 1, 0, 0>
Time   = unit<int, 0, 0, 1>

Length * Length -> unit<int, 2, 0, 0> // Area
Length / Time   -> unit<int, 1, 0,-1> // Velocity
```

Constants

```
Acceleration = unit<T, 1, 0,-2, S>;
constants<int> {
    Acceleration<int, std::centi> g{982};
};
constants<double> {
    Acceleration<double, std::ratio<1>> g{9.82};
};

template<typename T>
constexpr auto g = constants<T>::g;
```

Constants usage

```
auto t = Second<int>(3);
auto d = t*t*g<int>/Id<int>(2);
assert(d == Centimeter<int>(4419));
```

Other units

Other units

Things are getting complicated again

boost::units

https://www.boost.org/doc/libs/1_68_0/doc/html/boost_units.html

Voltage, current, power

```
auto P = (110 * volt) * (0.5 * milli * ampere);
std::cout << P;
```

```
55 mW
```

Length in boost::units

```
auto len = 3 * meter
         + quantity<length>(10 * milli * meter);

std::cout << len << std::endl;

static const foot_base_unit::unit_type foot;
auto combined = 1 * meter +
                quantity<length>(1 * foot);

std::cout << combined << std::endl;
```

```
3.01 m
1.3048 m
```

Links

- https://github.com/f00ale/strong_type_examples
- https://github.com/f00ale/stype
- https://github.com/f00ale/units

Conclusions

- C++ can be used in a type safe manner
- Use strong types
- A lot can be made without performance impact

Thanks for listening

arno@lepisk.se / arno.lepisk@hiq.se

@arno_l